January, 1996

WILD SOFTWARE META-SYSTEMS

Kent D. Palmer
WILD SOFTWARE
META-SYSTEMS

Essays in Software Engineering and Philosophy

Kent D. Palmer, Ph.D.
Software Engineering Technologist

Apeiron Press
Orange
California
Software Engineering Design Methodologies and General Systems Theory

Kent D. Palmer, Ph.D.
Software Engineering Technologist
PO Box 1632 Orange CA 92866
kent@palmer.name


Software Engineering\(^1\) is slowly emerging as a discipline in its own right from the craft of computer programming\(^2\). It is a discipline grown up in industry rather than invented in academia. It addresses the problems that occur when a group of people attempt to construct a large software system. The key word is \textit{large} because when the software product gets large enough that it can no longer be completely understood by one person, or a few, certain problems emerge that are not seen in smaller systems. These are problems like the need to communicate the design to others; the need for concurrent cooperative software design by a large number of people working on different aspects of the same problem; the need to control the artifacts produced by the essential transformations in the software development process; and the need to reuse as much of previous software systems as possible. These are the kinds of issues that emerge when software development attacks large problems and builds hundreds of thousands or millions of lines of software code.

Software Engineering is still in its infancy as a discipline, yet already it is experiencing paradigm shifts\(^3\). A paradigm shift is a radical change in the way of looking at known facts of a discipline. In Software Engineering the current paradigm shift is away from functional design\(^1\) toward what is called object-oriented design\(^5\). This is a shift away from looking at pieces of software as functions that transform a set of inputs into outputs, toward looking at the persistent data in a system, and seeing the functions that operate on these pieces of persistent data as being grouped around that private data. This paradigm shift is causing old problems to be seen in a new light. And it is causing a rethinking of the way we produce software systems.

Although Software Engineering is developing and changing rapidly within itself, the relationship it has to other disciplines is still unclear because it is not well integrated into the academic establishment as yet. Thus, the new inventions of Software Engineering within its own field may seem baffling or be seen as an exercise in re-invention by other more established disciplines. This problem is exacerbated by the fact that Software Engineering is perceived as a renegade newcomer. At the same time these more established disciplines are themselves developing software systems to support their own applications. So, although they may not have a good connection to what is happening in Software Engineering as a discipline, they have a connection to software products that they use in their own ongoing work. There is a vital need for other disciplines to learn what is happening in Software Engineering, and for software engineers to understand what other disciplines have to offer toward the development of the Software Engineering discipline.

One of the most vital of these connections is between Software Engineering and General System Theory. Software engineers profess to be building software \textit{systems}. But, beyond the loose generic use of this term, there is not a lot of material within the corpus of Software Engineering literature concerning the generic nature of systems of which software systems is an instance. On the other hand, General Systems Theory, which gives itself wholly to attempting to understand generic systems like

---

5. “The evolution of the Software Engineering Discipline” Mary Shaw ?????
6. The Structure of Scientific Revolutions Thomas Kuhn ????
7. Structured Design Yourdon / Constantine ?????
8. Some ref on OOD ???

Copyright 1994 Kent Palmer. All rights reserved. Not for distribution. Rough draft for review only.
many other disciplines, uses software applications in its own work, and perhaps is unaware of the developments within Software Engineering as a discipline; developments which are interesting because of the additional light they shed on systems in general. Because software systems emulate many different types of systems and allow them to be tested and interrogated as simulations, these software simulations are a valuable tool that function as the instruments by which General Systems theorists do much of their investigation. Yet these tools may not be designed and implemented using the methods which are currently state-of-the-art within Software Engineering. Those who build software models are generally operating within the programming craft stage of software discipline.

Thus, the question arises: What do General Systems Theory and Software Engineering have to learn form one another? The answer that will be explored in this paper is that these are both meta-disciplines, and that they actually have a symbiotic relation with each other. General Systems Theory looks at all kinds of systems that appear in the world, and attempts to abstract their generic features. This allows similar systems in widely different fields to be recognized. Software Engineering builds simulations of systems in widely varying fields by applying the same techniques for understanding and automating those systems. Thus, with General Systems Theory it is the representation of the systems that are abstracted, while in Software Engineering it is the means of embodying the abstracted or concrete system that is generic. The telling thing is that General Systems Theory itself uses software representations of abstract systems such as the General Systems Problem Solver (GSPS) provides. Software Engineering, on the other hand, designs and builds software 'systems'. So, not only are both meta-disciplines which abstract from and are used by all other disciplines, but each of these meta-disciplines need and use each other. For General Systems Theory software is the means of simulation and instrumentation of the systems being studied. For Software Engineering, a specific kind of 'system' is being built.

Software Engineering and General Systems Theory are symbiotic. The question is, what does this really mean for the relation between the two meta-disciplines? This will be the subject of the rest of this paper.


It is somewhat understandable why software engineers might ignore their sister meta-discipline. Generally Systems Theory itself has had a hard time getting recognition of its claim to a central role in science. It has been difficult to show that Systems Theory has anything to contribute other than generalizations which, because they are too abstract, do not really tell you anything. (Software Engineering, on the other hand, suffers from the criticism that everything it has to offer is too concrete.) Unfortunately, the connections between disparate disciplines that treat specific kinds of systems through General Systems Theory have not yet happened. Recognizing that the systems treated by different disciplines are really projections of the same general system has not yielded enough fruit to make this an important aspect of science.

General Systems Theory stands in the same relation to the sciences as Mathematical Category Theory does to the separate kinds of mathematics called categories. We cannot yet see the functors (homeomorphic relations) between different systems that General Systems Theory should eventually provide. However, with the advent of the structural General Systems Theory of George Kli, it is possible to see the outlines of this science emerging. It is only right that this system should be expressed in terms of Mathematical Category notation as it is in another article in this issue. We might expect software to have some affinity with category theory as well. Since software is the universal implementation medium of all mathematical categories, and because software has only mathematical limitations to its structure and shape, we can see that Category Theory is well suited to the understanding of the relationship between software designs and methods.

7. Categories, Types, Structures Andrea Asperti Giuseppe Longo QA76.7A76 1991
It is important to understand why GSPS is an advance over previous systems theories. I would characterize GSPS as a structural-formal systems representation, as opposed to merely a formal theory. Early systems theory really did little more than posit that “everything is a system.” What GSPS adds is the structural dimension that differentiates between different systems architectures. This is accomplished by invoking a hierarchy of epistemological levels\(^8\). The recognition of these levels is very important, as it gives clear definition to the subject matter of General Systems Theory and the differentiation of the field of possible systems architectures. Knowing what the structure of this field is, must be an essential step in the recognition of different systems with the same architecture. Since software allows any mathematically possible structure, software designs may be seen to cover the same field. It is the absolute malleability of software (within the constraints of hardware) that makes it fit to model all the diverse actual systems that are specific points in the field of all possible system designs mapped by General Systems Theory.

Given the epistemological levels posited by GSPS, the obvious question occurs as to what the relationship of software design methods (which allow different designs to be represented) is to the overall epistemological framework. But, first we must understand something about the underpinnings of the GSPS framework in its role as the articulator of design space for systems. There is a very important point about this framework which is perhaps not widely appreciated: the framework goes to infinity in two directions. The semi-lattice diagram\(^9\) shows meta-systems diverging from meta-structures as two completely different infinite regresses of meta-levels of description in orthogonal dimensions. The first

---

8. See Klir ASPS Figure 1.3 Hierarchy of epistemological levels of systems: a simplified overview. Page 16. Represented in Figure 1.

9. See Klir ASPS Figure 9.1; page 420
dimension is structural with patterns within patterns in part-whole relationships. The second dimension is meta-systemic where discontinuous changes are handled by positing modal changes of the system. These two dimensions are very different. Meta-systemic patterns are diachronic and have to do with the changing of patterning regimes within time. Pattern meta-levels are synchronic embeddings of higher logical types of order. They have to do not with time, but with space where data has a different fractal patterning at each partial dimensional level. We might say that the second dimension that deals with meta-models is the process dimension because it allows embedded patterns or structures to vary in time. The meta-models determine the discontinuous changes between embedded patterns or structures to vary in time.

Together, these two infinite regresses show that ultimately any system is unrepresentable in its entirety. This is because after just a few meta-levels of spatial or temporal embedding we get lost in the ultra complexity of all but the simplest dynamical systems. The most we can hope to do for all, but inherently deterministic systems, is give an approximation that goes up just a few meta-levels in relation to structural pattern and process meta-systems.

This framework in General Systems Theory both makes its limits clear, and also allows it to represent the architecture of diverse systems by allowing higher and higher logical types of description along both of these dimensions. But, we are led deeper to ask why this limitation exists and what it represents. This limitation has to do essentially with the way in which systems themselves are manifested. It is not merely an artifact of our descriptive conventions. Because General Systems Theory purports to be a science, it operates with the same self-imposed metaphysical blinders as all sciences. However, this does not mean that it does not have metaphysical presuppositions built into it. GSPS has its own metaphysical presuppositions, and one of those is that the ultimate system can never be known. That limit system that is unknowable is an image of the Kantian noumena. This unknowability is built into the theory as the twin infinite regresses. By building in this presupposition, GSPS is able to fine tune its description of systems by providing a structural model of their possible systems architectures. It is able to provide a representation that can track the changes in the system over time and to any level of sub-patterning. On the other hand, it can handle multiple interlaced part-whole patterns. However, the non-manifestation of the system as noumena is a steep price to pay for this structure and flexibility. It is this metaphysical dimension that needs to be appreciated because it is exactly here that the innate affinity with software can be shown to exist most strongly.

Dr. Klir has in several papers pointed out the new information aspect of science that he proposes to explore. This new aspect has its own metaphysical attributes which it shares with software. We can best characterize this new aspect by relating it to the concept of “Being” which is generally thought to be the most general, and thus most empty, concept. Heidegger gives the concept of Being content by associating it with ‘presencing’ or ‘manifestation.’ Being is the process of manifestation of “whatever manifests.” From the time of Aristotle, through Kant, and up to that of Husserl, this manifestation has been thought of as pure presenting of clear and distinct objects that can be seen from an ideal point of view that sees all sides of the object at once. In this ideal presenting the noumena, the inner coherence of the objects, remains hidden because it is on a transcendental plane. However, the object itself seems to be fully available. This is the type of manifestation that science assumes when it looks at its objects. It assumes that the object is fully available for inspection in every detail. This is the type of Being Klir assumes when he looks at his Object systems and turns them into Source systems. However, by differentiating the Object from the Source systems, he is making a crucial admission that the Source system is only seen to the extent it is turned into an object viewed by a passive observing subject. The process by which such a system is turned into an object is unspecified except that it entails some type of special focusing, such as that Descartes introduced into science, that makes clear and distinct (delimited) objects out of the blurry masses presented to the natural senses.

Seeing a system rather than an object is not differentiated by Klir. To him a system is an object which has been instrumented, or made available, for observation. However, I believe that a system is on a completely different ontological meta-level from the object, and I think this is implied by Klir’s usage. I think this difference between object and system is the same difference that Heidegger makes between Pure Presence type of Being and what may be called Process Being, which is “Being mixed with time.” In Process Being the process of

---

10. Klir on information sciences
12. Heidegger on Manifestation???
manifestation of the object is taken into account. The major effect in this manifestation is called showing and hiding. In manifestation not all aspects of the system are seen at once. The way in which the object comes into and goes out of view is also taken into account. This has very important ramifications for systems science which is not widely appreciated. Because systems science sees itself as science at a meta-level above all other disciplines, it is really ignoring the fact that it is a ‘meta-physical’ discipline. The word ‘system’ is the catch-all that attempts to bridge this gap. When you ask what a system is, you get back the answer that “everything is a system” that you consider as such. So ‘system’ is really a way of seeing the world: it is a worldview, a metaphysical approach to reality.

Fortunately, Nicholas Rescher gives a clearer definition of the characteristics of a system:

Lambert contrasted a system with its contraries, all “that one might call a chaos, a mere mixture, an aggregate, an agglomeration, a confusion, an uprooting, etc.” ... And in synthesizing the discussions of the early theoreticians of the system-concept, one sees the following features emerge as the definitive characteristics of systematicity:

1. wholeness: unity and integrity as a genuine whole that embraces and integrates its constituent parts
2. completeness: comprehensiveness: avoidance of gaps or missing components, inclusiveness with nothing needful left out
3. self-sufficiency: independence, self-containment, autonomy
4. cohesiveness: connectedness, interrelationship, interlinkage, coherence (in one of its senses), a conjoining of the component parts, rules, laws, linking principles; if some components are changed or modified, then others will react to this alteration
5. consonance: consistency and compatibility, coherence (in another of its senses), absence of internal discord or dissonance; harmonious mutual collaboration or coordination of components “having all the pieces fall into place”
6. architectonic: a well-integrated structure of arrangement of duly ordered component parts; generally in an hierarchic ordering of sub- and super-ordination
7. functional unity: purposive interrelationship; a unifying rationale or telos that finds its expression in

some synthesizing principle of functional purport
8. functional regularity: rulishness and lawfulness, orderliness of operation, uniformity, normality (conformity to “the usual course of things”)
9. functional simplicity: elegance, harmony, and balance, structural economy, tidiness in the collaboration or coordination of components
10. mutual supportiveness: the components of a system are so combined under the aegis of a common purpose or principle as to conspire together in mutual collaboration of its realization; interrelatedness
11. functional efficacy: efficiency, effectiveness, adequacy to the common task.

These are the definite parameters of systematization. A system, properly speaking, must exhibit all of these characteristics, but it need not do to the same extent -- let alone perfectly. These various facets of systematicity reflect matters of degree, and systems can certainly vary in their embodiment.

These characteristics of system are ignored by the objectivists which want a simple operational concept of system which is fully available, but essentially empty of content. These characteristics are implicit in the concept of system as it is actually used. This is true even of the objectivists, which is shown by the fact that the concept of an “object” remains different from that of “system.” The added implicit meanings of system are those pointed out above by Rescher. All of these characteristics of the system relate to the difference between the object and what might be called the ‘temporal gestalt.’ The object is a synchronic slice out of the diachronic temporal gestalt at the idealized “now” point. The temporal gestalt itself includes the whole evolution of the object. It thus inhabits at least the so-called “specious present,” or the extended now, and moves out toward the entire temporal interval in which the object manifests. The temporal gestalt is modeled by the process dimension of Klir’s epistemological framework. The difference between Pure Presence and Process Being is the difference between the temporal slice and the whole process of unfolding. It is in this process of unfolding that the characteristics of system enumerated above unfold. As a temporal slice, the object seems to have almost arbitrary features because any slice of the whole process of unfolding might have been taken. But if you consider the whole process of temporal unfolding, you get a complete dynamic picture or gestalt. One gets a sense of the wholeness of the system and can judge the completeness of any slice. The temporal gestalt exhibits apparent self-sufficiency because it is an entirely

17. William James
If we release our metaphysical imaginations for a moment, for itself through its own self-articulation over time. The observed phenomena to have its own voice and speak subjective whim. The system as temporal gestalt allows ordering it as a passive strawman determined by our with phenomena. The object controls the phenomena by the aim of controlling, rather than engaging in dialogue we will want to stick to the objectivist metaphysic that has the world and exclude whatever does not fit with our ordering, To the extent we want to project our own order on the world and exclude whatever does not fit with our ordering, we will want to stick to the objectivist metaphysic that has the aim of controlling, rather than engaging in dialogue with phenomena. The object controls the phenomena by ordering it as a passive strawman determined by our subjective whim. The system as temporal gestalt allows the observed phenomena to have its own voice and speak for itself through its own self-articulation over time.

If we release our metaphysical imaginations for a moment, we might see that systems are objects caught in webs of showing and hiding. Thus, the difference between an object and a system is whether it is looked on as something that is completely available or something that is only partially available and at least partially determines its own availability. The partial availability prevents us from capturing all of the Object systems in our source system. The partial availability is what separates the object from the source. They are really two ways of looking at the same thing. Moving from source system to object system, one has moved from blurs of perception to idealized objects that are purely available as sources for observation and experimentation by a subject. We have put on the metaphysical blinders which makes us think our scientific objects are “objectively” accessible to an intersubjective community.

For Heidegger, the realm of showing and hiding is not just something to be escaped from by science. Instead, this realm is the infrastructure that science itself simultaneously depends on and ignores. It is the realm of technological means. A simple demonstration can give you the feeling for this. In writing, you are unaware of your pencil. Your focus is on what you want to say, not on the movement of your hand in recording what you are saying. This is the difference between present-at-hand (pure presence of the object) and the ready-to-hand way of relating to the pencil which is moving in time and supporting the objective focus. So it is with all technology. Technology supports the endeavors of science. And the major instrument through which this is done today is the computer. In fact, it is interesting that the computer has in its make up all the elements of the structural system which join present-at-hand and ready-to-hand modes of relating to things in a single unified mechanism. This is what makes the computer a general purpose machine that is applicable to almost every task.

One might look at the difference between “object” and “system” as being like the difference between figure and ground in gestalt psychology. The object is the focus of attention. The system is the interaction of all the possible focuses in the field of the system. However, you can only see one or a few objects at one time, so in looking over the entire field, the center of focus must shift from object to object. This is the manifestation of showing and hiding within the field, that is, in fact, what we call the system. The system is a group of objects in the same field of showing and hiding relations.

Another way to look at this is by considering the difference between Formal systems and Structural systems. In a Formal system such as symbolic logic, geometry, or any mathematical category, there is a set of axioms, and it is through the relationship between the axioms that the full Formal system is built. However, this system is made of purely logical relations, and time does not effect the system except to the extent it takes time to go through the proofs step by step. On the other hand, a Structural system is a powerful means for understanding discontinuous transformations. The Structural system attempts to apply formal rules to the content of the Formal system in order to track the changes that occur across discontinuities. These discontinuities can be structural or changes in pattern at different levels of magnification in one synchronic slice. The Structural system is basically descriptive. It gives up the rigor of deduction for the ability to handle change. George Klir’s GSPS is essentially a Formal-Structural system. Its epistemological levels allow different Formal systems to be connected together to form a Structural system that can handle change. It is, in fact, the best example of a Structural system that I have found. Structuralism has been most effective in understanding the universe in the hands of physicists and chemists. It has even made strides in biology with the discovery of the DNA genetic code. And some use of it has been made in the social sciences.

---

18. Natural Complexes

with Structuralists like Noam Chomsky\textsuperscript{20}, Claude Levi-Strauss\textsuperscript{21}, Jean Piaget\textsuperscript{22} and others, using it to describe aspects of human nature and culture.

However, the difference between object and system as two different ways of looking at things does not exhaust the implicit metaphysics in GSPS. By introducing the concept of the two infinite regresses, GSPS implicitly hints at the presence of another metaphysical assumption which has also appeared in Modern Ontology. The question occurs to us, where do these two infinite regresses lead? That place is where the system exists as unreachable noumena. The system as noumena is very different from the objective noumena of the idealists. In modern ontology, various philosophers have sought to describe what the next higher meta-level of Being is above Process Being. Heidegger calls that next meta-level Being (crossed out). Derrida has called it DifferAnce\textsuperscript{23} which he describes as the differing/deferring of writing. In this conception we differentiate between speech and writing. In speech, a showing and hiding process occurs, and it carries us along, presenting us a series of ideas that appear fully available in language. The flow of language is the emanation of the temporal gestalt in action. It is the basic pattern for the ready-to-hand's technological support of ideation. However, writing is different. Writing makes it possible to create partial texts which are scrambled into a different order than that in which they were written. The text is a field of pure difference in which things that were together may appear apart, or in different orders, than those in which they were composed. Another ontologist, Michael Henry, speaks of what is called the Essence Of Manifestation\textsuperscript{24} which is the pure immanence underlying the manifestation (aka transcendence) that we see as showing and hiding. What is purely immanent never appears, but shapes what does appear. It becomes particularly conspicuous in texts which are susceptible to unconscious distortion. You know it is there by the way what appears is distorted or malformed. Taking these two ideas (differAnce and the unconscious essence of presencing) together, we can see their relevance to GSPS. GSPS posits two infinite regresses which hide the noumena of the system. We would like to call the noumena of the system a meta-system or proto-gestalt. In other words the temporal gestalt arises from an always already lost origin from which many systems (gestalts) emanate and toward which they all point. This origin is the next higher meta-level beyond the system. The noumena of the system (essence of presencing as such) never appears because it has been first covered over multiple times. Firstly, it is covered over by the extraction of the object from the source system into the realm of pure availability. Secondly, it is covered over by the showing and hiding relations between multiple objects which makes the system itself something that is never seen because it is always in the background. All we really see is the multiple showing and hiding relations. Thirdly, the system is covered over by the infinite regress of meta-layers of descriptions. These twin infinite regresses (in space and time) have as their origin, which is never seen, a singularity which is the noumena that represents the inner coherence of the system itself. That singularity is the source of the unity of all the systems characteristics spoken of by Rescher. Rescher points out that even the systems characteristics appear as independent of each other. This reminds us of Deleuze and Guattari's dictum\textsuperscript{25} that all contents that genuinely emanate from the unconscious must be orthogonal and independent. If there is any relation between the apparently emanating contents then we are dealing with projections of consciousness and not with the unconscious as such. This is why the systems characteristics are independent of each other. They emanate directly from the essence of manifestation or the unconscious origin of the system in the meta-system or proto-gestalt. They are ultimately empirically derived from the observation of organisms\textsuperscript{26}. The systems characteristics form a system which is a complex gestalt of “closely interrelated and mutually complementary” elements; but “the crucial characteristic of such cases is the conjointing of various factors that are in theory separable from one another but in practice generally found in conjunction\textsuperscript{27}.” So the systems characteristics themselves form a broken whole with no synthetic unity (totalization). Because of the discontinuities within the set of systems characteristics, there is room left for the action of the essence of manifestation. The inner coherence of the system must be inferred from the fractal and temporal patterning, but nowhere does this appear explicitly. This is the realm of pure immanence that never appears except as distortions and breaks in the always imperfect descriptions, that may be fruitfully compared with the unconscious discovered by psychologists in the system of

\textsuperscript{20} N. Chomsky on Transformational Grammar
\textsuperscript{22} J. Piaget, Structuralism. ????
\textsuperscript{24} M. Henry, The Essence Of Manifestation. ????
\textsuperscript{25} Deleuze & Guattari, Anti-Oedipus. University of Minnesota Press, Minneapolis
\textsuperscript{26} op. cit. page 12
\textsuperscript{27} N. Rescher Cognitive Systematization. page??

Kent Palmer
consciousness. The systemic unconscious (un-conscious means non-manifesting) was called by Freud 28 from the first, the ID, meaning “it.” The “IT” of systems is the essence of their manifestation that no-where appears as concrete foci, but appears globally in the distortions and breaks in system functioning that causes all the meta-levels of description to be at least slightly out of “sync” with the real system.

You may begin to get the idea that the meta-levels of Being, as we see them covertly in General Systems Theory, have more to do with our perception and understanding of the system than with the things being observed themselves. Well, you would be right because meta-levels of Being have to do with the manifestation of everything, including scientific objects. Science believes it can be clever by claiming metaphysical ignorance. But the best of scientific theory ends up mimicking the ontological distribution of entities in the world, and thus begrudgingly gives an accurate metaphysical picture in spite of claiming ignorance. So it is also with GSPS. GSPS accurately portrays three of the four possible 29 different meta-levels of Being discovered by modern ontologists. And it is through this accurate portrayal of these meta-levels that Klir manages to claim the high ground from which he can look down on all the other sciences supported by their implicit technologies. Each science has a focus, and it keeps that specialized focus and works on its object by applying the requisite technology. The application of that technology makes the science into a system, and becoming a system allows it to treat its object as a system. But General Systems Theory correctly thinks of itself at a meta-level above systems. It is the supra-system of all possible systems. It therefore maps out the terrain of all possible systems designs. By mapping this terrain, it allows the theorist to identify a particular instance of an abstract system and to recognize the generalization of an instance.

Figure 2: Essential View Points on Software Design Theory

An important point is to recognize that software also operates on this third meta-level of Being. It is by representing this third meta-level, that software has its true kinship with General Systems Theory. However, software appears on the third meta-level of Being in a completely different way from General Systems Theory. Software is a form of writing. It is an automated writing. It is writing that can change itself. Thus, Software is an excellent example in the real world of what Derrida calls “differAne.” In software the operation of differing and deferring appears in the spaghetti code of multiple intertwined GOTOs; in self-rewriting code; in side effects; and most importantly for us, in delocalization. 30 Delocalization is, in effect, exactly the type of phenomenon that Derrida wants to describe with the term “differAne.” Delocalization causes the ideas, or design

elements, expressed in the text to be spread out and interspersed within the text. It represents the opacity of the text itself. Any one part of the text seems perfectly clear. But we cannot hold all of it in our memory at once. So the text is not all perfectly available. There is “showing and hiding” as we scroll the source code up and down in our editor. But there is an even more severe “showing and hiding” effect as the software executes. That execution may be very different than one imagines just looking at the text of the program. This is called a bug or sometimes a side-effect, depending on the context. It is the cognitive dissonance between what we see on the screen in the code and what the actual action of the code turns out to be when it is complied and executed. Getting rid of these bugs, or dealing with side effects, normally determines the extent of delocalization as statements are moved so they do not conflict with each other. Between the code and the action in execution there is a blin spot in which difference appears.

So, Software and General Systems Theory are two completely different manifestations on the same meta-level of Being. There may be others at this level, but one thing for sure is that General Systems Theory and software are complementary and need each other. Software objects are always systems. They are inherently means of showing and hiding which are automated. General Systems Theory uses these mechanisms of showing and hiding to simulate concrete systems at some level of abstraction. General Systems Theory rises above all concrete systems by displaying the inherent mathematical structure of the field of all possible systems structures. So, where software is a very concrete manifestation of level three meta-level of Being, GSPS is a very abstract manifestation. But it is necessary to go to this third level because naive systems theories (that do not go to this level) tend to appear empty. By describing the field of all possible systems structures, GSPS proves useful because it surveys the field of combinatorial structural differences that underlie all systems.

Here we do not explore the fourth meta-level of Being called Wild Being by Merleau-Ponty. This is because both GST and Software are articulated at the third meta-level and do not address the fourth meta-level. But we can point the reader to the work of Deleuze and Guattari who attempt to define a philosophy that addresses this meta-level of Being. This meta-level is the highest articulation of Being into kinds and beyond that lies the unthinkable. The meta-levels of Being are analogous to the meta-levels of learning that Bateson defines and which will appear later in our argument. Both hierarchies have a finite endpoint. For phenomena related to computation that exist at this highest level of the articulation of manifestation we must turn to artificial life and intelligence techniques as examples. Software engineering attempts to separate itself from all techniques of programming that do not have deterministic or easily predicted results. Those techniques tend to appear in Artificial Intelligence systems.

The four kinds of Being arise from an analysis of modern Continental philosophies where the phenomena of the fragmentation of Being into different kinds was first noticed and appropriated as the central motif of modern phenomenological inquiry. The first philosopher who signaled this possibility was Husserl who was followed by his student Heidegger. Heidegger defined the difference between the first two meta-levels of Being in his seminal work Being and Time based on his teachers definition of essence perception (eidetic intuition) as a mode of apprehension radically different from the modes of apprehension associated with induction and deduction. From that beginning modern ontology has attempted to explore the relation between these two modes of being and their implication for our relation to the world. In the process other kinds of Being were discovered. Foremost in this work has been Merleau-Ponty who translated Heidegger’s modalities of Being called present-at-hand and ready-to-hand into the psychologically understandable ways or relating to objects through pointing and grasping in his work The Phenomenology of Perception. At the end of that work he discovers the next modality of relating to things in the world which I call the in-hand and which he later names Hyper Being. Finally in his last and incomplete work The Visible and the Invisible he defines Hyper Being formally and then points to the possibility of a fourth meta-level of Being called Wild Being. It is only with the work of Deleuze and Guattari in Anti-Oedipus that this last highest meta-level of being is fully explored philosophically. These meta-levels of Being discovered by modern European philosophers have deep significance for our understanding of how the world is projected within our worldview. Here we use them as a back drop for analyzing the relation of General Systems Theory to Software Engineering and show that both disciplines have the properties they do because they exist at the third meta-level of Being.
3. Software Design Methods

Software Design Methods are the major contributions of Software Engineering as a discipline which sets it apart from Computer Science. Sometimes methods are referred to as programming-in-the-large, as opposed to programming-in-the-small. Programming-in-the-small is done using computer languages and by writing source lines of code, one at a time. Programming-in-the-large attempts to see the global picture of what is happening and make it globally coherent. This means rising to higher levels of abstraction where delocalization does not effect design elements. Software Methodologies give the software designer a means of doing programming-in-the-large systematically. There are a myriad of such methodologies being proposed. The methodologies breakdown into two basic kinds:

- Understanding the problem.
  - Hatley-Pirbhai Structured Analysis
  - Ward-Mellor Structured Analysis
  - Shaler-Mellor Object Oriented Analysis

- Designing the Solution.
  - SPC-Gomaa (Ada Design & Analysis for Real Time Systems) ADARTS
  - Yourdon-Constantine Structured Design
  - Neilson-Shumate Object Oriented Design/Virtual Layered Machine

These are some representative examples of current design methodologies. There is a lot of change in this field today. Many new methodologies are being proposed every year. Major paradigm changes are underway, such as that from functional design to object-oriented design. This field has not yet begun to look at itself critically. However, what we can see emerging is a series of different representations for abstractly portraying all automated real-time embedded systems. Embedded systems function within machines by controlling sensors and actuators. Such systems must necessarily be real-time because they must react to actual events that are sensed and make things happen within the technological system. Thus, Software Design Methodologies (SDM) are the representation by which software systems will be adapted to myriad applications in the world. In this sense they are a specific type of systems architecture, but with very wide applicability. One might say that it is the class of architectural elements necessary for the technological system as a whole. The technological system as a whole, to the extent it becomes integrated, will be tied together through software which is designed using these representations. For this reason we can think of software as a meta-technology that integrates myriad systems within the technological infra-structure making it a whole.

Software Design Methodologies were not, for the most part, invented by academicians. Instead, they were developed by software engineering practitioners to deal with the problems of conceptualizing the abstract structure of software designs, and to allow those to be recorded and constructed. Software design methods allow major problems with the design of software to be addressed. Normally, for small systems programmers can design the software as they write it. This is called “direct programming” or sometimes “hacking.” They design the application, working within the mesh of delocalization as they attempt to think about the whole design. This practice, which works for small systems, leads to disaster when large systems are designed. The heart of the problem is delocalization that cannot be controlled under the interaction of dispersed design elements. Any particular design element at the software architecture level may be spread out throughout the code at the implementation level so that its side effects become a totally unknown factor. This fragmentation of the design elements makes it nearly impossible for the design to be recovered from the code. Object-oriented design attempts to reduce the effects of delocalization, but there is always some residue left over within the software system. Design elements which interpenetrate within the code become indistinguishable, so that the architecture is submerged and cannot be reconstructed without a lot of effort. This problem is exacerbated when performance issues produce

---

36. E. Yourdon & L. Constantine, Structured Design Book ???
38. E. Yourdon & L. Constantine, Structured Design Book ????
Software Engineering Design Methodologies and General Systems Theory

distortions in the software architecture that compound the problem of delocalization. Software methods solve this problem by using abstraction to let the designer look at the architecture before it becomes submerged and delocalized in implementation. However, at the software design level there is another important effect: the fragmentation of views of the software design. Software design has been called non-representable by Peter Nauer\textsuperscript{41}. This means that the design itself, in its entirety, cannot ever be represented. This is similar to the effect in General Systems theory in which we say that no system can be completely known. All representations of systems are partial, no matter how much they attempt to represent the details. Similarly, representations of software designs are also always partial. However, there are at least four points of view on any software system:

- **AGENT VIEW** - WHO? - The view of the hierarchy of independent processing units.
- **FUNCTIONAL VIEW** - WHAT? - The view of the transformations performed by the system on either material or information.
- **EVENT VIEW** - WHEN? - The view of the temporal ordering of events in the system.
- **DATA VIEW** - WHERE? - The view of the arrangement of design elements in the memory of the system.

These four views of every software system are probably canonical\textsuperscript{42}. They are the principle views from which each software system should be looked at as it is being designed. The design itself will have an endless number of WHYS or reasons for being exactly as it is and no other way. In fact, it is the infinite number of reasons for it being “just so” that cause the design to be ultimately unrepresentable. However, every design can be represented finitely from each of these views in spite of the fact that all its reasons cannot be enumerated. This fact allows software design to occur, but it is always limited by the fact that its representation is fragmented into at least these different views.

The fragmentation of the views onto the software design is the same effect as delocalization at the higher level of abstraction. The unconscious, or non-manifesting essential character of software does not go away at higher levels of abstraction, but merely transforms. Thus, the split in views is a manifestation of differAnced that hides the immanent essence of manifestation. It is the positive appearance of this form of unconsciousness. In effect there is appearing and disappearing if different kinds of design elements as we move from viewpoint to viewpoint when looking at any one design. The unconscious appears through the interaction of design elements that cannot appear in representations together yet still interact within the software systems as a whole. It, however, indicates the inner nature of technology as system. Technology forms systems which support various endeavors. Those systems are inherently perspectival\textsuperscript{43} because it takes various specialties to keep them operating. The need for those specialties is built into the technology itself because it has so many aspects which are each very detailed, so that no one person can know it all. When we abstract away all this detail at the software design level, the perspectival nature still adheres to the abstract view of the system. This tells us that the perspectival nature is essential, not accidental. In General Systems Theory the perspectives are the various specific Sciences that it attempts to serve. The relation between General Systems and the special sciences is essential. Without the special sciences, no General Science of systems would be possible. On the contrary, the question is whether a General Systems Science is indeed possible or useful. If it is useful, one of those uses is as a bridge between special sciences. As a bridge, General Systems Science must be built to move continuously between the perspectives of the special sciences. In that movement there is showing and hiding as different specific aspects of things move out of view when we switch from one way of looking at things (physics) to another (biology). The question is whether there is some area of exclusion that never comes into view at all, and is completely immanent. This is possible because one could say that the inner coherence of all the views taken together never appears anywhere in any of the views. Thus, the singularity to which the horns of infinite regress asymptotically approach is probably identical with the inner coherence of all the perspectives of the sciences which General Systems Theory attempts to integrate.

Now this brings us to the question of what software methodologies are, and how they work. Software methods are techniques for rationally proceeding to undertake the design of a software system. As such, they are composed of five fundamental categories of elements:

- **Design Elements**: The artifacts represented by the


\textsuperscript{42} Meaning that these are the only four possible views. This is still to be proven.

Artifacts: which are descriptions/specifications of entities involved in software design activities (e.g., the interface specification for modules).

Note that we did not decompose artifacts into sub-types. The reason for this is that our earlier experiments have indicated that the most effective way to aid the comparisons of artifacts is to examine the functions of the artifacts.

Concepts: which are ideas that underlie a Software Development Methodology (SDM). Concepts can be rationales and theories behind a SDM, or strategies and heuristics used in the SDM for specifying/evaluating artifacts (e.g., abstraction, information hiding).

Properties: which are desired characteristics of artifacts. An SDM is always aimed at producing artifacts that have some superior properties, (e.g., artifacts should be easy to understand and modify).

Principles: which are concepts used to help in producing artifacts that have the properties desired by customers and designers. An SDM either justifies new design principles or adopts some existing principles as the basis of the SDM, (e.g., object oriented SDMs use principles of information hiding and abstraction as their bases).

Criteria: which are rules advocated for use by designers in deciding what constitutes an artifact. An SDM usually provides a few criteria that serve as the necessary conditions for deciding what an artifact is, (e.g., Jackson System Design (JSD)\textsuperscript{45} defines the criteria for deciding a JSD entity. Rational Design Methodology\textsuperscript{46} provides a set of rules used to decide how to decompose a design document into a tree of module specifications).

Guidelines: which are concrete strategies, heuristics or existing techniques advocated for use in identification and specification of artifacts. Guidelines are often described by giving examples or artifacts. (For example, Booch Object Oriented Design (BOOD)\textsuperscript{47} indicates that device, system, people, and location might be objects. BOOD also suggests a guideline that using informal English analysis technique can help in identifying objects.)

Measures: which are references with respect to some standard, or samples used for quantitative comparison or evaluation of the quality of artifacts. Some existing SDMs define measures to help in quantifying the degree to which various artifacts demonstrate desired properties, (e.g., Structured Design defines different bindings (e.g., functional and logical) and uses them as a basis for a measure for evaluating the cohesiveness of the program design).

Representations: which are the means used for expressing artifacts. They are aimed at improving the precision with which an artifact is specified and at improving comprehensibility of artifacts. They could be languages, sets of diagrammatic notations, (e.g., data flow diagram), etc.

Guidelines: which are concrete strategies, heuristics or existing techniques advocated for use in identification and specification of artifacts. Guidelines are often described by giving examples or artifacts. (For example, Booch Object Oriented Design (BOOD)\textsuperscript{47} indicates that device, system, people, and location might be objects. BOOD also suggests a guideline that using informal English analysis technique can help in identifying objects.)

Measures: which are references with respect to some standard, or samples used for quantitative comparison or evaluation of the quality of artifacts. Some existing SDMs define measures to help in quantifying the degree to which various artifacts demonstrate desired properties, (e.g., Structured Design defines different bindings (e.g., functional and logical) and uses them as a basis for a measure for evaluating the cohesiveness of the program design).

Representations: which are the means used for expressing artifacts. They are aimed at improving the precision with which an artifact is specified and at improving comprehensibility of artifacts. They could be languages, sets of diagrammatic notations, (e.g., data flow diagram), etc.

Actions: which are physical or mental processing steps used for the developing artifacts. As action may create or modify an artifact. An action may also evaluate an artifact and then decide if it needs further development.

\textsuperscript{44} “A Framework for Classifying Parts of Software Design Methodologies” by Xiping Song & Leon J. Osterweil ISS’92 Second Irvine Software Symposium; Irvine Research Unit in Software University of California, Irvine


This more elaborate analysis of what actual methodologies contain will give the reader a better idea of what is meant by a methodology. However, in this article we will focus on the five more broadly defined categories of elements described above which are implied within the Song/Osterweil breakdown. Here we are defining a methodology as composed of various methods or techniques, which are sequenced by the methodology, and to which the methodology adds some overarching heuristics, or tactics for discovery of the best possible design. Each method has an associated notation for the representation of design elements which may take various forms. We assume that a method is composed of a set of actions by which the basic transformations of information defined by the methodology is undertaken. We will assume that the method is a theory of the best way to approach the transformation of the requirements into a feasible design. We will also assume that the method produces and manipulates design elements, so-called artifacts, according to the theory which, when operationalized, has its own properties and measures. The Song/Osterweil more detailed characterization is an important step in the evolution of design methods because they are the first to analyze the existing design methods in this manner. However, in this exposition we will settle for less precision concerning the categories of elements of any methodology.

The reason for separating methodologies into the categories of design elements, methods, notation, sequence, and heuristics is that it gives a clear picture of the structure of the methodology. The methodology is not a monolithic way of approaching design. In almost every case it is a combination of approaches and techniques. Each of these techniques of approach to the design within the methodology, called here methods, has their own representation which is separate from the theoretical structure of the method itself. A method may have many notations in various forms such as diagrammatic, formal language, mathematical formulation or prose description. The notation chosen is inessential. The essential thing is the set of interlocking concepts that form the foundation of any method. The next most basic aspect is the interface between methods which is usually specified in terms of the sequence of application of the methods to the problem. Methods must work together in order for the designer to get his arms around the problem. Finally, the methods are not algorithms that with certainty produce predetermined results. So, every set of methods that make up a methodology must be accompanied by a set of heuristics, or discovery guidelines, which show the way toward the best design in the face of the necessity for the partial solution of wicked problems. In this case our dissection of methodologies at a grosser level into five elements is meant to focus in on the salient aspects for our consideration here. For other purposes it is correct to go to the lower level of analysis carried out by Song/Osterweil.

We must distinguish between software process, software methods, and software tools. Software process consists of the kinds of work that are done in order to produce software. The activities identified by Song/Osterweil are examples of the kinds of work undertaken in the design process. These are described by industry standards such as the IEEE Standard for Software Development Process. Among these, one kind of work is Software Design which is a crucial transformation of the software system. In doing design work, one uses software methodologies. Methodologies organize the design work, highlighting and abstracting essential features along with the creation of design elements that are represented using the notations associated with the methods. Normally, a methodology is made up of several methods that are sequenced in a particular order. By following this idealized order, one is led to think about different aspects of the system and how they interrelate. As the methods are applied, one uses the heuristics to find optimal designs within the design space of all possible systems designs. Designing a system in this way is greatly accelerated when the methodology is supported by automated design tools. These tools enable the methodology and support the design process. Thus, methodologies enhance and rationalize design work which are further enhanced by automation of the methodology by Computer Aided Software Engineering (CASE) tools.

Notice that the methods that are sequenced by the methodology allow the software designer to look at different aspects of the design one at a time. Thus, methods are broken apart from one another by perspectival views. In fact, we can say that the only views a designer has are those listed above (i.e., Data, Agent, Function, and

---

49. Besides the essential transformations there are other kinds of work such as Planning, Metrics, Monitoring and Control, Configuration Management, Reuse, Technology Infusion, Environmental Maintenance, and many others.
50. IEEE Software Process Lifecycle Standard, P1074
51. These essential transformations are Requirements Analysis, DESIGN, Code, Test and Integration.
Event). Therefore, it is possible to see that methods are a way of rendering these views concrete by providing instantiatated abstractions or design elements with specific well-defined characteristics for the designer to work with as he is considering those views. In fact, we can go further and say that methods are the bridges between points of view, and that the sequencing of methods is the movement from viewpoint to viewpoint as one circles around the unrepresentable core of the design. That unrepresentable core is like the coherence of all the specialized sciences. The inner coherence of the views is never seen. However, one has the ability to circle around and around that non-representable core and make it partially visible from a variety of perspectives. The fact that methods are bridges between a finite number of viewpoints means that there are, in fact, a finite number of “minimal methods” out of which any particular methodology may be built. This is important because for the first time we can survey the whole field of methods and say how each particular methodology concocted by an expert fits into the whole field.

Given this framework of viewpoints and bridges between them, I have constructed a model of the field of all possible design methods. This framework draws from a survey of existing methodologies, and attempts to identify “minimal methods” that allow movement from one conceptional viewpoint on the design to another. The results of this survey were very interesting because it showed that methodologies were amenable to this type of analysis. In fact, it revealed certain properties of well known methods which might not be appreciated otherwise. For instance, the best known of the design methods is the dataflow diagram. It consists of bubbles that represent data transformations and lines that represent dataflows between transformations. Data stores may also be represented. It was quickly seen that the dataflow acted as a bridge between functional and data points of view. And the success of the dataflow diagram can be understood by the fact that it actually represents two “minimal methods” combined into one notational technique. A ‘minimal method,’ as I use the term, means that it is the simplest conceptual threshold that will allow the movement from one viewpoint to another. It is possible to endlessly add elegant notational or conceptual nuances to a method as seen by many methodologies that represent extremely subtle distinctions. However, there is some minimum theory that is needed to bridge the gap between two viewpoints that represents a critical threshold of conceptual complexity. The minimal method attempts to capture this simplest possible method that still allows the transition between viewpoints. In the case of the dataflow, the data lines and transformational bubbles are enough to show the relation between data and function from the function’s point of view. Data stores are unnecessary for this purpose. However, to represent function from the point of view of data, the dataines between bubbles are not necessary, and all one needs is the dataines between function bubbles and datastores. Thus, the complete dataflow diagram that has both dataines, datastores, and transformations allows one to consider data from the point of view of function or vice versa with equal ease. Thus, the dataflow diagram is really two minimal methods merged into one technique that has the extra advantage of conceptual reversibility. Looking at the data flow from this perspective shows how perfectly suited it is to the task of representing programs that are generally thought of as algorithms plus data.

Figure 3: Dataflow diagram minimal methods example.

The empirical survey of the available design methods was

---

52. See Figure 4.
interesting in that in many instances it was obvious which methods were conducive to transition between viewpoints. In other cases it was difficult to fill in the blanks from existing methods. In those cases, minimal methods had to be discovered which filled the blanks in the matrix. This occurred particularly for the transitions between the data and event views, which was quite unexpected. Another point of surprise was that what constituted minimality was different for different minimal methods. Some were very complex and others very simple so that the resultant field description appeared lopsided. Finally, while several of the minimal methods allowed two-way bridges between viewpoints, in other cases two completely separate minimal methods that were very different constituted two independent one-way bridges. This occurred between the event-process, event-data, and process-agent transitions. The following is the list of minimal methods generated by this survey of the field of all possible software design methods:

1. Function alone: Functional Decomposition. Functional decomposition is a hierarchical tree of functions and sub-functions which is standard in structured analysis, such as that of Tom De Marco54.

2. Agent alone: Tasking Tree. The Ada language55 introduces into programming the concept of tasking trees, which is a hierarchical decomposition of tasks, each of which are an independent thread of execution.

3. Event alone: Interval Logic. Interval Logic was constructed by J.F. Allen56 as a means of relating events in terms of their temporal dependencies (before, during, after, etc.).

4. Data alone: Entity, Relationship, Attribute Diagram. Chen57 introduced Entity Relation diagrams as a way to represent the relations between data items.

5. Function --> Event: State Transition diagram. A standard part of the Hatley-Pirbhai58 and Ward-Mellor59 real-time structured analysis method is to add state transition diagrams to dataflow diagrams. State diagrams are connected to the dataflow through decision tables and process activation tables. The State Transition Diagram (Finite State Machine) is the traditional way to represent computer algorithms abstractly.

6. Event --> Function: Petri Net60. Petri invented these nets to show how control flows are starting to be used more and more to analyze complex control structures. Colored Petri Nets61 is the preferred representation. Petri nets are composed of places and transitions. Markers move though the places by the firing of the transitions. Petri nets are the dual of State Machines.

7. Function --> Data: Input/Output Dataflow. This is the dataflow diagram with only transforms and dataflow lines.

8. Data --> Function: Objects with operations62. This is essentially the data flow diagram with only datastores and transforms now called operations or even “methods.” The operations are grouped around the data they change and the entire set is packaged together.

9. Function --> Agent: Mapping of function to task. Process Allocation is described by Mellor & Ward63. An explicit mapping is maintained as design progresses between the functional transforms and the discrete processors (tasks) that will perform those functions.

10. Agent --> Function: Virtual Machine instruction set. This concept was introduced by Neilson & Schumate64 and says that the implementation component is made up of instructions that solve a problem at a particular level of abstraction. The set of instructions working together solve the problem and constitute a virtual machine. Instructions are, in turn, lower level virtual machines, operations on data objects, or pure data transformations.

11. Agent --> Event: World Line. Explicitly introduced by Agha65 in his exposition of ACTORS, the world line concept comes from relativity theory which seems to apply to multiple agents acting concurrently within a system. A world line follows all the events that occur to a single agent.


13. Data --> Event: Data Transitions. This is a very simple method of watching a data value change in a set of variables as the program executes. It has dual configurations in which you follow the data from variable to variable. Or you follow the changes in the

---

57. ???????
58. op. cit.
59. op. cit.
61. Ref for Colored Petri Nets ?????
63. op. cit.
64. op. cit.
data values in one variable and compare that to the changes in other values. This method is not represented in the literature, but is practiced by every programmer who has ever debugged a program.

14. Event --> Data: Design Element Flow. This is a very simple method which says that system states and design element states are coordinated. It has dual configurations in which you compare the transitions of design elements as the system state changes. Or you compare the states of the design elements when system transitions occur. This method is also not represented in the literature, but is implied by the fact that design elements are expected to cooperate in a software system.

15. Agent --> Data: Communication Transport Mechanisms. A key feature of the ADARTS\(^6\) method developed by SPC/Gomaa is the display of communications mechanisms between multiple tasks. These are an implementation of the dataflow lines which specify the nature of the data channel as Flag, Semaphore, Mailbox, Rendezvous, etc.

16. Data --> Agent: Data Monitors. Introduced by Hoare\(^6\)\(^7\) data monitors protect data from multiple conflicting access and are used in operating system design to allow different tasks to share the same data.

This set of minimal methods fulfills all the requirements of our description of the field of design methods. It allows each transition to be made by letting one viewpoint take the other viewpoint as an object. Thus, in the dataflow example, process reifies and objectifies data as dataflow lines. On the other hand, data objectifies and reifies process as operations clustered around data that has the sole purpose of transforming the data. The objectification of one viewpoint by the other is a very interesting process. It has to do with stillness and motion in this case. The objectifying viewpoint sees itself as still, while the elements related to the other viewpoint are seen to be moving. Different criteria for objectification exist in each set of minimal methods. But the end result is that the active viewpoint uses the other viewpoints as its material which it organizes in order to get a view of the system. Viewpoints organize, and in these minimal methods we see that organization at work. Not all the design features of the system are visible at once, but it is possible to circle around a particular viewpoint moving through the other three possible viewpoints one at a time to get a coherent picture of the system from that particular viewpoint. As one rotates through the other viewpoints treated as material for the active viewpoint a coherent picture of the system from that active viewpoint becomes accessible. Then, at some point specified by the overall methodology, one changes to one of the other viewpoints and makes them active to repeat the process. The analogy is Rubric’s Tetrahedron\(^6\)\(^8\). As one moves between active view points, different design features rotate out of view, and others come into view. The whole thing is never visible at once, so showing and hiding is in effect. But all the views are connected and form a web around the designed software which can be traversed. With enough passes the one begins to intuit the complete picture of the software design which is never actually available. The coherence of that design, which lies hidden beyond the fragmented perspectives from which the infinite regress of WHY’s issues, is the unmanifest non-representable part of the design, the essence of manifestation of the design.

Figure 4: View Points Combine to Give Minimal Design Methods

Now this description of the field of all methods was empirically derived. The main reason for doing it was to study the interaction between elements visible in different views. However, once it had been formulated, I began to wonder about the strange lopsided nature of the description of the field. I had expected all the minimal

---

\(^{66}\) op. cit.


\(^{68}\) This is a toy related to the Rubric’s cube which is turned to match up the colors on the sides.
methods to be equally complex, for all minimal methods to support two-way bridges like the dataflow diagram. One good thing was that I could see how many of the minimal methods were duals of each other in the sense used in Mathematical Category Theory. For instance, the state machine and the Petri Net appeared to be duals. It did appear that methods connecting the same two viewpoints were of about the same complexity. It was only as one traversed the field, that the variations in threshold complexity appeared. Yet, there was enough lopsidedness and irregularity in my field description to make me doubt the result of this analysis on purely aesthetic grounds. It is always open for someone else to try to construct a different set of minimal methods that equally provide transition between viewpoints. But, if you accept that there are only four viewpoints on design, and that methods are transitions between viewpoints, then it is just a matter of filling in the blanks, and this is the set of minimal methods that appeared most likely to me. The nagging question, though, is why the set is so lopsided and uneven. Is this due to my poor selection of candidates for minimal methods, or due to some inherent properties of the field itself?


It occurred to me that there was no way to answer this question of the adequacy of the set of minimal methods without a context in which to see these minimal methods functioning. Therefore, I decided to attempt to insert them into Klir's excellent model of a formal-structural system. This would provide a context that should make it clear whether the set fits together or not. GSPS was constructed, with description in mind to facilitate the two great descriptive operations of General Systems Theory: reconstruction and identification. The amount of space dedicated in Klir's book to design of systems in the sense common in software engineering was minimal at best. In fact, it seems to me that the vocabulary is skewed too much toward description and would have to be changed somewhat to become more familiar to the software design community. Yet, GSPS provides something crucial that is missing from the software engineering community which is a universal vocabulary for describing systems. Thus, if it were possible for software engineers to learn that vocabulary, it would be a boon to the profession. What is really needed is another version of that system which is presented in a way that would be easy for software engineers to relate to, and perhaps some of the terminology changed to be more what they are used to, especially where the different words mean the same thing. For instance, Klir speaks of the generative system, whereas for software engineers this is the software program. It is interesting that what appears as higher epistemological levels in General Systems Theory are more concrete levels from the point of view of software engineering. This is another indication of the symbiotic and complementary relation between the dual meta-disciplines.

Terminological issues aside, the architecture of the GSPS is an excellent view of what a system is, and the only question becomes, how does one use that General Systems view to guide the design of software? Searching though the ASPS book for a place to make the necessary connection between my set of minimal software design methods and the general structure of systems, I was gratified to find, as if the author knew exactly what I needed, a ready-made place to plug in my set of viewpoints and minimal methods. That place was found in the section on “backgrounds” and “support variables.”

When one lifts the source system out of the object system, it is by deciding which attributes of the object system will be included in the system to be studied. The object system is a bundle of attributes from which some are selected. How the observer recognizes which attributes form a coherent or “systematic” set is not treated. But, the narrower bundle of extracted attributes is the source system, and for each of those selected attributes, a variable needs to be defined and an observation channel constructed to create the data system which renders the system available. Here, Klir points out a fact usually overlooked, which is that for meaningful measurements to be made, there needs to be other special variables created that provide the context for measurement, the coordinates within which the measurements will be related to each other. These coordinate variables are also attributes taken from the global context of the source system. They must be attributes that are globally uniform. This means that background attributes are really a connection between the system and its environment. This means that the gestalt of the system on the background of its environment is formalized by identifying the background attributes. It is interesting that by selecting a particular background for a system, one is, in effect, attaching it to a specific discipline as well so that in using the four viewpoints (data, event, function, and agent), one is connecting the General

69. Ref reconstruction and identification. 70. ASPS Chapter 2 Section 2 “Variables and Supports” pp 38-44
Systems Theory to Software Engineering. Other selected backgrounds would entail connection to other disciplines. Thus, the disciplines color the background against which the Object system is seen. By switching backgrounds, one immediately switches disciplines. The general point is though, that the particular background attributes furnished by Software Engineering are generic in the same sense that the Object systems of General Systems Theory are generic. Therefore, this connection of backgrounds to foreground attributes is the specific site for the marriage of the two meta-disciplines. Further, we see that the relation of the foreground treated by General Systems Theory, and the background treated by Software Engineering, is another instance of the object/supporting technology dialectical relationship which is in some sense formalized by this proposed common law marriage of strange bedfellows. The background attributes, once selected, become special variables called “supports.” Klir mentions that the normal kinds of support variables are measures of space, time and/or population.

This concept of a “support variable” provided exactly what I needed. Event and Data were already directly translatable, for they were the way time and space were represented in the minimal methods. Data always means where, in the space of memory, a particular pattern of information lies. Time is always conceived in terms of cycle times of the central processing unit (CPU). And when I looked closer at the concept of “population,” I saw it could be construed to be made up of my two other viewpoints superimposed. Population usually refers to animals which are autonomous beings that move about independently. Here again, we see hints of Rescher’s reduction of systems characteristics to the empirically discovered characteristics of organisms. Demographics follows these movements and categorizes the different types of organisms which are spatially distributed. However, essential to the concept of population, as it is normally used to speak of organisms, autonomy of action is definitely implied. And all organisms do different things, and have very different behaviors as well as shapes, so the things they do can be construed as their functionality. In fact, there is a direct relation between functionality and intentionality. Functionality is an ill-defined and over-used concept in software engineering. Intentionality really refers to relevance and the distinguishing of kinds, as one’s attention moves from one type of thing to another. Organisms all have some form of consciousness, and the focus of that consciousness is their intentionality. Functionality is the focus of the autonomous agents within the population which directly relates to changes in their behaviors. Thus, from this narrow perspective, population can be seen as made up of autonomous agents exhibiting different kinds of behaviors (functioning) expressing their individual intentions. So I could see my way clear to saying that for each viewpoint on Software design there was at least one support variable. The interaction between these support variables is then described by the set of minimal methods. In this way there was a precise interface between software design methods and Klir’s formal-structural system.

However, once I had made this connection, I was fascinated by what immediately followed from it. Klir goes right on to discuss what he calls Methodological Distinctions. This boils down to the idea that any particular support variable can exhibit different ordering characteristics. Basically the possibilities are as follows:

<table>
<thead>
<tr>
<th>ORDERING</th>
<th>DISTANCE</th>
</tr>
</thead>
<tbody>
<tr>
<td>o No Ordering</td>
<td>o No Distance</td>
</tr>
<tr>
<td>o Partial Ordering</td>
<td>o Distance</td>
</tr>
<tr>
<td>o Linear Order</td>
<td>o Distance (have a scale)</td>
</tr>
<tr>
<td>o Distance (have a scale)</td>
<td></td>
</tr>
</tbody>
</table>

These types of orders determine the effectiveness of the support variables in locating a measured event/entity (eventity) within the overall system. They form a lattice of possible orderings which has the following elements:

a) No Order, No Distance: The coordinates are a myriad of distinctions which are not related to each other.

b) Partial Order, No Distance: Set theory produces partial ordering without distance.

c) Linear Order, No Distance: A continuous line exists, but how far apart the points are cannot be determined.

d) Partial Order with Distance: Sets in which you know how many boundaries you have crossed to get somewhere give this type of measure.

e) Linear Order with Distance: This is the type of measure we are used to in mathematics that uses the real number line.

In this lattice position c and d are at the same level, lying at the same level, lying
as two separate routes between \( b \) and \( e \). The methodological distinctions tell you how good your measurement system will be. We naturally assume that full ordering will be available to us in each support that we select. So we assume that we will not have any problems determining the exact position of each eventity in relation to our coordinates. However, we do not always have full choice as to what support variable from which we can choose, and this is why Klir explains the important distinctions between them. If it hadn't been that support variables were such a good fit for my methods into the GSPS framework, I would not have thought twice about these distinctions between types of coordinate systems. But as I started looking at my proposed support variables and the methodological distinctions, I noticed that two types of support variables existed in my schema:

- Support variables with Partial Order and no Distance
  - \( \text{AGENT} \)
  - \( \text{FUNCTION} \)
- Fully ordered support variables
  - \( \text{EVENT} \)
  - \( \text{DATA} \)

*Figure 5: Lattice of Methodological Distinctions*

This caused me to look deeper into the relation between these two types of support variables, and what I discovered there, I think, is very significant for the development of software design methodologies. Basically what I saw was that there was an irreconcilable gulf between my two types of support variables, and that gulf intrinsically determined the nature of software designs. In building a software system, we start off with requirements which are basically a set of pure diacritical distinctions without order or distance. In systems design we decompose functionality and pick a set of agents (boxes with multiple processors, processors, tasks within a processor) that become the architectural elements of the system. A miracle of design must occur. In hard real-time systems it is assumed that a certain event has to occur in the full ordering of spacetime. The point is, that there is no way to connect the architecture of agents and the functional decomposition to the necessities of spacetime occurrence. The design must perform a miracle, and there is no real support for that miracle. The designer must find a way to leap the gap from global architecture and functionality to spacetime occurrence, and this is the essence of the real-time design problem. Implied here is that this gap is essential and can never be bridged except through intuition and basically inadequate bridging techniques.

This said, it then quickly became clear that the intermediate positions in the lattice of methodological distinctions were very important. They were, in effect, the generic prototypes of each minimal method pair. And this is very significant because it means that minimal methods have a root in some sort of mathematical necessity. In each case they attempt to describe an intermediary position between pure partial order without distance and full ordering. Now the lopsidedness of the set of minimal methods started to make sense. The agent-process (functionality) bridge which was composed of function-to-agent mapping and the virtual machine (augmented structure chart) was very complex in order to compensate for the lack of structure to the ordering itself; whereas the event-data bridge could afford to be simple because the ordering was robust. This means that the lopsidedness in the set of minimal methods was a direct result of the expressive poverty of partial ordering for which the agent-function methods attempted to make up in their higher threshold of inherent complexity.

Another important point was that the nature of the intermediate lattice positions could be understood as duals of each other. Linearity without order is the dual of partial
ordering with distance. When you see that partial ordering basically means “sets,” or Venn Diagrams, it is possible to see each of these intermediate duals as a use of two sets against each other in the only two possible ways. In partial order with distance, the agent set is used against the process set, and you get measurement by counting boundary crossings. In the case of linearity without distance, one set becomes the coordinate line, and the other becomes the thing arrayed on that coordinate line. Thus, the second set can be arrayed against the first in only two possible ways. It can become a coordinate line, in which case it has no tick marks to provide a scale. Or it can be used as an alternative categorization of the same elements, in which case it provides the scale without an independent coordinate reference. Either way, to move to full ordering, another element besides the original two sets must be provided. That third element is either an external coordinate or an external scale. This, in software, is provided by the real world itself. The spacetime continuum has the extra dimension necessary for full ordering to appear.

The minimal methods that connect event-data and process-agent strive to fill the gap between the two types of viewpoint. However for hard real-time systems the minimal-methods connecting event-data cannot be relaxed, and so the gap appears as an unreconcilable abyss across which the design must leap. This abyss, in the heart of the viewpoints is the negative representation as an essential absence or lack of the non-manifesting essence of manifestation from which all the difference conjured up by software issues forth. It is an important absence haunting every software design, making it an essentially impossible task which we do anyway using our intuition (which has an crucial unconscious factor) as a guide.

The presence of this lack in the background of every Source system is important for General Systems Theory also. Because General Systems Theory generally detaches itself from all possible backgrounds associated with specific disciplines, that lack becomes observed. General Systems Theory seems to be able to detach itself from that blindspot in the background. But then the blindspot reappears at the center of General Systems Theory itself as the “always already lost” origin of the two infinite regresses of meta-structures and models. The blindspot appears both in the background and in the foreground, and cannot be escaped without lowering our expectations significantly with respect to the robustness of our descriptions of empirical Source systems. It is not much of a leap to realize that the fore and background blindspots are merely two reflections of a single hidden source from which the whole temporal gestalt is emanating. Not only are we caught in webs of showing and hiding, we are also tied to something which never appears, but constantly distorts and conditions everything that does appear. This is hard for all naive realists to swallow, but is hidden in the very structures that objectivists build because it is an aspect of manifestation or Being itself, and whatever is defined by the objectivists undergoes manifestation or presenceing which even they cannot escape.

Each of the different meta-levels of Being can be derived naturally by first making the distinction of “ontological difference” between beings and Being. Once we recognize Being as different from the beings that participate in Being then we can begin to explore the nature of Being itself. For most of our philosophical tradition that Being was considered as unified and unitary. But since Parmenides it was considered as also frozen and static. It was Heidegger who exploited the insights of Husserl to show that there must be some kind of Being mixed with Time. This dynamic Being which underlies all processes and support temporal gestalts that last longer than the Now moment was shown to have a radically different nature from the Being assumed by Aristotle and Kant and the other major philosophers in the Western tradition. Eventually it was shown by Heidegger than yet a further kind of Being could be defined called Being (crossed out) which Derrida went on to define as Differance and exploit in his works such as Of Grammatology. Michael Henry also explored this level showing that it had some fundamental non-manifesting aspect which like the unconscious was always hidden from view and only seen as distortions within the things that were manifest. Henry called this always hidden aspect of Manifestation the Essence of Manifestation. As each meta-level of Being is broached it becomes successively harder to think about its nature and consequences for our comprehension of manifestation. The final stage in the unfolding of Manifestation was defined by Merleau-Ponty as Wild Being and explored by Deleuze and Guattari. Wild Being is the substrate of Being within perception and is exemplified by the phenomena of touch-touching itself. In that phenomena there is an opacity which cannot be made transparent to itself which reveals itself as a chiasm or reversibility between perceiver and perceived. Deleuze and Guattari speak of this same phenomena in terms of the relation between desiring machines and the body without organs or the embedding of partial objects within the rhizome. It is difficult to think about this level of the infra-structure of ideation except in terms of the negation
of dualities by the realization that duals imply each other and mutually define each other so that what really exists is the relation between the duals not the duals themselves. Thus at this level everything is composed of partialities and propensities which exist in a Chaotic field which combines order and disorder in which order is embedded in disorder and vice versa. Both GST and Software Engineering attempt to delimit themselves in relation to the meta-level of Wild Being and exclude its chaotic nature from their consideration. For Software Engineering the catch all category for all such phenomena associated with Wild Being is Artificial Intelligence. For General Systems Theory it is Complexity Theory. The limit of GST is systems too complex to be analyzed into component parts and synthesized as formal-structural systems. Such systems must be considered at a macro level without looking in detail at their parts or must be considered at a micro level and generated by uncontrolled interactions to produce macro phenomena. Complexity Theory assumes that certain emergent phenomena emerge from very complex systems that cannot be reduced by analysis to constituent parts. These emergences are the way that Wild Being manifests in these systems. In this paper very complex systems and their emergent phenomena are not considered further but the reader should be aware of the existence of this further horizon of exploration that opens up out of GST and Software Engineering as we attempt to think what the next meta-level of Being from that at which they are situated must be like.

5. Systems Architecture from the point of view of Software

Now that we have seen how the software methods plug into the support variables of the GSPS architecture, it is necessary to interpret the GSPS semi-lattice of meta-levels from the point of view of software. This interpretation will include the mapping of the software design minimal methods into the support variables so that, in effect, a complete software design framework is produced out of the generalized GSPS structure. Here we are turning GSPS into a specific design system for software. However, since software is also at a meta-level above all the specific sciences, we could be seen as extending the realm of GSPS to produce a broader set of modeling systems, by which discrete simulations might be built. As such GSPS is extended by attaching a certain set of generic modeling methods found useful in designing software, but which could be applied to the modeling of any discrete system. This combination of the two meta-disciplines, which takes into account both foreground, background, and their relations, might be called Software Systems Meta-Methodology.

5.1. OBJECT SYSTEM

The object system is what is differentiated from the environment and designated as a “system.” As has been said, if this is a system instead of an object then it exists in a web of showing and hiding relations as a complete gestalt. As such it is not just a gestalt in space but also in time. If the system can be represented in space alone then a formal system is adequate to describe it, but if the system is dynamic then we need a structural-formal description of it.

The formal-structural description has two components related to its patterning and its formation. Patterning considers order on the backdrop of disorder while Formation considers disorder on the backdrop of order. These are complementary views of the same system which combine to give the formal-structural description of the system.

The “object” system is delimited as a form on the background of everything else. That form may move and change over time and so we might have structural descriptions of it and process models of its changes. Within the form the contents may move and change over time in which case we construct micro-formalisms to describe transformations in content which delineate the structure of the content and model its changes over time. When we combine these two descriptions of the form and its content then we have a complete formal-structural description. But this is not a description of a system unless we include the showing and hiding webs that the objects within this system participate in through the dynamics of its form and pattern configurations.

At the level of the “object” system we have merely focused in on the target system as a gestalt and discriminated its orthogonal pattern and form components that blend together in webs of showing and hiding to create a formal-structural system.

5.2. SOURCE SYSTEM

Having isolated the object system we are ready to look for its attributes and the backgrounds that give significance to
those attributes. As we have seen for the software system those backgrounds become visible as we observe the system from the four canonical viewpoints: Agent, Function, Data, and Event. These viewpoints each reveal a background specific to it upon which the system might be seen. In the light of that viewpoint on a specific background the system will have certain significant attributes which will appear relevant to our modeling of the system. At the level of the source system we isolate those attributes against the system wide backgrounds. So the source system becomes a bundle of selected attributes in relation to a few well selected global system backgrounds.

At this level the system description begins to ramify. That ramification is precisely the same kind of ramification that occurs in Russell’s theory of logical types. In other words we can see the source system as attempting to resolve all the paradoxes that it finds in the object system by applying something similar to Russell’s ramified Type Theory. In the case of the Klir epistemological lattice the ramification occurs in a spacelike direction and in a timelike direction. In the spacelike direction variables change on constant backgrounds while in the timelike direction backgrounds change in relation to constant variable sets. At the point of fusion both backgrounds and variable sets vary together. The spacelike direction relates to the structuring of the parts and wholes within the system into echelons of holons. In this case we are calling the “structure” the combination of formation and patterning within the source system. The timelike direction considers the models by which different structural (form/pattern) configurations are changed over time. A model is the same as a structural configuration in space only applied to time. Models are composed of processes and sub-processes. During a certain sub-process a specific part-whole holonic structure is in force and a set of transformations are applied to it. When this sub-process changes to a different sub-process then a new set of transformations are applied to the holonic structure. Transformations may be of a number of different kinds. Transhaping changes the form from one formation to another. Transcription which changes the patterning from one ordering scheme to another. Transfiguration the changes of both form and pattern simultaneously. We would call transubstantiation the change of the medium which supports the form and pattern configuration. The change of form, pattern and substance all simultaneously we might dub transmutation. All these kinds of transformations are possible that bring us from one form-pattern configuration to another as the system changes as a dynamical process according to some model of changes. We can see this in terms of the system “working” where it brings to bear different kinds of work at different times in a specific sequences in order to effect changes. We notice that “work” can effect changes in pattern, form, or substance according to goals. When we look at a system from a process perspective we project these goals in order to see the coherent kinds of work being applied to effect transformations.

Since we understand that space and time dimensions are intimately related in a fusion which we normally think of as “spacetime,” it is clear that the structural and process modeled dimensions of ramification must be interlinked. We see exactly this in Klir’s epistemological levels where there are certain nodes that mix the spacelike and timelike aspects of the system description emphasizing one or the other. This means that the ramification of the structures and models in the epistemological lattice are mutually entailing which means that the spacelike characteristics articulate the time like characteristics and vice versa. Now we also notice that of our four viewpoints there are two that represent spacelike (Data) and timelike (Event) characteristics of our design. This indicates that the embedding in spacetime is complete in as much as the total dynamic system is encompassed by structures and process models but also there are design elements that specifically embody the architecture of the system in spacetime. We also need to note that spacetime has a dual called timespace74 which looks at the system and the propagation of events within it from a causal perspective. Thus the MATRIX of spacetime/timespace has two meta-viewspoints that encompass the specific background viewpoints on software design. The epistemological lattice exists as a half way house between these two levels of viewpoints75. In other words the epistemological lattice exists only as a spacetime structure which encompasses the four viewpoints on software design two of which embody the spacetime aspect of the system itself. We might ask why the spacetime component of the system differentiates itself on these three different levels. We note that embodiment within spacetime through the DATA and EVENT viewpoints is encompassed along with the AGENT and FUNCTION viewpoints by the structural and process model ways of looking at the system as a whole which in turn gives us a view of the whole within the ultimate background of spacetime/timespace. If we want to look at the causal relations between elements through the timespace view then in effect we switch from

---

74. Normally called Minkowski “spacetime.”
75. Spacetime / Timespace viewpoints or Design viewpoints including Agent, Function, Data (space) and Event (time).
emphasizing the independence of the structures and process models to the emphasis of the fused structural-processes or process-structures that exist in the epistemological lattice. At the level of viewpoints this fusion comes about though the agent and function views in relation to timespace embodiments by design elements. The timespace view is almost like the fusion of the spacelike and timelike orthogonal components that can be seen as separate in spacetime. The epistemological lattice allows us to see this fusion when viewing the system from the outside in terms of the fusion of process models and structures of pattern and form.

5.3. DATA SYSTEM

Klir calls the data system the observed data streams flowing though variables (that characterize attributes) and supports (that characterize backgrounds). Here he has already reduced the system in his mind into a data image. But we have seen that a software system has four viewpoints of which data is only one. All of those can be reduced to data images but this reduction does violence to our understanding of the workings of the system. Instead we prefer to understand that these data images preserve the different kinds of information related to the different viewpoints that were projected on the object system to produce the source system. At the data system level we are reducing the system to streams of data that are recorded and analyzed in order to discover the workings of the system. But from a software point of view we are seeing the software system as flowing streams of data related to the different views on the dynamic system we are designing. There is information about the Agents within the system, about the functions they are performing, about the events that are occurring and the data that the system is consuming and producing. We note that all this data about the different viewpoints are mixed together and they ramify and fuse in exactly the same way as the source system did in spacelike and timelike dimensions. The difference is now that the whole system is seen as a set of variables taking on values at specific locations in time and space. If we are studying such a system then we wish to understand the configuration of structures and process models that are implicit in the multitude of changing patterns of information in the forms of the variables over time. If we are designing such a system then it is our hope to get these changing patterns of data in the forms of the variables to behave correctly which is not an easy task.

What we notice at the data level is that the whole object system which was reduced to attributes has now again been reduced to a string of memory locations and their values over time. The next question becomes for the Systems Scientist if we can create a generative system that simulates the data values changing over time. For the designer the question is whether it is possible to write a program that creates precise configuration of values not as a simulation but as the origin of that pattern.

5.4. GENERATIVE SYSTEM

The program is the generative system that produces the required data streams when it executes. The program can be either compiled or interpreted. The difference is whether it executes in small snatches or as a monolithic block. An interpreted program is more malleable and flexible but is normally slower because it is compiling as it is going along. Either way the program when it is executing becomes pure behavior which acts blindly on the instructions that occur in the source code. It produces configurations of changing values in spacetime and if those correspond to those that are needed then we say that the program is running without defects. To write a program that imitates a value sequence exactly is taken as demonstration that one understands that data sequence. This is the notion of understanding within the objectivist viewpoint on existence. This is a very limited or one dimensional view of understanding. Exact imitation is not necessarily a display of understanding. Instead going beyond the information given is understanding. In other words real understanding is not merely the production of a frozen image of the thing being understood that can be made perfectly available as something Purely Present, but must exemplify a process of moving beyond what is given through the application of a hermeneutic circle which interprets what is given in different contexts to give an overall interpretation which is never complete. So we should posit a stage beyond the generative system. That next stage is the Knowledge stage. It moves beyond the epistemological lattice for general systems into Knowledge systems which exhibit artificial intelligence and life. Klir indicates this level when he deals with goal seeking systems which use feed forward and feedback separately or together to produce meta levels of the generative system which display goal seeking or adaptive behavior. He ends that exposition with an explanation of the autopoietic system whose goal is to maintain its own organization and especially its boundary with the environment.

But here at the generative level we must attempt to put our design for the dynamical formal-structural system into a program that produces the right data pattern. That
program when it is not executing is itself a data pattern as well. The data pattern of the designed program exemplifies the interembedded viewpoints of Agent, Function, Data, and Event. When we reduce the design to a program then delocalization begins to play a role as design elements actually interfere with each other within the source code of the program. It is the warps of delocalization which are increased exponentially as performance characteristics become more and more important. These warpages account for the difficulty of understanding code and many of the errors that creep in despite the rigors of structured programming and other good software engineering practices.

What for General systems theory is the most general level of understanding of the system is for software engineering the most concrete where the system is represented as an executable program. Software has its own special ontological basis which exemplifies what Derrida calls DifferAnce and makes it different from most things in the world that are either like nouns (frozen objects) or verbs (processes). The nouns are locatable in the spacelike dimension whereas the processes are locatable in the timelike dimension. Software with its branches and while loops that alter sequence instead has traces which place it between a noun and a verb. We can signify it with words like shape which can function as both a noun or a verb. The trace is the sum total of the side effects or mutual interferences of the running program. We look at the listing and compare it to the trace that occurs during execution. So . . . Shape shapes. The source listing executes and produces a trace. We trace our way thorough the source listing looking for what it is doing when in order to try to understand the complexities of its mechanism in operation. We might say . . . Trace Traces. There is a cognitive dissonance between the cumulative side effects as observed in execution and the flat text of the source code of the program. Due to delocalization and the mixture of design elements, that object oriented design can only partially stop, it is very difficult to understand the homeomorphisms between the two kinds of trace (noun and verb). We reach the point where we are lost between them, where we enter the realm of tracelessness in which it is impossible to decide what in the program is leaving a certain side effect. As we approach this undecidability we get closer and closer to the actual essence of software.

Attempts are being made to come to terms with the nature of software and its strange essence. The seminal article was written by F.P. Brooks called “No Silver Bullet.” A recent follow up article by R. G. Mays called “Forging a Silver Bullet from the Essence of Software” continues the introspection of the discipline into the nature of the peculiar object they are engaged in creating. The author has presented philosophical arguments that attempt to define the essence of software in the article “Software Ontology” which is the first part of the series on Software Engineering Foundations. In that article the author identifies the object of software with the third meta-level of Being called Hyper Being by Merleau-Ponty which exemplifies what Derrida calls DifferAnce with its chiasmic attributes of differing and deferring. This definition draws on work in modern ontology rather than the old Aristotelian roots of ontology that is used by Brooks and Mays. But if we look at the work of Mays we see that he points out three basic sources of the essence of software:

- **Conceptual Content**

  “A software entity is characterized by concepts that come from both the problem domain and the surrounding software entities with which it interfaces.” [page 21-22]

I would go further to say that software is in every case an embodied theory. Besides the concepts that Mays mentions there is the theoretical concepts that control the structure of the software itself that is inherent to the software which are represented by the concepts in Design Methodologies.

- **Representation.**

  “The concepts of a software entity are expressed as representations of both the data it uses and the function it performs.” [page 22]

I would go further and say following Naur that software design is inherently non-representable in toto and that all representations are partial. Those representations do not just represent Data and Function but also the viewpoints of Agents and Events. These viewpoints are “canonical” in that they are sufficient to represent the essentials of any real-time software design. Thus we would include the viewpoints and the minimal methods that span between them within a finer definition of the software essence.

- **Multiple Subdomains**

  “A software entity performs functions that consist of transformations on its data, based on conditions present at the time of execution. The presence of conditions splits the input domain into multiple subdomains of the function.” [page 22]
This is the combinatorial aspect of software that makes it impossible to formally prove any but the simplest systems. This also is the connection between the software and events, data, actions and transformations in the real world to which its events, data, actions, and transformations must be harmonized to operate properly.

Besides the splitting of subdomains over inputs of data to transformations there is also splitting of subdomains over event signals. Thus the combinatorial explosion of possible states is intimately related to the embedding of the real-time software system in spacetime.

- **Delocalized Incarnation**

To the three major aspects of the software essence mentioned by Mays I would add the effect of delocalization which causes the design elements at the conceptual level embodied by partial representations to be smeared out within the actual source code that incarnates the software design. Mays theory of software essence seems to ignore the incarnation of it within the text of the source code which is then compiled and executed. The major effects of cognitive dissonance occur because of our inability to see the relation between what is in the delocalized source code constructs and what actually occurs when the software is run. Also this is the place where the differences between Higher Order languages occur and their relation to assembly language and binary code. All software can be represented by the three constructs of structured programming (sequence, if, and while statements) but all languages have a bewildering array of constructs which gives rich choice as to how to implement any given feature of the design.

We can augment Mays’ summary by adding to his words saying, “Thus the software entity is in essence a construct of interlocking concepts characterized by a conceptual content” revolving around a non-representable kernel of the design “derived from its problem domain and the milieu of other software entities with which it interfaces” as well as software design specific constructs “by” partial “representations of its concepts both in the data it uses and in the functions it performs” as well as in terms of agents and events”, and by the multiple subdomains of its input domain that characterize the different transformations that will occur, depending on the conditions that are present during execution” which is carried out by delocalized design elements embedded within a text that is compiled and run which creates cognitive dissonance between the text and the actions of the program. [page 24-25]

From these he derives the following inherent properties:

1. The conceptual construct of the software is held in the developers thinking. Therefore the Software is Malleable and Changeable. (conceptual content)
2. The conditions in the software combine multiplicatively. Therefore the software is complex. (multiple subdomains)
3. The software representations are a “crystallization” of the conceptual construct. (representation)
4. The developer must anticipate the behavior of the software beforehand under all cases and conditions. (multiple subdomains)
5. Software is more broadly conceptual than mathematical or graphical. Therefore software is unvisualizable. (conceptual content)
6. Software development is an intensive activity of thinking. (multiple subdomains)
7. A higher-order verification coccus in the developers thinking. Formal verification is done relative to the software representations. (conceptual content)
8. Programs are objects in the world and operate in the world. Therefore software needs constantly to conform and change. (representation)
9. Software will always be an asset that must be maintained and enhanced. Therefore software developed is primarily though incremental enhancements. (representation)
10. The first task of development is to re-enliven the conceptual construct in the developers thinking. (multiple subdomains)

This series of deduced properties is very uneven. We could instead look at the minimal system of properties that occur in the essence of software constructing a minimal system.

![Diagram](image-url)
Mays draws the conclusion that because software is primarily theory that it is very changeable and malleable. Also because software has a world interface that is complex and that combinatorially explodes then it will be very complex. Representations need to be constantly verified in the face of this change and complexity. And as we have added the software incarnation leads to cognitive dissonance between source code and execution side effects. When we combine these essential factors we get even more difficult problems rearing their ugly heads.

A. Theory <-> Representation

In Mays’s scheme Theory does not interact with representation at all which is very odd. A non-represented theory is completely ethereal. Such designs only exist in the heads of the designers, where a good many designs of actual programs continue to live to this day. Without representation a theory cannot be shared or recorded or adequately studied even by the original designer himself. But these attributes are separated because the whole design theory cannot be captured by the representations. Representations are always partial and as we move from one representation to another different parts of the design appear and disappear. This is because the theories of design must be organized around the canonical viewpoints that organize the universe of discourse within which we build up our theories of designs.

B. Theory <-> World Interface (Mays #5, 6, & 7)

Theories are abstract precisely because we wish to cover the multiple subdomains of inputs and represent the system that can cover them nicely. But due to combinatorial explosion we cannot actually test all the interactions of constructs that we can finitely write down to occur in the operation of the software. Software cannot be adequately portrayed either graphically or mathematically because it has a behavioral aspect that cannot be portrayed adequately by these means. When we look at representations we are not visualizing the software itself but partial representations of aspects of it that can never be brought together in any satisfactory way to tell the whole story at once as graphics attempt to do. Nor can it be formally proved due to combinatorial explosion of states. Theories are by their nature abstract which as its good points and bad points. The good point is that we can say general things that apply in most cases about the software through the language of software methods. The bad point is that the world is full of exceptions that must all be handled and software theory cannot do that they must be handled at the level of implementation. The world as it exists as a collection of states of affairs presents software that tries to harmonize with the world a very severe problem.

C. Theory <-> Incarnation

Theory as design elements are encoded into software source code that embodies it and actually runs when it is compiled. Due to delocalization which smears the design elements out in the code it is very difficult to maintain the design once the code has been created. The code is in constant flux and the interference of design elements with each other especially in high performance code can cause design elements to be severely mangled in the this process of incarnation. Code and Design theory are fundamentally at odds and the code always wins because it actually does something whereas the design only makes things understandable to the humans that are creating the program.

D. Representation <-> World Interface (Mays #4 & 10)

Designers must attempt to use mental simulation to anticipate the workings of the software. That mental simulation needs to be revisited over and over again and remain true to the software design and the world that the program is going to function in. The representations no matter how augmented cannot show the complexity of the world completely. All that they can do is provide a gloss that stands halfway between the world and theory that is manipulable by the designer.

E. Representation <-> Incarnation

Representations by minimal methods create abstract design elements that gloss implementation issues which in turn gloss the actual implementation constructs at the level of delocalization which in turn glosses assembly code and which in turn glosses binary instructions on a particular hardware platform. What we have is a series of levels of representation. Within the design level we have hierarchies of Data, Events, Functions and Agents. These levels of design and implementation can be very complex and stacked deep. But each level must be perfectly matched to those above and below it for a good design to occur and for a complex system this is very difficult to do and sustain once accomplished in the face of changes.
F. World Interface <-> Incarnation

Not only is the world complex and unpredictable and the Incarnation of software on particular hardware platforms complex but these two things add together to produce a super complexity where the hardware platform is engaged in the world which the software is attempting to control things and make them play together properly. In another paper that the author has written this role of software is called meta-technology. Software binds together different technological structures into a meta-system which it attempts to operate so that harmony between the systems is maintained. It is the role of software to do this. So software is incarnated in the technological world as a meta-technology that makes other pieces of technology play together and integrate while performing complex tasks that can be safety critical. As such the software is embedded in spacetime/timespace making specific things happen at particular points in time and space among the hardware ensemble. Bridging the gap between the incarnated theories of its design and its embedding in the world in a way that works properly is a very difficult task for something as brittle as software.

We have gone into the nature of the essence of software to show what the generative system looks like from the point of view of software. The essence of the generative system relates our theories of the world to representations which in turn are incarnated within the world and tested against that world. Previously I have said in another paper that this gives software many of the same attributes that appear in the philosophy of science as problems. Science produces hypotheses from theories which it tests. Software produces representations from theories which it incarnates and tests against the world. So software has similar problems determining what is a good theory and how should it be tested. The difference is that with software you get more immediate feedback and one is not trying to extend knowledge but merely embody knowledge.

What is interesting is that one cannot demonstrate understanding by mere imitation. All that we have said about software merely shows it to us as something that imitates the world’s states in order to harmonize with it. But imitation even though it is a sincere form of flattery does not necessarily display true understanding of the world and it is only by embedding understanding in programs do they become less brittle. Software by itself is very brittle. Anything that is not explicitly programmed into it causes it to fail. In order to create robust programs we must produce those that imitate the living and the knowing things in the universe. That is we must strive to make them true systems that imitate the organisms that Rescher points out are the source of our ideas of systems. Thus to give robustness to software meta-systems we would extend Klir’s epistemological hierarchy by placing a level of Knowledge beyond the generative level. To display understanding we must go beyond the mere imitation of data systems and produce systems with knowledge and flexibility. Thus we arrive at the need for a living/cognitive epistemological level beyond the generative level.

5.5. ARTIFICIAL LIVING KNOWLEDGE SYSTEM

To have deep understanding of a generative system requires a knowledge representation scheme to be overlaid on the generative system. That supplemental system displays understanding of the workings of the generative system. It needs to ultimately be living/cognitive or what is called Autopoietic. Autopoiesis means self-producing or self-organizing. Thus we posit that the next level is most like an organism that is the root metaphor for the system. It is not just a knowledge level added to the generative but the knowledge is activated by being the self-knowledge of an autonomous being. This level actually allows us to understand software better because it is the next higher meta-level above software called the proto-technical and operating at the next higher meta-level of Being which is Wild Being as defined by Merleau-Ponty.

When we think of software we notice that the attempt is made to define it in such a way to get rid of all the paradoxes like self-modifying code and spaghetti goto statements and others. When we move to the Artificial Intelligence and Life level beyond software what we see is a mosaic of techniques with nothing like methodologies for us to hang our hats on. Each AI or ALife technique competes with all the others in a bewildering array of sophisticated but very basic programming techniques mostly realized at the implementation level. After studying this area for a long time I realized that there was a reason there were no equivalents to minimal methods for AI and ALife. That is because all the paradoxes that were pushed out of the software layer by the discipline of Software Engineering were pushed into AI and ALife. Each of these techniques revolved around some paradox in the software layer and because they were paradoxes they could never be resolved into a simple method that is easily
represented. All the monstrous aspects of software are collected here and combined to create specific techniques that will use the side effects of software to create imitations of life or cognition.

Another point about this level is that it uses software as an enabling machine instead of hardware. Because of that it is free to create theoretical structures that are completely disconnected from reality. Thus Virtual worlds arise as the abodes of artificial living and intelligent creatures that can be completely disconnected from any kind of recognizable reality enforced by the world we live in mundanely. When this detachment from reality is combined with network technology then you get the advent of cyberspace as the realm of all possible virtual worlds. Within these worlds artificial intelligent and living creatures roam which will be created by the opaque AI and ALife techniques that arise from the paradoxes in the software layer. Combinations of opaque techniques will render these creatures even more opaque and incomprehensible. Thus we are engaged in creating alien creatures within our virtual worlds which we can never understand. They are inherently incomprehensible since they are created using all the techniques banished from software engineering because they are not trusted to produce assured results in the real world.

Between the fantasy virtual world and the real world stands what Geleterner calls the “mirror world” which attempts to render an image of the real world in virtual reality. Mirror worlds stand between the real world and the fantasy worlds disconnected from reality. Mirror worlds give us more knowledge about the actual world than we would normally possess. They are worlds with superabundance of information and real-time connection to the actual world. They are the mirror between our world and the fantasy worlds that depart from reality in significant ways. We can say that the mirror worlds are super-real and form the reversible interface between reality and irreality. For instance a fantasy world may be a world where a fundamental assumption that is made in the designated as real world is changed to see what would happen. These fantasy worlds give us the possibility of conducting experiments in worlds that do not exist which will shed more light on the world that does exist thought intersubjective agreement. It is though mirror worlds and fantasy worlds that our ability to socially construct worlds is unleashed into realms that it was impossible to enter before. These mirror worlds and fantasy worlds will have a profound impact on the designated as real world as a hyper extension which when treated as part of the designated as real world actually has profound effects on that to which it is supplemented. This is because all of these worlds function in the realm of Hyper Being which as Derrida has shown has the form of a supplement which changes the meaning of the thing to which it is attached.

At this level generators become imitations of living knowing organisms. That is they imitate the most sophisticated systems we know which are living creatures. Thus it is only at this level that we have a true attempt to portray systems in relation to the root metaphor of organisms with cognitive capacity. These organisms have a fundamental capability to learn and adapt. And this must be taken into account in our model. Therefore an important part of this level of manifestation are the meta-levels of learning which were first defined by Bateson. There are four of these meta-levels of learning which scale the ladder of meta-levels until they reach the unthinkable which lies at the fifth meta-level beyond all forms of learning.

5.5.1. LEARNING SYSTEM

The knowledge system may learn about other systems or may expand to cover a domain of systems rather than a single system of a particular kind. Thus Learning systems supplement Knowledge systems. When software systems display learning then they cease to be fragile with respect to changes in their environment. A learning software system may also exhibit this learning with respect to itself producing internal images of itself and learning about itself.

5.5.2. META-LEARNING SYSTEMS

These systems as Bateson shows learn to learn. Learning to learn means exploring new ways of learning. This allows such a software system to cope with discontinuous changes in its environment and within itself. When we learn to learn we increase our learning capacity and also gain new learning skills. Such a software system would be very robust with respect to its environment being able to cope with environmental changes and changes in itself that are unexpected.

5.5.3. META-META-LEARNING SYSTEMS

Learning how to learn can be supplemented by Learning at

76. Geleterner. Mirror Worlds
77. Bateson. Steps to the Ecology of the Mind
the next meta-level which means changing paradigms of learning how to learn. There may be different paradigms of how to learn to learn which is to say different approaches to learning to learn. At this meta-meta-level the difference between self and environment become irrelevant. The environment and the self is considered a single meta-system where the environment learns from the self and vice versa. At this meta-levels the differences in paradigms in learning become important and the ability to switch paradigms of learning so that new self-other configurations become possible becomes important.

5.5.4. META-META-META-LEARNING SYSTEMS

Bateson says that the next level is one in which ones whole worldview changes and that this is the highest meta-level of learning. Beyond this is only the unthinkable. It is at this level that the projection of the world by the self-other meta-system is accomplished. The key feature of this level is the appearance of the emergent event. The emergent event is the possibility of a genuinely new thing to come into existence. A meta-system that operates at this meta-level could handle the appearance of the genuinely emergent event. The genuinely emergent event is defined as one that moves thought all four meta-levels of Being as it enters the clearing-in-Being and becomes part of the World.

An example of a Meta-meta-meta learning system is Western science. In school we are taught things in a certain pedological style. But as we encounter different teachers we realize that there are different ways of learning and we attempt to learn how to learn in these different ways. For instance, there are ways of learning suited to those who are language oriented, graphically oriented, and kinetically oriented. But we may combine these different ways of learning to achieve particular learning effects that are difficult to achieve in any other way. As an example, audio visual materials may be combined with an exercise. But eventually as we begin to achieve mastery of subjects we realize that we need to produce our own synthesis of the materials in order to show mastery. These syntheses appear like paradigms in that they go beyond the information given to posit theories which are not contained in what we have learned to learn. When we can advance these paradigms then we have in effect reached the fourth meta-level of learning where we advance the state of the discipline in which we are engaged. Finding these cutting edges at the fourth meta-level of learning is very difficult. In fact one can say that the whole problem of intellectual advance is to locate these cutting edges and make progress with respect to the disciplines at those edges. Persons who do not learn to learn to learn cannot locate these cutting edges. Those do locate them and contribute to our understanding at those cutting edges are the ones who bring genuinely new things into existence. They are the ones who transform the world.

5.5.5. THE UNTHINKABLE

The unthinkable is the meta-level beyond which we can create learning representations.

Notice that we have gone beyond Klir’s original formation to add levels of learning until we reached the unthinkable. We note that the unthinkable is equivalent to the infinite meta-levels to which structural and process models ramify and fuse.

We have also noted that when we reach the infinite meta-structures or meta-process models or the unthinkable we have reached a point identical with the “essence of manifestation” described by Henry that is the point of pure immanence which never manifests.

The unthinkable may be considered identical with the Buddhist non-concept non-experience called Emptiness. Emptiness is itself empty. It is the expression of the absolute middle between all nihilistic opposites. Emptiness is the center of the vortex around which the dynamic of worldview projection at each of the meta-levels of learning revolves. Understanding Emptiness is essential to understanding the projection of the worldview because Emptiness balances the whole action of worldview projection.

5.6. WINGS TO INFINITY

Now we will explore each of the wings that take us to infinity of process and structural meta-levels and see how they function at the multiple levels of the epistemological hierarchy. In what follows the word LEVEL can be replaced with any of the following levels we have discovered:
5.6.1. STRUCTURAL LEVEL SYSTEM

For software the structure appears as multiple whole-part relations exemplifying the relations between patterns and forms.

5.6.2. META-STRUCTURAL LEVEL SYSTEM

These whole-part relations become ever more inter-embedded. At the first level of inter-embedding there are structures within structures.

5.6.3. META-META-STRUCTURAL LEVEL SYSTEM

At the next interembedding level there are structures within structures within structures. This regress is infinite because we can imagine structures embedded within each other to infinite levels of logical typing. Ultimately these meta-levels of structures approach the unthinkable, which is a complexity of structure beyond which the human mind cannot conceive.

5.6.4. MODELED LEVEL SYSTEM

Models are the temporal structuring by which the spatial structuring is controlled and changed over time. We call these process models.

5.6.5. META-MODELED LEVEL SYSTEM

We can think of processes within processes controlling structures over time.

5.6.6. META-META-MODELED LEVEL SYSTEM

There is also an infinite regress for models of processes as we can think of processes within processes infinitely. These also approach the infinity of meta-levels of process which is unthinkable.

5.7. FUSION OF PROCESS AND STRUCTURE

There is an interference between our structural and process model reifications which sees them as fused. This fusion represents the timespace causal view of the system. In one fused view structure dominates time whereas in the other time dominates structure. These are equivalent to the proto-imaginaries found in Spencer-Brown’s Laws of Form. We follow Merleau-Ponty in calling these points of fusion between process and structure chiasms or points of reversibility. In fact, we will coin a new term called intaglio for the fused relation between process and structure. Intaglio is the engraving of an image within a stone so that it appears three dimensional usually though the other side of the transparent stone. Many times the intaglio is frosted to produce the appearance of solidity to the image. There are sculptures that exist made of glass where intaglio is used on both sides to give the appearance of intertwined figures connected thought the medium of the glass. Many times these are figures of men and women intertwined in some exotic fashion. In other words in these intaglio sculptures what exists is a fusion of the figures thought the connecting medium. The figures themselves have no reality other than the medium that holds the carving of the intaglio. So it is with the fusion of process and structure. They do not exist as separate entities but only exist as the chiasm or reversibility between them. We can talk of this fusion at three levels.

- PATTERN/FORM CHIASM = structuralized forms
- LIVING/COGNITIVE CHIASM = autopoietic systems
- SOCIAL/PSYCHIC CHIASM = reflexive systems

Here we understand that form and pattern together produces structures of forms and that processes model these over time. But form and pattern also have an intaglio relation in which one cannot be completely separated from the other. In that relation they exhibit interferences which reveal the trace structures below the level of manifestation of form and pattern. In those trace structures the intaglio of form and pattern as interference patterns between disorder and order appear. It is this trace level that give us the foundation for the understanding of the autopoietic systems that imitate living/cognitive organisms. The living and the cognitive also produce a fusion of process and structure that has a qualitative
difference from process or structure in isolation. The autopoietic theory of Maturana and Varela display these features of reversibility very well. However, these theories break down when we move to consider the social. Thus the social must be a new level of organization that goes beyond the autopoietic. Autopoietic system maintain their organization homeostatically. A reflexive system is defined as the next level beyond the autopoietic and it is seen as heterodynamic instead of homeostatic with respect to its organization. This means a reflexive system is ecstatic in projecting the world and changes its organization dynamically to different organizational regimes. Thus the reflexive system can accept emergent events as the way the worldview is projected changes radically over time. We say that such a fusion of process and structure lies right on the brink of the unthinkable because it accepts changes from the region of what is incomprehensible in relation to it and deals with these changes which are called emergent events. At this level there is a chiasm between the social and the psychic. From one point of view reflexive systems are social but from another point of view they are psychological. Thus there is a psychosocial dual-intaglio at the level of the reflexive heterodynamic system. The understanding of heterodynamic systems is the furthest reaches of all systems theory.

Each of these levels of dual intaglio that we have been laying out are extensions of General Systems Theory. They lay beyond the understanding of structural-process fusion. Structural-process and process-structural fusion exist at each level of the epistemological framework. We can view these merely as reversible process and structural modes of the framework or we can look beyond that to see the qualitative difference between the fused and the unfused aspects of structure and process. This qualitative difference points us toward the special systems that emerge from General Systems Theory. These are the systems theories regarding dissipative, autopoietic and reflexive systems. They appear as the fusion of process and structure from the timespace perspective. This fusion has a qualitative difference that expresses itself quantitatively as well.\textsuperscript{78} We see here that Dissipative systems can be looked at from the point of view of the object, source, data, and generative systems. The Autopoietic system can be looked at from the point of view of all these systems as well as from the point of view of Knowledge and Life. The levels of learning are the province of the reflexive system and can be considered up to the point of unthinkable.

Now that we have defined the special systems and their chiasmic fusion we can go back to consider the generic fusion from the process and structural perspectives.

5.7.1. STRUCTURAL MODELED LEVEL SYSTEM

At each level there is a fusion which emphasizes structure over process and one which emphasizes process over structure. These take on a different quality from the timespace viewpoint that reveals the special systems that emerge from GST. However if we go back and look at the structural-modeled system that exists at each level from the spacetime viewpoint we see that when space dominates time we get the equivalent of a knowledge representation system as in Prolog where connections in space are more important than the processing in time. In knowledge representation schemes the knowledge is coded into structures which are unified by a single logical algorithm. There is only one process and multiple knowledge representations on which it does its work.

5.7.2. MODELED STRUCTURAL LEVEL SYSTEM

When time dominates space from a spacetime perspective we see that we get a normal relation in programming between processing and memory where the processing controls the memory rather than the configuration of memory controlling the processing. But here we have an interpreted system where data and processing are more intimately connected rather than a precompiled program which operates on completely separate data.

Knowledge representation that emphasizes space over time is independent of interpretation which emphasizes time over space but still allows fusion of data and processing. These two fusions are orthogonal to each other in every case at each level of the epistemological framework.

So at the data level there can be control data and non-control data. This means that non-control data is dominated by processes while control data dominates processes and contains in the data stream the structure that controls processes.

At the generative level we see that data can be coded into tables which control processing or we can allow processing to contain many more control statements and we can code the functioning of the software into source

\textsuperscript{78} For further details see the author’s two series of papers On the Social Construction of Emergent Worlds and Steps Toward the Threshold of the Social (unpublished manuscripts).
code algorithm.

At the knowledge level we get the difference between Prolog which uses the unification algorithm to process static knowledge structures and Lisp which does its processing on lists where the list itself can be the program being executed. Thus list processing algorithms dominate the data representation but they are fused. Prolog expresses this fusion in the way it rewrites its knowledge representation causing the unification algorithm to give different results from pass to pass.

At the levels of learning we can either emphasize the materials being learned or the learning process itself. If we emphasize the materials being learned then the drive to learn is external and we call this teaching. If we emphasize the process of learning over the materials learned then the drive is internal and we call this self-realization which Maslow called a drive. This ramifies to all the meta-levels of learning. The drive to learn at any meta-level can be either internal or external but whatever the driving force learning has to be reciprocal and social. When we see this learning mirrored within the individual we call that the psychological realm. The psychological and the social are mirror opposites.

The fusion of the structure and process represents yet another way in which the unthinkable enters the epistemological framework. We already noted that the framework itself extends past the generative to the knowledge level and on up the hierarchy of the meta-levels of learning to the unthinkable. Then we saw that at each epistemological level there are two wings of extension to infinity. The point of infinity for both wings of meta-level extension is the same and is identical with the unthinkable. Now we see that each wing fuses with the other wing of the epistemological framework in a way that can either be seen causally from the point of view of timespace or in terms of separation from the viewpoint of spacetime. When we interpreted fusion from the point of view of timespace we recognized the levels of chiasm related to the generative system, the knowledge and living level and the levels of learning. These we defined as the special systems that emanate from General Systems Theory. The we turned around and saw that these fusions of the wings can be seen from spacetime viewpoint instead in terms of separation and we saw how that meant the difference between coding action into spatial configurations rather than writing algorithms and we can see how these may be expressed at every level of the epistemological hierarchy. But the reversibility between the spacetime and timespace views of fusion also points us toward the essence of manifestation because of the qualitative and quantitative differences between these two views of fusion that produce a blind spot in our view of chiasmically fused aspects of systems. We cannot understand easily the connection between timespacelike fusion and spacetime fusion of the two wings that tend toward and infinity of meta-levels.

In effect this shows that we need to understand better the extension of General Systems Theory into the realm of the special systems. The means for doing this is Software Engineering because it is software engineering that provides the connection to computability of systems. The special systems appear when we consider the fusion of structure and process from the causal or timespace perspective. They do not appear when we consider the spacetime perspective on fusion. Instead there we get a view of the computability of the combination of structure and process. Thus the special systems are bound to computability in a mysterious fashion which is not clear as we reverse our perspective from timespace to spacetime emphasis. In effect this calls for the development of a computational meta-system orthogonal to General Systems Theory. That computational meta-system is embodied in Goertzel’s Magician Systems first proposed in his work Chaotic Logic. If we see the expansion of the epistemological framework toward the unthinkable and the spreading of the wings of meta-levels of structure and process toward infinity as the dual opposites of the timespace and spacetime views of fusion then we see that these two duals define an interface which is orthogonal to GST within which the dual to GST must exist. That dual must deal with chaotic processes in a structured way which is computable. Goertzel’s Magician meta-systems is the only candidate yet found that fulfills the conditions...
that this dual must fulfill. And it turns out that the Magician meta-system is intimately connected to the special systems when it is expressed meta-algebraically. Magician systems also have the characteristic that they express formally all the different kinds of Being. So magician systems provide us with a model of the balance of heterodynamics and homeostasis within the realm of dynamical dissipation.

Therefore we see that General Systems Theory as the theory of gestalts or showing and hiding systems must have a dual which expresses the meta-system and meta-gestalt within which gestalts form. We posit that this dual is the Magician meta-system and that it is defined negatively by the relation of the fall into the essence of manifestation via infinite meta-levels and finite meta-levels to the embedding of fusion between process and structure seen in terms of spacetime and timespace. This reversibility between two views of fusion and two approaches to the essence of manifestation defines possibility of the Magician meta-system negatively. It is by studying the relation of the Magician meta-system and the special systems in this context that we realize their inner connection. And that connection is made possible by computability and ultimately by software as an embodiment within the matrix of spacetime and timespace.

GST is formed completely in the realm of Pure Presence. But it attempts to deal with processes in terms of models of temporal structuring as opposed to spatial structuring. Thus as a formal-structural system it gives us a view of processes while attempting to not fall into Process Being. When we realize that systems are gestalts of showing and hiding processes we fall into Process Being and we must reinterpret GST within that context. When we extend the GST epistemological framework we see that it truncates in the unthinkable which is either finite or infinite. We have seen that this is an expression of the essence of manifestation and that is what takes GST to the third meta-level of Hyper Being where the software essence also resides. At that level we see GST and Software Engineering as duals. But then when we look at the Epistemological Framework we see that there are nodes of fusion between process and structure. We can see these in terms of spacetime or timespace as we look at the embedding of the GST epistemological framework in the timespace/spacetime matrix. These two views show us the place of the special systems that emerge from GST with their chiasmic relations between fused components. But if we look at them from another angle we get a view of the computability of these fused structures seen externally in terms of process and structure. These two views of fusion indicate the presence of Wild Being which is the highest meta-level of Being beyond Hyper Being. Within the gap between fusion and the essence of manifestation the possibility of a dual to GST arises and we posit that this dual is a Magician meta-system that combines the inscription of traces with the computational emulation of chaotic processes. We posit that Magicians are the meta-system which combines all the special systems into a single proto-gestalt from which all the gestalts of systems arise within the clearing-in-Being. Magician systems combine all the kinds of Being into a single computable formal meta-system. The meta-system of Magicians is the dual of the structural-formal system of GST and it is software than provides the interface between them as the means of conferring computability to both.

The discovery of a dual to General Systems Theory is a surprising result when needs further study in order to explore all of its ramifications. That dual is a meta-system that defines the basis for the emanation of all the systems that arise within General Systems Theory. This dual of GST can only be appreciated from the point of view articulated by the definition of the different kinds of Being. But once this perspective on systems that looks explicitly at their ontological basis has been established it becomes clear that GST needs underpinnings that attach it to all the more fundamental ontological levels. Magician meta-systems perform that role. They unify all special systems theories and provide a meta system that defines their ontological basis. It is clear that a major extension to the foundations of General Systems Theory has been proposed based on these ontological ramifications of the fragmentation of Being which relates the most general system to a computational infrastructure and also to the thresholds of complexity that provide the basis for the emanation of dissipative, autopoietic and reflexive systems.

February 22, 2007
Figure 9:

U

Unthinkable

L4

L3

L2

L1

Learning

Autopoietic

Special System

Dissipative

Special System

Reflexive

Special System

Structural Models

Process MetaModels
Software ontology concerns the ultimate reality of the software process and product. The study of software ontology forms the necessary basis for understanding software engineering\(^1\) and particularly software engineering methodologies. Software ontology is a branch of metaphysics directed at a particular class of entities that have a different foundation in Being that is very rare. This same field of study has been called Grammatology, the science of traces, by J. Derrida\(^2\).

Being is not just an empty concept, but covers all ways things have of making themselves present within our world. These particular kinds of entities called "software" have a different way of making themselves present in showing and hiding relations than we are normally used to in our interactions with technological systems. Understanding these different ways of presencing peculiar to software must be the starting point for any truly deep paradigm for understanding software design methods, because these methodologies are ways of attempting to show software entities. Treating software as if it were merely another instance of the kinds of things we normally encounter within the world, will certainly lead to anomalies and conundrums which can only be avoided by attempting to understand the strange nature of software from a philosophical perspective.

Metaphysics gets its name from the type of philosophical questions it asks that are beyond the realm of questions answerable by physics, or any science that addresses a particular kind of thing. It normally is composed of epistemological questions about how we know what we know and ontological questions that concern the reality of the things we know about. The deepest level of questioning concerning the universe which is not religious, i.e. theological, is ontological. Ontology deals with the most general concept we have about the nature of things which is their Being. Being concerns the showing of things within our world. Not all things show themselves in the same way within the world. This is a discovery of modern ontology which we will use to attempt to understand the nature of software. If we wish to have a really deep foundation for our scientific paradigms, it must be ontological. Since we are not taught to think at this level of generality, it is very difficult to get a picture of the differences in the way things manifest. However, unless we understand these differences, we are likely to commit a category mistake in our reasoning about software. The following attempts to clarify these issues to prevent this mistake in our classification of software entities that might cloud our ability to understand the general phenomena of software.

Software is a relatively new kind of object that we might chance to discover when we look around us within the world. The strange thing is that most people never see software. They see automated machinery or occasionally computers, but rarely are the essential instructions that make computers function ever seen except by software engineers. Normally, software engineers are so busy trying to build software that they do not question whether its ontological foundations are any different from any other kind of object. In fact, normally we live without questioning the reality of anything within our lifeworld\(^3\).

It has been said:

*Philosophy may be ignored but not escaped; and those who most ignore least*

---

3. This term was introduced by Husserl in *Crisis of European Sciences and Transcendental Phenomenology* (Evanston: Northwestern U. P., 1970) and developed by Alfred Schutz in his two volumes on *The Structures of the Life-world* (Evanston IL: Northwestern U.P., Vol. 1, 1937; Vol. 2 with Thomas Luckmann, 1989). It refers to the world as experienced from a phenomenological perspective.
As software engineers, it is our failures in building software systems that cause us, at moments of desperation, to wax philosophical. Because engineers normally do not have philosophy as a part of their training, these moments of reflection on the ultimate truth of Murphy’s laws are experienced as dark nights of the soul where existential absurdity becomes palpable. Camus used the myth of Sisyphus, who eternally pushed a boulder up a mountain only to watch it roll down again, and again as the image for the absurdity of life. Murphy’s law—If anything can go wrong it will—is only an addendum to the basic message of the myth of Sisyphus that states that anything can start the boulder rolling back down the hill at anytime so be expecting it. For the software engineer and the user of computer software this image of the boulder is transformed into the disaster of spaghetti code that is totally incomprehensible, or the tragedy of the system going down, or the nightmare of losing a disk. Software engineering itself is a response to these existential problems as they relate to the software portion of the technical system. Software engineering takes as its premise that if we concentrate on the way we build software and attempt to introduce rigor into the development process, it will be possible to build higher quality systems and avoid the terrors of existential absurdity for both ourselves and the users of the software we build. Unfortunately, there appears to be a hitch. Software systems are not yielding readily to traditional pragmatic engineering disciplines. If they had, software engineering would have never separated from hardware engineering. The addition of the word “engineering” and recent modifications to standard software production practice has not been enough to subdue the monster (seen by some as a werewolf). Superficial changes in the discipline of programming and its management have only served to emphasize the enormity of the problem we face as software engineers. As we search for ways to apply engineering discipline to software production, we are faced more and more with the question, “What is software anyway?” and “How can such a simple thing as a few commands to a stupid processor cause so many problems?”

Normally, software is treated as a product by software engineers. However, some researchers have realized that it is an entity in the world which can be treated as the object of scientific inquiry. Software science is also a fledgling discipline. Software science treats software texts, compiled code, and running software systems as objects of study. Applying the scientific method to these objects, the software scientist attempts to understand the characteristics of the software. Halstead, who coined the term attempted to discover complexity metrics. However, anyone who treats software from an objective point of view as a subject of study may be classed as a software scientist. For instance, software scientists may be interested in measuring and modeling performance of computer systems and networks. Software engineers produce the product which is used by the end user. The software scientist observes the product, the production process, or the usage of the product and attempts to analyze and measure the resulting system. Software engineering and software science are complementary disciplines which at this time in practice merge into an ambiguous whole. However, in the future we must learn to separate the engineering perspective from the scientific perspective as has been done in other disciplines. We need to be able to switch back and forth between these perspectives in order to hone our software practices much the way mechanical and chemical engineers know and use physical and chemical scientific knowledge. Because our software engineering and software scientific disciplines are so new, we are hard put to answer just what knowledge software science has to offer the software engineer, or vice versa.

Software science and engineering hope to draw upon the tradition of science and engineering that has developed in relation to other more advanced disciplines. It naturally looks to those more mature disciplines for guidance in how to treat its products and objects. Unfortunately, it seems that a problem arises at this point. There doesn’t seem to be any direct mapping of techniques or approaches from these more developed science and engineering disciplines into the software disciplines. In fact, as we look for help, we often find that these other more mature disciplines, despite their successes, are themselves in trouble. They are themselves undergoing profound changes and are undergoing critical examinations of their own foundations. Thus, we do not discover a horde of certain knowledge and practices which our fledgling science can rely on unquestioningly. Instead,

---

we find that we are forced ultimately to invent much of our new software disciplines. So we end up asking ourselves, “What is engineering and does it apply to software?” and “What is science and how does it study software?” As we ask these questions, we find that our understanding of science and engineering is altered fundamentally. Ultimately this causes us to ask, “What is software, anyway, that it causes us to rethink science and engineering?” We know from experience that software is strange stuff. It is for us what quick silver was for the ancients (liquid metal?). It is a marginal object of some kind. More primitive societies always treated marginal objects as taboo. This is why the werewolf comes readily to mind as a mythical image. Marginal objects always require special initiation, purification, and ritual observances. In our culture this means special higher education like the Software Engineering Institute’s software engineering masters curriculum, special training on the job in producing software systems, and special methodological approaches which are very different from normal programming in the small. Marginal objects are hard to categorize, difficult to handle, impossible to pin down, and seem to be crazy combinations of normal things thrown together in an outlandish fashion. Every culture treats these objects with care using special precautions. The question is, “Why does software deserve this status?” and “Is there any way to turn software into just another normal thing that can be manipulated without special precautions?” This brings us back to the question of the nature of software. The nature of software is the domain of software ontology. This discipline is more basic that either software engineering or software science. It is the root of both of these disciplines. Software ontology could be defined as the interface between software science and software engineering. Software science seeks to discover the characteristics of software objects. Software engineering seeks to discover ways of making software products which may be considered as objects by the software scientists or tools by the end user. Between the making and the observing of software, somewhere lies the essence of the software itself. The observers and the producers of software deal with this essence of software every day. The essence of software provides the ultimate constraints on what may be observed about software or what can be created in a software system. Yet it is difficult for us to see what exactly the essence of software is. We are caught up in all the accidental characteristics of particular software systems. There is no ideal pure software system, but many different particular systems in different languages, running on different machines, doing various applications. Attempting to see the essence of software and thus define an ultimate statement of what software itself is, cannot be done either inductively or deductively. We recognize software when we see it, but we cannot deduce all its forms from axioms nor see it clearly through all the diversity of software systems. We recognize that what we call software is different from other things, but cannot pin down exactly what the difference is. Thus, it is necessary to start with the discipline of ontology itself. What can ontology tell us about any “thing”. It turns out that in this century many advances have been made in the discipline of ontology. These are not as well known as the advances in physics and other disciplines. This is because it appears to be a highly esoteric field that doesn’t seem to apply to anything because it applies to everything. What can be said with assurance about everything would appear to be very little. In fact, until the turn of the twentieth century ontology was little more than a recounting of the Western view of reality as defined by Kant in the 18th Century. Around the turn of the century, though, things began to get interesting because it was discovered that the received view of reality was more complex than first imagined. Since then the history of ontology has been a progressive dismantling of the Western worldview which has coincided with the discovery that the physical world was very different than expected as well. The development of physics and the dismantling of the Western worldview, along with the systematic research into the other worldviews as a result of global colonization of the third world by the West, has led us to an interesting existential situation. It turns out that our worldview, as expressed by philosophers, has been discovered to be groundless. This means that there is no ultimate philosophically secure basis for our view of the world that we have foisted upon everyone else via colonization. The groundlessness of our worldview is expressed within ontology as what will be described as the fragmentation of the concept of Being. On the other hand, other worldviews discovered in the process of colonization are not only more sophisticated, but agree in significant ways with the puzzling physical nature of the universe discovered by modern physicists. We have used our

---


worldview efficiently to dominate the world and discover
the nature of the physical universe, while at the same time
we have discovered that this worldview has no claim to
any ultimate truth. We could, perhaps, live with our
worldview which confers power over nature and other
nations regardless of doubts of its claims to truth if it were
not for the dangerous side effects which indicates a
fundamental imbalance. This imbalance in our worldview,
because of its groundlessness, causes us to destroy the
environment upon which we ourselves are dependent. The
list of palpable side-effects of the imbalance in our
Western worldview are too many to catalogue here, and
others have been more eloquent in bringing attention to
these areas of concern10. The point is that the combination
of the side-effects of the unbalanced and groundless
worldview, with its inability to comprehend physical
reality, throws everyone who thinks about the world in
which we live today into a tizzy. The power we exercise
over our environment and each other through the
knowledges derived from our scientific and engineering
disciplines is awesome, yet is offset by our powerlessness
to solve basic environmental and social problems. In
philosophy these core issues concerning the adequacy of
our worldview are dealt with under the rubric of the
relation between man and technology. Man creates
technology through a skillful combination of science and
engineering. Looking at technological systems produced
by men, we gain immediate insight into our own core. It is
our product just as the nests are the products of birds and
the webs are the products of spiders. By observing
technology, we gain insight into the nature of the Western
worldview in both its positive and negative aspects. We
see one of the possibilities of man taken to its ultimate
conclusions. Through this we learn more about the nature
of man and things.

Technology and the interlocking technological system
which is supported by software systems plays an important
role in our relation with our environment and each other.
Software plays a unique role in mediating this
relationship. Software may be thought of as a transitional
object11 mediating the relationship between man and
technology. In a way, software is a meta-technology
which allows us to control technological systems which
are themselves controlling physical, chemical or social
processes. This meta-technology allows us to adapt
technological systems to new circumstances without re-
tooling. It eases our interaction with the technological
systems and gives us more control and more information
concerning controlled processes. Thus, in observing
technology, we must pay special attention to the meta-
technology which software represents. There is a
symbiosis occurring between men and technology which
software allows to be flexible, adaptable, accessible, and
semi-intelligent. We are beginning to see the world in
terms of software systems. By decoding the genetic code,
it has become possible to reprogram the genes in
biological machines called organisms. Simultaneously,
we are destroying the gene-pool and thus radically limiting
the materials with which geneticists might work. We are
beginning to see the relation between mind and body in
terms of the relation between software and hardware, and
thus have developed neuro-linguistic programming to alter
behavior of individuals. Software is providing us with
new tools for organizing information about the world.
Different software tools run on the same computer
providing different applications. These tools are
extending the range of our understanding of the world. At
the same time we are learning to see the world itself in
terms of a software metaphor. The field of Cognitive
Psychology is an example of a discipline that primarily
sees the mind/brain as an information processor. This
strange development, accelerated by the advent of
personal computers and workstations, brings the question
of the nature of software to the fore for everyone, but
especially for software engineers. We make the stuff and
don’t even think to wonder what it is.

This series of essays12 will focus on the central concern of
software engineering. That concern is the definition of the
correct methods for designing software. Other aspects of
software engineering rigor appear to be straightforward,
even if they are expensive, difficult to implement, or hard
to define, measure and control. An excellent introduction
to these other aspects will be found in Software
Perspectives13 by Peter Freeman. This book offers a wide
perspective on the nature of the software engineering
discipline into which many of the other textbooks on
software engineering practices need to be fit. This broad
perspective offers a structural model that focuses on

10. See Morris Berman Coming To Our Senses (NY: Simon & Schuster,
1989).
11. This phrase is sometimes used in psychology for objects that make other
things accessible. See Winnicott, D.W., Playing and Reality (London:
Travistock, 1971).
12. This essay is the first in a series on Software Engineering Foundations. The
second essay concerns Software Systems Meta-methodology which goes
into depth concerning the structure of software engineering methods and its
links to General Systems Theory. The third essay presents the outlines of the
Integrated Software Engineering Methodology in terms of the formulation of a
design language for specifying real-time software designs.
External Organization, Objectives, Policies, People, Procedures, Information, Tools, Software Systems, and Supporting Technology. The perspective on software engineering offered by Peter Freeman will be assumed as the context for the rest of the discussion in this essay. The functional model consisting of the activities of Analysis, Design, Construction, Delivery, Support, and Management presented by Freeman is very basic and needs to be augmented with a fully articulated process model such as IEEE P1074 Standard for Software Lifecycle Processes¹⁴, or an equally relevant process model like that produced by Brian Dickinson in his book Developing Quality Systems.¹⁵ To these process models must be added a lifecycle model such as the waterfall model exemplified by the military software standard DOD-STD-2167A¹⁷, or the spiral model of Barry Boehm,¹⁸ or the stepwise refinement model defined by Vaclav Rajlich,¹⁹ among others, in order to get a complete picture of the complexity of software engineering regimen. These books set the stage for a discussion of the software engineering methods. Methods are particular implementations of crucial software engineering processes. Non-trivial processes are defined by procedures. Procedures are, according to Freeman, two dimensions answering the questions, “How the procedure works” and “what the procedure produces.” Procedures may work as Methods, Techniques, Guidelines, or Heuristics. Procedures may produce Technical, Quality Assessment, or Management outputs. Freeman defines a method as “a mostly complete set of rules for arriving at a desired result; in extreme cases, it may be an algorithm.” He defines the technical product of a procedure as “the primary result of the procedure is intended to be a technical workproduct such as a design or a program.” The technical methods of software engineering are crucial because it is through them that the primary transformations of the software system are performed. The transformation of the software system, from requirements to specifications, to architectural design, to detailed design, to code, to a tested and integrated deliverable product are each difficult to achieve steps. Methods attempt to make these transformations rational, rigorous, and achievable, if not easy and efficient.

Of these transformations the most difficult steps are from requirements to specifications to architectural design. These are called the upstream transformations as distinguished from the others called the downstream transformations. This paper will concentrate upon these upstream transformations. They are still ill-defined, and no known published method completely covers these transformations. Examples of published methods are those of Hatley-Pirbhai,²⁰ Ward-Mellor²¹, Project Technology (Shaler-Mellor)²², etc. All of these methods fit Freeman’s basic definition of a technical method. Xiaping Song has compared many of these methodologies and has developed a framework for understanding methodologies²³.

A software method is a means of abstracting certain aspects of a software system from all the other aspects of that system in order to make the design of the whole system tractable. Currently, most software is produced with little reliance on methods. Production of software without the use of methods is called by practitioners “hacking”. Hacking consists of programming a software product directly, using the software language alone as the representation of the system under construction. Software engineering is distinguished from direct programming by the mandatory implementation of methods for transforming abstracted representations of the system under construction. Direct programming of systems may be suitable for small, non-production systems which can be understood by one person or a small team. Even the usefulness of direct programming on small systems is questionable to the advocates of the use of software methods. Direct programming, though, is widespread and is still being taught to computer science students as the exemplary means of arriving at a working program. Thus, software methods are only learned in production environments where direct programming has been demonstrated to break down. Software methods demand that direct programming not occur until late in the lifecycle of the product except on a trial or experimental basis in the case of prototypes. Software methods abstract the basic characteristics of the software system so that problems may be solved within the abstracted form, which

---

¹⁴. IEEE publication.
¹⁶. See my paper “The Future of Software Process” in which an autopoietic life-cycle is developed as an alternative to the others mentioned. Autopoiesis means self-organizing.
¹⁷. The new version of this, so called 2167b, is officially MIL-STD-SDD which now allows spiral life-cycle models to be used.
¹⁸. The Software Productivity Consortium of Herndon VA has taken these ideas first presented by Dr. Boehm and have developed them.
¹⁹. He has developed stepwise refinement into a formal life-cycle with the same rigor of the waterfall.
is simpler to repair and modify than the programmed representation of the system in a programming language.

Both software engineering processes, which define the work context in which methods operate, and methods for transforming the software system must evolve to solve the problems faced by software engineering. Watts Humphries of the Software Engineering Institute in his book Managing the Software Process has explained a basic framework for the evolution of software engineering practice into a rigorous discipline. This evolution is seen as a judicious mixture of software science and software engineering. Product and process measurement go hand in hand with control and optimization of the software development process. Assuming that this evolutionarily path is correct, it is based on the assumption that adequate methods will be created which will make software design of large, complex real-time software systems tractable. At this point that is a questionable assumption. There is no doubt that methods will be introduced. There are already a plethora. The question is whether they will be effective in dealing with the basic difficulties of software production. The basic difficulties, which have been described by many practitioners, all lead us back to a questioning of the nature of software itself. The methods which will ultimately be effective in dealing with these problems will directly address the basic nature of software entities. The failure of existing methods to fully address the entire spectrum of software characteristics in a systematic way has led to the current confusion of practitioners concerning the usefulness of particular methods and the efficacy of the use of methods in general. Software engineering methods are in a very primitive, nascent, stage of their development. We are all so happy just to have something to guide us in our work, that it is very difficult to think critically about the methods as they exist now. All the current criticism seems to be of a partisan nature directed against the methods which do not catch our fancy and are balanced by unabashed praise for the methods that inexplicably appeal to us. It is significant that these uncritical attitudes are referred to by practitioners as “religious wars”. Each of us has our favorite methods from among the many that are being proposed. However, the ability to be critical of methods in general has not dawned yet.

Informed criticism is not partisan. It is applied uniformly to everything regardless of personal preferences. This type of “constructive criticism” is Cartesian and is properly scientific. When we are finally able to be properly critical concerning software engineering methods, we will begin to ask different kinds of questions about methods. These questions will not focus so much on methods themselves and the comparisons between individual techniques proposed by different authorities. Instead, we will begin to focus on the field within which a method is proposed and its functioning within that field. What is meant here can be explained simply with reference to Gestalt psychology. A Gestalt is a figure/ground relationship. Psychologists have shown that there is a dynamic relation between figures and their backgrounds. At the moment each method is a figure. We look from figure to figure, comparing the methods without recognizing that these figures stand out on a common ground. They are all a function of a common field. At the moment that field is invisible to us. All we can see are the figures of individual methods. In order to see the common field from which they arise, it will be necessary to develop meta-methods that explicitly render that background visible. When one attempts to develop these meta-methods, called paradigms in philosophy of science, then the study of software methodologies will begin to enter the first phases of scientific discipline.

In this paper I will propose a paradigm for understanding the field of software methods. Paradigms, like the methods themselves, are tentative and exploratory at this point. If we do not begin exploring, however faltering our first steps, we will not be able to understand what software methods are really all about. We will be trapped in the partisan wars between methods and authorities without any critical apparatus to decide issues and judge methodological theories. The general theory of software engineering methods has not been developed yet, and this is a first attempt in this direction. Such a theory asks questions like the following:

- What is a software engineering method?
- What is the relation between software engineering methods?
- Is there a minimal necessary set of software engineering methods?
- Is there criteria for judging the quality and effectiveness of software engineering methods?
- How do software engineering methods relate to the foundations of computer science?
- How should software engineering methods be taught?
- How do software engineering methods relate to the definition of the discipline of software engineering?

24. (NY: Addison-Wesley, 1989)
There will be many different kinds of answers to these and other related questions. The answers will form what will become the paradigms of software engineering in the future. A few of these paradigms will achieve dominance because of their demonstrated value and will vie with each other for supremacy. Occasionally, fundamental changes to these paradigms or their relations will occur, and these will be called the revolutions in software engineering. We are now in the pre-Copernican phase of the development of our discipline. We are longing for the first revolution in which our new discipline will begin to look at itself and its assumptions about the world critically.

I will begin by outlining what, I believe, has the makings of a fruitful approach to the construction of a general paradigm for understanding the field of software engineering methods. This paradigm has a more general applicability to the understanding of structural systems 25, but software systems are a specific case of great interest that will be used here as a concrete example. Practitioners will become, as time passes, more sophisticated and begin to take account the relation between software engineering methodology and other disciplines. This is crucial for the development of our methodologies, but it demands that we become familiar with other disciplines in order to apply the advances made there to our own problems. Thus, the tutorial nature of what follows should be taken as an invitation to the reader to explore other disciplines, and to see the connections between our discipline and many others like philosophy of science, ontology and physical science. However, it is not necessary for the reader to have any deep knowledge of these other disciplines in order to understand the paradigm proposed here.

The departure for this paradigm definition is Peter Naur’s 26 idea that the object produced by engineering is a “theory”. The theory is not code, nor a design, nor documentation, nor any artifact of the software engineering process. Artifacts are secondary products which are spin-offs from the essential work of building a software theory. Naur declares that the software engineering product is not completely captured by any of the artifacts produced in the software engineering process. Naur cites the inability of new programmers on a project to fully understand how a product works without tutoring by more experienced team members. No amount of documentation seems to be sufficient to transfer a good understanding of the software system. In fact, documentation can actually obscure the software product it is meant to elucidate. The relation between the software product and its documentation is not as clear cut as the writers of military standards might have hoped. In fact, the relation between the software theory, that is the real product of software engineering, and the representations of that theory dictated by software engineering methods is enigmatic and peculiarly complex. It is the recognition of this strange relation that makes software methods intrinsically interesting, and necessitates the development of software engineering paradigms which explain and help the practitioner to navigate through what is quickly becoming a methodological morass.

The recognition that software engineering does not produce just artifacts, but primarily theoretical structures, is extremely significant. First of all, it immediately allows us to connect our discipline to the core issues of Western science in a way that few engineering disciplines have been connected. All disciplines which function within our Western philosophical and scientific tradition strive to be scientific. However, few disciplines are as “inherently scientific” as software engineering appears to be. In fact, the fundamental problems in philosophy of science are perfectly exemplified in software engineering in a way which is unique. Science produces a theory of nature. In physical science, which is the preeminent scientific discipline, there is a direct relation between these theories

---

<table>
<thead>
<tr>
<th>prajna {Buddhism}, wisdom source of meaning</th>
<th>traceless trackless lostness</th>
<th>difference⁵ emptiness unthinkable</th>
<th>a decernment of a decision on a discrimination</th>
</tr>
</thead>
<tbody>
<tr>
<td>touch touching {Merleau-Ponty}</td>
<td>no trace</td>
<td>difference⁴ interference</td>
<td>a decision on a discrimination of a demarcation</td>
</tr>
<tr>
<td>inscription in the flesh {Deluze}</td>
<td>no trace</td>
<td>difference³</td>
<td>a discrimination of a demarcation of a demarcation</td>
</tr>
<tr>
<td>chiasm {Merleau-Ponty}</td>
<td>no trace</td>
<td>difference²</td>
<td>a distinguished distinction or a demarcation</td>
</tr>
<tr>
<td>reversibility</td>
<td>no trace</td>
<td>difference¹</td>
<td>a discrimination of a demarcation of a demarcation</td>
</tr>
<tr>
<td>origin, essence, beginning</td>
<td>trace</td>
<td>difference⁰</td>
<td>variation, heterogeniety, undistinguished, indeter</td>
</tr>
<tr>
<td>hinge</td>
<td>trace</td>
<td>difference⁰</td>
<td>indistinct, indecisive, undecernable, indiscrimina</td>
</tr>
<tr>
<td>spacing</td>
<td>trace</td>
<td>difference⁰</td>
<td></td>
</tr>
<tr>
<td>sign</td>
<td>trace</td>
<td>difference⁰</td>
<td></td>
</tr>
<tr>
<td>system &amp; structure</td>
<td>trace</td>
<td>difference⁰</td>
<td></td>
</tr>
<tr>
<td>text production</td>
<td>trace</td>
<td>difference⁰</td>
<td></td>
</tr>
<tr>
<td>noesis/noema</td>
<td>trace</td>
<td>difference⁰</td>
<td></td>
</tr>
<tr>
<td>mark, code</td>
<td>trace</td>
<td>difference⁰</td>
<td></td>
</tr>
<tr>
<td>form</td>
<td>trace</td>
<td>difference⁰</td>
<td></td>
</tr>
<tr>
<td>letter, picture, word</td>
<td>trace</td>
<td>difference⁰</td>
<td></td>
</tr>
<tr>
<td>colors, tints, hues, tones, styles</td>
<td>trace</td>
<td>difference⁰</td>
<td></td>
</tr>
<tr>
<td>mood</td>
<td>trace</td>
<td>difference⁰</td>
<td></td>
</tr>
<tr>
<td>tablet</td>
<td>trace</td>
<td>difference⁰</td>
<td></td>
</tr>
<tr>
<td>the writing surface</td>
<td>trace</td>
<td>difference⁰</td>
<td></td>
</tr>
<tr>
<td>the clean slate</td>
<td>trace</td>
<td>difference⁰</td>
<td></td>
</tr>
<tr>
<td>blank sheet</td>
<td>trace</td>
<td>difference⁰</td>
<td></td>
</tr>
<tr>
<td>hewn</td>
<td>trace</td>
<td>difference⁰</td>
<td></td>
</tr>
<tr>
<td>cleared, cleaned, leveled</td>
<td>trace</td>
<td>difference⁰</td>
<td></td>
</tr>
<tr>
<td>prepared surface</td>
<td>trace</td>
<td>difference⁰</td>
<td></td>
</tr>
<tr>
<td>genocid, ethnic cleansing</td>
<td>trace</td>
<td>difference⁰</td>
<td></td>
</tr>
<tr>
<td>partially ordered</td>
<td>trace</td>
<td>difference⁰</td>
<td></td>
</tr>
<tr>
<td>arranged, fitted together</td>
<td>trace</td>
<td>difference⁰</td>
<td></td>
</tr>
<tr>
<td>texture</td>
<td>trace</td>
<td>difference⁰</td>
<td></td>
</tr>
<tr>
<td>rough hewn</td>
<td>trace</td>
<td>difference⁰</td>
<td></td>
</tr>
<tr>
<td>garden</td>
<td>trace</td>
<td>difference⁰</td>
<td></td>
</tr>
<tr>
<td>selected variety, minimal modification</td>
<td>trace</td>
<td>difference⁰</td>
<td></td>
</tr>
<tr>
<td>of what is</td>
<td>trace</td>
<td>difference⁰</td>
<td></td>
</tr>
<tr>
<td>no excluded middle, prior to identity</td>
<td>trace</td>
<td>difference⁰</td>
<td></td>
</tr>
<tr>
<td>all things embedded in the social fabric</td>
<td>trace</td>
<td>difference⁰</td>
<td></td>
</tr>
<tr>
<td>natural variety</td>
<td>trace</td>
<td>difference⁰</td>
<td></td>
</tr>
<tr>
<td>unordered, unarranged</td>
<td>trace</td>
<td>difference⁰</td>
<td></td>
</tr>
<tr>
<td>natural complexes</td>
<td>trace</td>
<td>difference⁰</td>
<td></td>
</tr>
<tr>
<td>unhewn materials</td>
<td>trace</td>
<td>difference⁰</td>
<td></td>
</tr>
<tr>
<td>resources</td>
<td>trace</td>
<td>difference⁰</td>
<td></td>
</tr>
<tr>
<td>alterity</td>
<td>trace</td>
<td>difference⁰</td>
<td></td>
</tr>
<tr>
<td>unexcluded</td>
<td>trace</td>
<td>difference⁰</td>
<td></td>
</tr>
<tr>
<td>discovered</td>
<td>trace</td>
<td>difference⁰</td>
<td></td>
</tr>
<tr>
<td>unadulterated</td>
<td>trace</td>
<td>difference⁰</td>
<td></td>
</tr>
<tr>
<td>variety</td>
<td>trace</td>
<td>difference⁰</td>
<td></td>
</tr>
<tr>
<td>deep ecology</td>
<td>trace</td>
<td>difference⁰</td>
<td></td>
</tr>
<tr>
<td>biology, are mostly classificatory and</td>
<td>trace</td>
<td>difference⁰</td>
<td></td>
</tr>
<tr>
<td>observational. Still others, like social</td>
<td>trace</td>
<td>difference⁰</td>
<td></td>
</tr>
<tr>
<td>sciences, have a very difficult time</td>
<td>trace</td>
<td>difference⁰</td>
<td></td>
</tr>
<tr>
<td>connecting theories to observations. In</td>
<td>trace</td>
<td>difference⁰</td>
<td></td>
</tr>
<tr>
<td>fact, it seems that as more and more</td>
<td>trace</td>
<td>difference⁰</td>
<td></td>
</tr>
<tr>
<td>complex dynamical systems are studied,</td>
<td>trace</td>
<td>difference⁰</td>
<td></td>
</tr>
<tr>
<td>it is more and more difficult to</td>
<td>trace</td>
<td>difference⁰</td>
<td></td>
</tr>
<tr>
<td>construct adequate theories and to test</td>
<td>trace</td>
<td>difference⁰</td>
<td></td>
</tr>
<tr>
<td>them in ways that are convincing.</td>
<td>trace</td>
<td>difference⁰</td>
<td></td>
</tr>
<tr>
<td>Software engineering is a very interesting case. In</td>
<td>trace</td>
<td>difference⁰</td>
<td></td>
</tr>
<tr>
<td>software engineering we construct a</td>
<td>trace</td>
<td>difference⁰</td>
<td></td>
</tr>
<tr>
<td>theory, which like all theories can be</td>
<td>trace</td>
<td>difference⁰</td>
<td></td>
</tr>
<tr>
<td>described in various ways, but it cannot</td>
<td>trace</td>
<td>difference⁰</td>
<td></td>
</tr>
<tr>
<td>be completely captured by its</td>
<td>trace</td>
<td>difference⁰</td>
<td></td>
</tr>
<tr>
<td>representations. From this theory, one</td>
<td>trace</td>
<td>difference⁰</td>
<td></td>
</tr>
<tr>
<td>representation is constructed called the</td>
<td>trace</td>
<td>difference⁰</td>
<td></td>
</tr>
<tr>
<td>“code” that runs on a computing machine.</td>
<td>trace</td>
<td>difference⁰</td>
<td></td>
</tr>
<tr>
<td>We have hypotheses, which are tested</td>
<td>trace</td>
<td>difference⁰</td>
<td></td>
</tr>
<tr>
<td>concerning how this representation</td>
<td>trace</td>
<td>difference⁰</td>
<td></td>
</tr>
<tr>
<td>should function. These hypotheses are</td>
<td>trace</td>
<td>difference⁰</td>
<td></td>
</tr>
<tr>
<td>either borne out or</td>
<td>trace</td>
<td>difference⁰</td>
<td></td>
</tr>
</tbody>
</table>

TABLE 1

and observations. The closer this relation between theory and observation, the more a discipline comes to exemplifying the inner structure of science. Scientists classically produce theories of how nature works. From these theories, they develop hypotheses which are then tested. The results of these tests cause changes in the theory which lead to new questions and new theories. As we have seen in the history of physics, this leads us to very interesting views of the world that can be totally counter to our intuition and received authoritative explanations. Different sciences have different traditional relations between theory and observation which cause them to be graded in terms of how “scientific” the discipline appears. Some disciplines are purely formal, like mathematics and logic, with little room for experimentation. Others, like

Wild Software Meta-systems
refuted by the tests. Thus, in software engineering there is a close, direct relation between the production of the code representation of the theory and the hypotheses of how this representation should function which is held up against actual performance in test. Testing causes us to alter our code representation of the theory, to a greater or lesser degree, in order to bring the hypothesis into sync with the results of testing. Sometimes we are forced to change the theory upon which the code representation is based. This is a fundamental change called a redesign. This close, immediate relation between software theory and test results, which is so important in software engineering, is in some ways even more direct than the same connection that appears in physics. We are not dealing with cosmic questions of how the universe works, but the practical question of how a piece of software performs its intended function. But the recognition that the piece of software is a good or bad representation of a theory, allows us to see here a more general phenomena than might be expected.

As Persig notes in *Zen & the Art of Motorcycle Maintenance*, all machines are embodied theories. Machines have been developed into a high art by our civilization. We have mechanical machines, electronic machines, and are even developing light-based machines. However, software machines have a special and unusual place in the scheme of things. Our appreciation of the peculiarity of software machines, as opposed to all other types of hardware machines, is crucial to the founding of the software engineering discipline. Software machines are written. They cannot function without their hardware computational platforms, whether mechanical, electronic or light based. When software is written, it is a static representation which, when executed, becomes dynamic and does things in the world. The relation between code and test is mediated by the relation between the static representation and its dynamic realization executed on hardware. The transition between the written representation and the execution of instruction is a fundamental transformation. We never really know what will happen when we cross that boundary. We create the code representation which we hope is fully understood. Then we test that representation within the dynamic environment. What happens is always slightly different than what was expected. Closing the gap between the code representation and its dynamic effects is a crucial phase of software engineering. The written nature of software allows us to, relatively easily, change it and to narrow the gap between hypothesis and test results. Once a change is made and the software works, it does not change back so that it no longer works unless there is a random coding error. Perfected aspects of software remain perfect until the world changes or we want something else. Software, once it works, has minuscule reproduction costs in contrast to hardware production costs. Software does not suffer from the effects of entropy, except in terms of its relation to its media. These properties are very different from normal machines which deteriorate through the action of entropy, and have material costs associated with production, which mixes different kinds of materials together in very precise ways to achieve the desired results.

The fact that software is written is a key to understanding why it functions as it does. It has recently been realized by Western philosophers that writing is inherently different from speaking. Almost all our models of reality are based upon vision and speech. However, writing has always served as the shadowy bearer of our civilization. It has always been assumed that it worked the same way as speech, but was merely frozen in a different form. The French philosopher Derrida has recently pointed out that, in fact, writing functions very differently from speech, and our society has constantly denied this difference. Writing is what makes continuous civilization possible, yet it has, from the time of Plato, been denied and denigrated in relation to the preeminent place of speech. Writing lacks the inherent continuity of speech. Writing, in fact, falls apart into separate units almost immediately upon being

---

27. (NY: Bantam, 1974)
written down, and has an opacity which speech lacks. Writing is hard to produce, and has a more formal dimension because it will be scrutinized over and over. Speech vanishes almost as it is spoken (unless it is tape recorded). But the key difference between speech and writing is what Derrida calls “differance”. Writing is constantly differing/deferring. Writing refers constantly beyond itself, and differs from itself while deferring the resolution of meaning. Derrida illustrates this with the example of a book which has a preface. The meaning of the book may be changed entirely by the addition of a preface written after the book is completed. Thus, writing in its spatial organization allows the juxtaposition of passages written at different times in any order. The arrangement of these passages can completely alter the meaning of what was written earlier, but presented later, or vice-versa. This is unlike speech which has an inherent relation between speech acts and temporal ordering. Speech acts always appear in the order of performance. In writing, meaning is deferred until the whole has been seen. In speech, meaning is more immediate and localized to the utterance. Also, writing is a heterogeneous plenum of differences. Any text appears as a string of signs whose differences are the crucial feature. Speech, on the other hand, creates an illusory continuity which is its strongest feature. Speech organizes the stream of differences and causes the hearer to apprehend one speech performance at a time in a series determined by the speaker, not determined by the listener. Speech is a focusing narration that creates a series of figure/ground relationships in a specific order. In speech the speaker is in control instead of the listener. A book, on the other hand, can be opened anywhere. There is no dynamic focusing mechanism, other than the reader’s own attention and sense of what is relevant, that forces the reader to apprehend the presented figures in a particular order. There is, instead, just the suggested order linear text itself.

This linearization of the text is exactly what is challenged by grammatology, that is the science of traces. In the Phaedrus, Socrates challenges the speech of Lysias on the grounds that it has no beginning, that each line may appear in any order. The example of the plaque commemorating the death of Midas that also appears in the dialogue is precisely the same, a text without inherent order. When we produce writings, we attempt to impose form on them using book, section, chapter, paragraph, sentence in order to subjugate the text to our will and make it accessible. However, inherently each line of text is independent and can be moved to any place within the full range of the collection of the text itself. This ability of the text to become a pure plenum of difference without externally imposed form is the basis for the major difference between speech, which naturally organizes itself into gestalts which linguistics studies and thus differentiates itself from text that must be artificially organized by the writer. Text lends itself much more to a multidimensional writing and reading which today is known as hypertext. There is no beginning and no end--merely a labyrinth of differences. Derrida’s grammatology is aimed at understanding these differences; and, in fact, it is possible to locate Grammatology in a hierarchy of differences that parallels what will be called the meta-levels of Being.

Table 1 attempts to locate difference in a series of stages that follows the articulation of the meta-levels of Being by describing different meta-levels of difference. What is clear from the table is that difference level zero does not occur in a vacuum. There are levels of articulation that form the context for that lowest level of difference that produces the context within which that possibility arises. The ultimate context is pure alterity from which all man’s production takes its resources. There is a natural variety, which we discover ourselves situated within, which is unordered and not arranged by us. That primal condition is full of unsuppressed, discovered, unadulterated natural variety. The human being may begin by rearranging and ordering this natural variety without suppressing it. This brings about a rough hewn world much like the hunter gatherers must have lived within before the advent of the earliest agrarian civilizations. In this environment it is the sameness of things and not their differences that is most important. Things are not yet abstractable and perceptible as beings; there is, instead, a myriad of textures which itself is all the Same in the sense of belonging together. By selecting some of the variety, and ordering and arranging it, the unhewn becomes rough hewn. It is difficult for us to imagine this world which is neither alien nor shot through with differences. It is the archaic proto-world of ambiguity, indeterminateness, and flux in which everything is swimming in the soup of everything else merged and blending together. The word “thing” in Old English was a social event. The things within the sameness are part of the social fabric. That fabric ties the archaic proto-world together by a mutual recognition. What is important about the texture of this sameness is that it must be suppressed for difference to arise.

---


29. This ontological model will be fully developed later in the paper.
Difference arises out of identity. Without identity difference cannot appear. Before identity, which is the destruction of the sameness by making things too much the same, there was an apprehension of the world that did not know differences but only variations within the same. Alterity and anti-difference become known in one moment. Alterity, that other, like Grendel and his mother in the Beowulf saga, which may not be made the Same, which can never be identical, arrises at the same time as the identical, that which suppresses all differences and thereby makes difference visible as difference, rather than as variations in the sameness. Anti-difference created the surface, stage or tablet on which differences may be seen. It produces a plenum of pure difference which is indistinguishable from absolute homogeneity. This tablet becomes the writing surface. It is a cleared, cleaned, leveled surface where all the variations of sameness have been converted into differences that make no difference, a nihilistic landscape which suppresses all original and genuine variety. It is important to realize that without identity, difference would not arise. Difference is a counterpoint to identity; it is the counterbalancing overcompensation to the “clear cutting” that destroys the sameness, producing indifference and an indefinite extent of no-thing out of which beings might appear as abstractions of things. Upon the platform of anti-difference the meta-levels of difference are erected. Sameness has been completely suppressed, along with alterity. Alterity appears reflected in the shattered spider web of differences of self from itself. But true alterity has been destroyed along with sameness. Anti-difference erects the enclosure in which difference may appear against the background of a totalitarian imposed identity.

Software has exactly this kind of basis in anti-difference as well. In software there is a pure plenum of bits which are all identical except for location. For software everything beyond this pure plenum of indifferent differences is an alterity. Software can only exist where the plenum of pure difference, based on identity, has been constituted. The sameness of the human world is reduced to this tablet of arrayed memory locations. To software the sameness and the alterity are identical. The hardware within which the array of memory locations exist, and the processor, are already a structural system which has been precisely produced to create the plane of anti-difference which exists as identical cpu cycles and identical memory locations. The hardware has abstracted the entire field of differences, reducing all similarities and variations to a minimum and defining everything outside the hardware as completely alien to the system. Within the system the samenesses that would prevent the discreteness of the differences between the identical space and time places have been isolated and reduced with the same vigor as the alien-ness of dust particles that might blow in from the outside environment beyond the boundaries of the system. Due to these rigid boundaries, internally and externally, the computer system hardware becomes a very fragile kind of a system which must function perfectly for software to work within it.

Upon the pure plenum of identical differences, a whole series of differences in a cascade of meta-levels is formed. The first level is the pure content, which in the case of software is the momentarily magnetized or energized bit. It is these on-bits that form the content which is formed into words or numbers. This formation takes place through the creation of codes. The codes are at a higher meta-level than the forms because it is through the codes that the forms are transformed by the changing of the contents of the forms. So within the field of pure difference there is a distinction between off and on that is the minimal possible variety upon which the whole edifice of software is built. When the bits are randomly set, all on or all off, then the status of the bits are pure content. When the bits are set to particular patterns, this is the production of a distinction which allows forms to appear. These forms appear as particular patterns of on and off bits which are described and transformed according to a code. The code is at the meta level of the sign that allows a difference that makes a difference to be seen. We see forms directly on the screens of the computer. They are what the software presents to the human being in the form of a presentation. But behind these forms mediating between the pure content of ones and zeros, is the signs and codes that are actually manipulated by the processor as directed by the software. We have described three meta-levels of difference: the variation between on and off which is level zero difference; the distinction, or thresholds, which are based on the variation of the variations that produce the forms of letters on the screen that the user sees which is level one difference; and the differences that make a difference that produces varieties and kinds as codes through the articulation of the semiotic level of the software which is level two difference.

There is another level of difference beyond the sign, upon

---

which it is based, called the trace. Derrida speaks of this level using the terms discreteness, differance and reserve. Within the software system all the codes, as they are animated, produce a myriad of discontinuities which are dynamically changing. What the codes represent are discrete patterns with limits. Those limits, like the limits to an iterating loop, interact with each other in inexplicable ways to produce a myriad of effects within the software system. The articulation of all those interactions of limits creates a pattern of non-manifestation which stands in reserve, i.e. is held back and never is seen. Derrida calls this manifesting of what is absent (as absence) arche-writing, as opposed to the active writing or speaking which appears on the surface of the system. It is seen in stress testing as the limits of the software system are discovered. Sometimes a system will just stop at a certain point because some constant has been hardcoded, or some resource or design limitation has been reached. Working through the system which is only there to present things is a non-manifestation which patterns the whole. The traces of this non-manifestation, the unconscious of the software system, the limits placed upon the system by the designer often because he did not think of the particular combination of events that occurred during the design, are constantly being manifested as an emphasized absence, underlined and made “present” by its very lack.

One good way to envisage the trace is to think of the impression on a piece of paper below the one being written upon which can be rubbed with color to see the traces within the tablet of prior writing. Traces are a relation between the substance of the tablet and the act of writing. We can see the traces as the deepest deep-structure of the system, where the natural breaks occur for carving up the problem. It is the frontier of structuralism where the deep-structures deal with non-presence as well as understanding the patterns of presence. The traces are the anomalies in the structures. Traces are the inner relation between the off/on-ness and the physical substrate. What particular bits are off or on are arbitrary within limits. It is within the play that is built into the software system that makes it flexible, that these limitations arise through the actualization of particular unforeseen patterns. Signs operate on probabilities, whereas traces invoke
possibilities within the combinatorial labyrinths of the ultimately untestable software system. For any software system of any size testing all possible conditions is impossible given the limits of computational time. When the limits are breached, catastrophes occur. Thus the trace is the inner relation between the bit and its state which is a trace of the unconscious of the software system itself. Derrida also talks of spacing which concretely is the part of the design that never appears. That non-appearing non-computable untestable aspect of every design leaves its trace within manifestation.

Derrida speaks of the hinge which is at once both join and break. They are an absence that is paradoxically always already present as a lack. They are perturbations and holes in the field that holds the inscription or representation. For software theories they are the impossibility of seeing the whole of the formal-structural system at one time. There are holes, or warpages, in the topology of the field within which software design methods operate. These holes, or warps, can only be seen obliquely by their effects on the representations of software theories. We see the traces by their distorting, decentering, and displacing effects. Within software the hinge is that aspect of the software which allows it to fold back on itself without breaking in order to continue running. Many design elements of software have this property. For instance, memory management systems allow the computer to keep reusing memory instead of meeting the limit of system memory. Sometimes there may be what are called memory leaks within the software system that cause it to meet those limits that the memory management system is suppose to avoid. Here the correct return of memory to the pool for reuse is a folding of the software system back on itself so it can avoid a hardware limit. The fold in the software produces a break, but allows it to join across that break in order to continue to create the illusory continuity which would shatter if the limit were encountered and could not be avoided.

Derrida also talks of spacing. Spacing temporalizes space and spatializes time. Spacing occurs in software in the memory management system; for instance, in the obtaining of a chunk of memory for a certain time. Spacings are the internal juggling of the limits within the software system which allows it to avoid those limits and still manifest what is suppose to be seen. Software continually works around its own limits as well as those provided by the hardware environment. The active work of spacing allows it to recognize those limits and to produce realms of activity within those limits which continually fold back on itself through the hinges to allow the processing to continue. With iterative loops the software exists as wheels within wheels which continually check themselves so as not to get off track. This constant checking of itself produces the spacing by which the software gives itself room to maneuver and to continue executing. Ultimately Derrida calls this continual spacing of itself Differance, the work of differing and deferring, because in the spacing the software is continually avoiding built-in limits. It is the interaction of these limits which cause unintended results that are different from what should occur. The appearance of those unintended results differing from the requirements and differed until the right conditions manifest so that eventually failure occurs and the hidden aspects of the design, always there but never manifested, become suddenly known. This difference between what should have occurred and what did occur in the failure is a differance, that is a third order meta-difference.

Derrida writes under erasure in order to show that the actual action of the traces within writing. When we erase what was written before is usually seen through what is overwritten. There is a discordance between what was there before and what is there now. Even if we cannot see what was erased the knowledge that erasure has occurred calls the text into question. Heidegger was the first to use this tactic to move from the conceptual realm of Process Being to the realm of Hyper Being. Derrida took over this ruse as a means of showing the insubstantial nature of the text and the subject that produces the text. For Derrida the text has a special nature where the rules for what is inside and outside are violated. He indicates this with the statement “The inside is the outside.” The crossed-out is refers to the fact that between speech and writing it is difficult to say what is inside and what is outside. Naively we think along with the linguists that thought is covered by speech which is in turn covered by writing. But as Derrida shows we can also think of speech as a kind of writing so that the normal relations are inverted. Writing under erasure is a way to make the special point that in the arena of the trace the natural relations between things are

31. La Bresure; See Of Grammatology, pages 65-73.  
32. See Of Grammatology, pages 68-69.  
33. Of Grammatology, page 44.
likely to be distorted and some would say perverted. For instance we normally think of software as coming after and as a supplement to the hardware. But in reference to the genetic code we see that in relation to our organisms software comes first before we even dream of the possibility of hardware. Software is one of those dangerous supplements that Derrida speaks of that displaces what they are added to so that the relations between supplement and original become paradoxical. Software itself in its memory management as well as many other aspects continually uses the possibility of writing under erasure as a positive feature. It reuses memory by writing over what was there before as a general feature of its functioning. In certain kinds of software systems, like genetic algorithms, this feature of erasure is used as a positive part of the system. In genetic programming\textsuperscript{34}, which uses genetic algorithms to synthesize and evolve computer programs, the pieces of memory which operate under erasure are programs that are evolving through the combination of other programs. Programs of the last generation are broken apart and their pieces swapped then compared against some fitness measure to see if the program has gotten any better at doing the job it is being evolved for. Here the software \textit{writes} itself. We can see a system like this as erasing each older generation and building the new generation out of the parts of what has been broken up and reconstituted. Erasure is an effect that is normally not used in a positive way in the building of software. But because software can write, read, and erase its own memory location this is definitely a positive feature of software itself.

Above the third order meta-difference there are two higher levels which will be briefly introduced. The next level up is the palimpsest where all the traces are accumulated. Different traces cancel each other out so that a chiasmic effect of reversibility is created. This chiasm is referred to by Merleau-Ponty as “touch touching”\textsuperscript{35} and by Deleuze and Guattari as the inscription of the flesh by the savage social mega-machine\textsuperscript{36}. Here one set of limits might be replaced by another set of limits, and the software system might move from one regime to another. When this occurs, the software is oscillating between chaotically-induced states. This occurs quite often in networks. Here the interferences between limits and spacings is so complex that the system becomes opaque. That opacity is the prelude to transparency. The transparency is the ultimate emptiness of all things. It is due to that emptiness that things may have meaning. In software this is the pure discontinuity between all the bits, whether on or off. If each bit is considered independently, the system vanishes. It is only because we consider the bits to be related to each other that we see a software system at all. Software ultimately disperses as a cloud of electrical or magnetic pulses. But before we see that dispersion, there is the moment when software attains to utter alien-ness or blends to utter sameness. At that moment it is incomprehensible without yet having hit the ultimate limit of impossibility. These higher meta-levels of difference are not important for defining software itself, but only the entities that appear within software. The level of no trace where opacity appears is important for artificial life and intelligence and defines the essence of these kinds of computer programs. The level of emptiness is important for defining the phenomena of virtual reality and cyberspace, another incarnation of what the Japanese Buddhists called the “floating world”.

Derrida speaks in several places in \textit{Of Grammatology} of cybernetics and the relation of the gramme to the programme. He seems to be aware of the possibility of software as a kind of entity that embodies difference, that is structured as a series of folds or hinges in which spacing occurs. He does not pursue the point\textsuperscript{37}. However, it is clear that the programme is dynamic, and the gramme is static, and it is the dynamic gramme that more closely relates to his concept of the trace as a dynamic meta\textsuperscript{3}-difference. So software, as the informing of information, is the real subject of grammatology, the science of traces. Traces are the bit patterns which transform other bit patterns, but beyond that they are the internalized limitations of the hardware and software which never manifest but guide the manifestation of what is presented by the software system. The pro-gramme is the active trace that leaves a trace in memory and the phosphorescent dots of the display screen. The active trace traces itself, folding through itself and keeping distances from itself as it avoids its own limits. Whenever we trace the execution of code within a debugger, we are pursuing this active trace which quickly becomes lost in the incomprehensibility through its interaction with other traces. The entire system of multiple interacting agents across a plenum of distributed computing nodes is just the palimpsest which holds all these non-linear interfering traces. This morass is opaque to human understanding to the extent it is operating fast enough to produce an illusory continuity for us. But beyond this opacity, it is ultimately,

\textsuperscript{34} See Koza, J.R.; Genetic Programming (Cambridge MA: MIT Press; 1992)
\textsuperscript{35} See The Visible and the Invisible; op. cit.
\textsuperscript{36} See Anti-Oedipus; op. cit.
\textsuperscript{37} See Of Grammatology pages 9-10 and 81-87.
like everything else—empty because the traces reduced to a non-related array of bits can be seen to not be a system at all. Seeing the traces in the context of multiple overlapping traces and the tracelessness of pure dispersion, allows us to keep software within context as the meta-level where the traces arise but have not yet cancelled themselves out and disappeared.

This diversion into the esoteric philosophy of Derrida may not be so far afield as it may first seem. Speech is a metaphor for the illusory continuity created by the fast execution of instructions by a micro-processor. In drama a predefined text is spoken in an order determined by the playwright. The written text is executed by the actors in the theater. Thus, the written text from the earliest times was executed on human processors. The writing of the text allowed the playwright to see the text as a whole before it was performed. He could change it to achieve greater effect on the audience. Performance allows the attributes of speech and writing to be combined to heighten the effect of both. At a minimum, the lone reader is always the performer of the text he has chosen to read. Unread books are totally lifeless. It is the introduction of this early form of software, to the dynamic environment of the human processor, that causes interesting things to happen that do not happen in a totally oral culture.

The work of Derrida38 is crucial for understanding the problems confronting software engineering, for which, as Brooks says, there does not seem to be any silver bullet. The monstrosity of software is the presentation as lack, absence or arbitrariness of its nature as trace. Software programs are animated writing. The hardware produces an illusory continuity by executing commands at lightening speed. The illusory continuity experienced in the film theater or on TV now becomes easily modifiable, so that the interaction between the program and the user becomes possible. Yet, the nature of the writing which holds the static representation is very different from the dynamic “speech acts” of the processor executing operations. The gap between static writing and dynamic “speech acts” is a transformation fundamental to the software engineering discipline. Thus, it is important for us to understand the differences, now widely recognized, between speech and writing. These differences in static and dynamic representations, and their interaction, cause many of the problems recognized as crucial in philosophy of science.

For instance, until Karl Popper39 everyone thought the relation between theory, hypothesis, and observation was straightforward. Popper then showed that theories could never be proved, but only disproved. This caused a major crisis within the scientific community. It meant that science could never ultimately know the truth which it had set out to know. It meant that all undisproven theories were possibly valid. It also made more stringent what could be called a proper theory. Theories had to be falsifiable in order to be considered scientific. All parts of theories that were not falsifiable were not theory but suddenly philosophy. This made a lot of physical science suddenly disappear into another discipline. Popper still believed, however, that it was possible to have a logic of scientific discovery which depended on systematic disproving and disqualification of theories. This assumption of Karl Popper was challenged by I. Lakatos40 and P. Feyerabend41, who showed that all methods were only backtracking. How a particular theory was arrived at was arbitrary. Good theories were never achieved historically by applying methods. Systematic application of methods, almost always, resulted in bad theories. Methods were only there to help others get to the same view of things the discoverer had achieved by unknown or idiosyncratic means. Thus, science not only could not know what the truth was, but did not know how to get where it was going in a rigorous and well-disciplined way.

The crowning blow to the classical view of science was Kuhn’s concept that science was based on paradigms—meta-theories—which periodically undergo radical transformations. Why these transformations occur, and when they will occur, is only understandable by hindsight. The most significant study of these radical transformations is G.H. Mead’s The Philosophy of the Present42 in which they are called emergent events43. What we experience is a build-up of accumulated anomalies unexplained by current theories. Elaborate work-arounds are constructed by normal science to explain and avoid these anomalies. Then at some point, a new paradigm coalesces in the mind of one or more individuals who are working at the cutting edge within their discipline. This paradigm shift radically alters our understanding of the field within which all the theories operate. A scientific revolution causes the whole

42. (La Salle IL: Open Court Pub. Co., 1959)
43. See the Ph. D. dissertation of the author titled The Structure Of Theoretical Systems In Relation To Emergence, London School of Economics, University of London, United Kingdom, 1982.
world from the point of view of the discipline in question to change. These revolutions always accent the relation between observations and theories and cause that relation to change fundamentally. This is because our observations lead us to embrace theories which we would not arrive at otherwise. Not only did science not know how to get where it was going systematically, it could at any moment have the rug pulled out from under it. Shockingly we discover that normal science works completely different than the aspect of science that allows progress to be made.

Software engineering theory has many of these same features exhibited by scientific theories. Software defects, which cause observed behavior to differ from hypothesized behavior, cannot be proven to not exist in a given piece of software. Mathematically, NP-completeness theorems and computational limitations provide upper limits to what is achievable in terms of proof of correctness. Proving the verification and validation of a program in relation to requirements, and many other aspects of software engineering, suffer from the same problems. Software methods function similarly to scientific methods. They are good as explanatory tools; but good designs do not come from the slavish use of methods. In fact, the use of methods can be a positive hindrance to the solution of a highly constrained, “wicked” problem. We must remember that good designers rely on experience, and rules of thumb derived from experience, to develop a software theory. How they arrive at the theory is an aspect of their craft which is not under the auspices of method. Method is an auxiliary tool that provides a language, or set of languages, to understand and communicate the problem and potential solutions to others.

Finally, although it has not been generally recognized yet, it is clear that software theories, and their associated methodical representations, are based on underlying sets of assumptions which may be called paradigms. The standard computational paradigm is that developed by von Neuman and defined by Turing. Examples of nonstandard paradigms are cellular automata, neural nets, massively parallel computers with actors, expert systems. These computational paradigms cause the software theorist to have radically different approaches to designing and coding software. Within the standard software engineering arena, as opposed to computer science where most of these alternative computational paradigms live, there are also “software centered” methodological paradigms that are developing concerning how a network of von Neuman machines should be programmed. These “software centered” software engineering paradigms are still, at this stage, mostly unconscious, i.e. unarticulated. Although software engineering is evolving rapidly with the addition of networks, processors are still thought of individually and programmed as isolated units. The software centered paradigms are based on those structured programming constructs developed in computer science. The first major indication that other software paradigms are possible is the rise of object-oriented programming to challenge functional programming. We are now dealing with the trade-offs and attempting to understand the differences between these two approaches. However, this change is probably a small tremor compared to the paradigm shifts to come.

These strange aspects of software engineering which appear so similar to scientific theorizing are intimately connected to software’s relation to writing. It is only possible for us to have theories because of written culture.

Written culture allows us to formulate hypotheses and write them down for comparison to observations. Thus, our ideas about the world are represented in a static picture that is inspected after the experiment has been set up and run. Theories exist in a special space made possible within our culture by writing. Theories are, in fact, a special case of the deferred meaning of the text. The written observations appended to the theoretical text and the test descriptions give the theoretical text a different meaning. In software these relations between writing and theorizing are accentuated even more than in standard physical science. The pure difference of the software text is exhibited in its intrinsic unreadableness. The text is a string of commands to a processor that is not present. One must defer finding out what the significance of the text is until it is executed. In writing software texts, we are confronted by the utter alienness of writing in a way that prose texts never achieve. This utter alienness and difficulty of understanding flows from the fact that the meaning of the text is the actions performed upon its commands. These actions are mainly invisible. We read prose for pleasure. Code representations are never read for pleasure. Pleasure only comes when the code is executing properly. The limited visible outcome of the performed text is an extreme form of delayed meaning. The action controlled by the program executing on an automata is a special kind of speech. The transformation across the gap from static to dynamic, from writing to speech, is also a transformation from our programming words to the actions of the slave-other. Our programming words have no intrinsic meaning as the meaning of prose texts have for us. Our programming words are in a language of action whose meaning is only apparent once the radical barrier between static and dynamic has been passed.

We do not think about these underlying structures within which we work every day. But it is precisely these rigid structures which have never been confronted by human beings before, that we must understand if we are to comprehend what software engineering is really about. The fundamental concepts of how formal and structural systems work have been knit together in a unique configuration in the computing environment. These concepts have direct links to some sophisticated philosophical ideas recently developed within our Western scientific tradition. For instance, the preeminent philosopher of our time is Martin Heidegger. His work Being and Time is probably the greatest work in philosophy since Kant’s Critique of Pure Reason.

Heidegger presents in that book a fundamental reorientation of Western philosophy from that given it by Kant. From one point of view this book written in the 1920s could be seen as the fundamental statement of a philosophical basis for computer science. Heidegger’s work has as a central focus the relation between man and technology. In computer science the expression of this relation is extremely intense. Therefore, it is interesting to note that in this book the major concepts developed are isomorphic with the basic model of computer hardware structures. Looked at from the perspective of computer science, definite mappings between computational structures and philosophical concepts may be seen. Space is equivalent to memory. Time is represented by CPU cycles. And the key thing that makes the CPU work is the difference between accumulators and index registers. This difference between types of CPU registers appears, strangely enough, also in another form as a fundamental distinction in Heidegger’s thought. Heidegger tells us that Kant thought there was only one kind of Being, i.e. Pure Presence. He distinguishes between Kant’s kind of Being and a new kind he describes as mixed with Time, dynamic, instead of static. Being is the most general philosophical category signifying the ontological status of “remaining present”. As software engineers, we should be interested in this most abstract of all categories, because the ontological status of our product is questionable, i.e. it is merely a theory. We have serious problems presenting these theories in our representations. The science of Being, i.e. Ontology, helps to explain why we are having these problems that other branches of science have run into before us, though perhaps not in such extreme forms.

“Being” is whatever is verifiably present. Each entity in existence has Being as its most basic attribute. Heidegger makes the point that Being, as an idea is not contained in any individual entity but is a field that embraces all entities. This concept he calls ontological difference--it is the difference between the being of the entity itself and the field of Being in general shared by all entities. The human being is a special kind of entity that projects Being on other things in existence. At least Indo-European cultures traditionally project Being as the basis of the world. It is unclear whether other human cultures do this. This process of projecting Being is called “Dasein” which means “being-there” interpreted as being-in-the-world.

46. (NY: St. Martin’s Press, 1965; Macmillan & Co. 1929)
Projecting Being on things is in Heidegger's view intimately connected with the creation of our world by ourselves. Technology plays a large part in this process of world building which is one aspect of the social construction of reality. Heidegger points out that there are at least two kinds of Being, instead of only the one, the static kind, traditionally identified since Parmenides. The traditional static sort of Being will be identified with a superscript as "Being\(^1\). The other sort of Being, "Being\(^2\), he claims, is hidden but ever present. It is seen when the traditional category of Being\(^1\) is seen in relation to the category of Time. Being\(^1\) mixed with time produces Being\(^2\). A.N. Whitehead's Process Philosophy explores this same realm of ideas. Whenever what is important is not what remains present but the flux of existence itself, then Being\(^2\) must be considered. This is Heraclitus' perspective on existence. The human being relates to existence either from the perspective of Heraclitus or that of Parmenides. These represent age old modalities for humans to relate to things. Until recently, since things have sped up in our civilization, we were mostly interested in what remained the same. Now, however, the flux of existence itself is becoming more important to us in our everyday lives. Thus, each modality is becoming important to us, and we are often having to switch rapidly between these different perceptual modalities.

Heidegger calls the modality associated with the second kind of Being the "Ready-to-hand" as against the modality related to the first kind of Being called "Present-at-hand". He says these are two completely different types of modalities by which human beings, considered as processes, relate to entities within the world. This second kind of Being has a different criteria of truth. Instead of verification as its criteria of truth, Being\(^2\) has as its criteria of truth whether the thing can be re-presented or not. Presentation is a constant process of re-showing, or revealing, that entails showing and hiding. Things that can be re-shown over and over have a different kind of existence than those which cannot. For Heidegger, this second modality of Being\(^2\) is the modality which all technology inhabits. Technology is primarily a mechanism for controlling showing and hiding. Tools are ready-to-hand and disappear from presence as they are being used to "work on" what is present-at-hand. Technology hides itself, so that something else can be shown. What is shown is the center of attention, the thing being verified over and over. Technology allows this center of attention to change by making it possible to replace the figure with a new figure from out of the background. Technology controls the coherence of the Gestalt. Like the Gestalt, it is hidden in the background. It is only when the technology breaks down that Being\(^2\) is brought into focus and becomes a figure.

Maurice Merleau-Ponty, a French philosopher, wrote a companion book to Being and Time called The Phenomenology of Perception. This book showed the roots of the distinction between present-at-hand and ready-to-hand in human psychology as revealed by experiments on brain-damaged people. In that book, Merleau-Ponty shows that "Pointing" and "Grasping" in the psychological make-up of human beings have exactly the same attributes as the abstract philosophical concepts of Heidegger. This is important to us here because of the analogy we have drawn with computer science structures such as accumulators in artificial processors that grasp and

---

50. This distinction between the philosophies of the two Pre-Socratics is a generalization for illustration purposes.
52. M. Heidegger, op. cit.
53. (Evanston, IL: Northwestern U.P., 1966)
Software needs this independent processor to exist. Software seems to be all purely present-at-hand as a pure plenum of text. We like to think of this text as if it were all perfectly observable at one time, as open to an ideal inspection that could encompass the whole. This is, of course, impossible. We can only inspect a small part of the whole at any one time because of human limitations. Software, when compiled, presents the processor sequentially with the operations to perform. When compilation occurs, we no longer are able to inspect the software. Machine code at the binary level is notoriously difficult to comprehend. Human beings cannot easily understand that magnitude of diverse detailed intrinsically meaningless data. Thus, when software is compiled, it becomes even more hidden. The performance of sequential machine level instructions appears as an illusory continuity of actions in time. When the software disappears as presentable text and is compiled, it becomes purely ready-to-hand. When it is doing its work properly, it is not seen itself. Only the images and sounds that it causes are made present-at-hand. Yet, when the software has defects, it becomes the focus of our attention and is rendered present-at-hand itself. It ceases to be able to hide itself and show us something else. We relate to computer programs through our ability to point and grasp. For instance, when writing we grasp the pen which points to what we are writing. What we are saying in prose has our full attention, and we do not even notice the pen in our hand or our movement of it. This ability to do one thing while focusing on something else is what allows us to program computers whether we write code by hand or use a key-board. There is a radical discontinuity between words and action which has been appropriated by the computer technology. This strange relation between words and actions in relation to all processors, human or automata, is what we use every day without thinking twice. The secret of software engineering appears in this close juxtaposition of words and actions, and the differentiation of words into spoken and written.

Another point worth mentioning is that the model upon which the computer is built is the mnemonic techniques developed during the Renaissance. Francis Yates in the Art of Memory55 describes these techniques in their early forms. Essentially, the mnemonic device was a series of

---

54. An in-depth analysis of meta-technology and its difference from just plain technology is contained in the second part of this series of essays.

55. (London: Routledge, Kegan, Paul, 1966)
places like the rooms of a familiar house. In each place was mentally placed a fantastic object associated with and representing a fact that the person wanted to remember. The person would later mentally simulate a walk through the house saying, “what did I put in this room?” Amazingly an individual’s memory would yield up the secret he had hidden in each place as a recollection. A large number of facts could be memorized in this fashion. The structure of computers echoes this mnemonic model. The memory is an array of byte locations. Either the program counter or a moved index register allows the processor to walk through the rooms of the house to remember what was stored there. The things stored in those locations are fantastic objects made up of semiotic codes ordered by syntax. The codes and syntax allow a composite object to be built up of other objects or a parts of other objects. The composite figure placed in the memory location is the first glimmering of structuralism. Forms are made up of contents, which are merely rearranged contents taken from a whole set of similar objects, that are connected in a semiotic system. The mnemonic device is the first form of a showing/hiding mechanism designed to facilitate recollection by the human processor. It was also used for mental simulation. In the computer this mental simulation has been refined and externalized, but the structure is fundamentally the same.

The automation of the mnemonic device through the use of pointing and grasping gave a concrete realization of the structural system as an independent processing machine. The differentiation of formal and structural systems is important for the understanding of the development of computer technology. The best example of a formal system is G. Spencer-Brown’s Laws of Form. A formal system concerns the rules for the generation of forms or figures. Geometry is the classical formal system. From a few axioms a myriad of forms are defined and manipulated. They have formal static relations between each other which can be described by theorems with logical proofs. Formal systems allow verification of things not immediately self evident. Formal systems, such as symbolic logic and most of mathematics, make strict verification possible. They extend the range of what can be verified beyond the immediately present and allow a system of links which make it possible to retrace our steps to a previous position. It is formal systems that make methods possible. A method is the application of a formal system to a problem to produce a representation of the problem. The formal system is independent of the problem space. It is used as a tool for representing and then verifying formal representations. Forms are represented using rules of formal systems to govern their composition and changes from one representation to the next.

A structural system has a different purpose from a formal system. Structural systems arise at the limits of formal systems. Formal systems do not deal with the action of time. (Spencer-Brown stops the development of his formal system at exactly the point where time is introduced.) They express purely spatial or logically static relations between elements and are used for navigating the spaces inhabited by forms. Formal systems reach their limits when time is introduced into the system being verified. Structural systems are specifically designed to handle the case of temporal transformations. Good examples of structural systems are Transformational Grammar developed by N. Chomsky in linguistics, the structural anthropology of C. Levi-Strauss, and J. Monod’s genetic theory of evolution in biology. Also Heidegger’s theory of Hermeneutics in Being and Time evokes the same general conceptual structure. In General Systems Theory the best example of a structural system is the “General Systems Problem Solver” work of G. Klir as seen in his book Architecture of Systems Problem Solving. To formal systems temporal changes appear as

---

<table>
<thead>
<tr>
<th>BEING¹</th>
<th>BEING²</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pure Presence</td>
<td>Process Being</td>
</tr>
<tr>
<td>the presented</td>
<td>the presentation</td>
</tr>
</tbody>
</table>

| Determinate Probabilities |
| Calculus Statistics |

---

FIGURE 8

---


57. Cartesian Linguistics (U. P. of America, 1983)

---

Wild Software Meta-systems
discontinuities within the system space. Structural systems allow the construction of bridges across these discontinuities. The best example of this is how atomic theory is a structural bridge across the discontinuity witnessed at the macro level in chemical reactions. Physics has been very successful at applying structural models to the description of nature at many levels. The key feature of a structural model is that the content of the forms are analyzed and specified by a semiotic system such that any desired form may be composed by a specified content substitution. Structural systems govern the transformation and transduction (change of media as well as content) of forms across temporal discontinuities.

The ASCII code is an excellent example of the kind of coding that is necessary to control the different letter forms. The transformation from one form to another is made by changing a given content bit. By changing regularized content codes, transformations from one form to another are made possible. The fantastic figures planted in the mnemonic landscape (like the house layout) are structural in intent. They are pieces of different forms that are combined arbitrarily to create these fantastic figures. These pieces of content could be rearranged at will to transform one fantastic figure into another. Pointing to the memory location, we grasp its contents. Those contents are a figure made up of visual codes. The fantastic figure indicates the remembered fact by an established association, just as the structural system of software methods indicates the non-representable software theory. The structured form of the figure has a direct relation to the represented fact or idea which is to be recollected. By speeding up our movement through memory, and then executing the instructions found in a series of memory locations which modify forms by manipulating their semiotic infra-structures, and then by displaying the figures, we get a moving picture or a computer program. Within the computer the structural model has been animated and is executed at a speed greater than the eye can catch the discontinuities between formal presentations.

Software engineering mimics these same formal and structural underpinnings in the software we build. Using mathematics, logic, structural programming constructs, boolean operations we construct formal representations. These representations, however, are animated, and we use structural transformations of data to model changes required in the systems outputs. All the structural transformations of formally defined objects rely on coding of forms and the embedding of knowledge within the application. Thus, we are so immersed in formalism and structuralism that they are invisible to us like the air we breathe. We do not see them. They are too close to us. But we need to see the technological infra-structure of these tools and artifacts that engulf us, because something strange is happening while we are too preoccupied with
our own productivity to notice. There are not just two types of Being. In fact, the concept of Being is rapidly fragmenting. M. Merleau-Ponty, J.P. Sartre, M. Heidegger, and others have discovered and described at least two other kinds of Being. There is a good possibility that the type of Being that software primarily exhibits is neither ready-to-hand nor present-at-hand. These are, in fact, the types of Being related to hardware. This means software cannot really be described accurately, either as just something static, or as a merely a process. We need some other way of describing software because it is intrinsically based on a completely different type of Being. Thus, when we use formal and structural systems to build our software, it is possible that we have not realized what software really is all about. We are still just imitating the hardware technological infra-structure in the software systems we now build.

I will attempt to briefly sketch the way I see the relation between the different types of Being that seem to exist and their possible significance for software engineering. We have already described present-at-hand and ready-to-hand. I will continue to develop this theme using examples from mathematics. The type of mathematics related to the present-at-hand is Calculus. This is no surprise since Kant’s philosophical system is intimately related to the physics of Newton and is, in fact, an idealization of the Calculus. This transformation of the calculus into a philosophical system, grounding all formal and structural systems, is a very significant event in Western history. Since this event we have been working out its implications in the technological systems we have developed. Calculus allows the mechanism for constructing artificial continuities to be displayed for the first time. Ideas are forms given this attribute of illusory continuity generated by this mechanism called ideation. The structure of Kant’s philosophy uses this mechanism to ground all formal systems. It makes possible structural systems because the discontinuities between transformations of forms are reduced via step-wise reduction taken to infinity. Once the calculus was thus embedded as the underlying structure for all ideation, a brilliant intellectual ploy, the stage was set for describing the whole of existence in terms of continuous functions that allow the mapping of forms in a very sophisticated manner. Calculus is totally determinate. It, and the illusory continuity of speech that it mimics, is used to present forms continuously in a seamless web of interlocking formal systems. From the point of view of determinism the world a mechanical clockwork that gives a continuous sweep to the hands of the clock. This gives us the impression of the regularization and objectification of time. Besides its seeming mechanical perfection, this deterministic system was hermetically sealed. Existence was locked out.

Strange as it may seem, this perfect world really attempted to, but ultimately could not, last eternally. There was no real relation between the perfect world of pure presentation of ideal forms and the observed world. Statistics then entered the picture. Heidegger’s philosophy may be seen as doing for statistics what Kant did for calculus. The ready-to-hand is the modality of human relation to the world that gives statistics its foothold. It was noticed that identical acts had slightly varying results.
Thus, forms did not move through the world flawlessly as functions suggested. A trajectory could be calculated precisely, but multiple shots would all be slightly different, varying according to a normal distribution. Actualizations of ideal forms are scattered in predictable patterns. Thus, although there were flaws in the relation between the pure world of ideas and the actual observed world, the differences were predictable. All technology is based on its imperfect, but predictable, mapping. The process of showing an ideal form repeatedly encountered resistance from the material world. That resistance reminded us that the material substrate was there. At first this seemed like a minor side track to the program of constructing an ideal world of representations which controlled every aspect of the material world. However, as physics progressed, it found many strange phenomena at the limits of our abilities to grasp nature. These strange phenomena coalesced into quantum mechanics which perfectly exemplified statistical representations but challenge our abilities to make ideal representations that made sense. In fact, we still have no coherent explanation for these phenomena, even though we have found ways of dealing with the existence of these anomalies. There is a gigantic structural rift between the kind of idealizing determinate representations of the world fostered on the macro scale by Einstein that do not connect with the statistically correct but indeterminate quantum world. Ultimately the mechanism that underlies statistical variation in the universe is unknowable in any determinate way. Determinateness is the very essence of the present-at-hand. Determinate means are fully presentable in all its aspects. Thus, the statistically predictable yet intrinsically indeterminate falls outside this kind of truth. The indeterminate has a different kind of truth. Its truth has to do with the process of repeated showing of the same thing—which is exactly what is needed for the act of verification to occur—and what it is that renders indeterminate is ultimately hidden. We see the individual actualizations as figures on a ground. We cannot see the actual process that causes the variation. This is particularly disquieting in the case of the double slit experiment.

Thus, it is proposed here that the two kinds of Being that Heidegger defines are, in fact, real phenomena with the psychological components of pointing and grasping described by Merleau-Ponty; and that these connect...
directly to the two major kinds of mathematics that engineers the world over use every day to make technology and physics happen. The idea that one is switching between different kinds of Being when using these two kinds of math does not have to be strange. Reality has multiple aspects as shown by the fact that truth is different when looked at in these different ways. We already know that reality is multifaceted. The fact that we have developed different kinds of mathematics to deal with its different aspects should not be surprising. What is surprising is that observed nature should carry its adherence to the mathematical rigors suggested by these different kinds of being to such fanatical extremes. The key thing is not to get mixed up and apply the criteria of one kind of truth to the wrong aspect of reality. This, of course, happens all the time, since most of us operate as if there were only present-at-hand beings; the news of the discovery of the ready-to-hand has not filtered down to everyone yet, even though we deal with it every day. This is particularly the case in software engineering where we attempt to construct very large formally defined computer systems without realizing that once past a certain threshold (one screen full of text) the showing of the whole program at once is impossible. Immediately we are dealing with problems of showing and hiding, and are constantly switching back and forth between what is the immediately presented figure and another realm of Being that controls recollection. This other realm of Being gives the system we are dealing with a subtle partially uncontrollable aspect. There is always some part of the system hidden. We cannot make the whole thing visible, no matter what we do, so we need to choose appropriate abstractions to represent the system. This inherently hidden character of a certain aspect of the software system is one of the things that makes the software product a non-representable theory. What is rendered visible is a portion of an abstract representation. Abstracting still hides some things while making visible other things. The larger the system, the more of a problem this becomes, because it soon becomes impossible for one human being to handle, and then teams working in close coordination become necessary. Software graduates into the realm of the social and organizational systems. This is where software engineering becomes necessary. It may be said that software engineering comes into being to handle the new modes of Being encountered in building by large systems.
It is different from computer science because those systems studied by that discipline are normally small and tractable enough that the present-at-hand mode can be stuck to very closely. Software engineering comes into its own when the ready-to-hand mode must be faced directly and structural techniques developed for bridging the gaps caused by extreme showing and hiding problems. The software theory is totally obscured in software engineering, whereas in computer science there seems to be a means of making it at least partially visible.

All of this is relatively sure ground, both ontologically, mathematically and in terms of software concepts. Now we enter a more hazy area were I will attempt to explain my view of how the other kinds of Being enter the software picture. My premise is that, although both ready-to-hand and present-at-hand effect software engineering, we are actually dealing primarily with yet another kind of Being which has not displayed itself fully as yet. These other kinds of Being are not as widely accepted as the previously discussed pair, so we must proceed slowly and carefully, continuing our mathematical analogies.

Each of these kinds of “Being” operate as if they are at a higher meta-level than the last kind of Being. The ready-to-hand is at a meta-level from the point of view of the theory of logical typing, as described most succinctly by I.M. Copi, that was inaugurated by Russell and Whitehead to solve logical paradoxes. We are familiar with meta-levels from physics. For instance, we define stillness, motion, acceleration, and jerkiness as meta-levels of physical movement in relation to each other. The interesting thing is that we cannot “think” what is at the next meta-level, i.e. acceleration (acceleration of acceleration). It is impossible to think any higher concept than acceleration. It is my hypothesis that it is exactly this kind of theoretical structure that is involved with the articulation of the different kinds of Being. There are only four kinds of Being, and we cannot think beyond the level Being. This sets formal limits to the possibilities of the fragmentation in Western ontology. Being is the most general characteristic shared by all entities which is their existence as a being among other beings. This characteristic is fundamentally related to the possibility of the presence of that entity within the field of consciousness of a human being. Unless being is related to presence, it is an empty conceptual construct with no real meaning. Being is identified with the “being”, with a small “b”, that is an attribute of every entity. Being is the present-at-hand formally called Pure Presence. It is what may appear clearly and distinctly, to quote Descartes, within the center of consciousness as a figure or center of focus. Being is conferred upon whatever is the figure in any Gestalt as the center of attention. Being is the ready-to-hand which is known as Process Being because it is a combination of Being and Time. Being entails the presentation of what is present and is conferred upon whatever is the Gestalt of a figure/ground relation. Being will be identified as the “in-hand” known as either “Being” by Heidegger or “Hyper-Being” by Merleau-Ponty. The term “in-hand” is used to describe how tools transform in our hands, and also transform our hands, giving us new and different ways of grasping the world. Being entails the offering of the presentation which occurs because of the differentiation between transcendence and immanence.

Being is conferred upon the relation between what appears in the Gestalt and what is truly unconscious so that it never appears directly in a Gestalt. What never appears is truly immanent, while what appears is transcendent, either as Being or Being. Being makes itself felt through the changes in the patterning of the Gestalt. Over time the patterning of the gestalt as a whole will change, usually discontinuously, so that different relations will exist between the possible forms which may be made figures in relation to the background. Kuhn’s concept of scientific revolutions, which change the whole field in which theories operate, is an example of this phenomenon. In philosophy it has been noticed that the concept of Being has historically changed from time to time in the history of Western philosophy. Heidegger has called this phenomena the “epochs of Being”. Foucault in The Order of Things has identified what he calls “Epistememes” which, in various eras, have conditioned our understandings of the world. All of these examples display how the whole world can be transformed in a way recently portrayed very well by James Burke in The Day The Universe Changed. The true nature of the “cutting edge” of any discipline or genuine nature of “Progress” must be based on Being. Also, Being has been

---

63. See their Principia Mathematica (Cambridge U.P., 1925-27)
64. I posit that what is beyond Being is Emptiness or Void in the sense propounded by the Buddhists were the concept of Emptiness is itself completely empty. So it is, in fact, possible to speculate on what is “beyond” all the meta-levels of Being but that takes outside the Western tradition.

66. The Visible and the Invisible (Evanston IL: Northwestern U.P., 1968)
68. (London: Travistock, 1970)
69. (Boston: Little, Brown, & Co., 1985)
discovered to be the formal cancellation or annihilation of Being\textsuperscript{2} and its antithetical opposite, Nothingness as defined by J.P. Sartre\textsuperscript{70}. E. Levinas\textsuperscript{71} speaks of the human psychological referent of this level of Being as “bearing” related to the other psychological referents of the lower meta-levels identified by Merleau-Ponty as pointing and grasping.

Being\textsuperscript{4} will be called the “out-of-hand” modality named Wild Being by Merleau-Ponty. Being\textsuperscript{4} entails our relation with genuine impossibility, and thus expresses the reciprocality between limitation and freedom in our relation to the world. The patterning of the Gestalt may change periodically due to unconscious factors, but there are limitations to what changes are ultimately possible and what are impossible. The fundamental relation between our freedom and our determination is expressed in the idea of Being\textsuperscript{4}. Being\textsuperscript{4} is conferred upon the set of changes that occur to the Gestalt as it precesses through the deep structural changes caused by unconscious or immanent factors. In a dynamical system this appears as a series of ground states or regimes into which the system settles which alternate from time to time. These ground states represent the limitation of the possible changes in the Gestalt patterning. The progressive bifurcation of non-linear dynamical systems on the way to a degenerate chaotic state defines this series of ground states. The unlimited possibilities open to the dynamical system are structured into a highly ordered subset. In a sense Being\textsuperscript{4} represents a contraction that balances the expansion represented by Being\textsuperscript{3}. The psychological referent of this last level is “encompassing.” When things get completely out of hand, one is overwhelmed and utterly encompassed.

An example of how these four meta-levels of Being work together can be given from physics. In physics there is the idea of virtual particles. These virtual particles manifest as opposites, and then they destroy each other again before the laws of conservation are disturbed. All space-time is thought to be a foam of virtual particles being created and annihilated. Occasionally a virtual particle escapes its annihilation, for instance at the event horizon of a black-hole, and becomes a real particle so that there can be some crossover between the vast potential energy of space-time and normal matter. But this is a rare event. The intriguing thing about the concept of virtual particles is that in this theory all the elements of the fragmentation of the concept of Being observed in modern ontology are present. It is a theory that appears to be meant to display this fragmentation as it operates within the arena of physical science. Briefly, we can say that the two “laves”\textsuperscript{72} (wavicles, i.e. particle/waves) that are produced are antithetical opposites. They each exist as both particle and wave. As waves they are probabilistic, while as particles they display discrete quantal characteristics. The discrete/ non-discrete natures are complementary dual natures of the same entity. Through these natures the intrinsic complementarity of Process Being and Pure Presence is displayed. Because the wave appears with its anti-lave, for example an electron/positron pair, annihilation will occur. The particle/anti-particle relation between the two laves expresses in their nature the possibility of Being\textsuperscript{3} or the cancellation exhibited by Hyper-Being. There are myriad possibilities of particle/anti-particle pairs which may be realized in a given manifestation of a given virtual particle pair in spacetime. These possibilities will all lead to annihilation, and thus uphold the law of conservation. The relation between the myriad possibilities and the law of conservation displays the relation between transcendence and immanence. The freedom of unlimited production of sets of particle/anti-particle pairs contrasts with the hidden immanent constraint that maintains the law of conservation. Finally, which particular pair of laves is actualized at a given point in space-time, out of all the possibilities, is a matter of the conversion of possibility into probability. This displays the chaotic nature of propensity that drives this transformation. In the epiphany of the particles we see the action of Being\textsuperscript{4} or Wild Being. Being\textsuperscript{3} and Being\textsuperscript{4} display a similar intrinsic opposition similar to that of Being\textsuperscript{1} and Being\textsuperscript{2}. All four types of Being work together to make the virtual particles manifestation in Being as beings possible. A similar thing happens at all levels of existence. The four kinds of Being work together whenever the manifestation of anything occurs. In the manifestation of technical beings this process is particularly apparent, because the different kinds of beings are conditioned by the nature of the technological system.

Each kind of Being in this series is a meta-level in relation to the last with a concomitant refinement of meaning. For instance, in the same vein, G. Bateson describes meta-levels for “learning”. We all know what learning is. It is the acquisition of knowledge. Building software entails learning how to fit the design to the requirements constraints in the best way. Learning to learn (learning\textsuperscript{2}) means finding out how to go about learning and getting

---

\textsuperscript{70} See Being and Nothingness (London: Methuen & Co., 1958)

\textsuperscript{71} Otherwise than Being or Beyond Essence (The Hague: Nijhoff, 1981); See also Totality and Infinity (Pittsburgh, PA: Duquesne U.P., 1969)

\textsuperscript{72} See Wilczek, F. & Devine, B.; Longing for the Harmonies (NY: W.W.Norton, 1988)
better at it. It means rendering the process of learning visible and changing it. In software this occurs when new methods are introduced into the development process. Learning to learn to learn (learning$^3$) is more difficult to think about. It means something like changing one's way of learning to learn--adapting it to circumstances and new knowledge about the learning process--getting feedback from previous attempts to learn how to learn and making appropriate changes that show how one has learned to learn learning better. In software this means switching to a new software-centered paradigm. The shift from the currently posed paradigm to another one at the same level of abstraction would be an example of this kind of learning$^3$. One is not just adopting a new technique useful for learning how a software system works or needs to work. Instead, one is shifting one’s fundamental perspectives on how software methods need to be expressed. This is starting to get very difficult to express. Learning to learn to learn to learn (learning$^4$) is just barely comprehensible. It means something like revolutionizing one’s whole worldview and self. A good example in relation to learning in everyday life might be the Zen Buddhist enlightenment. Beyond this fourth meta-level of learning it is impossible to even grasp what meta-level five might be. It is beyond human comprehension. A similar thing is happening with the various ontological categories of the fragmenting concept of Being.

We do not need to go deeply into ontology in order to get the general idea and connect reasonably with our mathematical analogies. Being$^3$ was discovered when Sartre wrote his book Being and Nothingness$^73$ in which he took Heidegger’s notion of Process Being and turned it inside out. Nothingness was the totally antithetical opposite of Heidegger’s concept of the mixture of Being and Time. After much scholarly debate, it was realized that these two concepts could not exist in the same conceptual system, and that they immediately annihilate each other like matter and anti-matter. This cancellation was realized to be another kind of Being, either Being$^2$ as cancelled or Being$^2$ as annihilated. It was thus given its own name and considered an ontological anomaly. Michael Henry in his work The Essence of Manifestation$^74$ showed that the fundamental assumption underlying all previous ontology was that of ontological monism. Ontological monism meant that Being was only defined as Transcendence, and Immanence was not considered to be an aspect of Being. It was clear that the cancellation of Being$^2$ and Nothingness was the cover for pure Immanence which could never be revealed. This was a much stronger form of hiding than appeared in Heidegger’s initial system. In attempting to deal with this new kind of Being, Being$^3$, Merleau-Ponty in The Visible and Invisible$^75$ ran directly into the fourth kind of Being we will discuss. He called this Wild Being because it defined everything that was left after the annihilation of Being$^2$ and Nothingness occurred. He set out to define this final kind of Being in that unfinished book$^76$. Looking at Hyper-Being and Wild Being, there is really a very simple explanation of what these esoteric terms mean. Consider a guitar player or a blind man. Each learns to use a piece of equipment to such a degree of proficiency that the equipment becomes part of their being. This expansion of the being of the person to encompass another piece of equipment has the status of Being$^3$ or Hyper-Being. This expansion is found whenever a new technical or scientific discovery expands our horizons. This is normally called the process of Technology Infusion or Transfer. There is a related contraction of being-in-the-world which is called Wild Being. That contraction is apparent as the inner coherence of the shifts of Gestalt formations over time. Each instrument we learn to play, or each apparatus the blind man adapts to in order to expand his horizons, further defines the being of the human in relation to his world.

Kent Palmer

---

73. (London: Methuen, 1969)
74. (The Hague: Nijhoff, 1973)
75. (Evanston: Northwestern U.P., 1968)
76. This unfinished work has been taken up from a new perspective by Deleuze & Guttari in Anti-Oedipus (1983) and Thousand Plateaus (1987) (both books; Minneapolis: U. Minnesota Press) their series on Capitalism and Schizophrenia. They solved the problem of how to describe cancellation within philosophy by allowing the disciplines of Marxian economics and Psychoanalysis to cancel instead of antimonies within philosophy so that philosophy was left free to describe what remains after the cancellation. See Massumi, B., A User’s Guide to Capitalism and Schizophrenia (Cambridge MA: MIT, 1992); Bogue, R., Deleuze and Guttari (NY: Routledge, 1989)
Thus, besides expanding our possibilities, discoveries also increase the specificity of our relation to the world, thus in a certain sense limiting us at the same time. This limitation of our freedom by the realization of possibilities is the core attribute of Being^4.

Let's go immediately to our mathematical metaphors for a more concrete explanation. The kind of mathematics associated with Being^3 is fuzzy sets which is called “possibility theory” and was developed initially by L. A. Zadeh^77. The kind of mathematics associated with Wild Being^4 is Chaos^78, and is called “propensity theory” by Watanabe. These two recently developed kinds of mathematics stand out as examples of completely different ways of approaching reality with their own associated criteria for truth. Simply put, probability theory deals only with actualizations and their after-the-fact distribution. It does not deal with the possibilities that resulted in the realized actualizations. These possibilities are endless, uncontained, and do not add up to one like probabilities. I have a myriad of possible routes home, yet my probable routes can be worked out by collecting actual routes taken. Fuzzy sets allow possibilities to be dealt with and mathematically described. They are analogous to linguistic hedges (very, sort of, etc.), and they are therefore very useful in describing situations that cannot be precisely defined. Yet, the transformation from possibility to probability is still unexplained. The transformation of possibilities into probabilities occurs through the introduction of a propensity, or tendency^79, to realize a certain possibility, converting it into an actualization. The individual actualizations are related by probability distributions. If you tried to convert the same possibility again, a distribution would occur, and the propensities would vary chaotically. The action of probability and chaos cause variations in the realization of possibilities from two independent directions. Chaos causes inward variation in the actualization process with respect to possibilities, whereas probability causes outward variation.

Each of these kinds of mathematics are explicitly linked to automata theory by S. Watanabe in his article “Creative Learning and Propensity Automation”^80. He develops definitions of deterministic, stochastic, and fuzzy automata and adds a definition of propensity automata to these to define the entire set. The propensity automata uses weightings on the states of the system determined by the previous history of the system to determine responses to current states. J. Monod called this a teleonomic filter^81. By using a teleonomic filter, the range of possible system states are successively narrowed. Watanabe calls this learning. This is the phenomena exploited in a more complex manner by neural nets. What Watanabe does not do is relate propensity automata to the mathematics of Chaos^82. Many deterministic systems have been discovered to be chaotic in certain regions. This chaos will cause the system to have internal variation that will effect its teleonomic filter’s performance. This ultimately means that the choice of possibilities is not fully deterministic. This means that in the actualization of any possibility, the propensity to realize a particular possibility cannot ever be known for sure. Propensities weight the possibilities and more or less determine which is more likely to be actualized in a given situation, but the chaotic residue hidden in deterministic systems makes this weighting ultimately non-deterministic. In different trainings the neural net weightings may end up slightly different. This is an extremely important connection which explains the variations in the actualizations of possibilities. It also means that this phenomenon of converting possibilities into probabilities will never be fully understood. Chaotic systems defy any ultimate determinate, stochastic, or fuzzy reduction. Each type of automaton has an important place in our modeling of technical beings.

Each type of automaton is connected intrinsically to a different perspective on existence by being related to a different kind of Being. Since all beings exhibit these modalities in relation to humans, we cannot get a full picture of any system without using all of these different kinds of mathematics. This makes systems modeling an extremely complex business. The primitive state of our development of systems science flows from the lack of recognition of the importance of these ontological categories in our everyday practice of technological arts. We treat everything as if it were possible to make it present-at-hand. Because of this, we run into tremendous problems, especially in fields like software engineering where we are essentially dealing with text on computers which we feel we ought to be able to make totally present because it is merely written symbols. What we are

---

80. IEEE SMC-5 #6 1975
82. Watanabe, Satoshi; "Creative Learning and Propensity Automation" in IEEE Transactions on Systems, Man, and Cybernetics; Volume SMC-5, No. 6, November 1975, pp. 603-609
ignoring are our own human limitations. As human beings, we relate to the world through the different kinds of Being mentioned above. It is our basic limitation; operating as we do within the auspices of the worldview defined by Kant, and elaborated by all others since him, as the dominant culture. This culture has been very powerful in the subjugation of nature, but we are beginning to reach the limits in the subjugation of nature because the nature we are subjugating is ourselves. The structure of our relation to the world is slowly becoming visible, and this structure is effecting the very ambitious technical projects that have been undertaken. Parnas’ doubts about the feasibility of fielding Star Wars sized systems of embedded software are well founded. We must recognize the importance of the fragments of Being for our software engineering endeavors because they tell us about our own intrinsic limits which will not be solved by any device or technique. This is because these fragments of Being tell us about our relation to technology itself. Technology is a “controlling nature” subdued and made part of our own most being in order to control other parts of nature. This “controlling nature” is forgotten in our project to control everything else. However, we have an essential relation to this “controlling nature” because it is now essentially defining our “human nature”. Human nature is not something separate from technology. Human nature has been overwhelmed by technology as “controlling nature”. This has been pointed out by philosophers who show the inner relation between Nihilism and Technology. In the master/slave dialectic, human beings have become the slaves to “controlling nature” where they intended to be masters over “controlled nature”. This is particularly evident in software engineering in which we have such absolute control over individual small fragments of text that make up programs. But in order to write that text and execute it, human beings must conform to a very alien computational environment where text is not prose, but formal computer languages that have no intrinsic meaning. These languages only have extrinsic, syntactically defined, meaning in terms of performed actions. Where the software system is more than can be represented on a screen at one time, elaborate means of recalling different parts of the software must be developed. The software must be described in different levels of abstract representations in order to be comprehended. Ultimately, unless we are very careful, the text of the software can become a morass which is incomprehensible, uncontrollable, alienating, and meaningless. Human beings confront their limits in software engineering in a way which is very extreme as compared to other disciplines in engineering. These limits show up in the modalities of Being which are displayed and which must be recognized in order for software engineering to be a successful enterprise.

A way to make this argument more explicit is to consider the formula:

\[ \text{IDEA} = \text{FORM} + \text{SIGN} + \text{TRACE} + \text{NULL} \]

An idea is a form given illusory continuity. This illusory continuity is achieved by adding to the form a substrata which functions in a way to confer the illusion of persisting through time. There are three layers that each correspond to a different meta-level of Being. For instance, the form of a letter on the screen of the computer which is presented has the modality of present-at-hand, and its Being is that of Pure Presence. Normally this is all we care to know, and we use these characters in everything we do in software production. Yet the forms of the letters are represented by codes which are bit patterns in memory. These ASCII codes are patterns of signs that form a substrate of the forms of the letters. We have a different way of relating to these signs than we do to the forms of the letters. They have a ready-to-hand status, and they are generally invisible. Only occasionally do we have to deal with the hexadecimal notation which rule these coded signs. The science of semiotics describes the use of signs in society in general, and here we are talking about a very specific example. Yet the connection between the hidden underlying diacritical sign system and the visible forms is undeniable. In computer systems both are necessary. One is for the human to perceive, and the other is for the machine to operate on. One is visible and part of the presentation of the computer system to the human being, while the other is not apparent but part of the technological substructure upon which the form depends. The system of binary codes is further dependent on the traces in the computer memory that allow its content to be retained over time. This is the bit level composed of off/on electromagnetic pulses. The trace level has yet a different modality in relation to the human being. This is the modality of the in-hand. We can hardly experience this level directly because the individual bit being set or not is normally not something which is important. When we compile or code, we produce the initial pattern of bit level traces which drives the computer. The “object” or “binary” module that is the result of compilation is called that because it denotes the objectification of our representation as it is transferred.

---

83. See Fandozi, P.R. (Washington, D.C.: U.P. of America, 1982)
into machine readable form. The incomprehensibility of compiled code is the establishment of the fundamental link in software with the strata of traces that exhibits the structure of Being in software.

Error correcting codes allow even errors in bit settings to be ignored by the system. Bits are affected by electromagnetic fields, and can be changed by environmental conditions and media or equipment failures. For instance, when a disk is formatted, bad sectors are collected together and marked to be ignored. The myriad of bits within the computer may, at any instant, be in any pattern of off/on configuration. We do not know what that configuration is and do not care. We are separated from it by many levels of abstraction. Yet, it is the changes in that overall patterning of traces which allow us to see what is presented to us on the computer screen. What we see on the computer screen must make sense. Anyone who sees a computer program garble the screen image gets an immediate image of the dynamism of this myriad of traces. The myriad of traces displays the ultimate set of possible system states which is, in most computers, so high that they could not be all realized within the time the universe has existed. We quickly reach the limits of what is computable. Yet, out of all these myriad possible states, we are only interested in a very limited number, and we want to rigidly control the movement of these from one state to another. Yet, we at the same time want to be protected from arbitrary fluctuations in the traces. Thus, traces represent the myriad possibilities of system states. The signs which we use to program the computer on a level of binary codes restrict these possibilities to a very limited number. We use these sign systems through error correcting codes to protect us from random variations in trace phenomena. Thus, sign systems have an intrinsic ability to deal with probabilities. Finally, what the sign system will display is a determinate figure--this character, not any other--which has meaning in this context. A breakdown at the level of sign will cause the wrong form to appear. A breakdown at the level of trace will cause the sign system to compensate for the errors which can be tolerated only up to a certain threshold.

The final level of substrata underlying the form which confers illusory continuity is that named NULL in the proceeding formula. It could also be called “No Trace”. The null state in a computer system is experienced when the system goes down. If you have not backed up, and you suddenly lose an hour’s work, the feeling of utter unexpected loss is horrifying. Suddenly one experiences the ephemeral nature of the electronic media. All those traces suddenly are null, and no trace is left. In other media such as text written on paper this experience is not nearly as immediate. A fire or flood, or some other macro level disaster, must occur to lose the information. Thus, other media that we are used to are far more resilient than the electronic memory within the RAM of the computer. Thus, computers are a fragile technology as compared with pen and paper. One trades fragility for processing power. The null state shows the intrinsic limits of the possible states of the computer. As vast as it is, there are still the limitations imposed when the plug is pulled. The modality of the null state is out-of-hand. It, by definition, is the point at which we lose control completely. What is left when the null state occurs has to be the starting point for booting the system again. From the point of view of software, the hardware when first turned on, is also in a null state even if it is a random pattern of off/on. This is why initialization is necessary. All on is just as constraining as all off. The initial all on, or random patterning of the memory bits, is the tablet upon which our software writes. The processor (CPU) writes on this tablet of memory locations like a pen on paper, changing each memory location to what its initialized values should be by following the programmed instructions “booted” into a segment of memory. In a computer the interesting aspect is that what controls the processor “pen” is a configuration of the very tablet that is being written on. This paradoxical nature of the software, which makes it possible to write self-modifying programs, has been explored fully in Hofstader’s book Godel Escher and Bach: Eternal Golden Braid. This inherent paradoxicality is a unique feature of software that flows from its ground in Being in software. Initialization gives form to the computer memory by defining the configuration of traces. Those traces are then manipulated in sets as signs by the boolean logic of the sign codes. The manipulation of the signs is controlled to present a stream of characters that are meaningful to the viewer. This stream is actually an illusory continuity created by the speed of screen refresh and the clever contrivance of the programmer in manipulating the stream of actions of the computer at an instruction level. What is seen by the user is a carefully contrived illusion.

What is said here about the ASCII code holds for whatever...
forms we project upon the sign and trace layers of the illusion. At various levels we create abstract forms (objects, databases, files, tools, etc.) whose contents are other forms (buffers, pointers, procedures, variables, data structures, etc.). All that constrains us are the structures inherent in finite mathematics and the structures of algorithms. Through the use of higher and higher level languages, we are able to work the signs and trace systems without dealing directly with them. Formal languages are our intermediary. Thus, formal languages allow us to express forms directly, and the aspects of sign, trace, and null are transformed so that they appear differently at these more abstract levels. For instance, signs appear in algorithms as the indexes of loops controlling iterations or as the indexes of arrays. Traces appear as transitory state variables, and as momentary data configurations, or as the activated threads of tasks and as the current blocks of allocated memory. All systems have transitory configurations of elements which are not under direct control. These transitory states are controlled by the system completely, but their exact configuration is not relevant to the functioning of the system. In fact, software systems are specifically designed to handle these variations such as network traffic patterns. Embedded systems must even deal with degraded modes of operating, and thus deal explicitly with null states of the traces. These null states could occur by a computer going down within a network so that other machines must take over its functions. PM/FL software monitors the health of the software, watching for these glitches in normal operation, and exception handling in ADA is a specific mechanism for handling these situations. Thus, these same strata that can be identified in the different aspects of the lowly ASCII code can be identified throughout the realm of software. The different modes of relating to technology signified by the fragments of Being may be seen on all levels of software technology. It is just a matter of reflecting upon the nature of each modality and noticing the unique way it manifests in each instance. Beyond programming constructs, these modalities are exhibited on the methodological level as well and this is the level in which we are the most interested here. Software engineering really exists because for large embedded software systems these modalities become very pronounced and have profound consequences for our ability to build software.

Having outlined these different kinds of Being, and shown in simple terms how they may be identified as operating at different levels of software in something as simple as the ASCII code, the next point is to show that of these modalities it is the in-hand which is the most important in relation to software. The Being of software is determined by the in-hand modality. Although all the other modalities are operating within the software system, it is the in-hand modality that is crucial to the nature of software. For software the out-of-hand modality is null. The present-at-hand and ready-to-hand modalities are representations of the formal and structural systems and the underlying hardware. It is the in-hand modality which is most strongly represented as bestowing the nature of the software system. Thus, all of what Derrida has to say about traces in his book Of Grammatology86 with respect to writing are directly applicable. Software is written and its nature is determined by the incomprehensible trace that sustains the writing. In the trace is a certain opacity which cannot be captured by either signs or forms. It is in this opacity that the major problems of software find their origin. These problems are just now being felt as we attempt to build large systems and discover that the difficulties uncovered in that project are many times more than might be expected from similar small systems. That opacity is the manifestation of Being3 which is also called Hyper-Being. This is the meta-level on which the distinction between immanence and transcendence occurs in ontology. The point is that the hiding of the essence of manifestation87, i.e. pure immanence, is a much stronger hiding than that of Being2. This means that the opacity of the trace is a very severe meta-technical phenomenon. It can be called meta-technological because it is at the next higher meta-level from Process Being which is where technology has its natural home. The problems of software engineering may be predicted to be very severe because its nature is meta-technical. We are not dealing merely with something that is hidden in order to allow something else to be shown, and which appears when the technical system breaks down. Instead, we are dealing with something that can never be made present. Thus, the work arounds that will make it possible to operate in spite of meta-technical constraints will have to be extreme. This is because, like in quantum mechanics and in relativity theory, there are absolute limits being expressed. These absolute limits are expressed because we have entered the realm of possibilities instead of determinism and probability. We can build any sort of system in software, and there are a myriad of possible ways to do that. The unlimited horizons of possibility contain within

86. (Baltimore: Johns Hopkins U.P., 1974)
them a corollary of the confrontations of absolute limits like those of computation time and NP-completeness. Those absolute limits display an intransigence which is unparalleled by anything offered by lower meta-levels. Forms contain meaning. Signs contain significance, i.e. differences that make a difference. Traces are mute. That muteness offers almost infinite possibilities, but at the same time there is an incomprehensibleness which cannot be escaped which flows from the fact that the muteness hides the essence of manifestation, i.e. pure immanence.

The image of the essence of software is peculiar. It is the image of a singularity constrained by a formal-structural system which has been fragmented by multiple perspectives. Normally we think of formal-structural systems as combining present-at-hand and ready-to-hand modalities in a specific way to produce a technical system. Computer hardware is a good example of this, as has been explained. The formal-structural systems are not in this case fragmented into aspects that can only be seen from different perspectives. On the level of software though, the formal-structural system is repeated in a strange kind of mirroring which causes us to impose structural and formal models on the new realm of software as our means of dominating it. However, at the software level we discover that the nature of the formal-structural systems that we create here is radically transformed by the necessity of different perspectives. The formal-structural system is discovered to be fragmented at the meta-technical level. This fragmentation hides a singularity that is embedded in the formal-structural system. This singularity has the same properties as those described in physics in relation to black holes. It is a point where all the rules fail which can never be seen. The singularity in a black hole is surrounded by the event horizon from which no light escapes. Beyond the event horizon all the laws of physics no longer apply. The singularity within a black hole, and that before the Big Bang, share this characteristic of being beyond the laws of physics. Singularities are truly meta-physical entities which we may by definition, never experience directly. They are images of pure immanence that appear in our physical theories because it is difficult to explain certain things, like what existed before the Big Bang, without their presence. It is interesting to note that Sartre’s philosophy of “nothingness,” which contains a similar structure to the theoretical concept of the singularity, predates its introduction into physics. The theory of “nothingness” cancels with Heidegger’s idea of Process Being to produce the anomaly of Hyper Being. With the introduction of Hyper Being, the possibility of pure immanence having

The singularity at the software level does not appear directly. It appears indirectly as the fragmentation of perspectives on the software theory. This fragmentation effects our apprehension of the formal-structural system represented at the software level. This fragmentation does not occur at the hardware level. The formal-structural system at the hardware level is monolithic. We see hardware as given in the world which adheres to formality of boolean logic, and the structuring of space/time as cycles and memory. The formal-structural system is then applied again to software in the structuring of such artifacts as the ASCII code. At first it appears as if the formal-structural system will be monolithic at this level as well. However, as we begin to enter the realm of software engineering and begin to build large, complex real-time systems, we encounter problems. Slowly it dawns on us that we are dealing with something different from the monolithic formal structural systems we are used to building without difficulty. As chip sizes increase, we apply the same concepts of logic and layout to greater and greater densities without encountering the same problems. Chip layouts are complex and intricate but involve the same principles that were first developed when the first chip was produced. The increasing complexity of hardware does not run into the same problems as the complexity of software. This is why reducing software solutions to hardware implementations can be profitable. Besides speed improvements, there is a conceptual simplification that occurs when we cross the boundary from software back to hardware. The non-monotonicity of the structural system at the software level appears gradually, but fundamentally alters the nature of the structural-system at the software level.

The non-monotonicity of the formal-structural system appears as the necessity of a set of viewpoints. The software formal-structural system has different aspects which can only be seen by certain viewpoints. These viewpoints require that not all of the software formal-structural system may be seen at once. This allows the
differing/deferring at the event horizon that hides advent of pure immanence within the formal-structural system. Software theory is ultimately non-representable because of this fragmentation of partial perspectives. Exactly what these perspectives are, and how they relate to each other, will be treated in the second part of this paper. Briefly though, it can be said that the perspectives are related to the pointing, grasping, space, and time. These are related to the four basic perspectives on software theory which are Agent, Function, Data, and Event. The fundamental supports of the formal-structural system become the basis for the articulation of these perspectives. When these supports that are articulated so that Being\(^1\) and Being\(^2\) become perspectives on the formal-structural system itself, then it becomes possible for Being\(^3\) to appear indirectly. This is really equivalent to turning the formal-structural system inside-out. The formal-structural system is used as a basis for viewing itself. It becomes fragmented in the process, and in that fragmentation that mirrors the fragmentation of Being\(^1\), allows Being\(^2\) to become manifest.

As will be shown in the next section of this paper, once the perspectives are articulated, they become related to each other by method bridges that allow movement between perspectives. These methods, taken together, form the representation of the software formal-structural system. When this representation is analyzed, it has some interesting peculiarities. From this analysis the structural infrastructure (Shell) can be isolated which, with further analysis, reveals the inner coherence (Core) of software theory. This inner coherence of software theory masks the singularity (Kernel) which is the token of the advent of pure immanence. This progressive analysis, which will be carried out in detail in the next section of this paper, shows exactly how pure immanence appears within the software formal-structural system from a theoretical perspective. However, it is difficult for us to imagine how the pure immanence of the meta-technical will effect our ability to design software systems. It is being recognized, though, that in distributed systems individual processors are relativistically related\(^89\). Also, there are many kinds of indeterminacy which can be related to quantum dynamical effects\(^90\). Relativity is a determinate system for the transformation between the perspectives of different actors. The lack of global system time makes this effect occur. Quantum indeterminateness, on the other hand, occurs because the relation of actors to each other may be constantly changing (this is similar to the problems of self-modifying code). The inherent dynamism of the system causes probabilities to escalate until the effect is indeterminate. These effects are beginning to be seen, and are proper to the expression of the first two ontological meta-levels in the software system. The third meta-level effects are not yet obvious. They will probably appear mostly in the design realm. In design of software there are unlimited possibilities. These are radically constrained by the requirements and hardware performance. This causes the occurrence of wicked problems for which there is no good solution. Thus, the natural software problems of indeterminacy and relativity of actors, i.e. the space-time constraints on the system and the non-linear dynamics of the system, when realized on a specific hardware platform and to do a specific job, cause major constraints on the problem space which severely limit the possibilities of system design. Here we have a design effect where the nearly unlimited possibilities that exist on the trace level are suddenly constrained very radically, limiting design possibilities. This intrusion of constraints within design is the first major glimpse of the effects of the meta-technical. Pure immanence has effects similar to those of the unconscious discovered by Freud and elaborated by Jung and others. The effects are seen in secondary alterations of phenomena, i.e. displacements. There will be no direct display of effects. All that can be observed are the displacements which when observed closely, can be seen to be strangely systematic. The phenomena of the constraints on design which are normally very profound, when in fact the nature of the design space is almost unbounded, is just such a strange effect. Design spaces are complex, highly idiosyncratic, multi-dimensionally constrained surfaces. Human beings are expected to deal with these strange problem spaces and get a best fit solution. The nature of the pattern of constraints is highly irregular, and finding a solution at all is sometimes impossible. The play between the seeming freedom of the designer and the irrational constraints imposed that steals that freedom, secretly is exactly an image of the relation between immanence and transcendence. Unlimited possibilities and freedom are signs of transcendence. The constraints imposed that limit this freedom in peculiar, irregular ways are signs of immanence.

We return now to the question of the non-representable nature of software design. Software design is non-representable because of the interference of the essence of

---

88. See the forthcoming book by the author titled: _The Fragmentation of Being and The Path Beyond the Void: Speculations in an Emergent Ontology_. This book explores the foundations of the Western worldview and shows the meta-levels of Being to be an ancient structure in Indo-European culture the meaning of which it attempts to unravel.

89. See Agha, G; _Actors_ (Cambridge MA: MIT Press, 1986)

manipulation, pure immanence, in its essential nature. It is non-representable because there is some aspect of it that can never be made present, and we are constantly working around this aspect of the design, attempting to get at it all, and being continually thwarted. Like a race around a ragged rock, as we move this way to see all of the design, there is some aspect of it that moves in anticipation, making some other part of the design disappear. This shows up as the four perspectives on software design that will be fully explored in the rest of the essays in this series. Ultimately, a language for stating software designs is constructed so that this appearance and disappearance of design element templates may be studied in detail. But Peter Nauer’s insight into the impossibility of making all of the design perfectly present-at-hand must be extended to say that even a process will not manifest the whole of a design. So the design is not ready-to-hand either. There is always some aspect of the design hidden, and this ultimately shows up as defects which cannot be proven not to exist. Software design is constantly struggling with this non-representableness that ultimately shows up as deeply embedded errors that may or may not be found. Our methodologies seek to reveal as many of these errors as possible as far upstream in the production process as possible. But hidden in the software lurks all these possibilities for error which come from the fact that all the design could not be seen at once nor be revealed by any methodological process. These traces of the discontinuities between views taken of the design in its development, which during implementation become smeared out through the code due to delocalization, lead to completely unexpected breakdowns in the software system as it actually interacts with the world. Finding and fixing these problems may be very difficult; doing it early in software design may be impossible.

From the point of view of this new paradigm for software engineering, it is now possible to answer the questions posed at the beginning of this paper. Some of these answers will be presentiments of positions fully explored in the next section of this paper:

o What is a software engineering method?

A software engineering method is a means of partially representing an inherently unrepresentable software design theory from a specific design perspective (i.e. AGENT, DATA, FUNCTION, or EVENT). Methods either purely represent a design perspective, or act as a one-way bridge between design viewpoints. These one-way methods are called minimal when they have only the information necessary to perform their bridging function. Minimal methods may be concatenated to create more robust methods that act as two-way methods.

Xaiping Song, who did his Ph.D. dissertation on the comparison of methodologies, has developed a profile which offers the best definition to date of the components of a methodology.

o What is the relation between software engineering methods?

Software engineering methods are related via the intertransformability between design perspectives. Macro methods described by authorities, such as Ward & Mellor, Hatley & Perbhai, Shaler & Mellor, are essentially programs for applying minimal methods. Normally, they give a sequence of applying methods that moves between the design viewpoints in a certain order. The fundamental orientation of this paradigm is that all minimal methods are necessary tools, and that they need to be applied in different orders on different problems. Minimal methods should be seen as a tool kit for solving design problems, and the inter-transformability of design perspectives should be seen as a map whose viewpoints can be traversed in any order. The point that must be remembered, though, is that each different path between viewpoints will give a fundamentally different perspective on the system under design.

o Is there a minimal necessary set of software engineering methods?

Different applications will emphasize more or less the need of the different design


92. op. cit.
perspectives. For instance, database design will emphasize data-centered methods. For some problems a particular design viewpoint may be hardly needed at all. The design perspectives are meant to act as a reminder as to other possible perspectives on the design problem.

There are twelve bridges between the four software design perspectives. When four minimal methods are added to these twelve which represent each perspective in isolation from the others, this gives us sixteen minimal methods necessary to represent software designs.

Are there criteria for judging the quality and effectiveness of software engineering methods?

The software engineering methods used to design a program have a direct effect on the quality of that software. Ignored perspectives will tend to cause incoherence in the design. Design should move iteratively between the different design methods traversing different paths between the viewpoints by using the minimal methods that connect the viewpoints. The skill of the designer is seen in the ability to pick the best path between the perspectives in order to unfold the constraints operating on the design. However, the methods themselves are only representational tools. The methods will not lead to the best design. The quality of macro methods depends upon their flexibility in moving between design perspectives and the fullness with which they incorporate all the minimal methods.

Are software engineering methods related to the foundations of computer science?

Software methods display an emergent, sui generis, reality which cannot be reduced to computer science theoretical constructs. Software engineering confronts primarily a particular type of Being which computer science does not have to confront directly. Computer science entails the recreation in the software realm of the formal and structural systems related to Being\(^1\) and Being\(^2\). Computer science also attempts to explore alternative computational paradigms. Software engineering primarily deals with only one computational paradigm (networked von Neuman), and is oriented toward phenomena arising from Being\(^3\). Software engineering has a large social and psychological content, and thus is related to the social sciences as well as computer science.

How should software engineering methods be taught?

Software engineering should be taught as a mixture of social and computer sciences. Software process models should form the blueprint of the curriculum. Software methods should be systematically presented in terms of the concept of switching viewpoints. The skills needed in software engineers are those of theory building which should be given precedence over concrete computational structures such as diverse languages, databases, etc. Methods should be emphasized over programming skills. Programming in the large should be emphasized. Software engineering should include instruction in domain related knowledge.

Software design methods should be taught using the framework of all possible minimal methods as the context for teaching any particular software methodology composed of sequenced and elaborated minimal methods. This will allow the student to compare and contrast the different methodologies in order to select the best one for the job at hand.

How do software engineering methods relate to the definition of the discipline of software engineering?

Software engineering is a new type of discipline which will become very important as systems of the Star Wars (SDI) magnitude become commonplace. Methods are crucial
to this discipline because they allow abstraction at various levels of very large systems. Unless we formulate workable sets of methods which have automated support, we will not be able to build very large systems. Software methods are the core of the software engineering discipline and display its emergent aspect over against computer science. Knowledge engineering is also a sui generis discipline which has software engineering as its basis.

There will be many different answers to these questions proposed in the coming years. Most will assume that Software is a merely present-at-hand phenomenon which is easily controlled if we will only exert ourselves enough. Some more sophisticated answers will recognize the inherent blind spot within any structural system built in software. To these paradigm builders the dynamic nature of software will be its most interesting aspect. However, there may be some who recognize the inherent blind spot within any structural system built in software. These few will wonder at the strangeness of software and will attempt to discover the noumenal essence of software. This essence will appear to them as a singularity of pure immanence. They will be software ontologists.

EPILOGUE:

In this paper the ontological basis of software has been described. That basis appears at the third meta-level of Being. We have not described what appears at the fourth meta-level of Being. For those who are curious, it can be said that this level describes the fundamental basis of Artificial Intelligence (AI) techniques. It is interesting that there appears to be nothing in AI equivalent to a software engineering method. This is because AI techniques represent paradoxes in the software layer. All paradoxes are rigorously excluded from software engineering. As E.W. Dijkstra says, software engineering is itself a paradox because it entails the attempt to program when it is impossible. It does this by excluding everything in programming that may cause problems like spaghetti code, self-modifying code, etc. You will notice that the minimal methods which are rigorously defined in the next essay all use formalisms which do not have paradoxical features. There is nothing like self-reflexive reference in any of the minimal methods. All the excluded paradoxicality is pushed into AI's domain where it is used to imitate knowledge and intelligent or living systems. These paradoxes, such as "if statements" used against "if statements" to produce expert systems and all other non-deterministic techniques, are marked by their opacity. They are impossible for humans to understand. Each one is like a cognitive singularity within the field of software methods. Thus, we might say that they are like what Deleuze and Guattari call desiring machines. They are orthogonal protruberances out of the essence of manifestation, or the unconscious, of software. Their mode of being is the out-of-hand, and their psychological referent is encompassing. In short, they have their foundation in what Merleau-Ponty calls Wild Being. If this theory of AI techniques is correct, then it means that they will never be described fully by any method, formal or structural, and that they will remain incomprehensible to human beings. It is ironic that these are the very structures used by cognitive science to model the human mind because they are intrinsically non-understandable in their operation. AI techniques are right on the edge of the unknown; they are the points where things get completely out of control in the software realm.

There is a continuum which is differentiated by the meta-levels of Being. That continuum is best understood using what I call the Geode theory of meaning. A geode is a rock which when cut open, is discovered to be filled with crystals and many times is empty in the center. The geode looks like any other rounded rock on the outside. But instead of being worn to that shape by the action of water in a stream bed, it is formed in a bubble in mud which, over eons, has water flowing through it, leaving mineral deposits that form the crystals. All things that have meaning are empty like the geode. Thus, when we look at the different kinds of Being we see that Pure Presence represents the surface of that geode. At this level things have significance by their diacritical relations with other things. When any thing within the web of associations changes, all the associated things change their meanings. Process Being considers the geode from the point of view of the laying down of the mineral deposits that become the crystalline infrastructure. From the point of view of Process Being the geode is a temporal gestalt that only has significance when you look at the whole life-cycle of the formation of the particular rock. It is a system that has a
structure. That structure is the crystals inside the rock which are a picture of the deep structural relations that have built up over time. Thus, process, as temporal gestalt, is a system with a deep structure. In the deep structure of things, the associations are constrained by a weighting of some relations as more significant than others. At this level significance is intensified as we learn the underlying relations between things which are most important. Hyper-Being exists in terms of the analogy of the geode as the flaws in the crystalline structure. Different minerals interact to cause the crystals of different minerals to interfere with each other within the geode. This can cause the crystalline structure to be very irregular and impure. These impurities are the traces of other minerals within the lattice of the predominant mineral. Likewise, as we explore the deep structures of things, we find anomalies which cause us to call into question the regularity of the deep structures we find. When we see the significance of those anomalies, we have an increase of our apprehension of significance through the contrast between the regularity of the deep structures and the inexplicable exceptions that occur because of the differing and deferring of the essence of manifestation. The pattern of the anomalies give us some feeling of the presence of what cannot be known which interferes with the deep structures of what can be known. Wild Being sees the play of light off the crystals within the geode itself. The crystals are semi-regular multi-colored faceted forms. In the Geode there is never any light. It is always dark, and so those faceted forms never catch the light unless we cut them open. However, there is always that possibility of the light bouncing off their jagged forms from many different angles. This is possible because the geode is normally hollow inside. Cancellation of the antimonies of Hyper-Being is the equivalent to the manifestation of immanence. All the flaws in the crystals, if they are brought together, form a pattern. This would be the attempt to make what can never be seen appear. Since this is impossible, cancellation occurs to prevent the betrayal of immanence. What is left after this occurrence is merely the reflections of the surfaces of the crystals. Ideation vanishes into the flesh of perception turned in on itself without the domination of the ideas. Ultimately, Wild Being is merely the interface with the Void.

From the point of view of the technological system, the surface is the myriad relations between the different components of the technology. These relations are dynamic and produce a temporal gestalt in which the technical system evolves. In the evolution of the technical system, there are deep structural relations that hold throughout the evolution. The meta-technical system of software is what integrates and holds together the components of the formal structural system. The software is ultimately only the traces of ones and zeros within the hardware system. These traces, by their differences, show the fragmentation which reveals the absence that cannot be made present within the traces. At the level of abstraction of software design, this is the different design perspectives; at the code level, this is delocalization. By using structural formal systems, we attempt to isolate these anomalies and separate them from the code, calling them errors. When the possibilities for these errors are themselves studied and exploited, we call these artificial intelligence techniques. They are paradoxes in the software code or methods layers. They have interesting properties all their own. When we delve into them, they are opaque like human error proneness itself. They are the interface with the unknown within the technological system. That system is empty like everything else. It is because of this emptiness that things like the technological system can have meaning. As the Tao Te Ching\(^6\) says, “The wheel, bowl, door, and window are all useful because they are hollow.” The emptiness of things is what allows them to have unending unfolding realms of meaning. Meaning is the infinite limit of significance that lies at the heart of all things including the technological system. That hollowness is like a cornucopia from which meaning pours out into the world, and appears as pure meaning as well as the different kinds of significance related to the four meta-levels of Being that ground the technological system. Each kind of significance is related to a particular kind of truth that is associated with each meta-level. The truth of Pure Presence is verification. The truth of Process Being is manifestation itself. There is a truth in the mere presencing of something whether it can be verified or not. The truth of Hyper Being is the same as the truth of the individual unconscious. It is a truth of the inexplicable coherence of distortions within consciousness. The truth of Wild Being is the same as the truth of the collective unconscious. It is a truth which is related to the species and universal human experience. When those archetypal forces seize us they cannot be denied. Ultimately there is the truth of the emptiness of all things. The Buddhist concept of Emptiness, which is itself empty, was constructed as an antidote for Being in all its kinds. But it is because everything is empty that they can interpenetrate like a hologram, where each part carries a partial patterning of the whole. This is possible because each thing is only the sum of its difference from every

---

\(^6\) This is a paraphrase. Henricks, R.G. *Laotzu Te-Tao Ching* (Bantam 1989); p. 63; Chapter 11.
other thing, and those differences are discontinuities between kinds that are ultimately just voids. To the Buddhist the whole world is an illusion because this network of differences has no foundation and ultimately collapses into itself as the emptiness within the emptiness manifests. Notice that each type of truth associated with a meta-level of Being takes us deeper into the human being. The levels of significance within the technological system externalize these layers which are the different modes with which the human being projects his world.

If we take as an example the computer program that plays chess better than a chess master, we can see all the strata of the different meta-levels of Being within this cultural object. On the surface there would be no executing program without the hardware. The hardware is the formal structural platform of the program. The program itself is at a meta-technical level, pulling together all the hardware resources into a functioning whole. The program will normally use search algorithms to look at the space of all possible future games with the initial conditions of the pieces where they are now. The use of the search algorithms on possible future moves is considered an artificial intelligence technique, especially since it is necessary to constrain the search space by heuristics representing human knowledge. The software program is poised at the meta-level of Hyper Being, while the AI techniques used to approximate human cognition are at the meta-level of Wild Being. But ultimately the Chess program is empty, and because it is empty, it may be seen by human beings to have meaning as they interact with it. The meaning of a game of chess flows out through the illusion that it is playing as a human would. All the layers of significance and their corresponding truths are there in that game. But it only has meaning for the human being who is playing the game with the computer or observing two computers play. The technological system lacks intentionality and therefore cannot tap into the emptiness from which meaning arrives. The imitation of human responses is not the right criteria for judging intelligent or life-like activity.

The Turing test is not the proper measure of the technological system with all its meta-levels of Being. Actually, the technological system is utterly alien and exhibits only an alien intelligence as we combine different AI techniques which are all opaque in themselves as paradoxes. The fact that these alien intelligences can imitate human response is not the important point. The important point is that they can do intelligent things which humans cannot do. Thus, when the human confronts the technological system, it is confronting the alien-ness of technology itself, which is radically alien because it unfolds from out of the essence of humanity itself as a cultural product that is out of control. Utter alien-ness comes from ourselves, not from outer space. It will manifest in the social inner space of Virtual Reality. Cyberspace is the embodiment of the emptiness of the technological system with all its layers of difference and meta-levels of Being. Only humans have the ability to project a world. Projecting a world means tapping into the horizons of meaning beyond the technological system. Those horizons of meaning give a context to the significances of the technological system but that system, itself is allopoietic (other made), not self-organizing. If the humans went away and left these artifacts, they would be without meaning. Without the constant input of attention by the humans who see relevance, the meaning dries up, and the significances within the layers of the technological system lose whatever meaning has been attached to them. Only humans can discover and tap the sources of meaning which lies in the emptiness of all things. Thus, no technological artifact will be able to capture semantics except in a peripheral sense as embedded significances. Semantics are embedded in the human projection of the world. The projection of the world on the emptiness of things is the function of Ideation and expresses all the meta-levels of Being. The world is full of sources of meaning which the human being discovers. The technological system does not project a world. Without the human being, the technological system becomes discarded heaps of junk. Unused software is even more pathetic as it merely appears as random magnetic impulses on some discarded medium. All the embedded significances become static without the life given to them by human attention and valuing.

The technological system is a mirror for Western man. It is not a man, but his opposite. As his opposite, it embodies pure alterity which appears in the technological system as alien intelligences of artificial techniques. That mirror is ultimately groundless. As we stare into the groundlessness, we realize that it is groundless because Being, in all its kinds, is cancelling with Emptiness, which is empty. The confrontation between man and technology is the realization that this cancellation of Being and Emptiness is the essence of Western man himself. Suddenly we realize that it is only what is beyond this ultimate cancellation beyond the cancellation of Antimonic concepts that can be the source of meaning and

97. I am indebted to John Irwin for the suggestion of this example.
our world. So all derivative types of meaning, i.e. significances, within the technological system have no foundations. Semantics will always be the fundamental problem of the technological system because it points back to the human who is projecting the world and finding the sources of meaning within that world. Today we find the technological system to be our greatest source of meaning. But that is ultimately because it is, like everything else within the world, empty. That is to say, it manifests above all its own groundlessness that comes from the fragmentation of Being and the cancellation of the different kinds of being with emptiness. The fragmentation of Being is itself the manifestation of the void within Being itself.

Just as the fragmentation of perspectives on software designs reveals the unconscious, unexpectedly within the technological system, where we thought we were safe from the vagaries of the human self discovered by psychology, so we discover the archetypes of the collective unconscious as artificial intelligence techniques. The artifacts of the allopoiesis of the socio-technological system have not undergone the same critique as literature where the structures of the individual and collective unconscious reveal human foibles. But these adumbrations of the human spirit surely exist in this realm as well. All human cultural products can be expected to demonstrate the same universal structures that sociology and psychology find outside the realm of the technological system. But the technological system takes certain aspects of our world to extremes, so that we see the groundlessness of Being appear in surprising forms as technology strives against its limits. One of those limits is the expression of meaning. Semantics are successively embedded into the products of the socio-technological system as layers of significance, as it strives against the limits of the fact that meaning only pours out of emptiness. The emptiness of the emptiness is the very embodiment of that groundlessness of all things within the world, especially formal structural systems and their meta-technical web that reveals points of paradoxicality. The articulation of the points of paradoxicality and their combination into alien intelligences reaches toward the expression of meaning. But since the technological system isolated from its human sustenance does not have a world nor project a world, but is, in fact, only the remnants of a particular way Westerners project their world, there is no chance for meaning to arise from the technological system itself apart from the sociological and psychological realities of the humans projecting their world through the lens of that empty technological system.

KEYWORDS:


ABSTRACT OF SERIES:

Software Engineering exemplifies many of the major issues in Philosophy of Science. This is because of its close relation between a non-representable software theory that is the product of design and the testing of the software. Software Engineering methods are being developed to make various representations of the software design theory. These methods need to be related to each other by a general paradigm. This paper proposes a deep paradigm motivated by recent developments in Western ontology which explains the generally recognized difficulties developing software. The paradigm situates the underlying field within which methods operate and explains the nature of that field which necessitates the multiple perspectives on the software design theory. Part One deals with new perspectives on the ontological foundations of software engineering based on recent developments in Western philosophy. Part Two deals with the meta-methodology of software engineering by laying out the structural relations between various software architectural methods. Part Three presents an Integral Software Engineering Methodology based on these foundations.

OVERVIEW OF PART ONE:

In this first part a deep fundamental paradigm for the software engineering discipline will be proposed. The development of this paradigm will draw upon insights from modern ontology of various recent Western philosophers which will be treated systematically. A heuristic conceptual model of kinds of Being will be developed to facilitate the understanding of ontology and its relation to software engineering. Connections between the heuristic model and different forms of modern mathematics will be made to make the intellectual transition from a philosophical to an engineering universe of discourse possible. The relation of philosophical concepts which underlie philosophy of science to the software engineering discipline will be the main focus of
this essay.
The foundations of the Software Engineering discipline have yet to be articulated definitively. Like all engineering disciplines, little thought is given to the theoretical foundations of the discipline beyond borrowing from the residue of scientific theory and results. The first part of this essay suggested that this may be inappropriate with regard to the engineering of software, because it may have a different ontological foundation from other kinds of things we are more used to encountering in our scientific and technical endeavors. Because of this, a radical paradigm for understanding software engineering was ventured. Whether or not this paradigm which explains many of the difficulties in creating large software systems is ultimately proven correct, it is still necessary to understand the central features of the software discipline. These core features taken together make up the essence of software practice directed at producing and controlling software entities. The kernel of this set of practices concerns the nature of software design methods. Design methods are taken to be the heart of the software engineering practice. It is through these methods that the software entity comes into being in the software development process through a series of essential transformations. So moving from the nature of the software entity, we must next consider the nature of the software design methods. At this point, no general theory of software design methods exists. What does exist is a plethora of specific methods proposed by experts vying for predominance. What is necessary is a comprehensive examination of the field of methods through a kind of meta-methodological study of software methods. George Klir defines meta-methodology as follows.

In order to manage the complexity involved in the solution process, systems problems can rarely be handled without any simplifying assumptions. However, simplifying assumptions can be introduced in each problem in many different ways. Each set of assumptions reduces, in a particular manner, the range of possible solutions and , at the same time, reduces the complexity of the solution process.

Given a particular systems problem, a set of assumptions regarding its solutions is referred to as a methodological paradigm. When a problem is solved within a particular methodological paradigm, the solution does not contain any features inconsistent with the paradigm.

It is reasonable to view a paradigm which represents a proper subset of assumptions of another paradigm as a generalization of the latter. Given the set of all assumptions which are considered for a problem type, their relation "paradigm A is more general than paradigm B" ... forms a partial ordering among all meaningful paradigms associated with the problem type. ...

Paradigm generalization is a current trend stimulated primarily by the advances in computer technology. Any generalization of a paradigm extends the set of possible solutions to the problem and makes it possible in many cases to reach a better solution. At the same time, however, it usually requires a solution procedure with greater complexity. The study of the relationship between possible methodological paradigms and classes of systems problems is a subject of systems meta-methodology. This is an important new area of research in which little has been accomplished as yet. The central issue of systems meta-methodology is to determine those paradigms, for various classes of problems and the current state of computer technology, which represent the best compromise between the two conflicting criteria -- The quality of the solution and the complexity of the solution procedure. The main difficulty in this investigation is that there are usually many alternative solution procedures which can be developed for a given problem under the same methodological paradigm.

Another issue of systems meta-methodology is the determination and characterization of clusters of systems paradigms that usefully complement each other and may thus be effectively used in parallel for dealing with the same problem.

1. A definition of this discipline may be found in Peter Freeman’s Software Perspectives See Bib#. See also Bib#. 
Together, they may give the investigator much better insight than any one of them could provide alone.

Every mathematical theory that has some meaning in terms of a systems problem-solving framework ... is actually a methodological paradigm. It is associated with a problem type and represents a local frame within which methods can be developed for solving particular problems of this type. One of the roles of systems meta-methodology is to compile relevant mathematical theories and identify their place in the overall problem space. Another of its roles is to propose new meaningful paradigms; the ultimate goal is to characterize and order all possible paradigms for each problem type. Since the recognition of a new paradigm is an impetus for developing a new mathematical theory, comprehensive investigations in systems meta-methodology will undoubtedly be a tremendous stimulus for basic mathematical research of great pragmatic significance. Mathematics is thus a contributor to systems problem solving as well as a beneficiary of the latter.  

Software systems meta-methodology concerns the relation of software design methods for building large real-time software systems to each other. It specifically refers to the definition of the ground that allows different methods to be related to each other and compared. This ground for comparison has not as yet been the subject of study or debate within the software engineering community. This is due to the nascent state of the software methods themselves. There are a plethora of new software methods being proposed by experts in the field. Many of these are mentioned in the bibliography of this paper. Surveys of software methods already exist. Also, a few taxonomies of methods and enabling tools, such as that produced by the Software Engineering Institute (SEI), have been created. The efforts of surveying and cataloging various software methods is very important for the growth of software engineering design knowledge. However, this paper will seek to extend our appreciation of software methods in a completely different direction. Here the emphasis will be upon establishing a software engineering paradigm which will make it possible to define the grounds against which any software method can be understood and compared reasonably to other, both similar and dissimilar, software methods. A key part of the establishment of this paradigm was developed in the first section of this paper. There a foundation was established to approach the definition of the unique aspects of software. Building upon that foundation, this section will attempt an analysis of the grounds for software methods. This analysis will proceed from existing methods, but will focus on the creation of a hypothesis as to what the total field of software methods is like overall. The concentration on the field or ground that supports all possible methods is what distinguishes this study as meta-methodological rather than methodological.

Meta-methodology also concerns the relation of software methods to the rest of science and scientific method in general. In fact, this relation will be our starting point for extending the paradigm laid out in the first section of this essay. Software engineering has a special relation to scientific methods which has already been explored. A corollary of this special relation is the relation between software engineering and general systems theory. When we speak of general systems theory, we will be referring to the work of George Klir as it appears in Architecture of Systems Problem Solving 6 (ASPS) and his numerous articles. Klir has created an excellent abstract representation of what has been called formal-structural systems. In this study, we will take this representation to be definitive of what is meant by formal-structural systems even though many other representations exist in specific fields, like Chomsky’s Transformational Grammar. Klir has done us the service of expressing the essence of all the formal-structural systems in a single concrete representation. He is to be commended on the scope and detail of his generic rendering of what separately appears in many disciplines with various contents. Understanding how formal-structural systems work, regardless of the details of content contributed by the various disciplines, is an arduous task which is much simplified with an abstract model as a point, of departure. This essay will use Klir’s representation as its starting point and any questions as to what is meant by a formal-structural system should be referred to his book which merits serious study.

In fact, Klir’s ASPS has a special significance for software engineers. It should become the central reference for software engineering’s concept of what constitutes a ‘system.’ We produce ‘systems,’ yet it is difficult to get good definitions of what constitutes a ‘system’ and what it means to produce one. Definitions of ‘system’ tend to be vacuous. Like the many fundamental concepts, ‘system’ suffers from a lack of rigor, and most definitions of it are so vague as to be almost meaningless. When we define a

---

2. George Klir in ARCHITECTURE OF SYSTEMS PROBLEM SOLVING pages 18-19 (See Bib#)  
3. For instance Hatley/Pirbahi (Bib#), Neilson/Shumate(Bib#), Ward/Mellor(Bib#), Shaler/Mellor(Bib#), etc.  
4. MCC Survey (See?)  
5. SEI Taxonomy (See?)  
6. See Bib#
‘system,’ it is really its relation to formal and structural underpinnings that make it meaningful. The full articulation of the formal-structural underpinnings of systems gives real meaning to this all encompassing term. Software engineers build ‘software systems’. By knowing what software is, i.e. the manifestation of the singularity of pure immanence in the differing/deferring of automated writing, and what a system is, i.e. the fully articulated formal-structural relation between elements, then it becomes possible to take a deep look at what ‘software systems’ really are. This deep look is the purpose of software systems meta-methodology. Software methods are abstractions of software systems which we can manipulate and know that those manipulations are as good as manipulating the written code itself which may not yet exist. Our software design methods are pictures of the essentials of software systems. The better our picture of those essentials, then the better will we be able to design software systems. Meta-methodology concerns our getting the best possible picture of the essentials of software systems. We already know that in the very term ‘software systems’ there is a clash between the essential non-representability of software theory and the representable nature of formal-structural systems. Formal-structural systems are precisely a means of representing things that cannot be captured by formal systems alone. In fact, anything that involves time must be represented with structural additions to formal systems.

FIGURE 1

Essential View Points on Software Design Theory

---

7. See Bib#. See Also Bib#.
8. Singularity is used in the same sense here as in physics where it is a point within the area covered by a theory where the theory breaks down. See Bib#.
9. Immanance is the opposite of Transcendence. What is Immanent never appears in any form. Similar to the concept of the Unconscious in Psychology only applied to Ontology. See Bib#.
10. See Bib# and Bib#.
Formal-structural systems are a means of making things that cannot be fully made present-at-hand appear as if they could be rendered fully present-at-hand. Formal-structural representations have proved themselves in physics to be very powerful means of representing transforming objects in the world. Thus, in ‘software systems’ there is a confrontation between a powerful form of representation and something that is inherently non-representable. This clash is an interesting phenomenon by itself.

We need to know what formal-structural system are in order to understand their relation to software. Klir gives a good representation with which to work. Yet, beyond that there is a special relation between software and the general systems science that Klir is attempting to found. Software representations are the data for Klir’s general systems science which has empirical as well as theoretical aspects. Abstract representations of systems are represented as computer programs and tested. Those tests are meant to reveal inherent properties of the general systems architectures which have been abstracted from similar concrete images in various disciplines. This brings an interesting point to the foreground. General systems science itself has an intrinsic relation to software. It seeks to be more than an empty abstraction by discovering characteristics of systems by manipulating software representations of systems architectures. So on the empirical side of general systems theory, it too must confront the essential nature of software as effected by the manifestation of pure immanence. While general systems theory seeks to present a purely present-at-hand picture of what constitutes a formal-structural system, it is also dealing with the non-representable nature of software theory. The empirical discoveries of general systems science about systems architectures is in some sense an exploration of the same issues that are driving software engineering. In fact, software engineering has an intrinsic relation to general systems science which is similar to its relation to computer science. From the discipline of computer science it seeks the knowledge about how software works in a small scale on various kinds of hardware. From general systems science it seeks to understand the structural properties of general systems architectures, also on a small scale. The point is that software engineering uses this knowledge to build large-scale software systems. In these large-scale systems, quantitative changes in the dimensions of the systems cause qualitative changes in the nature of the systems. The facets of the software engineering discipline are responses to these unexpected large-scale effects. As general systems science explores larger and larger systems architectures, it becomes involved in building larger and larger software representations, and thus needs software engineering discipline. Also, as software engineering focuses more and more on design issues, it needs to know what general systems theory can tell it about general systems architectures.

The fact is, that software engineering draws knowledge from four disciplines. Besides general systems science and computer science, it also draws from management science and social science. Software engineering should emphasize it roots in social and management sciences more than it does at the present time. Software engineering concerns teams of people working together to produce the software system. It concerns the management of this development process which involves both general systems architecture and computer science knowledge, as well as application specific knowledge. By treating software engineering as if it arises directly from computer science alone, many of the key features of software engineering are distorted. Software engineering is seen as merely ‘programming in the large.’ Instead, software engineering is a mainly social phenomenon of human beings confronting the task of building large-scale software systems. In that task, there is a need for computer science specific knowledge and application specific knowledge. But it is the understanding of the general systems architecture which is key to making the whole system work properly. Beyond the general systems architecture, a large system is merely a large collection of small programs. This is why architectural design and the recently appearing myriad of software design methods are of interest. These methods are of interest not just because they provide the abstractions by which we can structure very large software systems. They are also of intrinsic interest because they provide the means of building very large architectural representations general systems science can experiment. Also, they are methods by which social and business systems can be described so that automated simulations of them can be constructed. They are a means of describing and controlling very large systems and their computer representations. Computer representations are a way of automating and controlling the described systems themselves. Software methods thus take on a key role of connecting social systems, business systems, and computer systems by describing at high levels of abstraction how these systems work in a way which can be automated. This allows a direct connection between the descriptions of systems, the simulation of systems, and the automation of systems, which has never before been possible.
Probably one of the most interesting aspects of software methods are their application to describing the software development process which is a social and managerial system. Traditionally these methods have been used to model the manual systems replaced by automations before modeling the changes that would be made in the system when it is automated. However, the modeling of the software production system itself shows the true versatility.
of these methods. What we are dealing with here is not something special to software engineering, but a general purpose system description language which can be used to describe any system. This description is the first step in the process of automation or simulation of that system. Since any object system could be described using software methods, they are a general purpose tool for understanding systems and mapping between system representations. For this reason, software engineering design methods have the same claim to generality that general systems science has. It is a universal discipline which can be applied to the description and modeling of any large-scale system. It is the means of creating mappings between specific images of systems in individual disciplines and their abstract counterparts. It is also the means of mapping between abstract representations of different kinds of systems. Finally, it is the means of mapping from an abstract representation of a system to its software simulation or architectural equivalent. Software engineering provides the glue that allows general systems science to create meaningful connections between systems and their representations and simulations. Software engineering has a much broader role in science than has been hitherto recognized. It is this broader role that causes it to have a structure that exemplifies many of the problems of scientific theory focused on by philosophy of science. Scientific theories are now reaching the stage where they are becoming complex enough that it is necessary to create computer representations of them. Scientific theories have structures that are merely specific instances of general systems architectures. Thus, scientific theories need to be engineered into specific systems architectures so that scientists can explore their ramifications through simulations and artificial experiments. In order to create these simulations, software engineering methods are necessary to translate from the structure and specific facts of the scientific theory into the software representation of those facts. In this way, software engineering methods will become more and more necessary to scientists as a means of structuring their representations of their theories and the simulations in software performed using those representations. Software engineering is a crucial component of future scientific explorations in many diverse fields. As simulations get larger and larger scientists themselves will start using the language of software methods to design the representations of their theories. Thus, it should not surprise us that many of the strange aspects of scientific theories are manifested in software theories. Software theories are the representations of scientific theories that can be tested in an artificial universe of a simulation. Software theories are the concrete representation of scientific theories in a dynamic form. This dynamic form is related to the nature of automated writing. Automated writing exhibits, as one of its side effects, the differing/deferring of difference which is the advent of pure immanence within the field of the structural system.

This concept of the embedding of a singularity within the structural system which results in the fragmentation of perspectives was elaborated in the first part of this paper. This is a key idea which is not recognized by Klir in ASPS. This is because Klir sees himself as operating at the level of Being and does not explicitly recognize any higher metalevel of Being. Thus, Klir believes his representation of the structural system is purely present-at-hand, even though the purpose of creating a formal-structural system is to handle ready-to-hand phenomena such as discontinuities in the temporal unfolding of formal systems. If Klir’s presentation of the formal-structural system had itself been at the level of the ready-to-hand modality, then it would have been more philosophical in nature. Alan Blum in his book *Theorizing* and John O’Malley in *Sociology of Meaning* have developed these kinds of self-conscious theoretical representations in sociology. They are more elegant because they say what they themselves do. It would be preferable to operate with these kinds of self-conscious representations in our explanation of software methods, but that would cause this text to become even more unreadable than it already is now to most readers. That kind of self-consciousness is intellectually more honest but leads to great difficulties in communication. Therefore, we will accept Klir’s naivete and will practice that same simplified approach that assumes that the representation of the theory can be at a different metalevel than the action of the theory. This is probably a false assumption.

Instead of attempting to make this presentation theoretically self-conscious in the way Blum or O’Malley have, we will posit the idea of the fragmentation of perspectives on the formal-structural system. This is equivalent to turning the formal-structural system inside out as demonstrated in the previous section of this essay. We will engage in working out the implications of this program as related to Klir’s representation of the formal-structural system. This intellectual trick will make it appear that we are still operating in present-at-hand mode in our theorizing, while in effect, we have represented both ready-to-hand and in-hand modalities in the more complex
theory. The ready-to-hand is represented in the fragmentation of the software level of the formal-structural system, while the in-hand is represented by the hypothesized presence of a singularity hidden by the multiple perspectives. The alternative to this formulation is to enter into O’Malley’s universe of discourse which is posed within the ready-to-hand mode, and introduce a single extra theoretical element to represent the singularity. Another alternative would be to construct a theory which was self-consciously operating within the in-hand modality. This would probably take the form of poetic dialectically cancelling aphorisms. Perhaps Frederich Nietzsche or Theodore Adorno were attempting this in their philosophical writings. We will not be so bold.

The software engineering profession is composed of many roles. One of those roles is the software engineering technologist. The software engineering technologist has four basic realms of specialization. These realms concern software engineering practices, processes, methods, and environments. The software engineering technologist is first and foremost a practicing software engineer. But as a software engineer, his focus has been shifted from the

---

**FIGURE 3**

*View Points Combine to Give Minimal Design Methods*
product to the processes of software production that result in the product. This shift in focus is from the present-at-hand software artifact (the ends) to the ready-to-hand means of production. This realm of specialization has been assigned as the responsibility of the Software Engineering Process Group (SEPG) as defined by Watts Humphrey of SEI. The SEI believes by defining processes and then by measuring the outputs of these processes, we will be able to control and improve upon our software development processes. One means of controlling software processes is to proceduralize the entire development cycle. Certain software processes are crucial, such as requirements analysis and specification, or architectural and detailed design. For these crucial processes we may develop software methods to guide us which go beyond rote procedures. These methods provide detailed guidance in how to solve the problems to which a particular process is addressed. Software methods are very important. They provide a language for communication concerning the problem and solution spaces. They provide a means of capturing requirements or design concepts. Examples of some more prominent software methods are as follows:

- **HATLEY / PIRBHAI**
  *Real-time Structured Analysis*

- **WARD / MELLOR**
  *Real-time Structured Analysis & Design*

- **SHALER / MELLOR**
  *Object Oriented Analysis & Recursive Design*

- **SPC / GOMAA ADARTS**
  *Ada Design Approach for Real-time Software*

- **NEILSON / SHUMATE**
  *Virtual Layered Machine /Object Oriented Design for Large Real-time Ada Systems*

- **CONSTANTINE / WASSERMAN**
  *Object-oriented Structured Design*

Because these methods cover crucial software production processes, they represent the best expertise in how to solve the important problems in software development. Different methods may be appropriate for different software projects. The software technologist should be able to act as a consultant concerning the tailoring of methods to the needs of projects and also the selection of particular methods. Beyond this comes the necessity to train staff in the use of adopted methods. It is at this point that software systems meta-methodology becomes important. As it now stands, software methods, in general, are fairly new. Many are being proposed each year. Thus, the software practitioner, whose main concern is building the software at hand, cannot be expected to keep up with the dizzying expansion of the field of software methods. The practitioner wants a tool box of techniques that will work for him. He wants to adopt a method which will facilitate the use of these techniques in a coherent way. The software technologist thus needs some overview of what is available so that he can help the practitioner select the appropriate techniques and methods. This overview can only be achieved by the development of software meta-methodology. Software meta-methodology gives a means of comparing methods with each other, and attempts to give an overview of the general field from which all software methods arise. Without a view of the field from which these methods arise, there is no way to adequately compare and discriminate between these myriad methods. Software meta-methodology is a theoretical discipline with very practical results. By establishing software systems meta-methodology, the software engineering technologist and the practitioner alike can establish the coverage of a particular method and its relation to other methods with different coverages. By coverage is meant how much of the field of possible software methods does a particular method cover.

The final realm of specialization for the software engineering technologist is enabling technology. The software engineering technologist should be part of a Software Engineering Technology Group which is concerned with enabling software processes and methods in a software engineering environment. This involves Computer Aided Software Engineering (CASE) and Computer Aided Cooperative Work (CACW). Implementing software engineering environments automates the tasks of software production. Up until recently, software production was a mostly manual process, except for editing and compilation. Now with many new software tools coming on the market, there is a great diversity of vendor-supported tools. These tools
sometimes have their own unique methods associated with them, and also call for special software processes. Thus, building software engineering environments in the current environment is not trivial. It usually means integrating vendor tools with home grown tools within a single environment.

This particular essay grew out of a very practical situation concerning software methods and practice. Our Software Technology Evolution Project (STEP) had, in 1986, just implemented a simple software engineering environment on SUN workstations using the IDE 20 Software-through-Pictures™ CASE tool and Frame Technologies FrameMaker™ document processor. Upon completion of this project, I began working on a project that was using this environment. Immediately I noticed something was strange in the way our engineers were using the tools we had given them. I was appalled to see that they had introduced their own ad hoc method rather than following that method supported by the tool. At this point, they had little training in methods or in the use of the tool. However, I wondered why they had changed the Hatley/Pirbhai 21 method. As I began studying their modifications to the method, which were still in the range of what was allowed by the graphical editor but were not allowed by the design analysis tools, an interesting picture became clear. All of the changes made by the software engineers were attempts to introduce tasking constructs into their design representations. On further study, it became clear that the popular tools all supported methods that did not express tasking. It took some research to find tools like DURRA by Mario Barbacci, 22 Task Builder™ by Ready Systems, 23 and StateMate™ by iLogics 24 which did support some sort of tasking constructs. Since that time, IDE has introduced OOSD 25 and SPC has introduced ADARTS. 26 Vincent Shen of MCC has also introduced RADDLE and its visual editor VERDI. 27

What is interesting is that since most software methods were introduced to support information system development, the original methods, like Yourdon/DeMarco’s Structured Analysis 28 and Yourdon/Constantine’s Structured Design 29 did not deal with time. Hatley/Pirbhai and Ward/Mellor introduced Real-time Structured Design at about the same time in which state diagrams were used to deal with the timing aspects of real-time systems. It was these methods which were implemented by most of the early vendors producing commercial CASE tools. Yet these tools did not deal with one crucial problem in software engineering of real-time systems. That is the problem of tasking. Research showed that Gomaa had provided the most straightforward solution to this problem, and our shop began using DARTS diagrams produced in the IDE picture editor. Other solutions were too expensive, and this proved to be an excellent interim solution. The realization that software architectural design required more than just temporal ordering being added to the basic Yourdon/DeMarco/Constantine methods, made it necessary to think through the relation between these methods. Paul Ward 30 provided an insight into this relation between software methods when he observed that different methods ordered the same techniques in different sequences, and by doing so, produced very different views of the same system. The best example of this is the difference between the Hatley/Pirbhai and the Shaler/Mellor methods. These use the exact same representation techniques, but use them in different orders.

**HATLEY/PIRBHAI**
1) Dataflow Diagram
2) State Diagram
3) Entity Relation Diagram (optional)

**SHALER/MELLOR**
1) Entity Relation Diagram
2) State Diagram
3) Dataflow Diagram

By applying both of these methods to the same problem, I saw that the kind of system that would be designed using these two different methodological approaches was very different. I felt that this was an interesting phenomenon and wondered what would happen when one added the DARTS tasking structure method of Gomaa to the set of techniques to be used. The question was, where should it be added and what would occur if it were added at different points in both suggested sequences. These ruminations suggested that there was a field of knowledge here which was worthy of exploration in its own right. What was needed was a general approach to the whole field of software methods, a meta-methodological study,
which would give guidance on how to connect techniques into method sequences in order to get full coverage of the software engineering design solution space. The answer seemed to be to introduce the concept of software systems meta-methodology along the lines that Klir suggested.

In order to get a clear picture of the field of software methods, a good starting point seemed to be the abstraction from each technique used by the major methods (Hatley/Pirbhai and Shaler/Mellor), and add that of Gomaa (DARTS). This led to the conceptualization of the four perspectives on software theory. These developed through a process of further abstraction into what has been dubbed the four fundamental software engineering perspectives.

**The Four Fundamental Software Engineering Perspectives**

<table>
<thead>
<tr>
<th>Perspective</th>
<th>View</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>WHO</td>
<td>=&gt; AGENT view \ W software</td>
<td>Answers the question -- Who does it? Is modeled by tasking a hierarchy which shows nested independent threads of control within the system. The ‘actor’ paradigm developed by Hewitt and Agha is a development of this paradigm in a pure form, extended from the concept of Pointing as Being as represented by the index registers in the CPU. Pointing is observing, so the agent is an independent automated ‘daemon’ or observer and actor.</td>
</tr>
<tr>
<td>WHAT</td>
<td>=&gt; FUNCTION view \ H design</td>
<td>Answers the question -- What is to be done? Is modeled by hierarchical functional decomposition. Function models what happens in a system in terms of data transformations. This is related to the operators in the formal language which define what can be done within the system. Extended from the concept of Grasping as Being represented by the accumulator in the CPU, grasping allows manipulation which causes transformations in data which are called functions in Dataflow terminology. Function means transformation by the application of operators to operands.</td>
</tr>
<tr>
<td>WHEN</td>
<td>=&gt; EVENT view \ Y theory</td>
<td>Answers the question -- When is it to happen? Is modeled by timing diagrams that represent the temporal relations between signals, extended from the concept of Time as represented by the cycles of the CPU. It is also depicted through the use of interval and temporal logics.</td>
</tr>
<tr>
<td>WHERE</td>
<td>=&gt; DATA view \ ?(the idea)</td>
<td>Answers the question -- Where is it in memory? Is modeled by entity-attribute-relation diagram. Data in a computer system is basically a manifestation of the configuration of memory locations and thus answers the question ‘where?’ Data is modeled in its pure state using Chen’s entity-relationship diagrams extended from the concept of Space as represented by computer memory.</td>
</tr>
</tbody>
</table>

With each of these perspectives is associated a minimal method that is used to represent that perspective by itself:

**DATA viewpoint:** Answers the question -- Where is it in memory? Is modeled by entity-attribute-relation diagram. Data in a computer system is basically a manifestation of the configuration of memory locations and thus answers the question ‘where?’ Data is modeled in its pure state using Chen’s entity-relationship diagrams extended from the concept of Space as represented by computer memory.

**FUNCTION viewpoint:** Answers the question -- What is to be done? Is modeled by hierarchical functional decomposition. Function models what happens in a system in terms of data transformations. This is related to the operators in the formal language which define what can be done within the system. Extended from the concept of Grasping as Being represented by the accumulator in the CPU, grasping allows manipulation which causes transformations in data which are called functions in Dataflow terminology. Function means transformation by the application of operators to operands.

**EVENT viewpoint:** Answers the question -- When is it to happen? Is modeled by timing diagrams that represent the temporal relations between signals, extended from the concept of Time as represented by the cycles of the CPU. It is also depicted through the use of interval and temporal logics.

Later it was realized that these four perspectives could be seen as extensions from the ontological categories developed in the first part of this essay. But the main idea was that by taking these perspectives, it was possible to take fundamentally different views of the software system under development. Combining this with the concept of Software Theory developed by Peter Nauer, gave a means of understanding software representations as being tied to these perspectives. All of the representations produced by software methods attempt to approximate the software theory. The software theory is the **why** of the system under design. There are so many reasons why a system is exactly the way it is that they could never all be written down. Instead of asking ‘why’ through representations, we ask questions of reduced scope like: Who? What? When? and Where? If one adds the assumption that only one perspective can be attained by the designer at a time, then one immediately has a reasonable answer to the question why software theory cannot ever be completely represented. One also immediately introduces the hiding place for the singularity which hides within the software formal-structural system.

Once the four perspectives have been delineated, it is possible to get a handle on the interrelation between software methods. The techniques like dataflow and finite automata, which are strung together in ordinary software methods, can be seen as minimal methods. A minimal method is a means of representing the software system from a single perspective or as a means of transitioning from one perspective to another. Perspectives are envisioned as looking at each other. Perspective looks at...
Perspective $B$ from its own viewpoint. This is what causes the minimal methods to be different. The minimal methods, when defined, tell us the absolute minimum conceptual baggage needed to transition from one viewpoint to another. With the concept of minimal method we no longer need to take the methods given us for granted. We can analyze them to see what is absolutely necessary in our methods. Finally, we achieve a means of analyzing proposed methods by relating them to the field of minimal methods as a whole.

This leads to another interesting point. When the dataflow diagram technique is analyzed, it becomes apparent that it is really composed of two minimal methods -- one addressing data flowing between functions, and the other addressing the transformation of data in stores by functions. Thus, we see in the dataflow a merger of two minimal methods. Data looks at Function as transforming methods, while Function looks at Data as flowing streams between functions. The great success of the dataflow diagram may be attributed to the fact that it represents a two-way bridge between the Function and Data viewpoints. We see in this method an excellent example of how methods should be constructed to represent, in the simplest manner possible, the means of transitioning back and forth between design perspectives.

This kind of study leads inevitably to an attempt to isolate all the minimal methods connecting the different design perspectives. This is not a simple task. A first cut was made in an earlier paper given at CASE88 workshop.\textsuperscript{34} The version presented here is a further refinement. It is necessary to state that this version is still tentative. The process of attempting to isolate the minimal methods connecting all the design perspectives has demonstrated the ad hoc nature of the methods designers are using. Singling out those techniques that can serve as good one-way bridges between perspectives, is a matter of intuition and analysis of the relations between existing techniques used by different popular methods. This kind of analysis is necessary in order to create a concept of the field of design methods. The field connects all the minimal methods via the fundamental design perspectives.

**Viewpoint Pairs and Minimal Bridging Methods**

**FUNCTION > AGENT**

**Minimal Allocation / Mapping of Function to Task**

Looking at Agents from the Function’s point of view. Function views tasks as vehicles for system functionality. This method establishes the mapping from function to task, attempting to establish the logical coherence of the tasks by grouping logically-related functions as much as possible, given performance requirements. See Mellor and Ward, Vol. III of *Structured Development for Real Time Systems*\textsuperscript{35} for an explanation of how Functional Allocation is done.

**AGENT > FUNCTION**

**Minimal Virtual Machine**

Looking at Function from the Agent’s point of view. Agent views Function as Virtual Machine Instructions performed. This method attempts to establish the space/time coherence of the logical functions within the tasks. Space/time coherence allows the system of tasks to function within the performance specifications. Space/time coherence may conflict with logical Coherence. Space/time coherence is established through the construction of a virtual machine which executes instructions whose outcome displays the required system functionality. See Neilson and Shumate’s description of Object-Oriented Design / Virtual Layered Machine Methodology in *Designing Large Real-Time Systems with Ada*\textsuperscript{36} for an explanation of how to build virtual machines.

**FUNCTION > EVENT**

**Minimal State Transition / Function Activation Table**

Looking at Event from the Function’s point of view. Function views Events as state transitions. State Transition Dia-

\textsuperscript{34} See Bib#

\textsuperscript{35} See Bib#

\textsuperscript{36} See Bib#
grams activate functions on specific event occurrences given the current state of the system. See Hatley and Pirbhai or Ward and Mellor or Shaler and Mellor (Project Technology) for the use of state machines in real-time analysis and design.

**EVENT > FUNCTION**

**Minimal Petri Net**
Looking at Function from the Event’s point of view. Function views Events as Petri Net representations of the flow of control in the execution of virtual machine instructions. Petri Nets establish the firing order of functions. Functions are considered events which move markers through the petri network. See W. Resig Petri Nets: An Introduction for an explanation of the use of this method.

**FUNCTION > DATA**

**Minimal Input/Output on Dataflow**
Looking at Data form the Function’s point of view. Function views Data as dataflow. Function sees data in terms of inputs and outputs. The dataflow diagram represents this method very well. See DeMarco Structured Analysis and System Specification for explanation of the dataflow diagram.

**AGENT > EVENT**

**Minimal World-line**
Looking at the Event from the Agent’s point of view. Agent views Event as a Worldline (i.e., a track through space-time). Following one agent through time and watching what messages arrive in what order, can be represented by a worldline diagram. See Agha on Actors for an explanation of the use of this method.

**EVENT > AGENT**

**Minimal Scenarios**
Looking at the Agent from the Event’s point of view. Event views Agents as Scenarios. Scenarios show the sequence of agent’s actions triggered by a given event. It follows the flow of agent’s actions given a specific initial situation. Scenarios are the normal means of doing mental simulations to test a design. This is a standard method which is used in software engineering but little documented in the literature.

**AGENT > DATA**

**Minimal Transport Mechanisms**
Looking at the Data form the Agent’s point of view. Agent views Data as data transport mechanisms such as Queues, Mailboxes, Semaphores, Pipes, Etc. Agent looks at data in terms of the transport mechanisms that which allow communication between concurrent tasks. See Gomma "Software Development of Real-time Systems" for an explanation of a design method that displays transport mechanisms.

**DATA > AGENT**

**Minimal Data Structures**
Looking at the Agent from the Data’s
point of view. Data views Agent via a Data Monitor. The data structures used by agents to support their activities are represented by this method. Normally they are guarded by semaphores or Hoare’s "Monitors."\(^{42}\)

**EVENT \(\rightarrow\) DATA**

**Minimal Design Element Flow**

Looking at the Data from the Event’s point of view. Event views Data as flowing design elements. Event sees data as persistent. The most persistent elements of a system are the design elements that are operands created in the design process. Normally, these are considered a static clock work mechanism. However, in real time systems, they can usefully be considered as flowing through the states of the system with a life cycle of their own.

**DATA \(\rightarrow\) EVENT**

**Minimal Data Transitions**

Looking at the Event from the Data’s Point of view. Data views Events as changes in data values. Looking at printouts of a set of data elements as a program executes in the oldest form of checking out whether programs are working. Events are represented as state variables to the system. Data sees changes as changes in Data values.

Each minimal bridging method will only work in one direction, but it may combine with its opposite bridging method to provide a two-way method bridge. An example of this is the dataflow diagram which combines the function to data and data to function bridges.

Two way method bridge between view while representing all the currently known genuinely useful techniques. Thus, practicality is an important aspect of selecting and analyzing the relation between minimal methods. It is important that a designer be able to look at this as a map of the methodological territory. Yet, it is also important that each method represents a completely different 'orthogonal' aspect of the software system while at the same time being able to work together with the other methods to create two-way bridges and larger method sequences resulting from moving in a certain order through the design perspectives. This set of minimal methods will serve as a working hypothesis until it becomes obvious that it is no longer the best selection of minimal methods. By looking at the techniques that have been chosen, it will become clear what this set of minimal methods seeks to achieve: broad coverage, orthogonal independence, practical usefulness, and systematic coherence.

Others might derive a different set of minimal bridging methods because different criteria may be applied in the selection process. My major criteria was the need to get broad coverage of the space of possible different methods.

Since this set of minimal methods results basically from an empirical selection of existing techniques, there is little surprise that the complete set has some unusual features. These techniques were not developed in any systematic way. They have been invented by different practitioners for many different purposes in various circumstances. To
Event/Data relations are basically of two types. One may be looking at variables modified by a system and making sure they are altered in the correct way by the executing program. Or one may be looking at the data structures that make up the mechanism of the software itself and seeing whether these data structures, like pointers and buffers, are used correctly. The view that design elements flow through a system’s state is a major improvement over the conventional concept of design elements as a static clockwork mechanism. Each of these two views corresponds to the internal relations between Data and Event viewpoints. The fact that methods do not exist for this bridge is very interesting. At first, I could not even think of examples of these method bridges. It took some time to realize that this is because we look on our systems from the point of view of persistence of data structures, not their change. The primitive nature of these methods shows that we are still applying Being1 as the criteria for the systems we build, and the function perspective inherent in Being2 has not impacted our design approaches as yet.

The concept core shows how these two ill-defined methods might be seen as addressing a single set of problems. We are concerned with data as content (interior view) which varies according to transient system changes. Transiency is Events viewed as links. The persistent state is the length of time things remain the same, which is different from the coordinating state. Remaining the same depends on the sensitivity threshold applied. Significant changes trigger interrupts.
FUNCTION

what

function views agent as vehicles for system functionality

who

AGENT

agent views function as virtual machine instructions performed

Tasking Structure

Implementation Model

mapping of function task

Virtual Machine Instruction

Function Transformation

Mapping Arrow: Call Structure

FIGURE 7

The methods presented here are further articulated by Ward and Mellor (function allocation from and implementation mode.) and Neilson and Shumante (OOD/VLM methodology). The two minimal methods here do not combine into a simple two-way method bridge. Two separate methods are referred to in attempting to describe the relationship between Agent/Function. From the point of view of function, the main thing is that all the system functionality is allocated to the tasks so that everything that needs to be done gets implemented. The implementation model records the progressive incorporation of constraints into the essential model. However, the tasking model developed in the proceeding bridge is used here for the mapping of functions onto tasking agents that carry out these tasks. The tasking structure may consist of nested tasks. But ultimately, each task is composed of a virtual machine which is, in turn, composed of lower level virtual machines. From the task's point of view, it is the execution of virtual machine instructions that causes functional processes to be done. Thus, agents see function in terms of these procedures and functions that embody different aspects of system functionality.

The task here is seen as a white box which we are looking inside of, either to place functionality by mapping, or by seeing what functions are performed within the task. In terms of functions, we are concerned with the type of transformation that is to be performed, either in terms of a requirements statement or in terms of the actual performance vehicle. The virtual machine and the mapping arrows actualize the relations between these two concepts. This bridge is the most complex all.
The Dataflow is further articulated by Hatley and Pirbhai in terms of its relation to control models. The key concept that they have added is the idea of inhibition control signals to express system modality. This control is not adequate for real-time design, although it may suffice for the development of the essential requirements model. The Dataflow technique is very successful and shows the power of constructing two-way method bridges. The dataflow is used in requirements analysis and specification phase to build the essential model of the system. It is then used in architectural design phase to elaborate the essential model into the implementation model by introducing constraints. The conceptual core is isomorphic to the major elements in the representational scheme.
The methods articulated here are further developed by Hoare (Monitors) and Gomaa (Tasking Architecture Design). The two minimal methods combine easily into an overall architectural model similar to that of the dataflow diagram. However, here the emphasis is upon architectural issues of tasking rather than functional considerations.

The conceptual core is derived from the basic concepts displayed by this model.

Task here is treated as a blackbox because we are not concerned with what it does. Tasks are related to Protected data through their ownership shown by possession of semaphore tokens. Protected data is a liking aspect of data because protection is the inverse concept of ownership. What is owned must be protected by the owner. Tasks communicate via transport mechanisms. Transport mechanisms assure date integrity in a multitasking environment.

In this case the derivation of the conceptual core is straight forward because the methodical model has only four elements which are isomorphic to the concepts in the core.
EVENT ~ FUNCTION

**EVENT**

when

event views function as state transition

**FUNCTION**

state machine

```
  s1  _event_  _action_  s2
    _event_  _action_  s3
```

petri net

A series of transitions may be reduced to get minimal state machine for a given net

```
  entry

  _place_
  _transition_

  _output_  _state_
```

function views event as petri net representation of control flow

what

The state transition method is well explained by Ward and Mellor and by Hatley and Pirbhai. The use of petri nets for modeling control is not as well known. Its advantage is that dynamic models which pass tokens through the net can be used to track control flows throughout a complex system with multiple agents. The transition may be seen as the execution of an instruction in a virtual machine. The petri net may be reduced to form the minimal machine by collapsing all sequences of instructions into a single transitional block. In this case, it is through the petri net reduction that the transition between the petri net and the state machine is achieved. State machines invoke actions which may be seen as implementing functionality.

The concept core is abstracted from these two methods. The transition (seen as the transition in the petri net and the action called by the state machine) is considered as a linking aspect of function, that is function considered as transitions between states only. The event here is seen as the activating control signal (seen either as the event in the state machine or as the marker in the petri net). As an activating control signal (pure interrupt), the Event is displayed as a black box.

The relations between these are captured by the ideas of control flow and coordination state. The control flow shows up as the path between states in the state model, and as the path between transitions in the petri net. The coordination state shows up as the places in the petri net and the states in the state machine. Coordination states are differentiated from persistence states. Coordination states signify logical states that control the virtual machines and allow agents to coordinate.
The worldline method is elaborated by Agha in the book ACTORS. The scenario method is widely used for mental simulation, but no satisfactory description of it as a methodology is known. These two methods form a two-way bridge in an atheistically pleasing way. The world line is a means of understanding the relativistic relation between actors. The scenario is the major means of knowing whether the system being designed will work properly.

The concept core expresses the inherent ideas behind these two methods. The agent has illusory continuity, and this sees the agent in its linking aspect. The sequence of events can be looked at with different thresholds of refinement. One event may be made up of many sub-events so that the Event here is seen as a white box. The concept that agents may communicate with each other, and the fact that their communication is structured relativistically, are the two other concepts that serve to bring out the relations between the continuity of the agent and the decomposition of events.

These methods are very important to the development of networked systems of any size. Now they are not widely known or appreciated. The analysis of indeterminate relativistic systems will be of prime importance in the near future. One tool that models this method bridge is VERDI, which is derived from RADDLE, developed by Vincent Shen at MCC consortium in Austin, Texas.
make an assemblage of these techniques, after the fact, and claim unity for them, would seem to be ridiculous. However, taken as a representative of what a software metamethodological field might look like, it is possible to test it to see what its shortfalls might be so that the next attempt to construct a field might be more successful. Analysis of the field of minimal methods reveals some strange properties.

- Not all the method bridges are two way.

Three conceptual cores define a viewpoint. Each conceptual core contains an image \( V \) of the viewpoint being defined either in whitebox, blackbox or linking aspect. Each conceptual core contains an image \( n,o,p \) of the other viewpoint that it is linked to by the method bridge. Each conceptual core contains two connecting concepts \( \text{GiKebC} \) related to control, communication, coherence, or cohesion. Three conceptual cores define each viewpoint in a systematic way.

**DEFINITION OF VIEWPOINT USING METHOD CORES**

Linked conceptual cores systematically define a particular viewpoint on system design.

**FIGURE 17**
The whole field is lopsided with function-task bridge being very complex while event-data is very simple.

The elements of different minimal methods do seem to work together to a certain extent, but it is difficult to see an overall pattern.

Abstract analysis reveals asymmetries in the relation between the elements of minimal systems.

These properties probably flow from the ad hoc and empirical nature of the elements combined, but let us suspend judgement and treat this set of minimal methods as a united field. That unified field represents the full extent of necessary fundamental software methods. Any methodology must build upon these twelve minimal methods, then string them together by taking a particular path between the four perspectives in order to design a software system. Accepting this hypothesis, the rest of
The conceptual cores of the method bridges defined in the previous six diagrams will undoubtedly change as our understanding of the application of structural systems to design improves. They are wholly derived from an analysis of existent methods empirically present in design circles at this time. The analysis of these methods is difficult due to their different states of development. From what has been seen, sometimes the derivation is clear and obvious, while at other times it is unclear. This is usually related to whether the bridge is two-way or not.

Once these conceptual cores have been derived, it is possible to show how they can be used to define the particular viewpoints of system design. In the above diagram, the function viewpoint is being defined by moving systematically between method bridges that revolve around the function perspective. By revolving around a single design perspective iteratively, a better and better picture of the system from that point of view is represented. In revolving around the perspective, it is the concepts of the bridge conceptual cores that are being used to refine the system. In each revolution, we could use a different one of these key ideas as the guiding theme of our analysis and representation construction.

Notice that from each method bridge the Function perspective is treated differently. It is treated in turn as blackbox, whitebox and link. This variation of the treatment is precisely what allows us to articulate the system from the given viewpoint. In this step-by-step articulation the functional decomposition is developed.
The same process of refinement is being carried out there with respect to another perspective, i.e. that of data. Method cores should be looked at in their relation to each other as they relate to a particular perspective. One is only in one perspective at a time, looking at objects created from other perspectives. Perhaps different methods should be created that link all the methods that surround a particular viewpoint. It is this kind of analysis that needs to be carried out beyond the refinement of the idea of the method cores themselves. All the ideas that go into describing a real-time embedded software system form a formal-structural system that relate Being$^1$ and Being$^2$ concepts to describe something rooted in Being$^3$. This description must always be partial, but that does not mean we cannot make descriptions. Our descriptions need to isolate the discontinuities across which our formal representations cannot cross. Here, this is done by positing that those discontinuities are related to the shift between perspectives on the software design theory.

Thus, the set of conceptual cores around a particular perspective are able to form a formal-structural system which does not have discontinuities. What we are actually doing is venturing out from a particular perspective just enough to use its surrounding and thus defining methods, but we are avoiding actually crossing the bridge. This means of using the methods will result in the high definition of the software system from a particular viewpoint, and will also cause sharply defined discontinuities between a given viewpoint and another viewpoint to appear. The bridging methods will allow us to cross these discontinuities gracefully. The redundancy of method viewpoints surrounding the different viewpoints (for instance U is part of the surrounding system for both Data and Agent viewpoints) is what allows the movement across the discontinuities to be made without total disruption of the design. In other words, something is the same when you get to the other side. Speeding up the movement through the perspectives, give the illusion of encompassing the whole theory.
Really seeing the whole theory is impossible because of the hidden presence of pure immanence. However, the faster and faster movement between the viewpoints will give the impression of seeing the whole system. Our focus should not be upon glimpsing the whole system, but instead upon the discontinuities that become apparent as we define the system from each perspective. It is out of all these discontinuities that all of our really difficult problems in system design come. These wicked problems express immanence as it constrains our freedom to design. The set of method cores are, in fact, the very expression of our transcendence. Here we are conceptualizing the structural system that allows us to define the software theory abstractly. This abstract representation is our means of transcending the particular systems with which we are confronted, and allows us to view them in relation to the generic software system, i.e. the pure theory. The definition of method cores is thus the expression of the purely theoretical software theory. Because of this, the development of the method cores should allow us to develop our software design metrics. Currently, metrics for code representations are becoming fairly well developed. The most popular of these are those of Halstead and McCabe. We are still, though, in a quandary of how to apply metrics to designs. It is easy to see that if we could agree upon a set of method cores, then the concepts of those cores should be what is measured in order to get quantifiable design metrics that describe the software theory considered purely abstractly as fully as possible. I suggest it is those metrics related to method cores that should be related to quality measurement.
We are now ready to transition from the relation between method cores and their associated perspectives, to the concept of the infrastructure of the formal-structural system. Note that in the case of each method core, two of its components is related to the representation of perspectives. The other two concepts have the function of relating the two represented perspectives. The question then becomes whether it is possible to formulate any relations between these mediating concepts. There are twelve mediating concepts in all, and what will be shown here is that these form an infra-structure that reveals the deep structure of the methodical structural system. The infrastructure will be represented according to the form Buckminster Fuller calls the vector equilibrium. This is an important synergistic theoretical structure. Because of the empirical origin of our mediating concepts, we do not achieve a perfectly symmetrical theoretical structure. But this is exactly the point because by considering these theoretical structures, each mediating concept is related to all the others in ways that would not be achieved otherwise. It is the consideration of the lack of symmetry of the theoretical structure in relation to the pure geometry of thought that should give us some insight into what is ultimately wrong with this way of thinking about methods. Perhaps method cores should really be five-fold instead of four-fold. When one constructs a theoretical structure, it is exactly the anomalies that are interesting. The point is that there is enough coherence to this explanation that it is worth entertaining until someone can explain more adequately the relationship of software engineering methods to each other.

So the next figure shows all the mediating concepts, from the conceptual core going together to form the infrastructure of the formal-structural system by which the software design theory is represented.
If each of these core-mediating concepts are related, using the same schema of as the representations of the perspectives (whitebox, link, blackbox), then analysis reveals four major complexes. These complexes may be related to several deep structural ideas with which all software systems must deal. These deep structural ideas are Communication(i), Control(j), Causality(k), and Cohesion(l). What is interesting is that each of these ideas concerns the relation of immanence to transcendence in some basic way. Control represents the transcendence of master over slave, and what are automata but pure slaves? Causality is a traditional way of expressing transcendencies. The cause is transcendent in time to the caused. Communication expresses our ability to transcend the barriers between entities. Cohesion is the expression of the transcendental unity of the system. This reduction two four basic transcendental themes at the infra-structural level serves to show that the description of the software theory as being rooted in Being is accurate. This minimal system of deep structural ideas that are all representations of transcendence in various guises forms a core around the kernel of pure immanence that lies at the heart of the software design theory. It is this kernel of pure immanence that we must constantly deal in our software engineering practice. If we do not recognize what we are dealing with, it only makes the job more difficult. If instead, we learn to recognize the different ontological categories that effect our work and learn to use the work-arounds given to us by the formal-structural system, we can more easily understand the source of those difficulties. There is no ‘silver bullet’ but not because we are dealing with a werewolf. There is no silver bullet because we are dealing with our innermost relation to technology as ‘controlling nature.’ It is part of ourselves that we cannot see. It is a blind spot that grows darker the more we try to bring light there. We must learn to deal with it indirectly, and ultimately accept our own limitations.
this essay will attempt to understand some of the idiosyncratic attributes and the non-symmetrical relations between the various minimal methods as they are presented here. In the process, many interesting features of the minimal methods will come to light. In fact, it will become obvious that these features are highly significant and form an overall pattern which allows us to gain a deep insight into the nature of the methodological field. It will become apparent that the methods, although chosen from an empirical field of currently used and available methods, actually form a systematic field which has its own internal logic. This internal logic makes the fundamental differences between the points of view on software design clear.

What follows will be a graphical representation of the method bridges between software perspectives. Each bridge will be reduced to its idealized ‘conceptual core,’ and a brief summary of the sources of the methodology will be appended to each diagram. The reduction to ‘conceptual cores’ allows us to project a highly symmetrical model of relations on the whole field of the methods making asymmetries visible. It is necessary to have all the details of the method bridges before us as we begin to attempt to construct our software meta-methodology. Thus, a complete symmetrical model will be developed here which step by step abstracts and relates the minimal methods to give a picture of the whole field of methods. This model will show how an infrastructure for the field as a whole may be derived. As that field infrastructure is studied, the asymmetries begin to appear, the explanation of which is the focus of the rest of this essay.

In the foregoing, the method bridges have been abstracted into what are called ‘conceptual cores.’ These attempt to distill the minimal system of concepts from each method bridge. This has been done in order to allow easy manipulation of all the concepts within the field of software methods. If we wish to discover the structure underlying this field, it is necessary to reduce the methods to theoretical structures that can be compared and contrasted. For instance, these conceptual cores can be used to see how the methods define each particular perspective. There are three ‘cores’ per perspective. The relation between these cores can be seen as a diacritical system of concepts that work together to make a particular perspective visible in relation to a software system. Each perspective can be seen as being defined by its three adjacent conceptual cores, and all the conceptual cores working together define the system as a whole.

The definition of conceptual cores also allows us to isolate concepts associated with representing viewpoints from those connecting concepts which tie these representations together. Viewpoint representations are described here as of three types: whitebox, linking or blackbox. Each viewpoint has all three kinds of representation. Each conceptual core contains two viewpoint representations and two connecting concepts. The connecting concepts may be separated from representational concepts to isolate the infrastructure of the method field. This substructure is represented as a vector equilibrium. The vector equilibrium was identified by Buckminster Fuller as a key conceptual structure in his geometry of thought. It is an important threshold of theoretical complexity. In this case, the vector equilibrium represents the conceptual infrastructure of the field of software methods. This infrastructure of connecting concepts, once isolated, can be used to identify the core of the infrastructure which contains four key systems design ideas: communication, causality, cohesion, and control. These, in turn, mask the singularity of pure immanence.

This reduction of the field of methods to conceptual cores of the two-way method bridges leads to the separation of the connecting concepts from the viewpoint representations which isolates the infrastructure of the formal system of methods. The analysis of the infrastructure, in turn, leads us to identify its core concepts. And finally, the hypothesis of the singularity at the center of the infrastructure is posited. All these analytical moves are calculated to make the field of methods more understandable as a whole in preparation for the meta-methodological discussions that follow. Hopefully this brief discussion of the analysis of the field of software methods will serve to orient the reader. This is a complex subject which has many aspects. Using abstractions such as these to render the field and its underpinnings visible will help lay out the important landmarks in that field and allow it to appear as a single gestalt worthy of meta-methodological study.

The question to be posed now is how to fit these four basic

---

43. See Bib#  
44. See Bib#
Software Systems Meta-methodology

perspectives that have been identified into Klir’s representation of a general formal-structural system. The representation of the formal-structural system is too complex to explain in an essay of this length. Thus, only the highlights will be given to show the relationship of the four perspectives to what Klir defines as a ‘system.’ Klir’s definition is rather elaborate because it involves the delineation of several ‘epistemological levels’ upon which the term ‘system’ takes on different meanings.

Klir’s EPISTEMOLOGICAL LEVELS

- **infinite regress**
- **Level 4,5,... Meta-systems**
- **Level 3 Structure system**
- **Level 2 Generative system**
- **Level 1 Data system**
- **Level 0 Source system**
- **Level -1 Object system**

Grounding ONTOLOGICAL LEVELS

In order to establish the relation between Klir’s epistemological levels and the references to ontology made in part one of this essay, consider the negative levels added to those delineated in ASPS. For Klir, the object of investigation does not belong among his epistemological levels except as an adjunct to the source system which is the representation of the object system.

For our further considerations, let the term ‘object’ be defined as a part of the world that is distinguishable as a single entity from the rest of the world for an appreciable length of time.45 Klir goes on to differentiate material from abstract objects, each of which are, in turn, divided into those found within the world and those that are man-made. This definition carries a great deal of metaphysical baggage emanating from the development of the Western philosophical and scientific tradition. At the very least, the concept of ‘object’ depends upon the coequal definition of a ‘subject’ who is the perceiver of the object. Both of these terms are laden with many philosophical interpretations which provide competing grounds for their theoretical elaboration. Instead of delving into that morass of competing dialectically-related positions that make up the bulk of the Western philosophy, let us attempt to extend Klir’s hierarchy in the other direction into the realm of ontology in order to get a wider perspective on the problem of the epistemology of the object.

![FIGURE 25](image_url)

Both the subject and object exist in what has been called the ‘Lifeworld.’ Husserl first introduced this term in his book *The Crisis of European Sciences and Transcendental Phenomenology.*46 Subsequently, the concept was developed further by Alfred Schutz in *Structures of the Lifeworld.*47 The Lifeworld is the world of everyday life in which we all live. Science and Technology are cultural products that are built upon the Lifeworld.48 Without the Lifeworld they would have no reality at all. Objects are idealizations that exist based upon the concrete foundations of the Lifeworld. Subjects are also idealizations of real people that live their day-to-day lives and perceive ordinary things that are sometimes raised to the status of an object by elaborate focusing and isolating

---

45. ASPS pp33-44
46. See Bib#
47. See Bib#
48. See Bib#
conceptual procedures. These elaborate procedures for perceiving ordinary things ‘clearly and distinctly’ were first inaugurated by Descartes in his Discourse on Method. Klir calls these observational channels. These idealizations that make us subjects who perceive objects have over the course of centuries become second nature to many of us who were successful in being ‘educated.’ For us, there is no problem when Klir assumes he can pass over the Lifeworld with a scant three paragraphs and go straight to defining objects without even mentioning our subjectivity. Much of the energy of philosophy in the last hundred or so years has been spent by the rediscovery of the Lifeworld and how it has been glossed over by idealizations of science and philosophy within the Western tradition. Unfortunately this effort appears to have been lost on Mr. Klir. Yet he is not alone in this. Objective and subjective idealization that glosses over the Lifeworld is rampant in scientific discourse. These glosses attempt to maintain our stance in the present-at-hand mode of perceiving reality. The problem is that it reduces the multiplicity of life to a caricature.

The next level (-3) beneath the Lifeworld is the realm of ‘existence.’ Existence is the one ontologically-related concept excluded from the Western concept of ‘Being.’ Existence concerns the concrete specifics of lived experience. It is the realm of accidents which make up so much of the world in which we live. All the individual particularities that are one way rather than another, seemingly by chance, that gives life its incredible diversity. Traditional philosophy always ignored existence in order to get on to more lofty ontological concepts. This realm was rediscovered in this century by the existentialist philosophers such as Heidegger, Camus, and Sartre, who proclaimed the slogan “existence proceeds essence” as the cornerstone of their ruminations on everything. The fact that the Lifeworld is filled with non-essential particulars which could be many different ways, but in each instance is one particular way, cannot be denied. Existentialism is a field of philosophy that addresses this issue and seeks its significance without skipping over it.

Beneath the level of existence is another strata (-4) which is called the phenomenological realm. This realm has also been explored a great deal since the turn of the century, predominately by the followers of Edmund Husserl. Phenomenology deals with the world of essences as perceived in pure consciousness. Essences of things are their necessary attributes and the constraints that give those attributes inner coherence. From essences, we build up simple ideas by a process of ideation. Ideas are glosses on essences. Once these glosses are established, we use induction and deduction to move between idealized abstraction and idealized particulars. Phenomenology seeks to unearth the roots of our idealized glosses in the perceived things. Phenomenology is a very significant advancement in modern thought. It replaces the naive ungrounded abstractions of traditional philosophy and science with a precise vocabulary for describing perceptions, both material and abstract. Klir’s division of objects into these two categories is a very crude approximation to the more subtle noetic and noematic phenomenological constructs that deal with our apprehension of essences and ideas.

Beneath the phenomenological strata is the ontological world which is level -5. Here we first encounter the concept of Being. Being is the most general concept we have. All the essences of things depend on Being. So phenomenology flows directly from ontology. The relation between individual beings and the concept of Being is called by Heidegger ‘ontological difference.’ Being itself is not contained in any one being. A class cannot be a member of itself, according to Russell and Whitehead in Principia Mathematica. This alone would justify the distinction of ontological difference to avoid paradox. Instead, all beings are contained in the world projected by human beings. We are beings-in-a-world which Heidegger calls Dasein (being there). It is this concept of Being and its supposed fragmentation that was treated in the first section of this essay at length. The fragmentation of Being is probably the most significant event for the Western philosophical and scientific tradition in this century. Before this century started, there was only one kind of Being which was defined by Kant and accepted by everyone as our inheritance from Aristotle. Now there can be recognized at least four types of Being. These four types of Being have been described in detail, and software’s essence flows from just one of them. It is the fragmentation of Being that makes software ontology

49. See Feyerabend speaks of the theories embedded in perception and the need to overcome these in order to ‘see the right thing’ in an experimental setting. The famous example is Galileo attempting to get others to see the same thing as himself when looking at the moon through a telescope. This is discussed at length in Bib#.
50. See Bib#
51. See Bib# page 33
52. See Bib#
53. See Bib#
54. See Bib#
55. See Bib#
56. See Bib#
57. See Bib#
important. If there was only one kind of Being now, like there used to be, then there would be no question of the ontological foundations of software engineering. But things have changed radically. And that change is the deepest possible kind of change. It is this realization of deep change that needs to inform our understanding of Klir’s epistemology of systems. The whole technological world is teetering on the edge of an abyss. The Crisis that Husserl recognized has deepened.58 We exemplify that unbalance in everything we do. The fractures at the ontological level percolate up through the phenomenological strata, the existential strata, the Life world strata, until the edifice of the idealizations of the structural system built so carefully by the philosophy and science begins to crumble. Outwardly there is a shiny exterior of technological progress and inwardly there is ‘mithraic’ active nihilism destroying meaning in the world. (We call this active nihilism ‘mithraic’59 to show that the roots of this nihilism are historically very deep. Active nihilism has its roots in the historical development of Zoroastrianism60 which was taken to Rome by converted soldiers as a mystery religion. This was a Greek style mystery cult that flourished all over the Roman empire and was related to Gnosticism61 and Manicheism62 and subsequently influenced the development of Christianity.63 The development of extreme conflicting opposites as ontological categories was inherited by all of these related religions from Zoroastrianism.64) We experience that loss of meaning in the world as an existential crisis described so well by Camus in The Myth of Sisyphus. We see it as crime rates, divorce rates, inflation rates, suicide rates, abortion rates, deficit increase rates, pollution rates, garbage dump fill rates, ozone hole expansion rates, resource depletion rates, poverty and starvation rates, along with a myriad of other indicators of modern ‘progress’ and our ‘manifest destiny’ working itself out in the world.65

The concept of Being is interesting because it is almost only the Indo-European languages that have this concept as part of the paraphernalia of grammar. Other languages like Chinese and Arabic do not have a concept of Being66 as such. Even though early Western scholars attempted to find it there, and in frustration, projected it upon these and many other languages]. Instead, these other languages have a concept of existence, and what lies beyond that is considered empty or void. Thus, the levels (-4, -5, and -6) do not exist for most other non-Indo-European languages.

The concepts of Being, with both essences and accidents, are a ridiculous fantasy from their perspectives. There is only the absolute, and from it flows existential beings. The concepts which are fused together into our concept of Being are handled grammatically in other ways. The concept of Being contains the idea of copula (A is B), ontos (A is !), attribution (A is blue), and identity (A is A). These concepts have been extracted from the grammatical Being over the centuries and make up our ontological concept of Being. The ancestors of the Indo-Europeans67 are called the Kurgen people because they built burial mounds called ‘kurgens.’ They came out of central Russia and devastated the entire world about the fourth millennia BC. Prior to that it is thought they originated between the Black and the Caspian Seas about 6000 years BC.68 They were the first to use the domesticated horse in warfare. They became the Anglo-Saxons, Germans, Gauls, Latins, Greeks, Hittites, Persians, and Aryan Indians. What is of interest is how their concept of Being came into existence. In all Indo-European languages, Being is the most irregular verb. It is made of several different Indo-European root verbs melded together over a long period of time to become what we think of as a single grammatical term today. The point is that Being started as a composite of different verbs, all meaning ‘to remain’ or ‘to persist’ which were blended together to form a single term with many different conceptual components. Thus, the current fragmentation of Being is merely a return of sorts to the original fragments which separately do not have the same meaning. Without the concept of Being, our language would be like Chinese or Arabic with only existence and the Absolute veiled by Emptiness or Void.

When we look to what Being means, we cannot help but see that the “remaining” and “persisting” denoted by verb fragments that make up the grammatical construct “Being” refer to what C.G. Chang calls a “subtle clinging.” This “subtle clinging” is clearly identified by Buddhist philosophy and connected to “clinging and craving” by human beings which is the source of all sorrow. The Buddhist antidote for this malaise is the concept of “emptiness” or “sunyata.” Through sunyata one may perceive and become immersed in the Absolute. The concept of Being is a barrier to this kind of experience of

58. See Bib#
59. See Bib#
60. See Bib# and ??? by same author.
61. See ??? on Gnosticism.
62. See ??? on Manicheism.
63. See Bib# and Bib#.
64. See Bib#
65. There are endless examples of tales of woe. See Bib# and Bib# Also Bib#
66. See Bib#
67. See Bib#
68. ??? Scientific American article on origins of Indo-Europeans.
Ultimate Reality. Instead, within the Western tradition, there is another strata projected upon the Absolute which Heidegger calls onto-theological. It imagines a Supreme Being who is called “God.” For Kant, the Supreme Being was an idealization of the concept of infinity. Each theologian imagines a different means of conceptualizing “God.” However, as long as “God” is thought of as having Being or as associated with anything with Being, there is a fundamental flaw which flows from the arising of the artificial concept of Being. As “God” Being faces the Absolute. These conceptual glosses of “God” are found by men to be empty of meaning. Nietzsche then comes along and declares that “God is dead.” He was killed by those who approached Him through empty abstractions or by associating Him with beings or Being. The death of “God” is the logical result of the creation of the concept of Being which flowed from the forging of the grammatical construct “Being.” Through the death of ‘God’ we lose contact with the ultimate reality or Absolute that permeates our existence in the lifeworld.

This brief excursion into the strata that underlies the structural system is only meant to give an overview of what is being glossed by Klir. From this it is possible to see how ontology is related to epistemology. These two specialties together make up the subject matter of metaphysics. If the concept of Being fragments, it effects all the levels above it. When “God” was declared “dead,” the empty conceptual unity cracked, being fragmented without any unifying force to keep it unified. Essences lost their meaning. Existence became absurd. The lifeworld was realized to be permeated by Mithraic active nihilism, and all the edifices beyond that found themselves resting on the abyss of groundless. The great chain of Being was suddenly broken. By saying that “God” is dead, Nietzsche merely pointed out that the emperor had no clothes. Instead of listening, we slay the messengers by attacking Nietzsche and his intellectual heirs. The technologist is caught in a world where the process of destruction appears in the guise of “progress.” This so-called “progress” is more aptly named “artificial emergence.” Artificial emergence is the production of empty novelty for its own sake. Newspapers, fashions, new model cars, and much technological innovation may be classed as artificial emergence or excrescences (abnormal growths). This is contrasted to genuine emergence such as scientific revolutions or the appearance of a genuinely new artifact that is not merely a combination of other artifacts by some method of permutation of characteristics. Genuine novelty stands out on the background of excrescence. The technologist is defined as the professional whose job it is to distinguish genuine emergence from artificial emergence. All other features of the technologist’s job flow from this one characteristic. The technologist, regardless of field, must survey the continuous production of novel technical gadgets and attempt to make use of what is worthwhile for the purposes at hand. The real discrimination of worth depends on making non-nihilistic distinctions between genuine emergence of significant new technologies from the background of marketing clutter over products that are only variations on the same theme. Distinguishing genuine emergence is the highest calling of the technologist. Where the scientist and inventor is trying to manifest genuine creativity by facilitating genuine emergence, the technologist is attempting to recognize it. This means the technologist is attempting to find significance within the field of artifacts. That significance is predicated on the recognition of genuine emergence. Without significance, then nihilism takes over, and all the technical artifacts become indistinguishable. Even the most mundane technological decision to use one artifact over another must reflect the recognition of significance. As Bateson says, “differences that make a difference” are important. The line separating genuine emergence from excrescences is the primordial significant difference from which all others flow. The technologist attempts to understand the whole field of emerging artifacts. Keeping up with this and distinguishing what is really useful among competing products, what is worth knowing among the myriad articles, what is worth knowing from the avalanche of new books, what is important in terms of standards selection, etc., is a dizzying and endless quest. The recognition of what is significant in the face of information overload is a major problem. Only by knowing the background of excrescence can the figures of genuine emergence of novelty be seen.

Through genuine emergence new meaning enters the world. New meaning is constantly created, and this is the inverse of the “god is dead” scenario. This new meaning must be recognized and capitalized upon for it is the only antidote to the domination of mithraic active nihilism. This is what makes genuine emergence so important. It causes the segmentation of our own scientific tradition (paradigm shifts > Kuhn), intellectual tradition (episteme changes > Foucault), and philosophical tradition (Epochs of Being > Heidegger). The point is that genuine

69. See Bib#.
70. See Bib# and Bib#.
71. See Bib#
72. See Bib#.
73. See Bib#.
emergence has a particular structure. That structure is written into the infra-structure of Western metaphysics. As genuinely new things come into existence from the void they pass through levels -4-5-6. In level -5, they encounter the fragmentation of Being. This encounter appears as the “thing” passing through each kind of Being as a phase of its becoming. It passes through these phases in the reverse order of their numerical ordering: Wild Being then Hyper Being, then Process Being, then finally Pure Presence. In each phase they are best described by a different mathematical formalism associated with each kind of Being: Chaos, then Fuzzy Sets, then Probability, then finally Determinate Calculus. These stages of emergence of the genuinely new distinguish them from artificial emergences which do not pass through all the stages. Instead artificial emergences are produced by the dynamic of the formal-structural system. The formal-structural system is not just a static representation, as Klir’s work may suggest. Formal-structural systems are non-linear dynamic systems that appear embodied in society, and to a lesser extent in the artifacts produced by our society. A natural output of dynamic formal-structural systems is erratic change (disorderly motion). It is significant that if the erratic changes produced naturally by the eye are cancelled out, the eye cannot see. The scene, moved perfectly in sync with the eye, just disappears. For a formal-structural system, erratic motion must be produced for the parts of the system to remain manifest or present. If erratic motion were to stop, the system would just vanish. Thus, a great deal of effort and energy goes into producing erratic change in every dynamic formal-structural system. Artificial emergence, or ex crescence, is one of the main sources of erratic change in technological systems. By producing myriads of slightly different variations, the system creates its own temporality. This temporality allows the system to “see” its environment and also maintain its own presence. If the ex crescence stops, the whole of the formal-structural system collapses. By maintaining its erratic changes, it appears ne-enthropic, producing structure as it feeds off the environment. As you can see, artificial emergence (ex crescence) and genuine emergence are opposites. The former is a product of the dynamic formal-structural system, while the other is the passing through of ontologically distinct phases as an entity comes into existence from the Void. The field of artificial emergence provides the background on which appear the genuinely emergent entities. Without the background, the genuinely emergent entity could not be seen. Without genuine emergence, there would be no advent of meaning to sustain significance. And with no significance (differences that make a difference), then the nihilistic aspect of the formal-structural system would engulf it. This would be the same as if erratic change stopped -- it would just disappear. Thus, there is an inner connection between genuine emergence and the production of erratic change. They are nihilistic opposites. Stopping one is like stopping the other. Like all nihilistic opposites, they only appear to be opposites, but really they amount to the same thing. This gives us a hint that the formal-structural system and its ontological foundations are merely two aspects of the same thing. Unless we consider them both together, they can never be understood. This is because what is hidden in one of the nihilistic opposites is revealed in its twin.

Understanding the relationship between the formal-structural system and its ontological grounds is important because it leads to a deeper appreciation of the relation between man and technology. For the technologist (regardless of field, and everyone engaged in the technological system is sometimes engaged in the role of technologist), it gives a clear view of the phenomena of never-ending variation and change with which he must cope on a daily basis. The technologist is concerned with using new technologies to improve efficiencies in the technological system. But the plethora of new ideas and artifacts is overwhelming. The technologist must establish his own criteria for what is significant out of the onrush of new entities that confront him. This criteria may be grounded by the realization that most of the effort to produce “new” things is merely erratic change necessitated to keep the formal-structural system manifest. Occasionally, a genuine new entity will surface. Its hallmark is that it will go through each of the stages of manifestation grounded by the different kinds of Being. The technologist must watch for these crucial changes which will cause a revolution in the apprehension of the whole field of new technologies. The emergence of genuinely new things causes paradigm shifts and restructures the technological field. Paradigm shifts are an important phenomenon that has been recognized in philosophy of science, and which applies to all areas of human endeavor controlled by formal-structural systems. Paradigm shifts dictate the temporality of the technological field. They give a structure to our approach to that field. By watching the field for indications of those shifts, we can orient ourselves toward the whole realm of technology. This orientation involves the realization of the significance of the differences between different kinds of changes. This means appreciating the difference.

73. See Bib#
74. See ?? [eye motion experiment]
between nihilistic or non-significant differences that are the result of excrescence, and the non-nihilistic distinctions that indicate what is really “relevant” in Alfred Schultz’s sense.  What is relevant to the technologist is what will keep him tuned to the changes in the technological field overall as it changes and occasionally undergoes restructuring. The technologist finds himself lost in a constantly transforming landscape like that of Stanislav Lem’s SOLARIS.  He must use the restructuring of that landscape itself as his guideposts. This is achieved by recognizing the difference between genuine emergence (or the non-nihilistic distinction) and artificial emergence (or excrescence) which exemplifies nihilism and is produced by the formal-structural system as erratic change (disorderly motion) in order to hold itself in manifestation. The formal-structural “remains in Being” or “clings to Being” through the manifestation of the inner structure of Being. This inner structure of Being is manifest in the passage of the genuine novel through each phase of manifestation supported by a different kind of Being which can only be seen against the nihilistic background of excrescence. But also the inner structure of Being is manifested by the formal-structural system itself.

As a static representation, the formal-structural system incorporates Being1 and Being2. When it becomes dynamic, it begins to produce erratic change in order to keep itself visible. Where does this erratic change come from? The answer can only be that it is produced by the differing/deferring of difference at the event horizon that surrounds the singularity embodied within the formal-structural system. Erratic change, like Brownian motion, is constantly produced. It is the ground state of the non-linear dynamic system. In the production of erratic change, all the different types of mathematics contribute. Chaos, Fuzzy Sets, Stochastic and Determinate Continuity all contribute to producing truly erratic change. This erratic change, which exists as the ground state of the dynamic formal-structural system, is the manifestation of all the different kinds of Being working together to hold the formal-structural system in existence. Within the realm of artificial existence, the variety of arbitrary peculiarities are the result of the action of the erratic change produced by the formal-structural system. This erratic change exemplifies the collusion between the different types of Being, but it is particularly exemplary of the differing/deferring that masks pure immanence. In fact, erratic change of the ground state does not exhibit chaotic behavior. Chaotic behavior is exemplified at higher energy levels of the dynamic formal-structural system. Chaotic behavior becomes clear in the changes of structure that occur when the dynamic formal-structural system reaches higher energy levels, or when genuine emergence occurs. In the ground state, it is fuzziness that dominates erratic change. There are myriad possibilities for combinations of the characteristics of existing artifacts. The permutation of these characteristics to produce hybrid artifacts contributes the greatest amount of variability to the erratic change of produced technological artifacts. This has been recognized by Koestler, Zwicky, and others who have studied innovation and who attribute it to the chance association of unlikely characteristics of existent artifacts. Chaos, on the other hand, contributes to the restructuring of the whole field of excrescences as it moves to another structuring regime at the advent of the genuine emergent entity, or in the case of the paradigm shift.

Since we are dealing with the particular area of Software Engineering Technology, it is proper to illustrate the foregoing abstract analysis of the relation between the formal-structural system and its ontological grounds with an example form this specific technological area. In order to facilitate this, let us analyze the relationship between embedded software and the characteristics of technology described by Fandozi in his study Nihilism and Technology. By establishing the relation between software and technology, we automatically create a link between software and nihilism because nihilism is the essence of technology. Nihilism is the face of erratic change within the social system that produces technological artifacts. Nihilism, as a form of differing/deferring, masks the presence of pure immanence. Nihilism exists so that non-nihilistic distinctions may be made. This is the equivalent of the old Zoroastrian moral dichotomy good (Spenta Mainu) and evil (Angra Mainu), within the technological sphere. Just like evil and good, nihilism and the non-nihilistic distinction cannot exist and be seen without each other. They are complementary interdependent opposites.

Fandozi enumerates the following characteristics of technology:

110

Wild Software Meta-systems
Technology is pervasive.  It effects all aspects of society.  It effects what we do and how we do it in all spheres of life.  It profoundly changes the lifeworld by altering structures of everyday behavior.  Our structures of behavior are conditioned by the artifacts that we produce and that operate within our environment.  These artifacts, such as telephones, radios, computers, automobiles, planes, etc., have a synergistic compounding effect.  They work together to create a complex technical environment which we work with everyday.  This interlocking of the aspects of the technical environment is what gives it a deep pervasiveness.  Think of the launch of the Hubble telescope satellite.  Here all the artifacts mentioned above, and many more, are used together, each fulfilling a particular function to bring about a successful launch.

And what is it that allows this synergistic effect of combined technologies to occur besides human coordination?  For the most part it is software embedded in various artifacts which allows the various technical artifacts to work together.  The technical environment is knit together with embedded software so that different machines can work together efficiently.  The deep pervasiveness of the interlocking technical environment is made possible by embedded software.  That environment is symbiotic with the technical human community.  It is a socio-technical system.  It is software that even further deepens that symbiosis.  The technical environment forms a continuous whole within which the human being functions.  The cockpit of a jet is a good example of this.  It is software that makes the technical environment whole, and also adapts it to the human whose behavior is adapted to the technical system.  Without software, the pervasiveness of the technical environment remains superficial.  The picture is of an environment filled with many different machines that do not work together except by human coordinated use.  These machines are not adapted to the human, but instead, it is the human that must adapt to each individual machine.  The deep pervasiveness allowed by embedded software occurs because the various machines can cooperate without human intervention, and because the machines can dynamically adapt themselves to the user.

It is thus clear that technology is pervasive, not just because wherever we turn we find ourselves falling over some machine.  It is pervasive because of coordinated use of machines so that the whole work structure is determined by the necessities of coordinating machines.  The human coordination of machines, as in a production line, is limited by natural human capacities.  Software has the potential to supplement and take over some of the work of coordinating machines.  This kind of software is called the Command and Control system.  Embedded software in each machine makes it able to be controlled, and for its current state to be sensed.  Embedded software, and command and control software, working together, allow a whole new level of adaptive and interlocking coordination far beyond what is humanly possible for complex technical environments.  Thus, we see that software allows an intensification of the pervasiveness of the technical system by allowing adaptation and interconnection in ways that are not possible without software.  This intensification of the pervasiveness of technology, through increased levels of coordination, may be deemed meta-technical.

Technology tends toward autonomy.  This is to say that technology has its own agenda which supersedes that of user and maker of the technology.  The agenda of technology is increased movement toward total control of everything, including the human that created or uses the technology.  The Parable of the Tribes expresses the

81.  See Bib#.
82.  See Bib#.
Dialectic out of which the autonomy of technology arises. If any tribe achieves a technological advantage and uses it against its neighbors, then all the tribes are forced to adopt that technological way of doing things or be dominated. Domination means the loss of autonomy. Colonization is just this type of domination through technological advantage. Under domination, the adoption of the dominant technology becomes necessary for different reasons. Only those tribes who adapt to technological dominance by adopting the technology which gives advantage can remain autonomous. By this dialectical relation between competing tribes, technology begins to evolve. This mutation of techniques takes on the appearance of a Darwinian evolution. In fact, the Darwinian model fits technology far better than it fits biological evolution. In biology, there is no mutation of one species to produce a wholly new species to be found in nature, even though this is demanded by the theory. There is only variance within species. The Darwinian model does not explain the discontinuous nature of biological evolution well. On the other hand, studies of the evolution of technology show that the Darwinian theory fits very well what happens when technologies evolve into newer technologies. The parable of the tribes is the story of the survival of -- not the fittest -- but those with the technological advantage. Over the long process of human history, since the Kurgan invasion of the whole of the known world in about 4000 BC, this technological advantage (in the case of the Kurgan people it was the use of the horse in warfare) has become recognized as the key element in economic and political warfare. Thus as time goes on, we recognize technology as an almost autonomous driver of human history. It gives unfair advantage to those who are at the technological cutting edge of their times. Everyone contributes to the growth of technology whether they like it or not. It is an out-of-control positive feedback spiral.

However, this autonomy of technology, which arises from the struggle for power between peoples and nations, where everyone contributes to the growth of technology, so that the agenda of technology is raised above everyone's agenda -- this is only the superficial autonomy of technology. The deeper type of autonomy is that given to technology by the presence of computing machinery and the software that controls it and its peripherals. The autonomy of independent artificial processors is a truer autonomy. Machines have always acted by themselves via a transformation of energy. Feedback circuits have allowed machines to be self-regulating. However, through the addition of computer control, machines achieve a degree of self-regulation which senses the environment and reacts in ways similar to an organism. This autonomy is set up by the human being because self-generation machines have not been produced yet. But once the machine has been set up, it can be programmed to sense its environment and react. Satellites are a good example of this sort of robot. Programming is a fundamental part of the “set up” for autonomy. It allows flexible adaptation and response of the robot. It is like a wind-up toy which is wound and then let go to walk about independently. Thus, programming manifests the slavery of the automated technical system to its programmers. But once programmed, the technological complex acts independently of the programmer; it appears in the environment as an autonomous independent agent. This is true even if the programmed system offers a control interface to the user. The user is controlling the autonomy of the programmed technical automaton. The automaton is still acting independently from the user. The point here is that the program user -- a technologically adapted human -- is part of the technological system. He is the wizard of OZ behind the curtain. He is the Sophist or magician that hides behind his sophistry or sleight of hand. The autonomy of the technological system, as an independent actor with its own agenda has appropriated the human slave. The master/slave dialectic of Hegel comes to play here, so that it is impossible to tell which is which. Technology has had its own agenda which has, through economic and political warfare been raised above the agendas of the players of the power game. Through the evolution of computing machines as a general autonomous processor, the technological system has itself achieved autonomous independence through which it can pursue its own agenda in a way hitherto not possible. The human being has been co-opted to serve the agenda of technology and provide it with the autonomy of movement and action as well. The autonomy of action given to technological products intensifies the application of the technological agenda to the world. Software is the mediator between the programmer -- slave/master -- and the automation -- master/slave. It is embedded software that makes the master/slave dialectic possible. Without software as a means of communication between automation and technical human, the actions of machines would remain disjointed and non-adaptive. Through software embedded in computing machines controlling

83. See Bib.
84. See Bib.
85. See ??? [Technology & Warfare]

86. See Bib.
other devices, technology begins to exhibit the kinds of independence which we attribute normally only to organisms. The autonomy of technology before was that of a preeminent cultural artifact. Now that artifact is automated and acting independently in the environment. The first killing of a human by a robot occurred in Japan not many years ago. Who is the murderer? An environment filled with autonomous technological artifacts is certainly different from one which is just filled machines of various types, closely watched by human manipulators. The autonomy of technology has deepened considerably with sensors and servo-mechanisms controlled by embedded software. The autonomy of the technological complex has deepened. This deepening, dependent on the presence of software and independent computing devices, may be described as meta-technical. The autonomy of the technological agenda has been supplemented with the true autonomy of the technological artifact.

Technology is repressive. This is a corollary of the surfacing of the technological agenda. Because the agenda of technology surfaces and comes to dominate the agendas of the players of the power game, all other agendas ultimately become submerged. The human diversity becomes subservient to the technological system and all features that are of no technical advantage are devalued. As such, they slowly become identified as hindrances to efficiency. This is happening to language barriers in the European Economic Community. These barriers cause joint technical projects to cost five times what the same project would cost by speakers of a single language. These human differences are inefficient, and thus, despite claims of cultural worth, English is becoming the technical language of choice everywhere in the world. The dominance of English is repressive because it causes other languages not to be used, and other kinds of human diversity connected to language become threatened. The success of American media that fosters English detracts from local cultural media products. Technology as a cultural system only accepts a very narrow band of human behavior. All other human behaviors are irrelevant to the technological society, and as such, they face extinction.

This repression of technology, which rejects inefficient or noncontributory human diversity, is a surface phenomenon. In many places around the world, this repression takes on a more ominous character as the police state using technical weaponry against its own citizens. Political prisoners all over the world are held enthralled by the sinister face of technological gadgetry for eavesdropping, information collection, processing and control, as well as torture. Most of this activity is aimed at controlling diverse human populations so they will be docile markets and work forces which do not threaten the stability of the dominant economic and political systems that foster the technological society. In many countries, the police state becomes identical with the technological system, and the sensing and information processing becomes essential control mechanisms. Without software, this machinery of repression could not blanket entire societies. Software is essential to the expanded control of the police state. Yet, there were police states much longer than there were the elaborate technical apparatuses that they use today. In fact, the police state first appeared as the Catholic church used the Inquisition to stamp out the Cathar heresy.

On a deeper level, repression of individual differences becomes the channeling of the human being into a technological mold. This is done by the education system. Technical societies require a highly educated, docile population. This is why Japan is so successful as a technical society. American individualism was good to elevate the technical agenda through the great strides of early innovation. But a technical society must repress the individual differences so that individualism becomes inefficient in the long run. Education is a kind of programming of youth that makes them docile and amicable to serving the technological system. It teaches them the doctrine of relativism which disarms all independent thought and action. Relativism is the positive face of nihilism. It is nihilism whose negative aspects have not yet become apparent. Relativism is the equivalent of religious dogma for the technocrats of the technological society. Education is an early repression of individual differences in order to mold good managers and workers.

Slowly, computers are entering education curriculums. Computer-based education materials are starting to be developed. Educational software is an expanding field. This kind of software makes learning without teachers or books possible. The educational program lets self-paced learning occur. It allows training to take place in a simulated environment. This type of software for teaching shows a completely different aspect of the usefulness of

87. See Bib#
88. See Bib#
89. See Bib#
90. See Bib#
software. It is universally regarded as a positive and fruitful aspect of software development. No one sees educational software as repressive. It is the means of turning talents into skills.

“Repressiveness” is a word with negative connotations. A better word might be “channeling.” Technology causes channeling. It channels the diverse human populations into a narrow range of alternatives. Those who resist this channeling are either controlled by the police state or join the ranks of the unemployed. Channeling, when directed at the young, is called education. Software takes educational channeling to a new level of sophistication. Education becomes a game. Education becomes adapted to individual needs and differences, allowing one-to-one teaching again by automated means. Computerized education is a channeling environment which prepares the student for control by making him docile in a computer-controlled environment. Thus, educational software allows us to see the deeper nature of technological repression. This repression of human differences is inherently meta-technical.

Technology is anonymous and tends to conceal its own nature. We do not think of ourselves as living in a culture controlled by technology, even though the artificial environment is everywhere around us. We see the artificial environment as a means to furthering human ends. Human actors that own and control technological means are what we see around us. The technological equipment are stage props to the actions of other humans. This is merely another way of saying that technology is not present-at-hand. It is not the focus of our projects unless we are building and maintaining it. Because it is usually not present-at-hand, it fades from view no matter how it clutters the environment. But this concealing of itself is but a prelude to its concealment of its own nature. The nature of things are their essential properties. Technology not only hides itself, but it hides what it really like from us. This is why technology has been part of human life for thousands of years, but it has not been focused upon by human society, except only exceptionally. Our era is one of those exceptions. Technology blends into human action. We see the actions and not the technical means. Humans rarely realized that the technical means had any nature different from the enabled actions. It is only in eras of great technological change that the distinction between the way we do things and the technical means can be made. Thus, the attributes of technology discussed here appear. They are normally hidden. It is only with great difficulty that they have been discerned by philosophers over the centuries. This concealment of “technical essence” or “technical nature” is itself an attribute of the technical. We might call this a meta-attribute of technology. Through the meta-attribute of essence concealment, appears the next level of Being. That has been described in the first section of this essay as Hyper Being. It is the absolute concealment of pure immanence. Software takes its nature from this meta-level of Being. Self concealment of attributes is a meta-technological manifestation. With preceding attributes, the surface phenomenon of technology has been differentiated from the deep meta-technical phenomenon. “Self concealment,” as opposed to the “concealment of nature” are two such levels. Fandozi recognizes both of these levels. He speaks of the anonymity of technology. This is self concealment of what is not present-at-hand. The concealment of concealment is the associated metatechnical attribute. Both of these apply directly to software. Software is almost never seen directly. Normally only its effects are seen. Embedded software even lacks a user interface. It is hidden in the bowels of the machinery. The user might not even know it is there. This software, unlike other kinds of writing, is nearly invisible to everyone except the programmer. When it is seen, it is very difficult to understand. Software is by nature hidden. The elegance of software designs are rarely appreciated. Besides hiding itself, software hides its nature. Software appears as “just a text.” Yet it hides through the operation of multiple perspectives the singularity of pure immanence beyond the event horizon of Derrida’s differing/deferring of differance. Probably no one would agree that software has this inner nature. The nature is hidden. We only see it in our nightmares, as yet another project fails, as a werewolf for which we seek an elusive silver bullet. So technological anonymity is the surface phenomenon that glosses over the meta-technical manifestation of technology hiding its own nature.

Technology emasculates ideology. Human differences are not the only kind that are repressed. Technology also represses ideological differences. Ideologies are the motive forces behind political systems. They occur when a particular set of ideas like freedom, liberty and fraternity become the central rationales of behavior. So it is interesting to note that although the agenda of technology is raised above all others in order to give power to the tribe or state, the state is ultimately a roadblock which must be removed by “progress.” Multinational companies express the agenda of technology more perfectly. The

91. See Bib#
92. See Bib#
corporation is an imaginary person created as a legal fiction. Imaginary persons have no need of ideology. Their behavior is not motivated. Only human beings need motivation. The corporation is the equivalent of the self for the technological system. It is a vortex of human activity, aided by technical means, which forms a system producing surplus value. The corporation is the point at which the economic and technical systems merge. By becoming a legal fictional entity, it is recognized by the state. But the state only serves as a host for the corporation. Its ideology that motivates the political system is not necessary for the technological system. In fact, as the agenda of technology is raised -- and that agenda is the intensification of its attributes -- these human motivators become inefficient, like other human differences. The Soviets are slowly recognizing how their ideology is counter-productive. They are losing technical advantage because they insist on ideological control. Free flow of information and open markets with free movement of people is more conducive to the flourishing of the technological system. The police state is ultimately counter-productive. However, the emasculation of ideology is a surface phenomenon. The deep phenomenon is the ideational phenomenon that produces ideologies. Ideologies are political theories. At a deeper level technology harnesses ideation but devalues the product of ideation which are ideas. Technology is practical. It functions by using the ideational process to produce theories. These theories are embedded in machinery for practical use. The pure ideas are discarded. Technology dwells in the realm of pragmatic intelligence defined by Kant practical reason. The discarding of the pure ideas is what separates science from technology. Science is the opposite of technology in this respect. Science reverses the pure ideas and is only interested in technology as a means of getting at those pure ideas. There is a natural partnership between science and technology because the end products of ideation are useless to technology. Technology uses the ideational process itself. It creates theories embedded in machines rather than free floating -- nonembedded -- theories. An example of an embedded theory is a software design. As shown before in the first part, software is a nonrepresentable theory which serves as a software design. Software design depends on ideation -- which produces illusory continuity -- to give wholeness to the design as it evolves. Without ideation as a capability of the mind to produce a differentiated theory and do mental simulations, the theory would not be a whole. The computing machine is an artifact that externalizes the ideational process. The software artifact, when represented as code, executes at such a speed that an illusory continuity for the processors actions is created. As seen in part one this illusory continuity is crucial to the manifestation of the software design. So at a deeper level technology rejects ideas (the products of ideation) for the process of ideation, which in software, is externalized in the form of the running computer animated by software.

Technology makes everything available. The anonymity of software shows it is not present-at-hand. But this attribute attaches it definitely to the ready-to-hand. “Made available” is equivalent to Being ready-to-hand. Total availability means ready to use to get some project we are focused on done. Unavailable is a hindrance to the technological project. Our conditioning to this attribute is shown by anger at the slightest inconvenience. Availability means the free flow of information, people and goods for immediate consumption. Any barriers to this flow is unacceptable. This free flow shows the likeness of technology to fire which consumes its fuel. The fire consumes resources and lives to produce artifacts which are consumed by the market. The fire is the vortex of activity within the corporation which feeds the frenzy of consumption within the free economy. Free economy makes everything available which fits into the technological system, including itself. Technology propagates its own availability even as it hides itself and its nature. This contradictory set of attributes gives us a hint that technology has its own mode of disclosing. It hides in relation to the present-at-hand, but becomes conspicuous in the ready-to-hand. As another mode of disclosing, we see that technology functions in a different kind of Being. Being is manifestation. Different kinds of manifestation will allow hiding in one mode with exposure in another. There is no meta-technical level associated with this attribute. Technology takes its Being from Heidegger’s strange mixture of Being and Time called “Process Being.”

Technology formalizes and functionalizes the world. Technology is structural and is best represented by the formal-structural system such as that described by Klr. Technology can formalize and functionalize precisely because it is structural. “Structural” means it has the ability to structure. To structure is to impose form. Functionality is an attribute of form. Structure determines the functionality of a form within a set of forms that can be transformed into each other. Functionality is the relation of a form from a structured set of forms to the whole set.
Structuralism handles the discontinuities between the forms of this set. Structuralism builds bridges between forms across these discontinuities. The forms evolve through time, changing into each other. If this process is fast enough, the change of forms appears continuous, and then this is a model of ideation. Form and function is the means by which we understand what is present-at-hand. We concentrate on forms, and apply our functions to transform them. This transformation changes the functions of the form. If that change is radical enough, they are replaced by another structurally-related form. Technology can only formalize because its nature is structural. This means that technology operates on the present-at-hand forms with their functions from the hidden space of the ready-to-hand. If it were not once removed in another mode of Being, it could not operate on forms. Thus, this attribute of technology connects it yet again closely to Process Being.

Software is a meta-technology. Where technology makes everything available and formalizes/functionalizes the world, software does that and more. Meta-technology intensifies technology. Meta-technology “technologizes” technology itself. This becomes apparent in the way computers make math skills more necessary, while at the same time taking away the tedium of the computation. It is no longer necessary to be able to do the computation. Yet is it necessary to know what the end product of the computation means. Without this knowledge “garbage in—garbage out” becomes more than a truism. It describes precisely the situation where there is reams of overly precise data which when processed, is meaningless. When everything is available, then relevance, significance and priority becomes crucial. Thus, education becomes more important. One can no longer do production jobs without basic computer and thus mathematical skills. Thus, while technology makes available, the question becomes “available for what?” What do we do with what has been made available to us? What are the significant facts in our data base? How do we find out? Meta-availability is relevance. Meta-availability is filtering. Software provides the means of automatic filtering of information. It can automatically signal the arrival of an important fact while filtering out irrelevant information. So while software makes more information available as a technology, it also allows filtering of what is available which is a meta-technical aspect that allows the true nature of software to be glimpsed. All we need to do is look at the evolution of abstracting journals to see this exemplified. Now the references of articles are cross-referenced to discern related articles. This has been taken another step where clusters of heavily referenced articles are identified and named as cutting edge subdisciplines. The identification of cutting edge subdisciplines allows a reader to immediately identify the crucial papers published each year by seeing how heavily referenced they are and whether they belong to a cluster of heavily referenced papers. This is a kind of information filtering which should intensify research and the dissemination of important findings.

Software plays a part in the formalization and functionalization of the world. Yet this has already been started by other technologies that foster the creation of formal-structural systems in the world. Software, as a meta-technology, must do more than merely formalize and functionalize. It must intensify the formalization and functionalization in some way to qualify as a meta-technology. The intensification of formalization occurs through the operationalization of formal languages. Formal languages are used to write software modules called functions that take inputs and transform them into outputs. Thus, in some way formalization and functionalization is epitomized by software. Software must have its input in a certain form to make use of it. Each piece of software fulfills a specific function. It demands a certain set of formalized behaviors on the part of the user in order to work properly. Installation must be done exactly right. The program accepts a certain limited range of user behaviors and responds consistently to the instilled behavior patterns associated with its user interface. Software simulates the world, and by running its simulation within an environment, it sets the pace and conditions the environment. So software allows the formalization and functionalization of the world to be automated. The automation is a simulation of some aspect of the world which ends up driving the world and causes behaviors of people to change to accommodate the simulation. By epitomizing formalization and functionalization, and confronting individuals in everyday life with formal and functional responses, it causes those individuals to change their behavior and conform. The result of conformity is that the individual enters into the simulated or artificial environment of the complex dynamic software artifact. Our vision of TRON being sucked into the computer is becoming fundamental mythology that haunts everyone who fails to get a good credit rating. The software has a replica of us which, in some sense, controls our basic economic behavior. The

95. See Bib9
96. The romantic Walt Disney movie about the hacker that is sucked into the computer and forced to play computer games in virtual space like a gladiator.
stock trader who does computer trading has an active replica -- called, strangely enough, a daemon\(^97\) -- that watches for the price of a stock to change beyond a certain threshold, and it automatically buys or sells based on that movement. Software allows us to enter the simulated artificial world, and that artificial world get mixed up with the tangible world of everyday life. The end result is a confusion about what is real. Fortunes are made and lost, not on paper, but within the electronic media of the computer circuits. Software makes the world a simulation. Through this analysis of Fandozi’s attributes of technology, we have defined a set of meta-technological attributes related specifically to the role of software in the world today.

The meta-technical (like software) is not just pervasive, but interlocking and adaptive; not just tending toward autonomy, but truly autonomous; not just repressive, but channeling; not just concealing (anonymity) its nature, but hiding a singularity of pure immanence; not just emasculating ideology, but harnessing ideation; not just making all available, but filtering for relevance; not just formalization and functionalization, but simulating in an artificial environment.

Recognizing the meta-technical in the midst of the technological milieu is very difficult. Yet it is necessary in order to respond appropriately. The meta-technical, as exemplified by software, is an interlocking and adaptive environment composed of truly autonomous “daemons” which channels behavior by harnessing ideation and filtering relevant information to produce a simulated artificial reality that hides a singularity of pure immanence. On the other hand, the technological exemplified by the formal-structural system is pervasive yet anonymous. It is repressive and emasculates ideology. It tends toward autonomy while making everything available. It conceals its nature while pursuing the agenda of formalizing and functionalizing the world.

Technology is intensified and guided by the meta-technical. Yet at the heart of the meta-technical is a phenomenon of cancellation. The meta-technology is the limit of technology.\(^98\) Upon reaching that limit, technology turns into its opposite which is the proto-technology of Wild Being.\(^99\) This is exemplified by the rise of interest in acupuncture\(^100\) and homeopathy\(^101\) and other types of “alternative traditional” medicines\(^102\) The meta-technical is like a veil which encounters the limits of technology and sees those limits clearly.\(^103\) From this encounter comes alternative or appropriate technologies. As one passes through those limits, there then appears proto-technologies which harken back to archaic technologies. In our time, each of these approaches to technology are flourishing side by side. Our culture as a whole favors technological means until the side effects become unbearable. Then small is suddenly beautiful,\(^104\) and we opt for appropriate alternative technologies.\(^105\)

Thus, we all use leaded fuel until it starts to appear in cow’s milk which causes it to go into our bodies from which it does not emerge. Then suddenly we are environmentalists attempting to manage the exploitation and pollution of the environment enough so that it does not effect us. When even management fails, we then become proto-technologists attempting not just to find alternatives, but to go back to archaic technologies that were hitherto rejected alternatives.\(^106\) No one tries acupuncture until all other forms of medicine fail. It is a last resort. In ancient China it used to be the first resort.\(^107\) When drugs fail, we try stress reduction and other health maintenance strategies. When these fail, we attempt to learn what the ancients know that we have forgotten. This harkening back to Shamanism,\(^108\) which is now called “New Age Thought,” is a reconstruction and not the original.\(^109\) The original has been lost, and only the archeological remains are left.\(^110\) The proto-technical is a kind of nostalgia for what has been lost.\(^111\)

In software engineering, we encounter the limits of technology by dealing with complex representations of systems we cannot see, which are nonintuitive and may act in counter-intuitive fashions which are hard to explain. We work in an environment where our tools are software artifacts that help us build software artifacts. These software artifacts are simulations of parts of the world which must be simulated themselves in order to know if they work correctly. This simulation is done with test software which simulates the environment of the simulation to see whether the built software will track
what is happening in the environment correctly. The software simulates without actually knowing anything about the environment. Whatever knowledge is not coded into the software product is thrown away. The attempt to regain and use this discarded knowledge is called “knowledge engineering.” Knowledge engineering is proto-technical in that it attempts to get at the knowledge in the human being that allows him to know what is significant. It is an attempt to code the knowledge into a software artifact that will not forget it. Software forgets knowledge and preserves behavior. Knowledgeware sacrifices behavioral coherence for knowledge preservation. Passing through the event horizon of differing/deferring, the meta-technical realm collapses. The singularity is not seen, but disappears. What is left after the collapse is the proto-technical. In software, this is the knowledge of the builder of the software. Automatic code generation is a proto-technical activity. Software reuse is a proto-technical activity. All activities which attempt to transcend the “software problem” are inherently proto-technical and deal with knowledge acquisition, preservation, use. Software is trapped at the behavioral level. It ultimately lacks the knowledge that would allow complete adaptation to its environment. It is the human expert that best exemplifies this total adaptation to the environment. In the proto-technical state, software attempts to become its own designer by replacing the expert designer. Software has become too complex for the human being. Now software defines software. The human reenters the arena of ignorance about software. Everyone is a user. The software engineer merely uses the software construction tools rather than their end products. At that point, we will have passed through the barrier of software which was too difficult for humans to handle directly. Software, with expert design knowledge, must handle software. Software is in this way destined to become a non-human artifact. Humans built software. Software extracted the knowledge of how it was built. It then started generating itself, and humans became excluded because it was too complex and costly for humans to handle the building process. This sounds like a science fiction scenario. Yet already code generating systems have been shown effective on simple problems. It is the nature of the meta-technical to collapse and push those who experience the limits of technology beyond technology into the proto-technical realm. In this, we do not solve the problem of software because we are the problem. When we are excluded and pushed into the proto-technical realm, we as a problem, have been solved by the technical system anonymously seeking autonomy. We have met the software problem, and it was us -- our incapacities and our limitations as human beings.

The essence of technology is nihilism. This essence can be seen in the intensification of the attributes of technology in the guise of the meta-technical. Thus, the attributes of the meta-technical must be the attributes of nihilism which is the summary of this inner core of technology that is exemplified by software. Nihilism has two aspects. There is the passive aspect which occurs when nihilistic opposites are produced that conceptually lead to cancellation. This cancellation of nihilistic opposites causes an entropic loss of meaning in the world. When nihilism becomes active, meaning is destroyed purposely. Active nihilism can properly be characterized as Mithraic and Manicheistic (i.e., heretical) and ultimately rooted in Zoroastrianism. Its roots are deep in history, and this is a virulent disease to which Western culture is all too prone. In the realm of the meta-technical, we are dealing only with passive nihilism which has been called previously excrescence (abnormal growth) or artificial emergence. This is a dynamic of the automated formal-structural system by which change is manufactured. This manufactured erratic change creates the possibility of seeing the formal-structural system within its environment. This has been called minimal change or disorderly motion.

Without this type of basic erratic change (a kind of Brownian motion for systems), no other changes would be visible. Since we are usually only interested in the changes made visible, this type of minimal change of formal-structural systems normally goes unnoticed. In software, this minimal change is seen in such phenomena as software bugs, designs thrashing between alternatives which are both bad for different reasons, and contradictory requirements. These are all phenomena that are endemic to software where changes occur which make other changes in which we are more interested invisible. It is the attributes of the meta-technical that make excrescence occur in the technical arena. Excrulence allows the formal-structural system to be seen on a ground of many different systems. Excrurence is a phenomenon located at the event horizon, surrounding the singularity of pure immanence. At that event horizon there is a constant shifting between possible states of the formal-structural system. It is constantly trying new permutations of its status that will allow greater interlocking or closer adaptability. This is an autonomous channeling function that acts as a teleononic filter as described by Monod.
It channels the system into narrower and narrower evolutionary changes in the interaction of the system with its environment. This excrescence appears as bursts of ideational relevance called newness. The novelty is purely artificial. It calls to the fore new figure/ground relations without changing the essential structure of relations within the overall gestalt. The scintillation of the event horizon where this phenomena is generated is the very stuff of which the artificially simulated world is made. Each spark of scintillation lights up the darkness of Plato’s cave. The sophists (sometimes called programmers) carry chairs and everyday objects (design elements) which cast shadows (sometimes called graphical interfaces). The world behind the glass of the computer screen is more alluring than that behind department store windows. The former moves by firing an electron gun at us. The activated phosphor dot takes a short time to fade. This shower of sparks in our direction is just enough of an indication to produce all sorts of imagined images upon which our preconceived ideas are projected. Ideation uses the illusory continuity of the moving display image as a springboard for its activity. We work all day with computers, then relax by watching TV or going to a movie. We enter wholly into the media-simulated environment. Its power to intrigue us is the ability to constantly produce slightly modified stimulation to which we become addicted. If the stimulation wasn’t always slightly different, we would quickly lose interest. Interest is stimulated relevance produced as excrescence by a dynamically permutating formal-structural system. When we lose interest because the image in the illusion is no longer relevant, then we break out, momentarily, from the confines of the cave. Real people and things are not nearly as interesting as the simulated world. In real life, if you wreok the car, it costs a bundle. In the simulation, you lose a few points. In the simulation, it is possible to do all the interesting exceptional things we avoid in the “real world.” But when the simulator is hooked up to the “real controls,” different rules apply even though it may be hard to tell the difference because the interface is virtually or actually the same.

When the point of cancellation is passed, the erratic changes turn into chaos. This is, in effect, raising the formal-structural system to higher energy levels in which its system states proliferate by bifurcation. The formal structural system produces nihilistic (i.e., self-cancelling) opposites. The self-cancelling opposites appear as bifurcation states embedded within chaos. Chaos occurs whenever a third thing is present to disturb the symmetry of the bifurcations. Each bifurcation represents a nihilistic opposite. Even within chaos, there occur clear spaces where the chaos abates and the bifurcation appears clearly again before it is again overwhelmed by chaos. The arising of nihilistic opposites out of the chaos of images clears the way for the third thing to appear, yet again as the embodiment of excrescence. The manifestation of the third thing breaks the bifurcation, and it is immediately overwhelmed again by chaotic states. In our culture, the production of simplistic self-cancelling opposite positions for the manipulation of “media-ignorant” masses has become a high art. These nihilistic opposites appear to solve the problems of the complex chaotic environment, giving a false sense of clarity. This false clarity of bifurcations arising from the chaos only appears to be broken by the arising of a third position dialectically related to the nihilistic opposites. This dialectical synthesis appears to be a “new thing,” but in fact it is the advent of a new wave of chaos which will give rise yet again to the production of false opposites. This dialectic within the chaotic field, moving from chaos to nihilistic opposites to the advent of the “new” synthesis of a third thing which brings again chaos, is the nature of a chaotically bound system. The chaotic substructure forms the basis of erratic change upon which the fuzzy, probabilistic and deterministic layers are built.

Understanding how nihilism and erratic change manifest from the chaotic substrata is important. The meta-technical arises from the proto-technical and brings possibilities into play adding to the chaotic dynamics. Possibilities are turned into actualizations that are properly stochastic by the action of chaotic dialectics.
The chaotic field represents a myriad of possible states that the system might assume. The arising of the third thing causes the system to leave the regime of bifurcation and reenter chaos. This is related to the point where chaos clears and a single state arises, which leads to bifurcation again. The clearing of the chaos and the arising of a third thing to bring back chaos, are related possibilities. This is again like the arising and cancellation of the virtual particle discussed in part one. There it was seen how the different kinds of Being participated in the manifestation of this theoretical complex made necessary by the infinite energy of space-time.

The exit from the chaos is an epiphany which is like a paradigm shift where a single way of looking at the world suddenly becomes dominant. This single track cannot maintain itself without splitting into nihilistic self-cancelling opposite states. The system will oscillate in these until suddenly the two positions themselves bifurcate. As bifurcation continues, the set of nihilistic opposites begins to have structure. The nihilistic opposites appear to arise out of nowhere. They are possibilities being actualized. They appear upon the background of chaos. They arise as figures on that ground, and the dynamic formal-structural system randomly pops from one nihilistic state to another. Thus, the actualization of possibilities includes the arising of the possibility out of the infinite possibilities of chaos and the random entry and exit of the possible states which have arisen.

Being is the current determinate state of the system.
Being² is the probabilistic movement of the system between its various actualized states. Being³ is the arising of actualized states from the chaotic background. The reentry of the system into chaos is also a function of Being³. It represents the cancellation of nihilistic opposites and their being overwhelmed by chaos. Being⁴ appears as the epiphany at exit from chaos and as the arising of a third thing (synthesis) which causes reentry into chaos.

The formal-structural system has its basis within the clearing of abated chaos. It arises as the infrastructure of bifurcated states. Bifurcations have a specific structure which allows the formalization of the relations between bifurcated states. This formal-structure allows the discontinuities between dynamical states to be bridged. Formalisms embody the image of a particular state. Structuralism comprehends the transmutation from dynamic state to dynamic state within the range of possible states. Formalism and structuralism arise within the abated chaos, perhaps analogous to what Heidegger calls the “clearing-in-being.” The formal-structural system that describes the bifurcation states of the dynamical system has its mathematical basis in topoi, which are sheaves of boolean systems. It has been shown that there is an inherent relation between topoi and fuzzy systems. Computing devices make use of this same boolean mathematical substructure as the electronic basis of software. Software exists in the boolean space, and makes use of the structure of boolean systems to model more abstract conceptual forms. Thus, there is a direct relation between bifurcation spaces and boolean discrete mathematical spaces where bit patterns model images of system states. In both cases, it is the substructure of binary systems which is the foundation of formal structural systems. This is an important perspective because it shows the deep relationship between the formal-structural system and both the bifurcation space within chaotic dynamics and the boolean mathematical traces upon which the semiotics of software is based.

As an aside, it is important to note that we cannot see past the veil of chaos which hides the origin of the dynamical formal-structural system. This origin is always hidden. Proto-technology and proto-science constantly attempt to piece together the archeological evidence and reconstruct the original archaic science beyond the veil of history. Proto-science and proto-technology is a nostalgic attempt to regain the past which is doomed to failure because the chaotic veil can never be lifted to see the “primal scene” of the true origin. Thus, the efforts of cultural anthropologists to reconstruct the archaic science and technology as a crude reflection of our own science and technology as Levi Strauss has done, in the end only emphasizes our own failures. There is, however, another route to the archeology of archaic knowledge before the arising of chaos. Taking this route necessitates a radical reappraisal of our own superiority in relation to other intellectual traditions. The fact is that the Western intellectual tradition is a diversion from the archaic traditions which it cannot understand because it refuses to

---

114. See ??? (N?)
115. See Bib#
116. See Bib#
see its own inherent inferiority. This blindness amounts to a hubris which is always ultimately punished by the “gods.” An excellent portrayal of this hubris, in relation to the archaic medical tradition of China, is given by Bruce Holbrook in the Stone Monkey.¹¹⁷ The attempt to understand archaic sciences in their own terms¹¹⁸ rather than as nostalgic proto-sciences is just the beginning. In order to participate in this adventure, it is necessary to be prepared to venture outside Being. For the descendants of the Indo-European Kurgen people, who are still enthralled in that subtle clinging, this is very difficult.

This excursion into the depths of technology is relevant because software methodologists are also technologists. We must deal with technology and learn to recognize its intensification into meta-technical facets. This perspective causes us to approach the definition of the formal-structural system a little less naively than we might have otherwise. We bring back from the philosophical substrata some insights into the significance of the formal-structural system that we can apply to understanding how software (meta-technical) perspectives alter our view of the formal-structural system (technical) as defined so succinctly by Klir.

In the foregoing, we have discussed in-depth the substrata upon which the formal-structural system is based. Now we will attempt to enter a particularly robust static representation of a formal-structural system provided by George Klir. We will enter, like visitors, into a medieval cathedral. As visitors, we cannot study every detail, but only get an overall impression. The formal-structural system is the cathedral of the Twentieth Century Western intellectual tradition. Unlike Chartres, it is not built of stone. Instead it is an ideational structure upon which all of Western science and technology is founded. Like medieval cathedrals, there are many different versions of the formal-structural system. We have selected one with the most parts intact. Tourists go to Chartres because the original glass is intact, and we can see the relation between the images in stone and the images in glass as the medieval peasant might well have done. In this case, the stonework is like the technological system, and the patterns in glass are like the scientific super structure. In the cathedral, the local interpretation of scripture was read off the walls and windows by the illiterate. In the formal-structured system, each discipline has its own variations on the same themes. The technologically illiterate are dazzled by the spectacle of the high priests. As Nietzsche says, the “last man” blinks. In later ages, these different kinds of formal-structural system representations may be studied by intellectual historians, like art historians study the cathedrals. When they were built too high, cathedrals collapsed. High Gothic had its limits; so too, these intellectual edifices will eventually realize their own inherent limitations. Until then, they are the main means for understanding everything in the world. Unfortunately, they are built on the shifting sands of fragmented Being. Thus, it is possible for the entire edifice to collapse at any moment. So we had better make our tour quickly before any tremor might occur.

A quick overview reveals that the edifice of this particular structural system contains several layers called epistemological levels. The first level is the object layer which is the system raised to a figure against the ground of all possible systems. This figure has a set of attributes, and it is the relation between these perceived attributes that constitutes the system. Attributes are compared to what Klir calls backdrops -- the name for very general environmental attributes whose values vary continuously systemwide. The object system obviously entails the subject/object dichotomy, and all that this implies in traditional philosophical discourse. In other words, the object system is assumed by Klir to be present-at-hand. The source system is a general representation of the object system prepared for study. Attributes become variables and, “backdrops” become “supports.” In other words, observational channels are set up for studying the object system. This instrumentation for objective measurement becomes the source for data about the object system. The source system is the experimental set-up. It becomes the data system when measurement, according to scientific protocol, begins. When enough data has been taken and analyzed so that the rules governing data generation have been discovered, then the data system may be compared to the generative system that simulates the object system. The generative system is often composed of different simulating models which work at different times. The models may generate data adequately when the dynamical system is in a particular dynamic bifurcation state. When the dynamic system pops to another bifurcation state, the model must be changed. Models are formal and generally generate linear behavior that simulates the system under study. Changing models brings out the relation between models which is the structure underlying the system. Note that structure handles discontinuous changes in the data.

---

¹¹⁷. See Bib#
¹¹⁸. See Bib#
Nonlinearity is mapped into structural concepts. This is exactly the relation between the formal and structural aspects of the formal-structural system. Meta-levels of relations between relations form an infinite regress of finer and finer structural systems modeling. In fact, meta-systems and structure-systems regress infinitely, forming two horns that arise from the basic source, data and generative systems. The meta-systems are an infinite progression of models-of models-of models . . . Whereas the structure systems form an infinite progression of relations-between relations-between relations. . . Klir has well represented the relations between these two infinite regresses which always occur to purely formal systems of the type first created by Russell and Whitehead in *Principia Mathematica*. 119

Each bifurcation creates another meta/structural level of analysis. Eventually these become impractical to understand, and the formal structural system, with too many meta-levels, self destructs as it is overwhelmed by chaos. A model is necessary for each bifurcation state. When a bifurcation state bifurcates again, a meta-model must be abstracted to cover both models covering the new bifurcation states. The meta-model explains how the two new states relate to the old single model. Models are composed of entities and their relations. Structural relations between models allow the explanation of discontinuous jumps between submodels. Each level of bifurcation demands another infra-structural level to explain the new sets of discontinuities between models. Thus, structure represents the increase of fragmentation, whereas meta-levels represent the increasing levels of abstraction necessary to hold the system together as a single object. Meta-levels represent analysis. Bifurcation drives the need for ever deeper levels of synthesis and analysis. Ultimately, the human mind, even with the aid of computers, can only stand a few of these meta levels of embedding. As bifurcation progresses, the analyst is quickly driven beyond his capacity to model, and the whole formal-structural system cracks up. The whole representation cancels itself out, and one is forced to start over looking for a new, more robust model.

To this static representation of the formal-structural system produced by Klir, we bring a demand which lies outside of his purview. We need to represent the perspectives that have been found to be an important element of the meta-technical. Fandozi, in his discussion of technology, almost equates perspectivalism with nihilism.

Although Marcuse’s analysis was concerned primarily with the theoretical background of science, it is important to see that there emerges in his results a concept of reality which is the correlate of the scientific endeavors and which lets the world appear as perspectival.* Entities in the world no longer possess a definite nature as such, but are rather defined by their context, e.g., a tree is an obstacle for the new freeway, potential lumber for a house, or an aspect of aesthetic appreciation. The destructiveness of technology is not just based on the fact that lumber takes priority over aesthetics, but that any perspective as an interpretation of a thing becomes equally possible, equally cogent. A thing is defined according to its use, and that use itself is a function of the current level of technology and other extraneous factors - factors which do not directly confront the thing in question. The thing itself is not allowed to address

---

119. See Bib#.
This paragraph in many ways shows the relation between the meta-technical perspectives and nihilism better than anything else said here. It is perspectives which hide the singularity. It is the equi-possibility of perspectives which constitute their nihilistic aspect. So, as has been shown in many ways in this essay, possibility is married to passive nihilism of the meta-technical. Perspectivalism and possibility are intimately related. We see new possibilities when we switch perspectives. Those possibilities are hidden, and new ones appear as we switch perspectives again. Possibilities are the showing and hiding avenues related to perspectives. Yet, this showing and hiding is deeper than that of the ready-to-hand. It is the showing and hiding which is inherently social. Intersubjectivity, the famed problem of phenomenologists, appears here. Intersubjectivity hides the “essence of manifestation” which amounts to the singularity of pure immanence. In intersubjectivity, there is something hidden irrevocably that may never be shown. It is embedded in the interchange of perspectives. It is that arbitrary interchange that causes passive nihilism to eat up meaning. Sartre calls it “nausea” as he contemplates the waiter in the cafe. Most philosophical analysis -- socially naive -- concentrates on the self/other dichotomy. More sophisticated analysis realizes that this is an abstraction from more complex social interaction.

Yet it is possible to hypothesize that perspectives bifurcate just like dynamical system states on the way to chaos. In our case, we wish to maintain our analysis at the threshold of 2. We recognize that the chaos of social interaction is primary. The individual appears for our culture as an isolated ego from this milieu. He experiences bifurcation at separation from the state of immersion in the presence of the mother. Mother, in time, becomes mother/father, and ego becomes self/other (I/thou). However, we are not dealing with these basic archetypal psycho-social dichotomies. Here it is a more abstract analysis which will consider the perspectival fragmentation of the formal-structural system. A set of four specific perspectives to be suggested here are relevant to the analysis of software systems. Yet the point being made is more fundamental.

The perspectives form a minimal system in the sense defined by B. Fuller in Synergetics I and II. It is the minimal system of perspectives that is complex enough to hide the singularity. Most analysis of formal-structural systems use only one or two perspectives. They do not attain the required complexity necessary to represent meta-technological systems. The advance suggested here is to consider how the formal-structural system is distorted by the introduction of a minimal system of perspectives. This is a concrete example of the intensification of the technical into the meta-technical. In effect, we are introducing the equivalent of relativistic intervals into the space-time continuum within which the formal-structural system exists.

One might well ask at this point what a perspective really is. An excellent semiotic study concerning this issue is Signifying Nothing. In that study the structural equivalence between “the number zero” -- “vanishing point” -- “money” is suggested. Suffice it to say for us that a perspective is the cutting of the world into multiple related horizons. The intersection of the structures that define these horizons is a null point which is “nowhere” in relation to those defining structures. This “nowhere” is represented by the vanishing point within the horizon. By signifying the vanishing point, the opposite observation point may be specified. Thus, the perspectival structure gives a means of orderly movement from observation point to observation point by moving the vanishing point. The multiple horizons reveal different realms of possibility open to exploration. What is open to exploration from one perspective is closed to another. This only gets complex when different subjects are simultaneously exploring different perspectives. Then
relativistic rules apply between simultaneously exploring subjects. Perspectives can either be quantitatively different in terms of space, time, position; or qualitatively different. Qualitatively different perspectives are the kind Fandozi referred to earlier. From qualitatively different perspectives, one sees different kinds of timings instead of different sides of the same things. Qualitatively different perspectives are much harder to deal with. Since we are dealing with a specific example of meta-technology, i.e. software, this problem is not as difficult as it might seem. In fact, we will postulate, as explained in part one of this essay, that it is the foundations of the technological system that provide the basis for distinguishing between the quantitative perspectives relevant to software. Two of these are quantitative and related to space/time. The other two are qualitative and related to the first and second meta-levels of Being. The quality to be distinguished is the present-at-hand from the ready-to-hand. The Western philosophical tradition contributes amply to the possibility of this discrimination. Now how these four perspectives (named here: event, data, agent, function) unfold has already been laid out. The question now is what the insertion point for these perspectives into Klir’s representation of the formal-structural system are.

According to Klir, the object system is composed of a set of attributes and a set of backdrops. Any one attribute or backdrop has a set of possible values whose range is circumscribed. Klir maintains that a position of ontological ignorance may be maintained concerning the “reality” of attributes and backdrops. This is possible because the entire “object system” is disposable. It is merely the starting point for the definition of the source system which is an instrumentally defined measuring device levied against the object system. In the source system object attributes translate into operational representations called “variables.” Likewise, object backdrops, i.e. global environmental attributes, translate into operational representations called “supports.” Supports allow different observations of the same variable to be distinguished.

When more than one support is involved, the overall support set is the Cartesian product of the individual supports sets. Properties recognized in each individual support set have to be properly combined to express recognizable properties of this Cartesian product. These properties of the overall support set (the Cartesian product) are then used in characterizing, together with properties of the associated state set, an elementary methodological distinction. If the same properties are recognized in each of the individual support sets, it is easy to combine them, and the derived overall properties are homogeneous over the whole Cartesian product. The situation becomes more difficult when properties recognized in the individual support sets are not the same. In such cases, there are, at best, some overall properties that do not extend over the whole Cartesian product. [My emphasis]

The last two sentences of the quoted paragraph are important for this study as they point the way to understanding the fundamental properties of software methods. When multiple support sets are used which do not share the same mathematical properties, then the Cartesian space represented by these supports is intrinsically fragmented. It turns out that this is true of the supports for the software system. Each of the four fundamental perspectives supply the supports for the

---

126. See ASPS p 37.
127. Klir ASPS page ??
software system. These supports, as noted in part one, are event, data, agent and function. These derive from time, space, pointing and grasping which are the ontological foundation of any formal-structural system. In this case, the environment which is posited for the object system is its own ontological foundations. These foundations provide the essential backdrops for all formal-structural systems. When we drop the pretense of ontological ignorance, it is readily realized that the object system and its environment are the same thing turned inside out. In our case, the object system is the formal-structural system itself. Its only possible environment is its ontological presuppositions. These presuppositions lead to groundlessness. The groundlessness of the ontological environment is nothing but the reflection of the erratic change of the formal-structural system itself. It is these ontological foundations which manifest groundlessness that are the systemwide universals that can be used as backdrops for measuring the attributes of the formal-structural system. As backdrops, they are space/time and Being /Being”, i.e. pointing/grasping. These are turned into supports in the source system which appear as data/
event/agent/function. The transition from “backdrop” to “support” occurs through the articulation of perspectives and their projection back onto the ontological foundations. The ontological foundations become buttresses which allow one to get a view of a portion of the formal-structural system. The key point here is that the place where perspectives plug into Klir’s representation is right at the base where he defines backdrops and supports.

When supports do not share mathematical properties, then the inherent fragmentation of perspectives occurs within the total support set. Different horizons open up to different perspectives due to this fragmentation of the total support set. Klir is very precise about the methodological distinctions that derive from the mathematical structures of the various supports. It turns out that there are a limited number of possibilities for combination of these
mathematical properties as shown in Figure 31.

Each of the supports may be of any of these five methodological distinction types. Which type (a, b, c, d, e in Figure 32) each support is, determines how easily the supports may be combined and the degree of fragmentation that occurs in the total support space. It is possible to assign each of our perspectival supports to a particular methodological distinction as shown in Figure 33.

Space and time are, as any physics student knows, a fully linear ordering in which distance may be described. Without this continuous backdrop, the description of physical systems would be well nigh impossible. The fact that the continuity is itself an illusion projected upon time and space and maintained by the ideational mechanism of the real number line and calculus need hardly be mentioned. Agent and function, on the other hand, are represented by partially ordered sets. They cannot support the concept of either linearity or distance. In software methods, the pure description of each of these perspectival minimal methods is a hierarchy represented as a tree structure. Agents are seen as a tree of tasks (tasks within tasks), and function is seen as a tree of functions (functions within functions). How far a task is from another task makes no sense. Functions are not transitive. It cannot be said whether a function is “between” to other functions or not, unless data flow is added. Functions themselves, without time or space ordering added to them are just a library of routines, some of which call others in a tree-like structure. Thus, agent/function supports are very different from event/data supports in terms of the methodological distinctions that apply to each perspective.
If you ask “who?” or “what?” -- these questions do not refer to continuums but to differences in accidental and essential characteristics. Who you are is dependent on the accidents of your life. Who were your parents. What name they decided to give you. What house was available when you could afford to buy. What number was next when you applied for a social security number. What day you happened to be born on. Who you are is determined by all the accidental circumstances that distinguish you from everyone else. This set of accidental facts that makes you unique is a set which has only, at most, partial ordering. Likewise, all the essential facts about you, like what you believe in or don’t, your skills, your profession, your hobbies, that indicate what kind of person you are, at most, partially ordered. You belong to this set which is a subgroup of another set. The complex Venn diagram that singles out both in terms of accidents and essentials from all others in the population cannot be fully ordered.

Pointing and grasping are modalities for the human encounter with things in the world. The set of things

---

FIGURE 35

autonomous behavior

---

FIGURE 34

data/event
space/time

competing closely coupled
multiple economies
distributed closely
coupled
interconnected
special devices
unix processes

competing hetero-
genous
distributed hetero-
genous
integrated special
devices
multiple interrupt
server

competing homo-
genous
distributed homo-
genous
parallel processors
multi tasking

pointed at are distinguished from those which are not. Likewise, the set of things grasped forms a distinct set over and against what is not grasped. Without adding space/time, at best these can be partially ordered into sets of things pointed to or sets of things grasped. Inherently, there is no distance nor linearity in pointing and grasping. Only hierarchies of distinctions can be supported by these modal concepts.

This realization that different mathematical distinctions apply to the various software methodological supports is a crucial step in understanding the way perspectives are fragmented. The fragmentation has a particular structure which has important implications for software design methods. These implications are best brought out by studying the diagram of methodological distinctions produced by Klir more closely.

There is a great imbalance between agent/function supports and event/data supports. Event/data supports have the benefit of full ordering which will allow any signal or datum to be pinpointed in the linear ordering of memory locations or CPU cycles. We can gauge the distance between any two signals or datum by counting discrete cycles or memory addresses. Thus, it is clear where and when each datum/ipsity (eventity) occurs in the discrete space/time environments provided by computing hardware. The mapping of this full discrete ordering onto an illusory continuity of the space/time environment made possible by the distribution in space of processors and the illusion of continuous execution due to fast instruction processing, allows the computer, with its software, to fit into our picture of the physical universe. Space/time eventities within the computer map to space/time eventities outside the computer. This mapping makes use of the power of the illusory continuity of the real number line to give a four-dimensional continuous support space. This four-dimensional support space (either space/time $[x+y+z-t]$ or Minkowski’s time-space $[\text{past}+\text{present}+\text{future-nowhere}]$) has very powerful representational features. Without this possibility of mapping, simulation spaces onto our idea of the physical universe, computing machines would be useless.

On the other hand, the agent/function supports are weakly ordered. This weak ordering is difficult for us to deal with. We are used to dealing with fully ordered mathematical objects. Sets are not as easy to manipulate as cartesian coordinates. Yet, we recognize that the weak ordering of agent/function has its own significance. Accidents and essences are weakly ordered by their nature. Space and time are like empty vessels for objects to move in. We cannot say anything about the objects themselves using space/time. The essences and accidents that define the objects as more than blank empty entities are what give us really important information about the things gliding through space and time. Essences and accidents give us a glimpse inside the black box. We pay for this glimpse inside the black box. We pay for this glimpse by losing ordering information. We gain by finding out who’s who and what’s what. These partial orderings, with no distance, can be seen as the delineations of kinds, either essential or accidental. In philosophy, these are called “sortals” by some who think it important to distinguish natural complexes from abstractions. We can basically sort things according to external coherences or internal coherences. External coherences relate to the accidental configurations of things that appear in the world. Internal coherences relate to the essential configurations of natural complexes within themselves. The internal coherences give one kind of sorting, while the external coherences lead to a completely different kind of sorting. Klir lumps both together under the concept of “population.” We think of a population as a set of organisms of a similar kind. It is composed of individuals in a particular space-time distribution. But important information is to be gleaned by looking at the internal and external coherences of the set of individuals taken as a whole. The external coherences will classify individuals on extrinsic features such as serial numbering. The internal coherences will classify individuals on the bases of intrinsic characteristics like their different behaviors. These intrinsic and extrinsic characteristics tell us a great deal about the set of individuals that space-time distribution alone would not be able to tell us. Yet, this information is dependent on our ability to distinguish. The most basic essential distinction of an individual is based on behavior. The sortals related to behavior are called functions. The most basic accidental distinction of individuals is based on autonomy. The sortals related to autonomy are called agency. The agent is the source of a behavior. The behavior is the expression of the autonomy of the agent. In the field of computational machines, the behavior of a processor is the transformation of data grasped in accumulators. The autonomy of a processor is the indication of which task is in control by pointing at an execution thread or context. The individual processor, like the minimal organism in a population, has both autonomy and behavior of its own. The concept of population contains both of these concepts which, from the viewpoint

128. See ??? {Natural Complexes}
of software methods, need to be vigorously distinguished.

All of this becomes clearer by realizing that in the course of the development of software, we traverse the set of methodological distinctions (a, b, c, d, e) from no ordering to full ordering of space-time eventities. Requirements are normally without ordering or state some boundary conditions for space time ordering of eventities. As analysis and design progresses, the exact space time ordering system eventities is determined. Software methods need to be able to describe these fully ordered eventities, and do so in terms of events and data perspectives. However, making certain events occur at particular points in space are the purely performance aspects of the system. It is also very important to analyze and design who does what in the system. The “what” characterizes the functionality of the system. That is the particular kind of expected behavior in space-time. The “who” characterizes the division of labor between cooperating agents. The “who” normally determines the structural aspects of the design. The “who” is structural, whereas the “what” is formal. This reversal occurs because point and grasping are mutually sustaining, like space and time. They are complementary opposites that make each other possible. The assignment of autonomy to individuals structures their interaction. The assignment of behaviors to individuals formalizes their actions. In the one case, structure is on the surface while formalism is deep, while in the other case formalism is on the surface while structure is deep. Pointing is the action of an autonomous individual. The deep aspect is the formalism of rendering a present-at-hand image at the point of focus. When this ability is assigned to an autonomous individual, the surface structural aspect becomes apparent where different individuals do different things in cooperation. Grasping is the behavior of an individual. The deep aspect is the structuralism of rendering a ready-to-hand means of moving between temporary gestalt images. When the behavior is assigned to a particular individual, the surface formal aspect appears where a particular individual has a language of transformations he can perform to transform one image into another.

Grasping is the behavior of an individual. The deep aspect is the structuralism of rendering a ready-to-hand means of moving between temporary gestalt images. When the behavior is assigned to a particular individual, the surface formal aspect appears where a particular individual has a language of transformations he can perform to transform one image into another. Structuralism and formalism belong together. Each fills a crucial role in the full articulation of the other. Thus, there appears both formal-structural and structural-formal phases similar to the difference between space-time and time-space. There is “behavioral autonomy” and “autonomous behavior.” The former emphasizes the behavior of the individual over autonomy, whereas the latter emphasizes the autonomy of the individual over behavior. The autonomy of competing processors is greater than the autonomy of tasks. The behavior of closely coupled environments is more complex than that of monolithic environments.129

Traversing the lattice of methodological distinctions (a, b, c, d, e) in the development from requirements to design to code, it is clear that at a certain stage each support is made up of non-ordered sets of distinction. Then at some stage,
all become partially ordered with no distance. In the final product, events and date (space and time) become fully ordered (linear with distance). However, at most, agent and function remain partial orderings. This is a big problem for software engineering. The fact that agent and function separately can, at most be partially ordered without distance causes a major break between the space and time supports and the model supports. This "gap" in the representational power of these two different kinds of perspective causes a rift in the designer’s ability to move from concepts of agents and functions to a final space/time ordering of software eventities. This rift is the major methodological problem of software engineering. Even though it is possible to appreciate that the lack of full ordering for model perspectives allows a fuller expression of internal and external coherences of the software’s functionality and structure, still it is difficult to deal with this rift in practical terms. In the development of software methods, it is this gap which must be addressed. It is, in fact, this intrinsic rift which is the fault line that hides the singularity of pure immanence. In addressing the fracture that separates software “supports,” we are orienting ourselves to the hidden heart of software.

Even though agent and function can each, by itself, only be partially ordered with no distance, together it is possible for them to approach closer to full ordering. In order to attempt to narrow the rift, the two partial orders are used together to imitate linearity without distance (structure chart) and to imitate partial order with distance (mapping). In mapping, distance is expressed in how many boundaries must be crossed from one end point to the other of a mapping arrow. The minimal formal calculus of G. Spencer-Brown in *Laws of Form* may be applied to measure the distance implied by mapping. In structure charts, linearity is expressed as a calling sequence. The control flow of a calling sequence is like passing a token of autonomy through the software system. Autonomy is passed from function to function, and the sequence of calls represents the linear structure of the program.

It is interesting to note that these two ways of using two partial orders together are duals. There is a one-to-one correspondence between their elements. The path of the token of autonomy is analogous to counting boundary crossings in the mapping. The point here is that structure charts and mappings are the two minimal methods that

---

130. See Figures 37A&B.
131. See Bib#.
connect the agent/function perspectives. These minimal methods are much more complex than those that connect the event/data perspectives. There is good reason for this lopsidedness in the complexity of minimal methods in these cases. The minimal methods connecting agent/function must make up for their weak orderings. They do this by working together to attempt to bridge the gap to full ordering. In so doing, two possible gap-bridging paths are articulated which lead to the minimal method of “mapping” and the minimal method of “structure charting.” Note that the inherent linearity of structure charts approaches the representation of linearity in time. The inherent distance of mapping approaches the representation of distance in space. So structure charts are close to the temporal ordering of software eventities, and mappings for traceability are close to the spatial or data orderings of software eventities. Yet being close is not the same as being completely isomorphic. In each case, a crucial ordering characteristic is missing. Yet, if the linearity of space and the distance of time is relaxed as it can be for some classes of systems, e.g. information systems rather than embedded systems, then this attempt at gap bridging is good enough to give full methodological convergence. However, in embedded real-time systems, this relaxation is not possible. So there is a class of software systems in which the rift cannot be hidden. In

FIGURE 42

Kent Palmer
No wonder software is hard to build, and software methods are difficult to use. There is a fundamental rift between what our perspectives can represent in an ordered fashion. This gap is, in principle, unbridgeable, even though for some classes of system the rift appears to vanish. Our most important methods (structure charts and mapping) attempt to bridge this gap. Because of this function, these methods are more complex than any of the other minimal methods that act as bridges between perspectives. Linearizations of the software system, together with many-to-many mappings are the two main ways we have to approach the mental simulation of our systems. Linearizations, through which tokens of autonomy move, express these systems in terms of autonomous behavior. Many to many mappings express the behavioral autonomy of the system. The former shows what part of the system is autonomous at what point in its execution. The latter shows how the overall system behavior is automated by deployment of functionality to different system segments. Autonomous behavior means... what part of the overall system is acting independently. Behavioral autonomy means... how has the behavior been partitioned across autonomous units. These two different aspects, like their counterparts space/time and time/space, are difficult to express. One looks at individual pieces of the system as encapsulating behaviors through which autonomy moves, given different inputs and states. The other looks at the overall system as expressing a set of behaviors and looks to how those behaviors are allocated to the different pieces of the system. This difference can be brought home by referring again to the gestalt. The gestalt is a number of dynamically-related figure-ground relations. Each figure-ground relation expresses a particular kind of behavior of the gestalt whole. The total behavior of the gestalt is the sum total of these relations. Yet, all cannot be seen at once. Our perception moves through the figure-ground relations one at a time. This is like the movement of autonomy through the structure chart. Looked at analytically, we can see behavior distributed among different figures. Looked at synthetically, we actually perceive the shifting from figure to figure as the center of attention changes. Perception snaps from one whole to the next. The background of the gestalt remains always ready-to-hand in relation to the present-at-hand figure. We point at the figure, but we grasp (understand) the whole gestalt which is more than the sum of its parts. Behavioral autonomy means giving autonomy to behaviors via mapping. Autonomous behavior means the automation of the behaviors seen in a particular linearization which moves through each of the figure-ground relationships. In autonomous behavior, the accidents of where control is within the linearized system predominants. In behavioral automation, the essence of the behavior of the whole system predominates.

We have taken our clue for how to deal with the relations of perspectives to the formal-structural system from Klir’s discussion of “backgrounds” (in the object system) and “supports” (in the source system) and the relevance of methodological distinctions. By looking at the different kinds of methodological distinctions that relate to agent/function perspective versus event/data perspectives, we

---

132. If this distinction between Behavioral Autonomy and Autonomous Behavior appears forced, there is good reason. They are nihilistic opposites now seen in action. Don’t worry; they cancel, leaving nothing at all. See Bib#.
have seen the advent of a fundamental rift between these pairs of perspectives. That rift gives us our first concrete evidence of the presence of a singularity within the set of software perspectives. It has also indicated an interesting way of looking at the relations between software methods. Methods arise as means of attempting to bridge the rift between pairs of perspectives that are methodologically distinct in Klir’s sense. Agent/function perspectives are augmented from partial order with no distance to have either linearization or distancing. Event/data are reduced from full ordering to emphasize either linearization or distancing. Thus, the meeting point of methods occurs in the methodological distinctions related to linearity with no distance or partial ordering with distance. Perspective pairs (agent/function or event/data) are more extreme in their ordering than the methods that connect them which exemplify linearization or distancing. This is a key point that makes a deeper understanding of software methods possible. All software methods come in pairs. Each pair relates two software perspectives. One method from a pair emphasizes linearity, while the other emphasizes distancing. The pairs may be thought of as looking from
one perspective toward the other perspective or vice versa. Thus, the mapping method looks at agent from the point of view of function, whereas the structure chart method looks at function from the point of view of agent. The point of view exemplified by the method uses the other perspective as an object. The subject/object dialectic gives each method its own peculiarity: what is a subject for one method is an object for its opposite method. This inversion of subject-object relations is then colored by the emphasis on either linearity or distancing that gives substance to the method pairs. Method pairs may be considered as bridges between particular pairs of perspectives. Methods give the designer a means of traveling from one perspective to another in his mental simulations of a software system. The designer may only be in one particular perspective at a time. Design work is a path of thought which traverses the perspectives one at a time in order to build abstract design representations of a software theory. The software theory is the ultimate synthesis of the software system projected by the designer.
The ultimate synthesis revolves around the subjectivity of the designer and cannot be completely represented. The designer, as subject, assumes the different software perspectives one at a time to get the particular view each offers. The terminology invented by Husserl\textsuperscript{133} in his phenomenology can be used to describe this process. For the designer as subject there is the projection of the “intentional morphe” upon the “hyle” or matter of the software system. In this case, the matter is a pure plenum of off/on bits. The “intentional morphe” is a forming intention that projects the software system. From this fundamental subject/object interaction, “noema” and “noesis” arise. The “noema” in this case are the various levels of software objects like “bit,” “byte,” “counter,” “statement,” “routine,” “program,” or “software systems.” The “noesis” is the various ideational structures such as go to, assignment, if statement, algorithm. The noesis is the thought structures which determine the structuring of the program. The noema is the structural concepts which ultimately, translate into patterns of bits.

Husserl’s terminology gives us a terminological precision for dealing with subjectivity similar to that offered for objects (called systems) by Klir. This terminological precision is required in order to situate our concept of “methods.” Methods are normally understood as sets of techniques used in a particular order to produce abstract representations of software objects. In fact, these representations are particular aspects of a software theory.

Methods are a kind of noesis at a particular level of abstraction which produce a corresponding noematic representation. The level of abstraction at which software methods arise is precisely where software comes to be considered a system. Klir’s lame definition of “a system” as a set of attributes avoids the controversy surrounding the synthetic nature of systems. Systems are natural or artificial complexes with internal and external coherence. They are normally sets of dynamically interacting objects. Klir’s position of ontological ignorance is compounded by his empty definition of a system as a collection of attributes. By avoiding the controversy surrounding the synthetic nature of systems, he is led to build the general science of systems on weak philosophical grounds. Operationalism pushes all substantive questions concerning the nature of the system under the carpet. It is ultimately the same position as that which claims technology is neutral. It is the epitome of nihilistic positions. Operationalism and technological neutrality allows the subject to identify with technology. We give up our humanity without a second thought. We are the technological system looking at the world through “observational channels.” When we lose our ability to distinguish between ourselves and technology, that dehumanization leaves us totally lost in the nihilistic essence of technology. Instead, we must struggle with the concept of a system and distinguish it from the concept of object. An object is a thing pointed at and focused on present-at-hand. A system, on the other hand, is a set of objects interacting dynamically as a whole. This is precisely the same idea of a set of figures with the same ground that dynamically form a series of gestalts. The idea of a system is an attempt to make the ground within which the objects of a system relate dynamically present-at-hand. In physics, this is represented by the concept of a field. In the field, system-wide interactions between objects are given a specific mathematical form. A system would be better recognized as a manifestation of ready-to-hand phenomena. As such, it can never wholly be made present-at-hand. Thus, come the problems of defining a system. Either definitions render it empty, as does Klir, or get tangled in messy ontological problems. The recognition that a system is more than an object with a different kind of coherence immediately leads us to suspect that it has different ontological grounds. These grounds are glossed over by futile attempts to turn it back into a mere object as Klir would have us do. Instead of recognizing these grounds outright, Klir reintroduces the distinction between general systems and specific systems. Systems science occupies a meta-level to the study of all particularized systems by various disciplines. The meta-level is introduced surreptitiously in the sophistic style so common in Western theorizing. The structure of Klir’s systems theory is brilliantly conceived. But it lacks the straightforward recognition of the difference between systems and objects. Objects are present-at-hand focuses of attention for subjects. Systems are present-at-hand representations of ready-to-hand phenomena. These representations always mix formal and structural representations to achieve their effect and always represent meta-level constructs, where the contents of forms are structured, to achieve control over the transformations from one form to another. The set of transforms is characterized as a system. In this manner, form and structure mediate between objects and meta-objects, i.e. systems.

The “intentional morphe” is what projects form on content (hyle). The noetic and noematic cognitive striations reveal pure form and pure structure as abstractions within the

\textsuperscript{133} See Bib#
field of the system. Form and structure are actually bound together as necessarily intertwined within the system. They are the internal and external coherences of the system. The external coherence appears as the series of forms arising within the gestalt whole. Structure is the transformations of contents allowed by the categorization of contents. These transformations of content allow the tracking between formal transmutations which makes it possible to relate a particular form to all the other manifestations governed by the system field.

According to Husserl, it is the arising of essences as distinct from simple ideas which is the key point. Essences are different from the results of induction or deduction. These latter are purely formal, logical performances. Charles Pierce contrasted these with “abductions.”¹³⁴ In abduction, one jumps to the conclusion. As Husserl noted, one knows “a lion” the moment it is seen. Its essence leaps forth without induction or deduction. Essences arise at the systematic level. Objects have noematic nucleuses which reveal their form-content coherence as objects. Noesis provides our ability to understand transmutations by categorizing contents and mapping between discontinuities informal presentations. The ability to perceive structural relations reveals the essence of the objects. The object’s essence relates to its possibilities of variation and reveals its inner necessary structure. Essences are separated from

---

¹³⁴ See Bib#
accidents as the inner coherences of objects. This separation of essence from accident appears for the first time at the systemic level. There is a direct line between that separation and the arising of point/grasp or agent/function as images of the model differentiation between the present-at-hand Being \(^1\) and the ready-to-hand Being \(^2\). Essences have a different mode of Being than objects as noematic nuclei. Husserl discovered this, and Heidegger based his whole philosophy on this difference in modality. Essences express the variability of the object within the systemic field. They also express the limits of that variability where one figure/form transforms into another, and the gestalt changes. Once it is realized that essences have a different modality from the noematic nucleus that supports them, then the importance of accidents becomes obvious. Accidents are what make different incarnations of a certain kind of essence unique. This uniqueness is the key to connecting essences to concrete lived reality. This is why existence occurs as a level between the phenomenal and the life world. Existence grounds essences. The agent/function dichotomy relates this same point back to software systems by the distinction between autonomy and behavior. Without processors, functional behavior would never be concretely realized in space/time.

Systems are meta-objects whose nature is technological. Systems are, in fact, the technical view of everything. We see everything as systems. In this way, we convert them into meta-objects which we can deal with technologically. Klir's epistemological levels show how this technologization of objects works in a clear way. The object is instrumented, observed and then simulated. Then the object itself is discarded and replaced with the simulation artifact. The simulation artifact at a technological level may be a machine that imitates the original but makes the process imitated more efficient. This process of technologization is intensified at the meta-technical level. There software allows the simulation to be adapted and integrated. The differences between the technological and the meta-technical have already been discussed. The meta-system appears in Klir's work as the infinite regress of meta-models and meta-structures. These two infinite regresses are similar to those generated

---

**FIGURE 4 6**

<table>
<thead>
<tr>
<th>agent</th>
<th>function</th>
<th>event</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td>peripheral</td>
<td>external</td>
<td>interrupt</td>
<td>datum</td>
</tr>
<tr>
<td>network</td>
<td>context</td>
<td>bubble</td>
<td>sheaf</td>
</tr>
<tr>
<td>processor</td>
<td>transform</td>
<td>signal</td>
<td>entity</td>
</tr>
<tr>
<td>task</td>
<td>function</td>
<td>event</td>
<td>instance</td>
</tr>
</tbody>
</table>
by Russel and Whitehead’s theory of logical types. This is where the concept of meta-levels as the solution to logical paradoxes was first tried on a large scale. The problem was that infinite regresses of meta-levels were produced so that solving the paradoxes just changed where the problem lies. That these same infinite regresses should appear in the description of the formal-structural system is an important fact. The fact that two infinite regresses appear together is also important. These are the expression of the in-hand modality Being within the description of the formal-structural system. The singularity of pure immanence is the ultimate source of these two horns of Klir’s dilemma. The dilemma is that infinities cannot be factored out of his model. They appear as the ultimate barrier to fully understanding systems.

Software systems are “systems” to the extent they imitate hardware. As software, they are really meta-systems. When we build software systems, this distinction between system and meta-system is lost for the most part. We do not think of the meta-system, but suppress it by outlawing such things as go\texttt{os} and self-modifying code thinking that the problems have been solved by this expedient. When we come to define the software method, we find that this distinction again becomes very important. Our approach has been to take the singularity which the infinite regress of meta-structures and meta-models banished by Klir to exotic realms beyond where it is possible for us to think clearly. (Remember, Bateson pointed out that we have difficulty thinking beyond the fourth meta-level.) Instead, our ruse is to take the singularity and embed it at the heart of the description of the formal-structural system. This is done by turning our four perspectives into “supports.” Thus the relation between perspectives as “supports” and system attributes, i.e., the component concepts of methods, expresses the relation between system and meta-system (between Process Being and Hyper Being). What Klir presents as a present-at-hand model has two other modes of Being hidden in it. We have drawn out the ready-to-hand (as system) and contrast it directly with the in-hand (as singularity that is source of two infinite regresses of meta-levels). The present-at-hand object which was Klir’s empty definition of object has been given life by introducing a true subject-object dialectic.

135. See Bibliography.
The subject “takes” perspectives and views from one perspective the object bound to another perspective. The observation channel becomes an introspection within the system of the formal-structural system itself. Each introspective observation channel is characterized as a minimal method. A method is a means of viewing other perspectives, and also a means of moving to another perspective. The dialectic between the perspectives and the methods provides the relation between the system and the meta-system. The perspectives are faceted, and embedded within their facets is the singularity of pure immanence -- the point of pure cancellation for all the relations between the perspectives. What is left when this cancellation has occurred is the proto-technical meta-meta-system.

This transformation of Klir’s formal-structural system representation causes methods to play a crucial role. They are essential bridges between the technical and the meta-technical realms. Through methods, the essential natures of systems become clear. The essence of systems must, in fact, be meta-essences because systems are the field within which essences are first apprehended. The essence of the field that gives rise to essences must be a meta-essence. Essences and accidents are delineated by partial orderings with no distance. These are contrast with full linear ordering with distance provided by space/time. The perspectives themselves arise from these two opposite methodological distinctions. It is, in fact, the methodological positions of linearization without distance and distancing of partial orders that support the meta-essences. Methods are meta-essences. This is why they can describe systems which are the fields within which essences and accidents are discovered. Only as meta-essences can methods be directly related to the nature of meta-systems composed of software. Only as meta-essences can methods deal with the presence of the singularity and represent the nonrepresentable software theory. Each introspective observation channel connecting perspectives on the formal-structural system contains a single meta-essence that bridges not only between the perspectives, but also between the perspectives and the system observing itself. What is a meta-essence? Sounds esoteric. An essence is the variability of the object. One would expect the meta-essence to express the variability of the system. It must express the limits of the variability of the system. Without the necessary attributes of the meta-essence, the system would no longer be a system. The set of meta-essences together make up the internal coherence of the system, and the meta-essences all cancel, making the systemic field vanish. Meta-essences must cancel, leaving only the residue of Wild Being which is what lies beyond cancellation.

The next step is to extend the concept of methods as meta-essences built upon intermediate methodological distinctions. This can be best done by comprehending the difference between linearization (L~D) and distancing (P+D). The difference between Spacetime (x + y + z - t) and Timespace (past + present + future - nowhere) has already been mentioned. This difference occurs because of the nature of the interval. Intervals consist of a temporal and spatial displacement. Because of relativistic effects, one observer may see the temporal phase to be larger or smaller than another observer. The theory of relativity in physics explains the relation of expanding and contracting phase structures for different observers in...
different inertial frames of reference. The understanding of phase structure of intervals is also important for understanding the relation of minimal methods as meta-essences.

In the interval between two perspectives, there is a phase structure. This phase structure may be shifted in relation to each perspective. In spacetime, this shift may cause the space phase to be displaced in relation to the time phase or vice versa. The two phases are separated by a point of reversability\(^{136}\) which is the area where the transformation from one phase to the other occurs. The phase structure can be looked at in two ways. If space is emphasized, it is spacetime, whereas if time is emphasized, it is timespace. Minkowski timespace model is an example of how time can be emphasized over space. Light cones distinguish past, present, and future, whereas the spatial component is reduced to the "nowhere" of nonoverlapping light cones. In spacetime, the time component is one dimensional whereas in timespace, it is the space component that is one dimensional. The important thing here is that the interval relates two perspectives, and there are always two ways of looking at these two perspectives. Minkowski's spacetime model is best for thinking about casualty in a four-dimensional plenum. Einstein's spacetime is a better model for thinking about communication and synchronization. These models emphasize either the linearity of time (timespace) or the distancing of space (spacetime). Both allow the phase structure necessary for relativity to appear within the interval. One model emphasizes one kind of phase as primary, whereas the other emphasizes the other kind of phase.

When we consider the minimal methods related to event/data, or time/space, the phase structure of the interval plays an important role. The two methods which appear here are called “data mutation” and “design element flow.” Data mutation is the most basic of methods for determining whether software is working properly. We enter print statements into the program and check their changing values as the program runs. Data mutation may be observed by the program itself. A threshold is set up which will cause a control signal to be generated when the data values cross a crucial threshold. The streams of changing values in different variables is a fundamental way of looking at a program which emphasizes linearization. The other method which serves as an introspective observation channel is design element flow. This minimal method concentrates upon the flow of design elements like pointers, counters, timers, etc., through system states. The emphasis is not on instantaneous values, but upon correct operation within a particular system state. Most designers consider design elements as clockwork-like mechanisms. These clockwork mechanisms work together in a static gear work fashion to manipulate input/output data. In fact, design elements flow through system states just like input/output data. With design elements, it is the relation between various kinds of elements which determine how the system works. A system needs several different design elements in order to work. It is the interlocking mechanism which introduces a certain semantic distance into the system. The more complex system will have a greater diversity of design element types working together. This can be seen by differences in Halstead metrics which compute total operators/operands versus unique operator/operands. This semantic diversity, together with the complexity of system states, determines paths of design element flows. The design elements crossing system state boundaries introduces distance into the software system. The distance is the opening up of difference between diverse kinds of design elements working together in an interlocking fashion which changes as system states change. Distance here means similar/dissimilar and near state/far state. The treatment of dissimilar types of design elements working together in the same state, and the treatment of the same element across various system states, is how distance appears. Distance is the opening up of difference in terms of differentiation.

Note that both data mutation and design element flow are two different ways of looking at the same thing. Data mutation may consider several different data object values instantaneously, or the history of a single data object. Design element flow may consider the state changes of a single design element or the differentiated set of design elements present working together in a single state. These are diachronic and synchronic views. When design elements are considered as merely data objects with instantaneous values, then data mutation has precedence. Every design element has, as its basis, a data object. Design elements flow is the event view of data, whereas data mutation is the data view of event. Design elements emphasize the systemic events that happen to design elements. Data mutation emphasize the data changes that are brought about by systemic events. The data-centered view watches the data objects themselves and looks for instantaneous changes. The event-centered view considers thresholds of system significant changes to key design elements flowing through system states as

\(^{136}\) Cf Merleau-Ponty’s ‘Chiasm’ in Bib®

Wild Software Meta-systems
important.

The crucial difference between these two methods is the emphasis of one phase over the other. One is a timespace emphasis, while the other is a spacetime emphasis. Data mutation emphasizes spatial aspects of the software system. Data is kept in different memory locations, and what is printed is the instantaneous value. Input/output variables are emphasized. This is a timespace view. History is important. Design element flow is the cartesian product of system diversity and system state diversity. Thresholds of values are important, and internal variables are emphasized. This is a spacetime phase space. The internal differentiation of values within system thresholds is important.

A way of perceiving the difference here is to examine the distinction between system dynamic simulations and discrete event simulations. Both are performed on digital computers. In system dynamics, it is the deltas between system variables over time that is important. In discrete event simulations with queues, it is the interlocking of system components that is important. Both are dynamic simulations, but these are two very pure examples of the difference between data mutation versus design element flow views of a system. The system dynamics simulation looks at a system as a series of attributes which change in relation to each other over time. The internal connection of the attributes is not considered, but only their outward behavior in concert. The discrete event simulation sets up structures similar to the system under study and allows the individual behaviors of discrete elements to add together to make up an overall system behavior. In discrete event simulation, it is the internal relation among system components that is important, more than the instantaneous readings of externally observed system attributes. One is an approximation working from the outside in, and the other is an approximation working from the inside out. System dynamics (outside -- in) emphasizes the history of attribute values read instantaneously (linearization). Discrete event simulation (inside-out) emphasizes the relation among internally differentiated system elements working together to create, as a side effect, overall system behavior (distancing). System dynamics has the premise that the whole of the system can be characterized by itself. Discrete event simulation says the system is the sum of individual component behaviors. Considering the examples of different types of simulation, it is possible to see that data mutation sees each data object as a disconnected system attribute. The history of these values over time is an external view of the system history.

Design element flow gives an internal picture of individual system components interaction. This interaction, when summarized, becomes a picture of system-wide actions. This is a restatement in terms of software systems of what Arthur Koestler called the Janus137 like nature of systems components. They are holons which act as systems viewed from below in the control hierarchy, and act as components working together when viewed from above. The conception of the holon is precisely the area of reversibility between the two phases of the interval. Linearization of the holon is related to an external, or view from below. Distancing of the holon is related to the interval, or a view from above, that sees differentiations interacting and working together to form a whole system. The system is a holon composed of holons.138 Each holon is both component working together with other components, and a whole system itself. Meta-essences describe holons from different perspectives. Where essences describe the variation of objects and their limits of variability, meta-essences describe the variation of holons and their limits of variability. Holons are the components of systems. They are subsystems. A software theory must strive to create holons that work together and also function independently as systems. Holons organize the system gestalt where a certain set of system objects appears. In another gestalt, a different set of system objects appears. The holons are the transitions between gestalts. They control localized transformations of objects between gestalts. Holons are not objects themselves. Software theory strives to construct holons because it is that which will make a software text into a software system. The interaction of holons are the meta-system. Design heuristics strive to impart paths of thought that will result in holonomic structure.139 Design heuristics are the most important aspect of design methodology. Through heuristics, we attempt to learn how to create holonomic structure. These holonomic meta-structures exist in the interstices between meta-models and meta-structures. They cannot be separated from the intertwining of meta-structures and meta-models. They are seen in the elegance, simplicity parsimony of systems design. Where software theory is nonrepresentable, software holons may be glimpsed by passing back and forth between linearizing and distancing methods and may be indicated with heuristic aphorisms.

137. See Bib#.
138. See Bib#.
139. See Bib#.

Wild Software Meta-systems
FIGURE 4.9

Singularity

essence of manifestation
At this point begins a derivation of the methods of software engineering. By derivation is meant a step-by-step explication of the relations between the methods, starting from primitives which are related to single perspectives that are combined in order to allow the bridging of the gaps between perspectives. Minimal methods are defined, and then finally the holons that lie between minimal methods are indicated. Derivation is a difficult process because it goes against traditional approaches to the development of methods which are normally ad hoc. Derivation treats the method “space” in a systematic and rigorous fashion. Because of the complexity and length of exposition needed by full derivation, only an outline will be presented here. In the outlined derivation, only the major structural elements will be treated. The treatment will attempt to place these methods in the context of the formal-structural system while building up all the major structural elements necessary to deal with the interrelation between methods fully. A series of entity-relation diagrams will be used to build up the relations between elements of the methods, step by step.

Let’s begin with an overview of what a system is to us before we attempt to be explicit about the definition of perspectives. A system is a gestalt of interrelated forms. As such, it has a foundation that depends on both formalism and structuralism. Formalism is a set of rules for manipulating contents. The contents remain unspecified within the formal system. Only the rules and their relations to each other are important. Structuralism is a subformalism in which contents are classified and become bound by their own rules. By shifting levels back and forth between formalism and subformalism.

FIGURE 50
(structuralism), functional changes are represented as transformations. The formal-structural foundation of the system allows the gestalt to be modeled explicitly. The gestalt itself is, however, always more than can be captured by a formal-structural model of a system. The formal-structural model of a system that appears within the horizon of the world is always a gloss or an abstraction. Systems are intrinsically both temporal and spatial in their differentiation as natural complexes within the world. The fact that they appear within the world means that they may be observed by multiple observers. It is from the observation of a system gestalt that Klir’s epistemological levels arise. These are balanced by the ontological levels already described. The observer uses the epistemological levels in order to focus on different aspects of the system gestalt. The observer applies the formal-structural matrix to the observed system gestalt in order to get a dynamic model that can be understood theoretically. There is no limit to the sophistication of theoretical understanding. An infinite regress of meta-models and meta-structures linking models assures this. Behind this infinite regress may stand the singularity of pure immanence as an unattainable limit.

What has been described is the superstructure which allows systems to be viewed theoretically and philosophically. This superstructure is essential for systems science and systems philosophy, but is inessential for the development of an approach to methods. For the development of an approach to methods, the concept of meta-system is important.

A system may be described in purely formal-structural terms. In this case, the attempt is to give a present-at-hand description of a ready-to-hand entity. However, a meta-system contains a singularity of pure immanence and is perspectively fragmented. This means that a meta-system breaks up the community of possible observers so that each observer must take a particular position in relation to the system in question. The meta-system structures the community of possible observers. The meta-system encompasses the infinite regresses of meta-models and meta-structures, giving them coherence as they revolve around the singularity of pure immanence. The perspectives present qualitatively different views of the system under study. Perspectives see a particular set of qualities to the exclusion of other qualities. This set of

![Figure 5.1 World Peripheri](image-url)
qualities acts as a filter which enhances certain aspects of the system and causes other aspects to be veiled. This qualitative filtering is an important phenomenon. It cannot be accounted for in formal-structural terms. Qualitative filtering gives an aesthetic dimension to the study of the meta-system which cannot be seen in relation to the system alone. Design elegance and optimum solutions enter into consideration by this door. The perspectives available in relation to the software meta-system are DATA, EVENT, AGENT, and FUNCTION. Data means information stored in a specific place in memory. Event means temporal modulation of signals. Agent means different locuses of independent action. Function means specified transformations. Software meta-systems organize the observation of formal-structural systems.

The observation of systems must account for their gestalt-like character. As such, each system has a set of facets. The observer may focus on any given facet conditioned by a particular perspective which will alter the quality of the facet. A given facet is conditioned by the environmental context to which it responds. The environment, which is the world from the point of view of a particular system, contains many contexts. Contexts interpenetrate within the world. This means that many different contexts apply simultaneously to a particular system without necessarily interfering with each other. This interpenetration of systemic contexts is called their holoidal character. They appear as different roles. The world has a set of environments depending on how many systems it appears to contain. An environment has a set of contexts which interface with the facets of any particular system. The observer has different roles related to the environmental contexts. The observer has four qualitatively different perspectives which will allow him to focus on a particular facet. The system synthesis organizes all the facets of the system. The focus highlights synthesis as it controls a particular facet that is the current center of attention. Systems also have modes of operation. A mode is a select set of system responses keyed to an environmental context.

Beyond the gestalt nature of the system, it is also important to note that systems have sub-systems. A subsystem is a set of systems that cooperate to make up an overall system. Subsystems are holons in the sense of Koestler in that they have a Janus face. They appear as parts working together from the outside, and as complete systems from the inside. Systems may or may not be seen to be composed of subsystems. This depends on whether the gestalt contains sub-gestalts. These sub-gestalts may or may not be related to the meta-system. For instance, a purely formal-structural subsystem may exist next to a subsystem with a meta-system within the same overall system. However, if a system has both a meta-system and nested sub-gestalts, then the meta-system organizes the nested sub-gestalts.

The meta-system extends the system in two directions. The meta-system allows the articulation of supersystems and subsystems that act together to form a whole. The meta-system also allows the differentiation of perspectives from which the qualitatively different facets of these systems might be viewed. Within the meta-system, the behavioral synthesis of the system becomes a harmony of subsystems each with their own synthesis and the perspectives of multiple observers. The perspectives of multiple observers mediate between the articulated subsystems and supersystems which bracket the system itself. Meta-systemic harmony is an important concept that gets little treatment by general systems theory. That theory sees systems as random collections of attributes without even the coherence of a gestalt. So how could they go on to deal with the harmony of meta-systems? This lack of insight, or short-sightedness, is a symptom of our cultural disintegrity. The inability to see harmony in meta-systems leads directly to a kind of cognitive dissonance that shows everywhere in our environment. The best discussion of harmony must come from the study of cultures which were not so blinded. Chung Ying Ching’s article "On Harmony as Transformation" is an excellent study in the meaning of harmony. He distinguishes four grades of harmony:

A. Logical consistency       Formal system
B. Interactive relation       Structural system
C. Mutual support              Meta-system -- Holon
D. Interpenetration           Meta2-system -- Holoid

Ching’s first level of harmony is that which occurs in a formal system. Within a formal system, logical
consistency of rules governs the harmony of propositions. The second level of harmony is that of interactive relation which can only occur when time impinges on the formal system turning it into a structural system. The third level of harmony goes beyond the formal and structural system. It is only captured by the meta-system. This is the mutual support by holons (Arthur Koestler) within the same system. Holons are systems within systems. For this reason, they can only be described adequately in terms of meta-systems. The fourth level of harmony is the interpenetration of mutually supporting subsystems. This is described by George Leonard in the *The Silent Pulse* as the holoid.143 The best exposition of interpenetration is by Francis Cook in his book on Hua-Yen Buddhism. Interpenetration may be described as a meta-system phenomenon which is generally demonstrated by referring to holograms. Holograms are lightwave interference patterns in which the whole form is contained in every part in potentia. The *Holographic Paradigm* by Ken Wilber explains these concepts.144 This definition of the different levels of harmony is important for the understanding of systems and meta-systems. It gives a view which has been totally neglected within Western science and engineering. This view is beginning to become important with such

FIGURE 5 2

![Diagram of Interpenetration and Mutual Support](image_url)

<table>
<thead>
<tr>
<th>Perspective</th>
<th>Data</th>
<th>Event</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Interpenetration</strong></td>
<td><strong>Horizon</strong></td>
<td><strong>Datum</strong></td>
</tr>
<tr>
<td><strong>Mutual Support</strong></td>
<td><strong>Context</strong></td>
<td><strong>Class</strong></td>
</tr>
<tr>
<td><strong>Interactive Relation</strong></td>
<td><strong>Facet</strong></td>
<td><strong>Entity</strong></td>
</tr>
<tr>
<td><strong>Local Consistency</strong></td>
<td><strong>Attribute</strong></td>
<td><strong>Instance</strong></td>
</tr>
<tr>
<td><strong>Truth o( Manifestation</strong></td>
<td><strong>(Fate)</strong></td>
<td><strong>Merged Spacetime/Timespace Openness</strong></td>
</tr>
</tbody>
</table>

Kent Palmer

---

143. See Bib#
144. See Bib#
movements as Deep Ecology.\textsuperscript{145} It is a view which attempts to understand the harmony we experience, but generally ignore, within the world. This harmony holds systems as holons together in mutual support to form meta-systems. It holds meta-systems together through mutual interpenetration to form worlds or meta-systems. Harmony is the aesthetically perceivable guide which should inform our systems design and construction work. This aesthetic dimension is generally lost because engineers do not consider themselves artisans in the same way as architects. For the software engineer, we might wonder what interest there would be in harmony when his product is never seen. Yet, what Alexander calls the “quality with no name” (nb the Tao)\textsuperscript{146,147} pervades everything that we create as human beings. Even software design must take seriously the inner harmony of the systems it produces in order to achieve elegance, simplicity and optimality. These can only be achieved by taking into account the harmony that becomes apparent in the relations of systems to each other. Mutual support is only made visible by comparing wholes to each other. Interpenetration only becomes visible when it is realized how differences are exactly what makes it possible for things to be the “same.” As Heidegger says, there is a belonging together among things that are the “same” which is different from and richer than the concept of identity.\textsuperscript{148} Interpenetration allows the mutual support of the details of different subsystems to be grasped as a holoid, where those differences allow the greater whole to be realized in actuality.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure53.png}
\caption{Wild Software Meta-systems}
\end{figure}

\textsuperscript{145} See Bib\#.
\textsuperscript{146} See Bib\#.
\textsuperscript{147} See Bib\#.
\textsuperscript{148} See Bib\#.
In our exploration of methods, it is precisely this perception of harmony which must be brought out into the open. This is because it is that aesthetic perception of harmony which counteracts the fundamental nihilism of technology and meta-technology. Because meta-technology deals prominently with meta-systems that may be seen as holons, it is possible to glimpse the non-nihilistic distinction through the veil of nihilistic erratic change and excrences. The technical system is also a human system which has the possibilities of realizing the non-nihilistic distinctions in the midst of nihilism. This makes the agenda of deep ecology the innermost possibility of the technological system.

Software engineering is taken here as the exemplary praxis of our age. Its essence is nonmaterial production. Work
has always been understood before as physical work that involved rearranging material matter to some end. Software engineering rearranges gigabytes of ASCII characters whose physical representation is perfectly maluable. This exemplary praxis changes the nature of all other praxis. Blue collar traditional manufacturing work is transformed by computer literacy. Production of physical products begins by programming the production line machines. This is most significantly borne out by the idea of the human Genome project. In this project it is suggested that the entire human genome may be read. The secret book of human life written in the DNA spiral in each of our cells might be unlocked. The DNA code could then be used to reprogram human development. Thus, we intend ultimately to reprogram ourselves. We see ourselves essentially as the programmers of everything in the world, including ourselves. Selective breeding is replaced by genetic engineering which is essentially a kind of biological software engineering.

FIGURE 5 3B

---

Wild Software Meta-systems
So software engineering is an important example of how the technological world is being transformed in our time. As such, it is interesting to note the relation between nihilism and non-nihilistic harmony within this new discipline. Nihilism is intensified at the meta-level of Being 3, Hyper Being. Nihilism becomes the active destruction of meaning in the technological society. Within this bleak and dire picture, we note the arising of the importance of harmony. Disharmony and strife in the world highlight the need for harmony. We experience harmony in stark contrast to the excrescence of erratic change or noise produced by the dynamic formal-structural system. I remember an exhibit at the 1960 World Fair that showed a single machine a mile long crawling through the rain forest of the future (which is now). In one side went everything living and dead from the rain forest in the machine’s path. At the other end came out a road, and on the road trucks carrying the transformed rain forest. This machine is the mythical image of the dynamic formal structural system transforming wild into tame in a single all encompassing transformation. Now with holes appearing in the O Zone, the “myth of the mega-machine” is slowly being replaced by furtive searches for harmony in a world which is fast on its way to becoming like Venus -- uninhabitable because of excessive greenhouse effect.

In software engineering, a realm which is safely within the technical establishment, there is no pollution. We use such a small amount of power that solar cells could furnish all the necessary energy. Now we waste paper by the ream, but these printouts are ultimately unnecessary. We could achieve the ideal of paperless environments if necessary. Thus, every way you look at software engineering, it is a kind of production that has risen above the dirty industries, i.e. polluting and resource destroying, of the industrial era. Post-industrial production, like software engineering although it is within the technical sphere, apparently has a radically different nature from traditional production. Software production is clean. It is the traditional industries controlled by software that are still dirty. And software helps those inherently dirty industries to be cleaner and more efficient. Retooling no longer means remachining. Resource utilization may be monitored.

---

149. Nb. Toynbee
Pollutants emissions may be monitored. Software allows traditional production to tighten its control on waste and inefficiency. Software exerts an influence on traditional production, which is subtle and profound, toward a cleaner, more efficient, more harmonious working of the myriad different parts of the industrial complex.

We might say that machines display the harmony of logical consistency (A) embodied by the formal system. Chemical processes display the harmony of interactive relations (B) embodied by the structural system. In interactive systems, catalysts which facilitate but do not enter into interactions are possible. Chemical and mechanical industries have been the traditional productive centers of our economy. However, it is only software which makes possible a harmony of mutual support between elements of the chemical and mechanical industrial complex. In the past, careful design has brought these elements into close interaction, as in the automobile. However, actual mutual support with multiple feedback relations between parts was primarily a side effect of design rather than being intrinsic to the system. With the advent of software, the mutual support is intrinsic to the system. This software-based harmony in airplanes is
called “fly by wire.” It is only the electronic signals between cooperating processors that allows the plane to fly at all. For instance, it is said that some aircraft with forward swept wings could not fly without computer-controlled compensation. Thus, software introduces the possibility of intrinsic mutual support between subsystems within the meta-system. In this way, the artificial meta-system becomes a reality only with the advent of software.

It is the holonomic harmony of the meta-system which is the ultimate aim of software. Thus, the aim of software methods must be the definition of the holons which make that harmony realizable. Methods, as formal systems, cannot capture these holons. They show up as the heuristics which are the real heart of any method. Instead, methods in their interaction with the structural system of the software itself, make the holon/heuristics visible for the first time. In our attempt to derive our methods rigorously, we must keep in mind the goal of making these heuristics visible. Methods are meta-essences of systems. Systems are composed of objects with attributes. These objects have essences in the traditional sense (attributes, which if varied past a certain limit, the object is no longer the “same” object). Systems compose meta-systems which are the active environments of the subsystems. Meta-systems exhibit the harmony of mutual support between subsystems. The presence of mutual support is the sign of a meta-system or active environment. The subsystems within a meta-system are holons in Arthur Koestler’s sense of being both parts and wholes simultaneously. It is software which makes mutual support an intrinsic rather than an extrinsic attribute of the meta-system. Software operates within the realm of Hyper Being 3 which is governed by the essence of manifestation -- the singularity of pure immanence. A meta-system with extrinsic mutual dependence is not governed by the action of the singularity, whereas the meta-system with intrinsic mutual dependence must be so governed. This makes us think about all organic creatures which are based on DNA coding. All such creatures achieve intrinsic mutual dependence by carrying a copy of the master software in the nucleus/cpu of each cell. This implies that all organisms operate on the level of Hyper-Being in some aspect of their existence and are governed by the law of pure immanence. It is obvious that mechanical and chemical processes play a large part in the functioning of the organism. The formal and structural relations of the mechanical (physical) chemical processes are not recognized to be controlled explicitly by the software of the DNA strand. At this point the physical and chemical analogies break down, and a new software metaphor must be used. Biologists are only slowly realizing the implications of this change in metaphor. The body is a multiprocessing distributed system where each cell is a holonomically independent/dependent processor. We realize the truth of this when dealing with neurons, but falter when it is necessary to extend the metaphor to all cells of the organism. The question is, how does the singularity manifest within the functioning of the organism?

Within the meta-system exhibiting intrinsic mutual dependence achieved on the basis of software, there is a fragmentation of perspectives that is necessary because of the presence of the singularity. With respect to software, this fragmentation appears first in the separation of space (memory-data) and time (cpu cycles-event). Then later there appears the separation of pointing (agent) and grasping (function). We will start by looking at the first separation more closely. As has been shown, these exhibit methodological distinctions (in Klir’s sense) of full metric ordering (distance + linear order). Thus, these two views are easier for us to understand since we are used to dealing with the full ordering of the x-y-z-t coordinate scheme from intermediate level mathematics. As also noted above, spacetime/timespace is originally a single entity which may be viewed in two distinct ways. These two views inform our separation of the time and space views. There can be no total separation of space from time, only shifting of perspectives in such a way as to emphasize one or the other momentarily. Our four basic perspectives are not totally divorced from each other, but grow out of each other in a way that is mutually reinforcing. Mutual support is real and basic. It is “actual” in the sense of Kubler in his book The Shape of Time. “Actuality” differentiates into the interval which can either be viewed in terms of timespace or spacetime. These are further abstracted into space as data and time as event.

What we wish to do is present first the pure space/data perspective and then the pure time/event perspective before blending these to get the minimal methods that act as a bridge between these two pure perspectives. This is the next exercise in the derivation of software methods. The derivation process aims to construct the minimal methods in such a way that the holon/heuristics may be glimpsed in the exchange between minimal methods on the bridge connecting pure viewpoints. However, it must always be kept in mind that the “pure perspectives” are a hypostasis from something which is organically bound

150. See Bib/.
together. For instance, “spacetime” and “timespace” are meta-views which allow the ideal views of “time” and “space” to appear as distinct. But what is spacetime/timespace before the meta-views came into existence? What is the “actuality” of spacetime/timespace? It is obviously buried deep in the singularity. This is exactly what the singularity is not showing us. There is a lost, unrecoverable origin which is always already lost. The singularity is the dark face of this origin. We construct primal scenes to cover over this primal lostness which explains it away. But the origin of spacetime/timespace before the Big Bang is lost forever, even though in a sense it is ever present. Like the relics of the Big Bang, we constantly live in the midst of the actuality of the origin of spacetime/timespace. The singularity is the dark face of the always lost origin of the four perspectives. We live within the nexus of their unfolding. Experiencing that unfolding we are what Heidegger names “Dasein” instead of subjects. The objects caught in that experience of unfolding might be called “ejects” because of their being thrown along with us. As Heidegger says, we realize our essential nature as falling within the world toward the abyss of groundlessness. It is within the ready-to-hand modality that Dasein relates to ejects within the world. In the present-at-hand mode, we are again subjects safely relating to objects. Subjects and objects do not experience the unfolding from the lost origin. They are static entities. Perfect specimens for nineteenth century science. However, once we begin to recognize systems and become observers of systems, suddenly the world is dynamic again. Those systems are not like objects -- as Klir would lead us to believe. He makes the minimal number of assumptions in order to avoid controversy. Systems exhibit formal and structural coherence wherein appear the harmonies of logical consistency and interactive relation. Yet, systems also appear as subsystems within meta-systems. When these meta-systems are purely extrinsic, then we can say that the meta-system is projected as an abstraction on the subsystems. In this case, the meta-system is an artifact. However, when the meta-system is intrinsic, then it must be held together with an ontic structure like software. In organisms it is the DNA spiral, and organisms are the exemplar of the dynamic interdependent mutually supportive meta-system. In fact, Whitehead used “organisms” as the metaphor in his process-based philosophy which attempted to describe what has been called “ejects” above. The meta-system corresponds to the meta-observer (meta 2 subject). The meta-observer is the observer observing himself, or the thinker thinking about thought. This is the final refuge of reason. It is a refuge in paradox which encapsulates the singularity in an analogous way as the meta-system. The best exposition of this is The Tain of the Mirror by Roddlphe Gasche.\textsuperscript{151} A good discussion also exists in

\textsuperscript{151} See Bib#.
Understanding the layers from Process Being as manifestation out to the full expression of the world is important for our study. Heidegger speaks of the world as a “clearing-in-being.” We can think of it more as a vortex which, like a galaxy, has spiral arms. The arms form successive veils that obscure the center of the galaxy. The center of the vortex is the point of manifestation -- pure upwelling of existence unfolding like a fountain -- out pouring. This is pure ecstasy. “Humankind cannot bear very much reality.” So each layer building out from the empty center of the vortex is a veil covering up that reality we cannot stand. The first layer is our experience of reality as Being. Being as C. Chang has said, is “a subtle form of clinging” almost exclusively the sole province of Indo-European languages. In all these languages, the conjugation of the verb of “Being” is the most irregular. This is because the concept was forged from multiple roots with similar meanings, all indicating “abiding” or “remaining.” This veil of Being, subtle clinging to phenomena, is so deeply rooted in the Western consciousness that we think of it as the foundation of the whole world. It is a surprise to us when we discover the groundlessness (the abyss) which is like quicksand beneath our feet. We discover ourselves to be falling, and other things (ejects) are thrown with us into existence. The confrontation with Being focuses us upon our own impermanence. The subtle clinging to things which Being represents is experienced as everything being torn from our grasp. At this level, our grasp of things in the world is most important.

The next layer of veils is where we have reified and made everything static. We separate ourselves from everything - - subjecting all things to our gaze. We are dialectically related as subjects to all the objects we have projected. We pretend to be transcendent subjects, untouched by the world, as if that could possibly reduce our vulnerability to death. Distancing is the watch word at this level. This is the level at which scientific technological subjugation of the world occurs. The distancing is represented by the pointing that creates the present at hand. Being is stultified by its separation from time. Being is no longer dynamic. This is the Kantian universe of discourse. Transcendental objects are projected as the ideal supports for this view of the world with God as infinite ideal connecting these two poles -- the third transcendental.

The next layer of veiling begins to become dynamic again. Only now the subject begins to be an observer of systems rather than the subjector of objects. From domination we move to a recognition of interactive relations. From the logical consistency of formal systems, we move to the harmony of structural systems based on interactive relations. Here, the observer sees the gestalts of system. This is a new entry into an engagement with Process Being in a completely different way. Here, Process Being supports the dynamism of the formal-structural system rather than the primordial encounter with Being. Observer and system are locked together in mutual definition. Suddenly the observer affects the observed system in ways we still find puzzling.

The next level of veiling appears as the reflectivity of thought thinking of itself. It is the observer observing itself caught unawares by Lacan155 in the mirror stage. This is the meta-observer caught in self paradox so well described by Douglas Hofstadter in Godel Escher and Bach.156 This paradoxicality of reflexivity is matched at this level by the less explored nature of the meta-system which may have the intrinsic harmony compared by software. At this level, mutual support appears in the environment of the system that becomes a holonomic subsystem.

All meta-systems ultimately interpenetrate to make up the world. Here, interpenetration is given C. Chang’s interpretation as non-impedance (interference) of one meta-system with another. This is a primitive interpretation of interpenetration which will suffice in this context. A fuller treatment may be found in F. Cook’s book on Hua-Yen Buddhism.157 The world is made up of a myriad of meta-systems, both intrinsic and extrinsic. These meta-systems interlock to form a coherent world where systems simultaneously function in different meta-systems as subsystems.

The world is the meshing together of all the different meta-systems to form a concrete “de-totalized totality” as Satre would call it.158 It is the totality of everything there
is without any explicit totalizing principle giving it inherent unity. We experience our world as the manifestation of all that is in a semi-organized formulation we learn from childhood throughout our education. Different cultures within the global village have different twists on the same basic superstructure that governs all aspects of our lives. Historically, the divergences in “world views” was much greater, but slowly but surely a global culture is replacing those divergent views. The de-totalized totality of post-modern techno-culture is becoming more and more pervasive. The world encompasses the limits of our understanding. Multiple worlds used to exist side by side within the cosmos, but these are becoming extinct faster than our planet’s wildlife. The world is the outer limit of the vortex of manifestation.

The vortex of the unfolding of the world is a metaphor that allows us to explore the fundamental aspects the harmonic nature of things. Each layer addresses a different kind of harmony, most of which were identified in the analysis of Ching Ying Chung. However, we need a way to see this harmony and analyze it in a concrete manner. This necessitates the development of a vocabulary which will allow us to talk about what is crucial at each level of the vortex. Each veil has its own necessity that must be respected if we are to deal with harmony in a concrete way rather than as a nebulous concept with no actual referents. The vocabulary to be introduced has already been shown diagrammatically. Working inward from the peripheral of

---

158. See Bib#.
the vortex, we encounter each of these terms in sequence. A brief explanation will follow.

The world’s limits are horizons within which all things appear. Meta-systems, or environments, provide contexts which to the reflective consciousness are situations. The observer has a focus on a particular facet of the system’s gestalt. Dasein is shown the truth of its own care, and the fate of the eject as they arise from pure manifestation which is the center of the vortex.

Each perspective within the meta-system participates in these harmonic levels. Each perspective has its limits when it touches horizons. These horizons mark the interpenetrating boundaries with other worlds. Each perspective has a context with respect to the immediate environment of the system. The system has a facet that is focused on by the observer. The observer can also look within the system at objects which have attributes. We will not deal with the fated aspects of ejects since these have no psychological distance from us yet. They are transitional objects which are not yet totally decoupled from Dasein. Dasein and eject merge indistinguishably like the sucked thumb. This is an important level of care for self and the fated appearance of the object as it is in itself phenomenologically. Our analysis of all perspectives really revolves around distinguishing the following harmonic levels as in Figure 52.

This vocabulary allows us to discuss for the first time the harmony of design for which we all strive. In the field of software engineering, this harmony is implicit in everything we do. It has long been recognized that simple, elegant solutions are superior. Yet, we lack a way of

---

159. See Bib#.
expressing the aesthetics of design. With this vocabulary we can talk about each level of harmonization with an appropriate word, and we can see how the different levels work together in design.

Recognizing the important place of harmony in design is crucial to software methodology. Methods without a sense of harmony are empty tools without a craftsman to give them life. The aesthetic aspects of engineering are very important and little developed in this age. By building a vocabulary explicitly linked to the grades of harmonization, we can begin to explore this unexplored territory.

The definition of the pure data perspective begins by recognizing the difference between “class,” “entity” and “instance.” A class is a set to which entities belong from which they could inherit attributes. For instance, the class “airplane” is a set of attributes such as wing span, number of engines, etc. This set of attributes is given coherence when mapped on to an entity. An entity is a generalized form that includes attributes as well as some other features. These features include the name, a set of relations with other entities, an assembly, a set of faces, and a set of slots. The name is a unique identifier for the generalized form. The set of relations, such as “has_one” or “has_set” shows how the entity fits together with other kinds of entities. The assembly is a set of parts that comprise the entity. Parts are other subentities of different kinds that, acting together, perform the actions of the entity. The faces of the entity are the external views that other entities may have of a specific entity. The set of slots are active equations or rules that recalculate if any values of attributes they are tied to change.

An entity is more than merely a collection of attributes. The entity is the concrete realization of an object which may exist within a system. This concrete realization is a general entity which has Being and persists through time. It has faces, parts, attributes, relations, and recalculating slots. As such, it can serve as a model of any passive object within a system. It represents the pure data view of an object within a system.

Such a pure data view sees the datum. The datum is the limit of the data perspective. It is the observed value of an attribute of the object. Klir’s treatment is perhaps adequate for the observer who experiences the system as an unknown ensemble which he is attempting to reconstruct or model. However, for the software engineer who is attempting to construct a system by design, such an empty object is not adequate. Instead, we need a model of a general entity which can form the building block of our system construction. We know that within a system there are various kinds of entities in multiple interlocking relations with each other. Each of these entities belong to classes that express their kind. Kind here is expressed as a collection of attributes. Since Aristotle, attributes have been divided into essential and nonessential (or accidental). Essential attributes must be present for the entity to remain the same kind. Accidental attributes may be varied without affecting the kind of the entity. Here, kind is reduced to arbitrary sets of attributes. Kind is, of course, more than just sets of attributes. This is an external, minimal view fostered by analytic “anti”-philosophy. “Kind” has internal structure as well as being a collection of attributes. The “entity” fills this void by providing a general basis for the internal structure of any kind. The entity has relations with other entities which form a synchronic whole within the system. The entity also has an assembly of parts, giving it internal differentiation. Through its external relations and internal differentiation, the entity fulfills its role as the general model of an object within a system. Each entity may form the prototype of a whole series of similar entities of the same kind. These are the instances of an entity. An instance will replicate all the attributes of the original class of the entity. An entity may inherit from several classes at once. So the entity is the locus of inheritance of attributes. Within the particular instance, these attributes are called replicas. The slot is the active portion of the entity. The slot is the result of an equation recalculating. This equation is based on the current values of attributes. Through the slot, the entity is given a dynamism related to the assignment statement.160 Through the equations related to slots, the inner structuring of the relations between attributes can be modeled. These inner relations are static but express the inner structure of the entity itself as a data transformer.

The pure data view sees entities as nets of relations between different kinds. The entity is composed of an assembly of parts, and it expresses a structuring of relations between attributes that produce other attributes. The entity has different faces which are seen by other different entities. The entity is a general purpose construct for expressing clusters of persistent data in definite structures. These data structures will be called data

160. See Bib#.
“objects.” The entity is a design concept necessary for the theoretical construction of systems from the point of view of data alone. The concept of entities have gained prevalence through the emergence of object-oriented design in software engineering. Entities and their relations are modeled via Chen’s entity-attribute-relation diagrams. A good introduction to this is found in Shaler/Mellor’s book on Object-Oriented Analysis. However, our formulation differs somewhat from their’s because we need a more robust design representation for software systems.

The event perspective is very different from that of data. Each system can be subjected to an event perspective. The key idea in the event perspective is the “signal.” A signal is a stream of pulses or a steady state of energy in differentiation from other signals. A signal implies a stream of energy whose differentiation remains fairly persistent. The other signals, along with the one under consideration, form a bundle of signals which have specific relations between each other. The signal needs the bundle of signals in the same way the “entity” needs the class to give it kind. Through the synchronic/diachronic unity of the bundle, the lacuna (blank spots) in the signal can be comprehended. The signal may either be broken by lacuna or modulated. These breaks or modulations, when they cross imposed thresholds, are events. An event is a change to a signal or group of signals that takes place at a specific point in time. Events have names and also have temporal relations with other events or lacuna. These temporal relations are represented in terms of interval logic in terms of before-after-during relations as defined by James F. Allen.

X before y
x equal y
x meets y
x overlaps y
x during y
x starts y
x finishes y

These relations, plus their inverses, form the temporal logic of Allen.

Interval logic allows time relations to be expressed without an objective timeline being established. This is important because an objective time scale is not always present.

---

161. See Bib#.
162. See Bib#.
163. See Bib#.

essential attributes relations
accidental attributes relations
numerical relations between instances
spacetime position relations

SYSTEM EVOLUTION
SYSTEM DYNAMICS
POPULATION FLUCTUATIONS
DEMOGRAPHY

essential diacriticality
accidental diacriticality
instantiation diacriticality
position sensitive diacriticality
possible. In many systems, the temporal points at which things occur is not as important as the linear ordering in time. Interval logic establishes linear ordering without necessarily having to establish distance between time points. Interval logic allows us to relax distance to get the pure linear ordering of time. Bundles of signals may belong to a sheaf of signals. The signal sheaf can express branching bundles of signals. This allows temporal, or modal, logic of necessity and possibility to be expressed. The core of a sheaf can be interpreted as the necessary core of a set of possible worlds. A bundle of signals may branch at a particular time point to show two different routes of system evolution. The signals that have necessary modulations, or breaks, belong to the core of the sheaf unaffected by the branch which is common to all possible worlds. The sheaf contains all the signals pertinent to the system. In this way, the temporal view of the system is articulated as a series of events on signals within a sheaf of bundles of signals where there are both synchronic relations between signals and diachronic relations between intervals.

This is a short introduction to the vocabulary and concepts which are necessary to understand the pure data and event views of systems. Every system can be understood as a kind of entity and as a sheaf of signals. Both entities and signals have the persistence necessary for Being. A system described as a sheaf of signals, or as a set of entities within the system entity, has the robustness necessary to be a full description from these particular points of view. However, at this point these perspectives are totally separate, even though they arose ultimately from the same lost source. Entities are static, and signals are dynamic. The attributes of an entity could be tied to a set of signals. Thus, the changes in attributes that cause recalculation of slots may occur. In this way, entities may appear to change over time by the changes in their attributes. However, this type of linkage, which is the traditional one, is totally superficial. What is needed is a deeper fusion between the data and event perspectives.

Notice above that we linked signals to attributes. This suggests that the concepts of entity and signal are actually orthogonal to each other. In fact, when signals are attached to attributes, then these attributes may experience events. It is only the instances of an object that have attributes with values. Thus, there is formed a nexus where instances of entities reflect events in bundles of signals. This nexus is given the name “eventity.” By “eventity” is meant the same as Whitehead called the “organism” in his process-centered philosophy. Its ontological foundation is the “eject” described above in relation to Dasein. The eventity combines fully the features of the data and event views into a single common element. The eventity is more like a process than a static object in a frozen world. The eventity is the nexus of the orthogonal connection of signals to the attributes of instances of entities. But the eventity brings to bare the entire structuring of the data and event views on a single nexus of transformation. Transformation becomes the key idea because the eventity is a dynamic process of unfolding where bundles of signals are actively gathered by the entities whose attributes they are linked to and control. As signals differentiate within a bundle associated with a entity, the associated attributes differentiate. Since parts are associated with sub-bundles, and since relations between entities can change after time, the eventity gives an excellent means of modeling the dynamic parts of a system diachronically. The eventity is seen as a series of phases within an overall process rather than a static ever persistent object. Even though entities and signals are purely persistent, their combination can describe a dynamic nexus which is more like the “organism” described by Whitehead in Process and Reality. It is bringing together the overall structure of the data and event views that allows this representation of dynamic subobjects of systems. The “eventity” contains:

- grid location plus movement (spatial attributes)
- start time plus extent (temporal attributes)
- local timeclock
- instances plus attribute’s replicas
- branch and bundles of signals
- signals attached to attributes
- parts lists
- classes
- slots and equations

---

164. See ??? (Temporal Logic)
165. See Figure 54.
166. See Bib#.
167. See Bib#.
This is a fairly robust design representation yet it still lacks a crucial ingredient which is automation.

The addition of the automation to the eventity is a key transformation from a means of representing the elements of systems passively to the active representation a virtual processor that can simulate the dynamics of a system. This is a move from the passive eventity to an active eventity, which is also an automation. The automation is added both to the eventity proper and to the assembly of its parts. Certain attributes are defined as input variables, states and output variables. The input variables accept a stream of signals, and the output variables generate a stream of signals. The output signals are based on the input signals and the states. This is the basic form of the finite state machine which is generally known as a foundation of computer science. The eventity becomes a transformer that transforms input signals into output signals. The slot, with its equation that fires, is the mechanism by which the automation is effected. So the need for including this mechanism in the entity becomes readily apparent.

However, the eventity also has an internal assembly of parts which must be coordinated. This is done by making the assembly an automation as well. The states of the parts together with the state of the eventity determines the

---

FIGURE 6.2

**Perspective**

**Agent**

**Function**

---

**Horizon**

**Peripheral**

sensor or actuator

**External**

source or sink

---

**Context**

Network cluster

Envelope context bubble skin

---

**Facet**

Processor vortex of action

Transform chain of functions

---

**Attribute**

Task

Function

---

**Fate**

Population

CLOTURE

---

Kent Palmer
reaction of the eventity and the activators of the parts. The working together of the assembly automata and the eventity external behavioral automata allows the eventity to organize the behavior of its component parts as well as to have a holistic behavior itself. The eventity contains both accepting automation (the assembly) and generating automations (the eventity state machine) in a single structure in order to effect its role as a holon. From the outside, it has the behavior of its state machine, while on the inside, it is organizing the behavior of its parts with the assembly automata. The assembly is a cybernetic mechanism which is married directly to the automation to produce a self-governing mechanism that at the same time controls its parts. This is an “autonetic,” or self-controlling, mechanism. Thus, the eventity, augmented by dual finite state machines, may be called an autonetic eventity. As such, it is composed of the following parts:

**AUTONETIC EVENTITY**

- state list \( \{S_n\} \)
- current state \( \{S_i\} \)
- input mapping \( \{I\} \)
- output mapping \( \{O\} \)
- reaction mapping \( \{R\} \)
- activation mapping \( \{A\} \)
- automata definition \( \{Ixs\} \)
- assembly definition \( \{PSxs\} \)
The “autonetic eventity” is a fairly complex theoretical structure. It combines the conceptional structures of the pure data and pure event views of the system with the dual mechanisms of the automation and the cybernetic control structure. The autonetic eventity stands as a combination of the temporal and spatial viewpoints on a system. These two fundamental viewpoints arose from the always lost origin of the singularity and separated at the level of Dasein/eject to become totally reified at the level of subject/object. Within the gestalt of the observer/system layer, the entity and signal appear as the persistent focus of these perspectives. We discern the independent structuring of concepts within each perspective and how these can be recombined into a synthetic unity of the eventity which is equivalent to Whitehead’s organism in the process philosophy. The eventity combines the persistence of the entity with the temporality of the events. By taking this hybrid structure that allows the perspectives to interface and adding inward and outward automation, we gain a general purpose simulator for dynamic formal/structural systems. A system may be viewed as an ensemble of autonetic eventities. This ensemble is itself an eventity with its own assembly called a “synthesis” by which the parts of the system are controlled and coordinated.

This view of the system as an autonetic eventity composed of autonetic eventities provides a foundation for beginning to attempt to understand software methods. A software system is a good example of a dynamic formal-structural system. The concept of the autonetic eventity can be used to analyze the design of this kind of dynamic system. Design analysis and design synthesis are dependent on software methods. For only with software methods can we attempt to understand the dynamics of the system at a level of abstraction higher than the execution of the code itself. Software methods are a representation at the “system” level of abstraction. This level of abstraction is above that of the higher level languages commonly in use today. “Ada” and “C” are examples of these higher level languages. They are idiosyncratic in their construction. They contain many features beyond the three constructs necessary for structured code (branch, iterator, sequence). They do not deal with the software system as a whole, but merely define each statement. The interaction of all the statements, together with inputs defines the software ensemble. Whether these statements act together as a system is difficult to discern by looking at the sequences of statements alone. This is because of the fundamental problem of non-locality. Software design elements may have to be spread throughout the code to achieve a certain effect. Unless all the places in which the design element appears is known, along with how they work together, the effect of the de-localized design element cannot be known. Design methods attempt to deal with non-localization of design elements by creating a level of abstraction above the code where de-localized design entities may be conceptually localized. Localization insures that the conceptual structure will not execute. It is not executable in the sense of code statements. However, the design elements at the more abstract level make the multiple de-localizations at the code level comprehensible.

This level of abstraction is where entity and signal appear. It is the level at which the software system might be represented as a set of interacting autonetic eventities. Since the autonetic eventity is foreign to our usual concept of a design element, it will remain useful to speak in terms of the kinds of design elements that normally appear in design as well in order to make our point clear. Design elements are normally thought of as ensembles of statements that work together to perform a single duty. For instance, a queue may be a design element which is used in many places in a system for different ends. The design element has a whole range of forms and levels of abstraction that it operates at to accomplish its work. The design element may be something as lowly as a pointer, counter, timer, buffer or as large as a data base or communications protocol. Like the code, the traditional software design elements form a diverse set of abstractions that are highly idiosyncratic. Thus, the concept of an autonetic eventity as a higher level design element serves as a means of finding the middle level of complexity and abstraction to refine our concept of the design element.

We expect software methods to allow us to focus in on the way design elements work together to form a single system out of diverse elements. The system is a gestalt which works by combining diverse parts in coordinated efforts to achieve common goals. Understanding the system as a whole, by looking at this cooperation of design elements, may only be done by using software methods at the proper level of abstraction. It has already been pointed out that methods appear as the relation between design points of view. Here we are dealing with only two points of view: event and data. We expect here two minimal methods to arise in order to let us see data from the point of view of event and vice versa. We are focusing in on

168. See Bib#.
these two minimal methods in order to get a sense of how
to derive all the other minimal methods which relate all
the other permutations of viewpoints.

The two minimal methods that arise at this point are called
“data mutation” and “design element flow” methods.
They arise from the relation of the autonetic eventity to the
software system as a whole. For instance, the design
element flow minimal method arises in order to formulate
the relation between system states to design element
states. This has two forms. One form shows the multiple
transitions of the eventities in relation to system states,
while the other shows the multiple eventity states that are
the nodes of system transitions. In this way, the overall
coherence of the software system can be designed and
maintained.169

These two forms of the design element flow minimal
method allow designers to formulate the coherence of
design elements working together. Here, event views data
as flowing design elements. The events are the transitions
(either eventity or system), and the data are the state
values (either eventity or system). The method focuses on
a crucial aspect of the eventities working together in a
coherent way to give the impression of coordinated
behavior. This method focuses on the coherent relations
between state machines at different levels of abstraction.
It could just as well be applied to the relations of the
eventity automata and its parts. It is clear that this method
is directed at a particular aspect of the systemic overall
functions of eventities working together.

The other minimal method that arises at this point is the
information flow diagram of data mutation. The changes
of system and eventity states are just a special case of the
changes in the data components of software systems.
Most of the attributes and slots of the eventities that make
up a system will be in constant flux throughout the
execution history of the software system. The data
mutation minimal method addresses this other very
important aspect of the functioning system. The data
mutation minimal method has two aspects. The first is the
trace of the signals attached to eventity attributes. As
signals are modulated, they cause changes in the values of
attributes of the eventity. These modulations may be
recognized as events by the system when a difference that
makes a difference to the system occurs. These
differences that make a difference are information. They
are no longer pure datum because the system makes a
judgement as to the importance of a particular change.
Information flow within a system occurs when data about
changes that make a difference is moved around the
system in order to cause reactions within the system.
Events are recognized by using the slot equations. A

169. See Figures 55 & 56.
change in a particular attribute when it passes a certain value causes a certain slot value to change, and this signals reactions by the state machine of the autonetic eventity. This may cause other attributes of the system to be altered. Only by watching the traces of the signals of general eventities together, can this kind of reaction be discerned. This is one of the most basic software testing techniques. Pick a set of relevant variables, and watch how they change as their values are printed to the screen. The information flow between program variables is the most basic way of verifying that a program is working correctly. As with the design element flow method, the data mutation has two basic dual representations.

The information flow diagram shows the traces of data mutation. The information network shows the flow of information between attributes divorced from the signal traces. The information flow within a system is the manifestation of the internal temporality of the system. The coordination of information flow within the system is crucial to a system’s overall functioning. In fact, software systems are primarily characterized by information flows. This is why they are called information processing systems. In many instances, it is primarily the storage and manipulation of information which is the primary concern of the non-real-time software system.

We see here that information flow mapping and state coordination are the fundamental aspects of the system that are highlighted by these two methods. These two aspects work together to give the spacetime coherence of the system. The different variables that represent both states and variables must be spatially distinguished. They each change over time, one within a finite set of state transitions on different levels which the other varies continuously over wide ranges of values. These variations are monitored by the system and reacted to so that information flows through the system are related to input values. The causality manifest in the autonetic eventity is the combination of input values of attributes and the reactions of the state machines to these inputs. Thus, the two minimal methods encapsulate very fundamental aspects of the functioning of the software system.

This derivation of the two minimal methods that relate data and event perspectives has been carried out in great detail in order to show how perspectives combine to give rise to methods. As a speculation, it is possible that these two minimal methods are related to E. S. Bainbridge’s formulations of the dual representations of automata. Studied from the point of view of category theory, one arrives at the dual of a category by reversing the arrows. Arbib and others have assumed that this is an opposite automata where time runs backward. Bainbridge makes the intriguing case that in fact, that the dual of the state machine representation of an automata is an information flow representation. If this is true, then it may well be that these dual representations of automata find their manifestation in these two minimal methods. The duality of automata representations would be a very good foundation for these minimal methods. However, it is not clear whether this connection to Bainbridge’s work is justified.

A general discussion of the impact of this derivation of these two minimal methods follows:

Software methods address the whole software system. Each one gives a unique view of that whole. Each perspective as subject views the other perspective as object. Thus, between each set of two perspectives on the software system, there are two one-way bridges. Associated with each of these bridges is a software method that combines elements of each perspective in a unique way. Thus “data mutation” combines time and event to produce information flows and nets while “design element flow” combine time and event in a different way to produce coordinations of state machines. These minimal methods are meta-essences. If objects have essences, then systems of objects must have meta-essences. An essence is the set of attributes within certain limits which an object needs intact to maintain itself. It has been said that essential attributes are distinguished from accidental attributes. A meta-essence refers to objects embedded within system gestalts. Meta-essences govern the presentations within gestalts. In a gestalt, a series of objects are presented in a certain order. Under certain circumstances, one object is brought to the fore for presentation, while others are submerged into the background. In another circumstance, other objects are brought to the fore. Meta-essences govern the presentational sequence in a gestalt. This presentational patterning is the heart of the dynamic formal-structural system. The patterning is the meta-essence of the system. It determines what is essential and what is accidental within the context of the system. It determines what

---

170. See Bib#.
objects are necessary and their ordering within the system. If a system has a single meta-essence, then it does not have a singularity. However, a system with a singularity has multiple meta-essences which are united by the singularity. The meta-system has the singularity as its core, while the system has the meta-essence as its core. Within the meta-system, the set of meta-essences organize the relations between perspectives. Meta-essences are thus very important in the study of both systems and meta-systems. The expression of meta-essences within software systems are as minimal software methods. They serve as bridges between perspectives and thus give unity to shattered perspectival fragments. Here we have focused in on the relation between data and event views of the software system. It has been seen how elements from these perspectives combine to give rise to these methods. The methods focus in on an aspect of the system which has global significance. What is true of these two minimal methods (data mutation and design element flow) are true of the other ten minimal methods as well. Each minimal method distinguishes the perspectives it connects, while pointing to a unique aspect of the global system.

What is important, however, are not so much the minimal methods themselves, as what is seen in the area of reversibility that is discovered by oscillating between them. As we develop systems, it is necessary to iterate between pairs of perspectives. In this oscillation, we glimpse the heuristics which guide toward harmonic design. Methods are only a language for representing system-wide design issues. They are a means for communicating design ideas and representing aspects of final design decisions. Methods usually include a set of procedures for applying minimal methods in order to elaborate the design. But methods do not tell you how to pull a good design out of a wicked problem which is multiplying constrained with no obvious optima. Methods usually include some heuristics that attempt to give some

---

**FIGURE 6.5**

Mathematical notation and relationships between macro and micro levels of virtual machines.
guidelines in this area. The heuristics are normally few and far between. They are the words of wisdom of very experienced system designers. An example from structured design of Yourdon-Constantine is to stress cohesion and minimize coupling. This is a famous example of a heuristic for software design. These heuristics normally appear very sparsely in texts on methods, buried as pearls of wisdom among the methodological paraphernalia (such as how to draw the right diagrams and what are the steps to follow). But these heuristics are very important pointers toward the way to instill an aesthetically pleasing and practical coherence into the designed system. These heuristics cannot be represented within the methodology, but only indicated. In designing a system, one is attempting to instill a mutual dependence among the varying parts of the system. This mutual support and internal coherence is an aspect of the meta-system. It must be instilled despite the barriers erected by perspectival fragmentation. It is only glimpsed as one jumps from one perspective to another. It is a holon that exists in the gaps between perspectives. Thus, it is not captured by any one perspectives nor by the minimal methods that attempt to bridge the gaps between perspectives. These heuristics are knowledge of what makes systems harmonic and how to instill that harmony. As holons, they are glimpses of the wholeness of the system. As knowledge, they exist at the level of knowledge, not information. Information is built into the software system. Knowledge informs the design of the software system, but is not captured by it. The holon/
heuristic is captured at the next meta-level of Being -- Wild Being. It is not captured at the level of the singularity in the meta-system. Artificial intelligence deals with the application of heuristics and knowledge representation. Software engineering encodes knowledge into the design of the system, but then throws it away during implementation.

The real work of developing software engineering methods is to attempt to capture these heuristics to guide design. They are seen by passing back and forth between minimal methods in design iterations. In order to make this discussion more concrete, I will suggest a possible heuristic to describe the holon that appears between the data-mutation and design element flow minimal methods.

**Heuristic 1. Do not treat design elements as clockwork mechanisms, but treat them as flowing, transforming elements with their own life cycle within the life cycle of the system.**

**Heuristic 2. Minimize coupling and maximize cohesion of system entities by reduction of information flow across eventity boundaries.**

Heuristics such as these attempt to tell you how to build a better system in the language of the methodology. They strive to fulfill the prerequisites of mutual support between parts of the designed system. They instill the harmony of mutual support into the system from the perspective of the meta-system. This harmony should lead to simplicity, elegance and aesthetic balance in the designed system. It leads to increase in what Alexander calls “the quality with no name” called by the Chinese “the Tao” or “the Way.”

Software engineering needs to discover these heuristics that indicate holonomy -- the wholeness of systems. Today our discussions focus on concrete computer science techniques, or, at best, methods. The true heart of the software engineering discipline is the discovery of the heuristics that should guide our use of methods. This should be the focus of our research, giving life to our methods and instilling harmony into our software systems. As with all other disciplines, the most harmonious is usually the optimal.

Up to this point the focus has been on those points of view associated with space and time. As was pointed out earlier, these points of view are easy for us to comprehend because they are fully ordered. Cartesian coordinates and the real number line allows us to give full ordering, in terms of linearity and distance, to all points in space and time. We are taught to deal with this illusory continuity in mathematics from an early age. This mimicking continuity found in the world, mathematically seems natural to us. So natural that it goes unquestioned. Unfortunately, we discover that digital computers have some difficulty mimicking continuity. It is necessary to construct elaborate work around like floating point numbers in order to maintain the illusion of continuity. The digital nature of computers leads us to view mathematics differently. Finite mathematics takes on a new significance. Calculus begins to take a back seat. It has provided the defining framework for the Western view of the world since its invention. Now we begin to explore the granularity of the illusory continuum it offered us as a haven for so long.

With respect to space and time, we notice that our descriptions of these fundamental categories need no longer mimic the continuity we find in the world at all times. In the artificial digital universe, it is possible to relax our simulation of continuity. This is done differently for space and time. With space it is possible to relax linearity to get partially ordered distances. With time it is possible to relax distance to get linearity without distance. In non-real time systems, these relaxations cause great simplifications to occur in system design. For instance in human interfaces, it is many times only necessary to get the sequence right, and as long as response times are not too long, it is not important exactly how long the response takes. Or again, in data base systems search trees may allow efficient retrieval of information which is not sorted into a linear order. The distances between information elements are just a few algorithm steps up and down the tree which would be a great distance from the perspective of a linear search. Multiple search trees for the same data base makes the stored sort order irrelevant. In digital systems, these and other similar relaxations of space and time constraints allow many useful information systems to be built. However, real-time systems demand the full articulation of spatial and temporal order in order for the system to adapt to the events in the world. Meeting real world time deadlines and the spatial movement of information across a computer network becomes a crucial aspect in these systems. This is unfortunate because even though space and time views of the system can expand to handle these constraints, the other important viewpoints, which will be discussed next, cannot handle this expansion. These other viewpoints are agent and function. As long as we are not dealing with real-time systems, this
discrepancy does not show up. The shift to real time systems opens a rift in the relation of software methods to each other which does not appear in conventional information systems.

This is important from the point of view of methods. The space and time-oriented methods we have just described are associated with the partial ordering of linearity without distance and partial ordering with distance. Partial ordering with distance is related to the design element flow minimal method by which state machines are coordinated. The linearity without distance is related to the data mutation minimal methods by which information flow within the system is coordinated. Information datums have no distance from one another. State machines are not necessarily sequential. These two minimal methods exemplify the digital system in a relaxed state, not necessarily coordinated with external real events in space and time. However, for the digital system to mimic external continuity, state mechanisms and information flow must be coordinated together. Information takes time to flow across the network, and state machines can be made to operate sequentially. For instance, the addition of an executive cycle to a real-time system is a crucial addition that adds a basic sequential aspect to a system that might merely be interrupt driven. The minimal methods of data mutation and design element flow must be used together to gain the leverage in system design necessary to produce an adequate spacetime representation of the real-time system. Because they must be used together, the oscillation between viewpoints (event and data) is guaranteed. This oscillation allows glimpses of the holons in the chiasm of reversibility that exists when one jumps between viewpoints. These holons are fleetingly represented by heuristic aphorisms that tell how to use the methods to achieve a harmonic design. Examples of these heuristics have been offered.

Methods give us a language by which we can talk about digital software systems design. Methods, used together, provide leverage to approach the difficult problems that arise when designing extreme systems. Real-time systems are extreme in the sense that the digital must adapt to and mimic the continuous. This goes against their nature. The advantages gained by the relaxation of time and space constraints is lost. The rift between minimal methods opens up. Seeing these rifts, we gain insight into the relations between the data and event viewpoints. The holons of harmony are glimpse. This means that the language of methods lead us to a point beyond this
language. Methods are meta-essences that make visible the harmonic cores of systems ensnared in meta-systems.

Now it is possible to switch from the discussion of the data event perspective to the discussion of the agent and function perspectives. These two perspectives are fundamentally different from data and event. A similar derivation of minimal software methods will be attempted in this case as well. But that derivation will take on a completely different character. This character is motivated by the nature of the inherent ordering of these perspectives. These perspectives, agent and function, are inherently only partially ordered. We are not used to dealing with partial orders. They do not fit our training that always assumes full ordering. So when we are confronted with partial orders, we think there must be some mistake.

The real question is why are different perspectives given to different levels of ordering. We would expect them all to be the same. Perhaps this difference between perspectives is an illusion? Because we expect full ordering, as if it were “natural,” we suspect there is something wrong with our analysis of these perspectives. Instead we should suspect our “natural” prejudice for continuity which is misplaced within the digital world of computers.

It should be clear by now that every conceptual structure has its dual. This is a fundamental “law” of category theory in mathematics. It should follow directly from this “law” that the conceptual structure just described that related the event and data perspectives requires a dual. That dual is the conceptual structure that appears between agent and function perspectives. As we explore this
“dual” conceptual structure it will become apparent just how different the duals can become while still bearing an internal relation to each other. The dual of spacetime/timespace is not just an inversion of superficial properties. It is a fundamental inversion of deep conceptual relations. Because of this, the complementarity between these duals has a profound significance for understanding systems and meta-systems.

The place to begin is a general consideration of the lopsided nature of the two sets of perspectives. In the case of the spatio-temporal perspectives, the definition of the perspective itself was quite complex, while the minimal methods which connected them were fairly simple. For these new perspectives, exactly the opposite is the case. The definition of the perspectives is simple, but the connecting minimal methods are very complex. This shift in complexity is an important phenomenon. It shows that the inner relation between the dual sets of perspectives is very unusual. It portends a deep structuring by the singularity of pure immanence. The displacement of complexity in the dual, like the displacement of images in dreams that belie the unconscious, gives us indirect evidence of the action of the always already lost source of

![Diagram](image-url)
the dual images of the interlocked perspectives.

In the technical world, it is taken for granted that everyone is asleep to the significance of images and their relations. The text of the technical is rarely read (in the sense of hermeneutics). Both because of schooling and proclivity, engineers disdain the efforts of literary critics. Because of fear of lack of competence, the literary critics are held at bay. Thus, the technical and meta-technical worlds form a closed system, seemingly immune from criticism. Yet it is precisely here that the deep structures are revealed in the most extraordinary fashion. For instance, this study finds great significance in the relations between software methods. Surely these significances will be denied. Yet the archeology of knowledge should be able to treat any cultural artifact, finding human significance in any of them. In the engineering realm, this significance is denied more strongly than any other field of human endeavor. This alone should make us suspicious. Yet Heidegger opened up the technical to us in Being and Time, giving it a deep meaning for the first time. This study seeks to do the same for the meta-technical. Meta-technology is the as yet unrecognized spinoff from technology. It is technology taken to its limits where it collapses into the proto-technical. The ultimate meta-technical event is nuclear war in which the super powers cancel each other, leaving only the residue of savagery -- if any humanity at all. This is the age in which nuclear war hangs like a pall over every event giving it a changed meaning. Software makes nuclear war feasible, even if it makes nuclear defense infeasible. Thus, the post-modern age is the meta-technical era where all meanings are changed by the possibility of annihilation.

Space and time comprise the field of action within which the system functions. Together they form an opening, an open place, within which manifestation occurs. Space and time together give a patterning to that open place that determine what can happen there. This intrinsic patterning has been reified into our mathematical metric which describe position and occurrences as "backdrops" (Klir) against which object attributes are measured. The openness of space and time, as a stage upon which action takes place, is of primary importance for the articulation of the system. However, there is a dual to this openness which is little noticed in the literature of systems theory. That dual may be characterized as a kind of closedness, or "closure." In this case, the closure appears as the relation between the observer and the observed. Within the openness of space and time, the observer "closes in" on the observed. Normally, we think of the system as the observed which is apprehended by an ideal scientific subject which is never specified. That observer exists in a kind of "nowhere" which is the origin of a spatial perspective that is associated with a particular vanishing point. The ideal observer becomes a position which we can all assume under the disguise of our scientific persona to view the system objectively. From that position, we feel safe from involvement in the workings of the system. However, recent quantum physics, and experiments in social sciences, both lead us to the conclusion that this separation of observer and observed is a delusion. In fact, there is an intimate inexplicable link between observer and observed as idealized concepts. The observation of the system changes its operations. As a corollary, we might guess that the observer is also changed by the interaction with the observed system. Thus, mutual effectivity of observer and observed is a key concept which must be taken into account by our system’s theory. The objective description of Klir, which all but forgets the action of the observer on the system, must be replaced by a system’s theory in which the closure of the observer on the observed and vice versa is recognized.

It is not surprising to find that once this closure of observer/observed is recognized, it moves right into the understanding of the system itself at a fundamental level. It constitutes another pair of perspectives on the workings of the system that now includes the observer within it. The observer becomes part of the system, and the system becomes part of the observer. Their interaction form the basis for gauging all other interactions within the system, and between the system and the meta system, or its subsystems. The openness of the clearing-in-being is balanced by the closure of the observer/observed interaction which it contains and supports. At a more basic level, the openness and closure merge into the manifestation of Process Being, witnessed by Dasein as being-in-the-world. However, after the reification of separating the nihilistic opposites of subject and object, which deny all other actions other than dominance that subject the object, then the differentiation of openness and closure become inevitable. When the reified elements become active and are seen as dynamic system/observer interactions, then openness and closure assume a primary importance as the dual means of the apprehension of these

171. See Bib#.

172. See Bib#.
interactions. The subject/object relations are purely static delineations of territory within the openness of the clearing-in-being. The observer/system interactions allow the dynamic interplay between the territorial positions. However, for the most part, the free play of exchange between positions is limited, and no territory is gained or lost. The subject/object dichotomy anchors the observer/observed relations, so there is no loss of identity for the ego in the interactions with a particular system. This kind of anchoring may be seen clearly in Klir’s general system’s theory divorced from any phenomenological sophistication. Unfortunately, it also leaves us with a system with no internal coherence that has been reduced to a mere object. The subject is safe from involvement, but at the price of not really ever understanding the system as anything other than a random collection of attributes chosen by the subject. The interplay in which the “objects themselves” suggest their own articulation and interact with the observer, is lost. The result is ignorance that masquerades as knowledge. Phenomenology suggests we
listen to the objects themselves and allow them to suggest their own coherence and the ways they are best understood. Objective science explores the terrain of nature only on its own terms. However, the will-to-power that motivates their need to dominate is based on a massive insecurity. One might be led to say an ontological insecurity because it stems from the projection of the subtle clinging of “Being” upon a world which is inherently void. The observer/system interaction exhibits a limited dynamism tied directly to the subject/object underpinnings which have reified the world into a totally static configuration. It is, however, necessary to allow the free play of these interactions in order to realize fully the character of systems that have been released from the prison of objectivizing dominance by the subject. In such systems, the observer/observed interaction becomes a fundamental relation upon which all other systemic relations are based. The closed borders of the subject and of the object are thrown open. In the interaction, observer and system merge into a new dynamic whole. The system is affected by the observer, and the observer is affected by the system. The observer becomes part of the system, and the system becomes part of the observer. In this merging, there is a closure in which these two, at times, become indistinguishable, or at least closely bound up with each other, so different aspects of the meta-system that embraces both, appear as one at one time and as the other at another time. Observer and system-as-observed form subsystems of a single meta-system which have dynamic interchanges with one another. The closure of the observer on the observed system, and the counter closure of the observed system on the observer might be called a “cloture.” The cloture is like an in-folding or mixing, which is the opposite of the unfolding of the openness within which the observer-system (ob-sys) occurs. Because the nature of cloture is closed, interlocking, and obvolute, it is not as well articulated as its dual of spacetime/timespace. Yet it is spacetime/timespace that ultimately bind the observer and system together, if nothing else. Yet we know that “spooky action at a distance” also exists because of the proof of Bell’s theorem. Thus, the intertwining of observer/observed is inherent in the nature of things.

This intertwining is called “cloture.” Space-time unfolds from the singularity. In the singularity, (the always already lost origin) there is a single limit. This limit splits to give rise to the interval in which space and time are mixed. The meta views of spacetime and timespace arise as two ways to look at the interval, and from these arise the concepts of separated time and space which are then artificially merged again in our metric conceptions. This opening out of the interval from the singularity is matched by an opposite movement which is here called cloture.

By Bell’s theorem’s proof, we now know that particles that are once related never cease to be related, no matter how far apart they become. Somehow the separation of things in timespace is inessential, whereas relatedness is essential. The openness of the interval of spacetime/timespace is somehow superficial. This means the universe, no matter how big it expands, is somehow fundamentally interrelated because it all started by means of a single event -- the Big Being. This interrelatedness pervades everything. The singularity from which the universe arose is, in some way, identical with the whole of the universe itself. The common origin ensures a deep interrelatedness that Chung has called “interpenetration.” This is the highest form of harmony. Beyond this fundamental interpenetration, due to common origin of everything in the universe, there is the further interrelation of mutual dependence which occurs whenever two things interact in the universe. The interaction of observer and system is no exception to this. Once interaction occurs, then mutual dependence follows, which ultimately is based on interpenetration. This interlocking occurs when one system impinges on another. The trace of the interaction is carried away with the two parties. No matter how far they get from each other in spacetime, there is a basic interrelation through the medium of their interpenetration which is the trace of the singularity (or their common origin within). The interlocking of observer/observed-system is a cloture by which interaction leads to mutual dependence based on interpenetration. In the cloture, there is a relation of belonging together or “sameness” like that described by Heidegger. A web is formed by everything that interacts as they leave their traces on each other. This web grows stronger with prolonged contact until the web becomes a seamless fabric of mutual interrelation. Mutual interrelation taps the source of the interpenetration that ultimately unites everything. The web of cloture is just as important as the opening in which it occurs. In fact, the web that fills the opening is in some ways more basic. Both the opening and the web arise from the singularity. The opening is an unfolding from the single limit of the singularity into the multiple limits of the intervals within spacetime/timespace. The web of cloture is the creation of mutual dependence through interaction based on the traces of the

173. See Bib#.

174. See Bib#.
singularity left on everything which appears as the interpenetration of all things. From the web of cloture arise ecological systems and cultures. Rupert Sheldrake has begun to explore phenomena related to cloture in The Presence of the Past.\(^{175}\)

In fact, all meta-systems are webs of cloture. Meta-systems that do not contain singularities are merely environments. In environments, different systems interact and form mutual dependences. These mutual dependences tap the interpenetration of the subsystems to produce a deep harmony. In some cases, the meta-system will represent to itself its own always already lost origin as a singularity. When the singularity appears, the perspectives on the system fragment. This reification and reintegration of perspectives allows the singularity, or lost origin (the impossible), to haunt the meta-system without ever being observed. Cloture exists not just between observer and observed within the meta-system; cloture exists between each element of a system. Any element of a system may be considered as observer or observed by any other element. The elements of systems take account of one another by mutually affecting each other in their interactions. They react to each other. It is the sum total of these natural reactions that make up a system. This is, in fact, mutual observation. Observation is not a passive perception. Observation is a reaction to the other. It is the sum of these reactions to others, and the internalization of this set of reactions, which defines the self of each entity within the system. G. H. Mead made this position clear in his analysis of society in terms of symbolic interaction.\(^{176}\)

Each entity is defined in terms of its multiple cloture with all other elements within a system. Multiple cloture is the key for understanding how the two other perspectives (agent and function) on a system arise. Think again of the difference between essence and accident. The essence is the set of attributes and their interrelation which are necessary for anything to be “what” it is. The accidental attributes are those that can change without causing a thing to change kind. Any entity within a system may be considered with respect to its essential or accidental characteristics. By considering its essential attributes, we assign it to a class. Normally, accidental attributes vary over time, while essential attributes tend to vary less over time. When we are attempting to pin down what remains of a thing, it is the essential attributes which are sought. The essential attributes give a thing its Being or persistence. An instance of an eventity is distinguished more by its accidental properties than by its essential properties. Accidental attributes distinguished instances of the same kind. Kindness is dependent on persistence of essential attributes.

When an eventity is thought of as an observer of all other eventities in a system, it becomes clear that the multiple cloture defines each eventity instance, giving each its unique position within the system. This diacritical definition, whereby the meaning of each element in the system flows from its relation to all other elements, obviously gives a new twist to the accidental/essential dichotomy. It is clear that any system is made up of a certain set of different kinds of eventities. These eventities have specific relations based on the interaction of their persistent attributes. But also, each eventity probably has multiple instances that all define each other diacritically by their accidental properties. Thus, diacriticality occurs on two levels. It occurs on an essential level by the relation between eventities of different classes, and on an accidental level by the accidents that define each unique instance. Essential diacriticality may be viewed as independent and orthogonal to accidental diacriticality regardless of the spacetime positions of the instances. The dynamic nature of systems are dependent on the interplay between these two forms of diacriticality. The system can be recognized as the same over time because of the essential diacriticality between kinds of eventities. In can be recognized to change because of the accidental diacriticality between different instances. Actual movements in space and time may be irrelevant here, but usually these form a third level of important change. Changes in essential diacriticality are called the evolution of the system. Changes of accidental diacriticality are called the system dynamics of the system. Change of levels of instantiation of different eventities is called population fluctuation. Changes in spacetime position of instances are called demography.

The essential diacriticality exists because within a system there are persistent and changing relations between the essential attributes of eventities. Accidental diacriticality exists because within a system there are persistent and changing relations between the accidental attributes of instances of eventities. These two types of diacriticality are different from the essential and accidental attributes of the eventities themselves. These types of diacriticality are two complementary aspects of the web of multiple cloture between all elements of the system. These two aspects are, in fact, the basis of the other two perspectives on the system called hitherto “agent” and “function.” The difference between agent and function has been

\(^{175}\) See Bib\#.

\(^{176}\) See Bib\#.
characterized by the difference between what and who. Who you are is almost entirely dependent on your accidental attributes. Your date of birth, address, social security number, name are all things that are inessential, but they are the very things that identify you. These accidental attributes answer the question as to who you are. On the other hand, what you are distinguishes your kind from all other kinds. For instance, what you believe, what you do for a living, your education, your genetic make-up. These are all essential to what kind of person you are. Perhaps they were originally random as well, but if any of them were changed, you would be a different kind of person. The difference between “who” and “what” is fundamental to our way of thinking about any population. Klir mentions space, time and population as basic backdrops for attributes of systems. But population is a complex concept. Separated from concerns of demography or numerical ratios, population becomes a combination of who and what. And these separate components of population actually separate into two independent perspectives on the eventities that make up a system. This is similar to the way space and time are separated from spacetime or timespace.

With respect to systems, just reducing these perspectives to “who” and “what” related to the attributes of entities and their instances, is too simplistic. Actually, it is not the essential and accidental attributes that determine these perspectives, but essential diacriticality and accidental diacriticality. Essential diacriticality is related to system evolution, and accidental diacriticality is related to system dynamics. System evolution is related to functional
process of transformation. System dynamics is related to a locus of action as an identifiable agent.

Essential diacritical relations are not static. If they were static, then there would be no system. Systems are normally characterized by the chains of functional relations that constitute it. These chains are called processes. A process is a specific set of transformational steps that lead to a specific end. All systems contain these chains of transformations made up of functional steps. The chain of transformations is held together by the persistent essential attributes of the different kinds of eventities that make up the system. On the other hand, each system will be made up of different vortexes of action where different kinds of eventities work together. A vortex of activity is not necessarily the same as a chain. Different vortexes of activities, called virtual processors, may perform separate functional steps in a single process, or all the functional steps of a single process may occur in a single virtual processor. Activity vortexes and chains of transformations depend on eventities and their instantiations. However, these are completely different concepts from that of the eventity. The eventity takes inputs and produces outputs. But the differences between inputs and outputs do not necessarily constitute a transformation. Nor is an eventity necessarily an activity vortex, a transformation or an activity vortex may be in the interspace between several eventities that cooperate together to perform the transformation or make up an activity vortex. In some sense, the transformation and activity vortex are absences rather than fullnesses. It is precisely because they are absences that an eventity can step into that space and fill the role. Yet the role was there even if there is no particular concrete thing that can be identified to do all the work. It is better to think of transformations and virtual processors as two types of operating areas between groups of cooperating eventities. Associating them with a positive structure is a mistake of misplaced concreteness and leads to misunderstanding the nature of systems. Systems affect chains of transformations called processes through the cooperation of different kinds of eventities working together. Systems are composed of many different virtual processors which are vortexes of activity by cooperating eventities. Virtual processors and a chain transformations may or may not overlap. The voids that are vortexes of activity, or functional processes, may or may not be filled by a single eventity affecting these changes by itself. More likely, in complex systems, vortexes do not overlap chains, and there is no functional or processing eventity. Groups of different kinds of eventities cooperate to form vortexes or chains. This is why we need the concept of system over and above the concept of object. Systems are precisely a set of cooperating eventities that form vortexes of activity.
that are virtual processors or chains of functional steps that form processes. In processes, things change kind. The process transforms its inputs from one kind of thing to another. In activity vortexes, things change who they are. If nothing else, who is active at a given time changes. Activity is the fluctuation in the accidental attributes of one or more instantiations. Transformation is the change in the essential attributes of one or more instantiations.

We have grave difficulty thinking about anything that is not a concrete thing. Vortexes and chains of transforms have no substantial thing to pin our hat on. Yet systems theory that concretizes everything within a system is in grave danger of misunderstanding the whole system. We will use terms like nexus, locus, vortex and chain to describe these sets of changing diacritical relations within the system. We will attempt to avoid false concretization. Vortexes of the actions of virtual processors are seen from the agent perspective. Chains of transformations that form processes are seen from the function perspective. These two perspectives appear as two aspects of the diacriticality of multiple closure within the system. The “who” of an eventity within a system is determined by accidental diacriticality between instances of different kinds of eventity. The “what” of an eventity within a system is determined by the essential diacriticality between eventities of different kinds of eventity. But who and what need not be tied to a particular eventity. An agent may be a group of interacting eventity instances. Likewise, a chain of transformations may be affected by a group of interacting eventities. These locuses of transformation or activity can change the what and who of other eventities within the system. An eventity may either be acting, acted upon, or both, in respect to a particular locus. The locus has a function, or an identity as an agent, which is independent of whether it is associated with a single or a group of eventities.

Once the agent and function perspectives have been clearly identified, it is possible to begin the derivation of the minimal methods associated with the bridge between these two perspectives. We will begin by differentiating how these perspectives view the different levels of harmony. At the horizon, the agent sees peripherals in the world which are either sensors or actuators. These are normally specialized hardware equipment that the system interfaces to in order to be connected to the world. At the horizon, the function sees the externals which are either sources or sinks for its transformation resources or products. The context for the agent is a network of processors which form a cluster interconnected by communication channels. The context for the functions are the skin of the system that forms an envelope around its chains of functions. This is normally denoted by a context bubble in structured analysis. The facet focused on through the agent perspective is the virtual processor. This is a locus of coherent activity. The facet focused on through the function perspective is the transformation. This is a chain of functions that lead to a particular result by a set of specific steps. The attribute that the agent looks for within the processor is the task. The attribute that the function sees within the transformation is the function. The task, or the function, are the threads of coherence within their respective locuses. A virtual processor may have several threads of coherence within its activity vortex. A transformation may have several threads of coherence within a transformation. These threads of coherence are very important. They make it possible for the system to be coherent as a whole. Weaving together all these threads of coherence within a system is what differentiates a system from an object. Having the locus of activity and transformation is not enough. These are only theaters of operation in which the coherence of the system may be expressed. Good design, using the language of minimal methods and guiding heuristics, together with insight, make it possible to confer coherence on the software system. Eventities are the warp of the system, while the threads of coherence form the woof of the system.

The entity-relation description of these two perspectives, as shown, is very simple, and has already been explained for the most part. A detail is that processors normally have a physical substrate in a central processing unit. However, this physical hardware substrate may be many levels removed by layers of virtual processors. Tasks time share the resources of this hardware substrate. In this essay we do not use the word “process” to signify an independent task as it is used in Unix. Tasks may or may not communicate or share memory. Tasks are independent threads of execution within a virtual processor. Tasks may contain a hierarchy of sub-tasks.

Each transformation within a system has a transformate and a transformand. The former is the operator, and the latter is the operand. Each transformation may contain
META-SYSTEM

DESIGNED SYSTEM

META-Structures

META-Models

Ant System

Anti-Source System

Anti-Data System

Source

Sink

Wild Software Meta-systems

FIGURE 74
independent functional threads that are interwoven functions that perform a specific step within one of these threads. They have specified inputs and outputs. Functions have implementations in standard programming languages like C or Ada. Functions are normally algorithmic. All functions or transformational chains may be expressed by the control structures of structured programming, i.e. iterator, selection and sequence. A transformation chain may contain multiple functions performed at each step. So chains may have subchains or branching chains. Functional chains are always expressed as action verbs. All functions must be executed by a processor within a task to become actualized. The activity of the action verb is performed in the vortex of activity. Outside that vortex it is static. Real transformations only occur when the vortex of activity in the virtual processor is brought together with the functions that are implemented. Implemented nonexecuting functions are static code. Implemented executed functions perform active transformations in time and space. A processor that is not executing implemented functions is idle, even if tasks are running. Pointing and grasping must work together for real work to be accomplished. The agent embodies pointing, and the function embodies grasping. The
processor is pointing at the currently executing instructions of the implemented function. The transform grasps and manipulates the inputs, turning them into outputs. A processing transform does actual work with concrete results in space and time. A processor without a transform is using time but not producing any spatial changes. A transform without a processor takes up memory but does not produce any changes of inputs to outputs in time. The processing transform executes the threads of coherence in the system. These are threads of coherence with the locuses of action and the locuses of transformation.

Within the system, eventities participate in processing transforms. All four perspectives from the meta-system onto the system are brought together in this statement. Eventities combine the event and data perspectives. Processing transforms combine the agent and function perspectives. When eventities participate in the processing transform, an executing software system is the result. The system viewed statically would discover several independent elements. There is the eventity, its autonetic substrata, the processors and the transformations. Processors and transformations only touch in a single moment in time. The processor plays the music of the transformation chain of functions. As it plays the music, the keys of the autonetic state machines are touched, producing different effects. The eventities are the different instruments in the orchestra of the system.

Notice that processor and transform fall apart when not held together in time and space. On the other hand, eventities persist to hold both space and time together, whether or not the system is executing. Openness produces unity, whereas closure produces separation.

Next it is necessary to explore the minimal methods that combine these perspectives and act as a bridge between them. In this case, the methods that are minimal are complex. This complexity stems from two sources:

1. Agent and function are only partially ordered, and
2. Agent and function do not combine easily but must be forced together unlike event and data. A processing transform is an overlapping of locuses, not a thing like an eventity.

Both agent and function are described independently as nested tasks or nested functions. Other than the partial ordering of that hierarchy, they have no internal structure of their own. This structural poverty allows them to express meaning. The structural richness of data and event cause them to not be able to express meaning very well. Processors and transforms are merely diacritical markers related by partial ordering. They have no real structure of their own. They simply reflect important points in the systems structure.

These hierarchies of agents and functions must combine to produce minimal methods. When the two hierarchies are played off one another, there is enough information to produce either partial order with distance or linearity without distance.

In this context, linearity without distance can be associated with the “virtual machine instructions” minimal method. Partial ordering with distance is associated with the “mapping between functions and tasks” which is the other minimal method connecting agent and function. Virtual machine instructions are the embodiment of functionality within the task. The “distance” between the instructions has no meaning. The instructions are executed in a linear sequence by the virtual processor. The virtual machine instructions are the implementation of the function which allows it to be executed by the virtual processor. Both virtual processors and virtual machines may exist on many levels within the software system. By turning a function into a set of virtual machine instructions, the processor is then able to execute the function. In this way functions are executed one by one in a chain to complete a transformational process. The processor does its processing by executing the virtual instructions that comprise a virtual machine.

A virtual machine is a composed set of instructions. Each instruction is a lower level virtual machine. Nested virtual machines are executed by nested virtual processors. The term “virtual” here indicates that the processor is not necessarily a hardware implementation. The processor could be emulated in software which runs on a lower level hardware or software processor. Virtual machines perform transformations. They are implementations of state machines. The instructions are fired in a certain order, given particular inputs to produce specific outputs. Part of the instructions functioning is the reset of the state of the machine for the next invocation of the macro instruction.

The relation of macro to micro instructions may be shown

---

177. See Bib#.
by a structure chart diagram, as those that Yourdon-Constantine used in structured design. The sequence of calling micro instructions is determined by the state of the macro machine and the inputs (or parameters) given to the macro machine. Virtual machines are used to implement the state machines of autonomic eventities. They are abstract implementations which make functionality processible in a way that also embodies the automata. An automata is a mapping from inputs to outputs and states. This mapping may be a table lookup, or more commonly it is an implementation of some functionality which makes the transformation from inputs to outputs. In this latter case, outputs are not already available but must be made. Making outputs from inputs and available data is a transformation performed by the virtual machine. That virtual machine also resets its own state to be ready for the next invocation. By resetting its own state, it is able to participate in the external functioning of the system which expects different responses in different circumstances. When a virtual instruction has its own state, it is a virtual machine. It is multi-functional. On the other hand, an instruction may not have its own internal state, in which case it is just a straight implementation of a function. Functions transform inputs into outputs regardless of external changes in the system as a whole. They always do the same job. Many times virtual instructions are merely functional implementations.

When virtual instructions are associated with particular eventities, they are called operations or “methods.” In this case, the operations change the values of attributes or slots within the eventity. In object-oriented design, “methods” are the only means of changing data encapsulated by the eventity. We will make the distinction here between operations and workers. An operation (or “method”) is a virtual instruction that changes an internal attribute of an eventity. A worker is invoked by the state machine of the eventity to transform input to output and change the state of the eventity. Operations are visible outside the eventity, while workers remain hidden. In constructing virtual layered machines, the lowest level instructions ideally all end in operations (or “methods”) on eventities. The word “method” for operations in persistent data will be avoided.

However, some virtual instructions are pure transformations of inputs to outputs with no persistent data being changed. These pure functions, which exist to some extent, cannot be associated with a persistent object. These tend to be garbage leftover when a purely object-oriented design method is used. This problem with purely object-oriented methodology has been identified by Vaclav Rajlich\(^{178}\) among others. The problem is solved by using the Object-Oriented Design/Virtual Layered Machine (OOD/VLM) method proposed by Nielson and Shumate in the book *Designing Large Real Time Systems with Ada*.\(^{179}\) This book, and their article with the same name, should serve as a major source on this methodology. The methodology gives concrete implementation to functional transformations for a particular level of virtual processor. Functional transformations are accomplished by the processor executing the virtual instruction. Virtual instructions may or may not be virtual machines with their own state values. A virtual instruction with no state values is a simple transformation. A virtual instruction that is also a lower level virtual machine is multi-functional, reacting differently to external system states at different times. The virtual processor and the virtual machine intersect at the executed instruction.

The other minimal method that serves as a bridge between agent and function is the mapping between tasks containing virtual machines and functionality. This mapping allows us to see how the total system functionality is to be carried out by the threads of coherence (tasks) executing the virtual instructions of virtual machines. This traceability, from the functional decomposition of the system to the specific elements that implement that functionality, is very important. In order for this traceability to be carried out, two other methods must exist. These methods map both function and agent perspectives onto the data perspective. The method connecting agent and data perspectives was defined by Gomaa, and is called DARTS.\(^{180}\) A new version developed in conjunction with the Software Productivity Consortium is called ADARTS\(^{181}\) and uses Ada. The method connecting function to data is the Data Flow Diagram used in structured analysis developed by Yourdon and DeMarco. These two methods are each composed of minimal methods which combine to produce a single macro method or two-way bridge between perspectives. There is a high degree of parallelism between these two macro methods. This parallelism allows isomorphic mappings between tasks and functional bubbles. That mapping illustrates how the macro functionality of the software system is accomplished by the threads of coherence (tasks) in the virtual processor.

178. See Bib#.
179. See Bib#.
180. See Bib#.
181. See Bib#.
Without this kind of mapping, there would be no way of seeing if the entire system would function properly. This is different from the perspective that looks at the individual virtual machine instruction within a particular task to see what micro function it embodies. Functionality is both system-wide and narrowed to the particular instruction executed now. Both types of functionality must be coordinated for the software system to work properly. The two minimal methods that connect agent and function assure this coordination by giving a means of representing it.

The mapping minimal method is not as complex as the virtual machine minimal method. The mapping is a web of traceability links between functions and tasks. The method is explained best in the third volume of the Ward-Mellor series called Structured Development for Real Time Systems. The process of creating the mapping is called “functional allocation.” In functional allocation one first allocates functionality to processors, then to tasks and finally to sequential modules (virtual machines) within tasks. At each step of this crucial system architectural design process, the essential model of system functionality as a data flow diagram should be modified to reflect implementation decisions. Functional bubbles may be split between processors or tasks, so new bubbles and associated data flows must be added. The resultant transformed essential model is called the implementation model. This implementation model is isomorphic to the processor-task-module implementation decomposition of the system into process-able units. Functional allocation is the process by which this mapping is produced. Functional allocation may be described as the transformation of the functional description of the system into an implementation description which results in traceable links. This transformation is evolved through a mental processing of the functional essential model. That mental processing results in an implementation model and a system architecture in terms of processor-task-module descriptions. It is here that the main effort of design work is done by mental simulation. Mental simulation in design is a virtually ignored aspect of the design process. There are almost no books or articles on how to perform mental simulation in design work. Yet this is exactly the most difficult part of the design process. Design simulations are necessary during functional allocation in order to make the system work properly. Without mental simulation during design, there can be no correct processing of a system at execution time. Without methods, these mental simulations are too complex for the mind to handle. Methods are the language in which mental simulations are executed by the mind of the designer. But the actual work of how to do mental simulations is not described anywhere. As a result, software design is a very difficult type of mental work. Software design is mentally exhausting. Each set of virtual instructions must be simulated mentally many times before that is ever executed by a virtual processor. In fact, the mind is the first virtual processor for all virtual machines. Mentally simulating virtual machines is different from ordinary thinking. There is a rigor to mental simulation that is not necessary in thinking, which goes against the grain of the mind and is particularly difficult to emulate. During this process, the designer is not just emulating the paths through the virtual machine under various conditions; he is, at the same time, changing the virtual machine that is being emulated so it will function correctly. This is a very complex task that is very difficult to perform, particularly because only a small part of the virtual machine (seven to nine chunks) can be kept in short-term memory at a single time. One is constantly losing one’s place, or realizing the relevance of some other aspect of the system to the problem at hand. Thus, mental simulations tend to be diverted and interrupted in mid-stream as long-term memory suggests other relevant connections within the system being designed. How to do mental simulation needs to be a high priority research area. Tools and meta-methods need to be developed to support this work and take the drudgery out of it. The definition of a framework of methods, such as that developed here, is a good start. But the methods and heuristics do not touch the heart of the problem of system design, which is the mechanics of mental simulation.

Having said that the virtual processor and the chain of transformations are locuses, it is still possible to substitute the entity into these locuses. When the entity is substituted into the place of the virtual processor, it becomes an “actor” after the meaning of Agha and Hewett. Here the entity becomes an agent and is identified with either the processor or the task thread of coherence. Likewise, the entity may be substituted into the chain of functional transformation. In that case, operations are associated with the entity that perform these functional transformations which may be independent of the modification of persistent data. The addition of these operations allow the entity to be identified with either the virtual instruction or with
persistent data objects.

By making these additions, the eventity comes to serve as a generalized computational emulator. It is a generic design element which can perform many different roles within the system. Yet this connection with the virtual processor and transform should not lead to the fallacy of misplaced concreteness. Processors and transforms are independent of their linking to an individual eventity. They are more likely to be represented as cooperating groups of eventities. However, the ability to represent processors and transforms as eventities does serve the purpose of giving us a single multifaceted design element that can be the basis of system design. The eventity is fairly complex. But this complexity allows faithful representation of the “systemic” nature software aggregates. The problem with using less complex design elements such as queues, pointers, counters, timers, etc., is that the systemic character of the mechanism they embody is hidden. In order for these systemic features to be presented in the design, the design element must represent the complexity of the spacetime backdrops as well as being an automation. In this case, the automation is dual in order to represent outward and inward control over parts. Thus, I argue that the eventity is just complex enough to represent the systemic features of software designs. And this is very important. Unless the systemic features are represented, then there is no way to hone in on a harmonic design. Harmonic design depends on the representation of the systemic figure/ground dynamics. Different eventities may serve as both elements of the figure or the ground within the system. Eventities give a generic design abstraction which is at the right level of complexity to capture the whole gambit of system functioning. Below the threshold of the eventity, which is robust enough to represent a subsystem, the systemic interdependencies are subsumed within the panoply of lower level design elements. It is as if the system were out of focus. When the granularity of the design elements is wrong, the system is blurred. Eventities are at the right level of design element complexity to bring the system into sharp focus and thus ease the burden of design work.

Now it is necessary to consider the relation of the minimal methods associated with the agent and function perspectives, i.e. mapping and virtual layered machine, to the metrics of system supports. As has been mentioned, both agent and function perspectives are radically different from the fully ordered space/time perspectives. The agent/function perspectives are only partially ordered. This means that within either of these perspectives that exhibit cloture, only partial order relations between elements may be adduced. Consider the agent perspective. An agent is an independent actor or center of activity. Our natural models for independent agents are organisms. For independent centers of activity, or actors, the concept of linearity and distance have no meaning. By their nature, actors are processing simultaneously independent of each other. They only take turns in a linear fashion if they are forced to through some synchronization mechanism. The distance between processors is also irrelevant to their processing capability, although it may be important with respect to their ability to communicate as a network. Now consider the function perspective. A functional transformation also lacks comprehension of distance and linearity. Like processors, transforms operate in space-time. But functionality, when abstracted from the space-time milieu, has no inherent linearity or distance component. Consider the data flow diagram. The distance between bubbles is irrelevant. The bubbles fire when all their inputs are available. Thus each bubble is really like an independent processor. The network of bubbles is ordered by the data flow arcs. But data flow arcs are a spatial feature flowing from the data perspective. A data flow diagram with no arcs would be static. Without inputs, there would be no firing. Without arcs the dataflow nested bubbles are merely a partial ordering of system functionality. There is no linearity because there are no data flow arcs, and the distance between bubbles is irrelevant.

This should suffice to show that linearity and distance do not have meaning for the pure agent and function perspectives. Yet obviously for the minimal methods of mapping and virtual machines, these concepts do have meaning. Mapping connects the hierarchy of agents with the hierarchy of processes. Through this connection between the two hierarchies, the notion of distance becomes meaningful. One can think of distance as the number of boundaries crossed in the background hierarchy. One hierarchy is used as foreground, usually function, and the other forms the background, usually agent. The background forms a grid against which foreground objects can be measured. For instance, a single function bubble might be split between two agents. The separation of the functionality by the agent demarcations induces distance. This is why we call this assignment of functionality to multiple agents the “spreading” of the functionality out within the system. In this case, two partial orders, when connected by mappings,
gives rise to meaningful demarcations of distance. Distance is measured by counting the boundaries crossed in the background grid by objects in the foreground grid. On the other hand, virtual machines exhibit linearity. The instructions of a virtual machine are fired in sequence. The idea of distance between instructions, however, has no meaning. Each instruction is an implementation of some functionality that can be executed by a virtual processor. The instruction is the intersection of the functional map of the system with the processor map. These single points of intersection are active, one at a time in sequence and their firing constitutes the execution of the software program. Here again, it is the relation between agent and function hierarchies at the single point of execution linearity through time that allows two partial orders to emulate a linear order without distance.

As a general principle, agent and function are inherently partial orders when viewed separately. When used together, they can emulate linearity without distance (virtual machine minimal method) or partial order with distance (mapping of functionality to agent minimal method). However, agent and function cannot emulate full ordering in the way event and data perspectives can. This means that there is no equivalent of cartesian coordinates for agent and function. In cartesian coordinates, both linearity and distance may be seen at once. Event and data may be represented within cartesian coordinates or via a timeline using real numbers on the x, y, z and t axes. There is no equivalent to this for agent and function perspectives. Processes and agents may be seen as moving in time and space, but they do not themselves give rise to fully ordered supports against which system variables might be measured. Agent and function only give rise, at most, to supports with linearity but without distance, or supports with partial order with distance. This seeming defect in the function and agent perspectives is a major barrier for software system modeling. In information systems, the barrier is not readily apparent. In these systems, the space-time constraints are relaxed. Thus, all the minimal methods suffice to work together to model the system. However, in real-time reactive systems, the event and data perspectives must interface with events in the world and move information through the network to meet the system needs. In that case, a gap appears between the fully ordered capabilities of the event and data perspectives, and the lack of that capability in the agent and function perspectives. This gap is the hiding place of the singularity of pure immanence. It is the unbridgeable gap between openness and cloture. It is what makes the design of real-time reactive systems so difficult. This is because ultimately the methods do not help at this extreme point. Ultimately, the methods break down, and there is no full coordination between system supports. There is only a partial coordination. The system must be designed according to the minimal methods as they overlap when space-time constraints are relaxed. Then the space-time constraints must be tightened to discover performance as the system is executed. The performance where the system is executed must be analyzed, and then the design is modified to attempt to solve performance problems. The gap between methods design and system execution performance is a fundamental element in the software system. It is this gap which gives software science its important role in observing executing systems. It is this gap that gives software its quasi-experimental nature. It is this gap which is the point of cancellation of Hyper-Being.

In fact, the gap is not a defect. It is an important indicator of the nature of software. Agent and function embody meaning in a way event and data do not. The poverty that agent and function have from the point of view of systems supports is made up for by their ability to serve as markers for meaning. Data and event have no internal dimension. They represent complete exteriority. Function and agent, on the other hand, have a dimension of interiority in which semantic value may be invested. The expressiveness of agent and function, in terms of semantic content, balances their poverty with respect to metric ordering as system support variables. This fact shows that software methods can be vehicles for semantic expression. They have a natural interface with language that appears whenever we say “what” or “who.” We anthropomorphize agents and project organism (or even human) qualities upon them. This is not correct, but it shows the viability of the semantic interface of these methods with language. This interface is even more pronounced with respect to saying what a function is. We speak of functionality in the same way we confer meaning on things in everyday life. By this means, the functioning system becomes a particular kind of thing -- i.e., it comes to inhabit our world along with us. The bank teller software with its hardware does the job of the human teller. It becomes an “automatic teller.” The emulation becomes the thing it is emulating. Our ability to say what a software system is, and thus incorporate it into our world flows from the interface of the function perspective with language. The function elements in the system serve as bearers of meaning, as do the agent elements. This connection between software systems and linguistic reality is very important. The fact that software methods have this interface shows that their structure in relation to the gap between openness and cloture is no
accident. In this we see an essential extension of technology into meta-technology.

The derivation of the minimal methods that directly relate agent and function perspectives, or event and data perspectives, has led to an appreciation of the difference between these two sets of perspectives. They can be contrasted in terms of the ordering metrics associated with them. Event/data perspectives are fully ordered, whereas agent/function perspectives are only partially ordered. We see clearly how the minimal methods derive from relaxation of ordering in the case of event/data and a tightening of constraint in terms of agent and function. The tightening of constraint occurs when partial orders are combined to attempt to mimic fuller ordering. There is, however, a limit to this mimicking, and full ordering is unobtainable. This impossibility that lurks at the heart of the matrix of software perspectives is an important phenomenon that determines the nature of software. It bears out the ontological analysis already performed in the earlier part of this essay. The gap between event/data and agent/function is the fundamental locus of differing/deferring within which the singularity of pure immanence resides. The gap itself is like the event horizon of the singularity. It only shows up through a detailed rigorous analysis which seeks to understand fully the interrelation of minimal methods that spring from the perspectives on software design.

The four minimal methods explored thus far are crucial to understanding the nature of the gap between openness and cloture. What has been learned by this analysis and attempted derivation may now be applied to the other minimal methods. There are five other methods that relate the perspectives (agent, function, event and data):

- **EA** Worldline/scenario Two-way bridge
- **PD** Data flow Two-way bridge
- **AD** Tasking and comm mech (DARTS) Two-way bridge
- **PE** State machine One-way bridge
- **EP** Petri net One-way bridge

Interestingly enough, three of the remaining methods are two-way bridges composed of two complementary minimal methods which fuse into a single macro method. Of these three macro methods, two are very similar. Those are the data flow method of Yourdon and DeMarco, and the DARTS method of Gomaa. In these methods, agent and function are almost interchangeable. There is a direct parallel between all the features in these two macro methods. This shows that the data perspective exerts a powerful organizing force which sees agent and function in very similar ways. Attempts have been made to synthesize these two macro methods into a single overarching representation. For instance, Mitchell D. Lubers of MCC developed "A General Design Representation" which attempts to combine the data flow and tasking/communication mechanisms into a single super method based on a Unix processes and pipes motif. This kind of collapse of two macro methods into a single super method has the danger of confusing the perspectives which are intrinsically distinct. From the separation of data flow and DARTS, is opened out the mapping minimal method and the virtual layered machine method. In fact, it is possible to see that the viewpoints from which two macro methods diverge, i.e., the data and the agent perspectives, have the greatest organizing capacity with respect to the software design. Within each pair of viewpoints separated by the gap, there is a dominant viewpoint (agent dominates function) (data dominates event). The dominant viewpoint has diverging macro methods (two-way bridges) which opens up the arena in which the methods connecting the opposite set of viewpoints exist.

The inferior perspectives from each pair (event and function) are connected by two one-way minimal methods which are state machines and petri nets. These minimal methods are important, but do not have the overall organizing capacity that the cones of methods arising from the dominant perspectives have.

- **DATA FLOW AND DARTS ORGANIZE MAPPING AND VLM**
- **AGENT WORLDLINE AND DARTS ORGANIZE STATE COORDINATION AND INFORMATION FLOW**

From this, we can see why the object-oriented design is beginning to achieve dominance over the functional orientation. The methodological field itself has an internal structure that would shift the balance in the favor of data-oriented minimal methods. We would expect the same sort of ultimate emphasis on “actor” like systems such as that described by Agha. The function/event axis would

---

183. See Bib#.
apparently have to be subservient to the two major triangles of super method groupings.

This does not mean that any of these methods should be neglected in system design. It means that there are intrinsic organizing principles in the field of the methods which dictate the perspectives from which the most powerful organizing forces will naturally radiate.

<table>
<thead>
<tr>
<th>Partial order</th>
<th>Linear with</th>
</tr>
</thead>
<tbody>
<tr>
<td>with distance</td>
<td>no distance</td>
</tr>
</tbody>
</table>

Actor  
Scenario  
DARTS  
Monitors and tasks  
Tasks and comm mechanisms  
Data flow  
Methods and data stores  
Data flow arrows  
Background  
State machine  
Petri net

Each of the eight minimal methods shown above can be divided into those which approximate linearity without distance metric, and those which approximate partial ordering with distance metric. The concept is that all minimal methods approximate one or the other of these two intermediary metrics. All of these minimal methods are combinations of the fully ordered and the incompletely ordered perspectives. Each of these methods operate across the gap between perspectives with similar metric orders and thus combine, in various ways, dissimilar ordering elements. However, in this case, the approximation to the intermediary metrics comes from the collapsing (relaxation) of the event perspective toward linearity, and the collapsing (relaxation) of the data perspective toward distance. The interaction of a partially ordered perspective with a fully ordered perspective that is related causes the fully ordered perspective to dominate. Thus, worldline and scenario have a dominant linearity, as does state machines and petri nets. Whereas in DARTS and data flow, their submethods have a dominant distance component. For this reason, the direct derivation of the two intermediate orders is not straightforward as it was in cases previously studied. Yet for the most part, the paradigm of which sees each minimal method as an incarnation of an intermediary level metric construct (L-D or PO + D) can be applied directly.

For example, the “actor” macro method which combines worldline and scenario minimal methods, has a mapping to the two intermediary level metrics.

<table>
<thead>
<tr>
<th>Scenario</th>
<th>Worldline</th>
</tr>
</thead>
<tbody>
<tr>
<td>Partial Order + Distance</td>
<td>Linearity-Distance</td>
</tr>
</tbody>
</table>

Linearity is dominant because the event perspective’s full ordering collapses into L-D. Thus both of these minimal methods are apparently linear. The worldline may be seen as a fully ordered timeline, or as a sequence of events for a single agent. Scenarios cut across multiple timelines with a linear ordering of events performed by multiple agents. However, because worldlines may be relativistically related in the absence of a global clock, this linearity of scenarios is not what it may seem at first glance. The distancing of agents in different inertial frames (associated with nonsynchronous clocks) plays a role in scenarios not present in worldlines. Worldlines normally collapse from full-time ordering using the real number timeline to a linear sequencing of events and actions. Scenarios involve the slicing across worldlines in non-orthogonal planes that allows causality between events and actions to be seen. When scenarios become involved with relativistically related worldlines -- usually this is not obvious -- then there is interval distancing between events and actions. The “actor” model of concurrent processes is designed to handle this kind of interval distancing which allow nonsynchronous clocks for multiple worldlines. Here it is clear that the dominance of linearity contributed by the event perspective to these two minimal methods is not straightforward. Distancing insinuates itself within the scenario minimal method in ways that may not be obvious. Thus, though the analysis is clouded, it is not impossible. Ultimately, it is possible to understand how the intermediate metrics show up within the two minimal methods being studied.

Another case is the DARTS and dataflow macro methods. These are very similar. In both, distance is emphasized through the relaxation of the space (or data) perspective. For this reason, it is clear that the data flow bubbles or tasks boxes strung together by data flow or
communication mechanism arcs is a linearization metric. Whereas the relation of function bubbles or task boxes to data stores or monitors is primarily a distancing metric. The linear sequence of bubbles strung together by arcs is contrasted with the mapping of methods to data objects. The linear sequence of tasks strung together by communication mechanisms (queues mailboxes, etc.) is contrasted with the mapping of tasks to monitors. In these cases, linearity and distancing appear clearly in the different related minimal methods. Distance appears in the mapping from active to passive elements. Multiple active objects are related to a single passive object. In object-oriented design, this mapping is made such that there is a close binding between the method and the data object. In functional design, the mapping is less direct, so that there is greater distance between the function and the data object. With respect to tasks and monitors, the distance appears as contention of tasks over the same resource.

The final example is the state machine and petrinet minimal methods. Petri nets are like refined control flow charts which are augmented with states and control flow tokens. The petri nets emphasize the linear flow of control; whereas the state machines emphasize the relation of state places to each other. Distance appears as the number of intermediary states exist between any two states. Linearity appears as the sequential path as the control token. The petri nets and state machines are intimately related. The emphasis on state diagrams is the static representation of the state places and their interrelationship. The emphasis on petri nets is the linear sequence of states that a control token moves through as the transitions fire. The former is static, while the latter is dynamic. The former emphasizes distance, while the latter emphasizes linearity. It is interesting that it is the state machine which must be added to the eventity to make it essentially autonetic. One state machine governs the coordination of parts while the other governs externally observable behavior. The fact that the state machine/petrinet minimal methods are left over as outside the agent and data super method complexes is very significant. It shows that the state machine/petrinet methods have a special place in the field of minimal methods. This special place has been characterized as the background because they are not included in the super method complexes. But they could just as well be characterized as the jewel that stands out on the background of the super method complexes. It is state machine and petrinet methods which are used to model the structure of automata. Automata are either active or passive, depending on whether they are parsing input or defining output. When the active and passive state machines are combined in a single two-part complex, they become autonetic or self governing. The self-governing autonetic automata can be described in terms of state machines or petri nets, depending on the static or dynamic emphasis. When the state machine/petrinet is combined with the eventity, then the autonetic eventity is the result. The autonetic eventity is the key to advanced design that takes into account all four perspectives on software design. It combines all four perspectives into a single multi-purpose design element. This integration of design perspectives by advanced knowledge of methods is the crucial work of software meta-methodology.

This essay began as an exercise in meta-methodology. Meta-methodology seeks to understand the interrelationships between methods. Software design methods derive from the four fundamental perspectives on the software system. These perspectives (agent, function, data and event) have been identified with what Klir calls “backdrops” or “support variables.” The minimal methods define the relationships between these supports. The support relationships form a substructure that serves as the foundation for the superstructure of the designed software system. Normally, the substructure is invisible because its main purpose is to provide something for designed system variables to be measured against. In this essay, an attempt to make the substructure visible has led to an appreciation of the complexity of the field of intra-method relationships. This substructural field is the backdrop against which all the relations between designed system variables must be understood. If a variable can only be measured against one perspectival support variable, then only the method associated with that perspective may be used to define that designed system variables in relation to other designed system variables. If a variable can be measured against two perspectival support variables, then the relevant minimal method may be brought to bear to define the place of that designed system variable within the whole designed system. The more perspectival supports that the designed system variable can be measured against, the more central that variable is to the internal structure of the designed system. The key system variables can be measured against all the perspectival support variables. As the design progresses, more and more of the designed system are brought within view of all the perspectives. In a fully defined system, a good portion of the superstructural variables should be, in
principle, situated in relation to the total field of the substructure of support variables and their bridging methods.

The development of design perspectives as support variables allows us to plug our design methods that flow from these perspectives into Klir’s overall definition of the epistemological levels of the formal-structural system. It is, however, necessary to read Klir’s overall definition of the formal-structural system in a particular way in order to see its significance for software design. Klir’s general systems theory should be considered the basis of software engineering formalism. Having defined the generic backdrops or perspectives applicable to all software systems, Klir’s general system theory then becomes a general theory of software systems. We need only look at it in terms of systems design instead of systems inquiry. This may be accomplished by realizing that every designed system must interface directly with its environmental anti-system. The designed system and the anti-system form two subsystems of a meta-system. In the design process, technical source systems for both the designed and the anti-systems must be defined. The external system boundary separates the designed source system from the anti-source system. Both the designed and the anti-source system variables are measured against the fundamental perspectival support variables. These fundamental software design perspectives mediate the interchange between the two source systems. Once the source systems have been defined, then the data flows across the external system boundary are isolated. In relation to these data flows, the anti-data system is defined. Its reciprocal designed data system is postulated on the basis of the needs of the anti-data system. The designed data system is the inverse of the anti-data system.

The next step is to define the designed generative behavior system. This is the software capable of generating the required data streams. This generative behavior system may require further development of structural and meta-models which drive the generative behavior (software) system. Support invariant constraints between variables necessarily change overtime. These changes from one model that generates data to another are structural relations. Meta-models record the changes in structural relations. Each level above the generative behavior system records the embedding of more and more sophisticated constraints into the software design. Structural systems record changes in design system variables relations. Meta-systems record changes in the support variable set in relation to the designed system variables. In the process of design, variables measured against a single perspective are extended to be measured against two perspectives, then three perspectives and then four perspectives. Each extension of a designed system variable to another support perspective allows another meta-system level to be defined. What is invariant in relation to two perspectives may be variable in relation to a third perspective. The addition of support variables other than those required by the software design perspectives, allows this process of generating meta-systems to become an endless regression like that of the endless levels of meta-structures. The two infinite regresses of meta-structures and meta-systems approach infinity, ending in the singularity of pure immanence.

Klir’s definition of the formal-structural system gives a firm basis for understanding the embedding of structure into the software design. Formal methods in software engineering normally apply logic and other discrete mathematical formalisms to the definition of software. These formalisms must be augmented by an appreciation of meta-structure and meta-system articulation. Software design must be defined by structural as well as formal relations. The formal relations are the basis for understanding these structural relations. Klir’s model of the formal-structural system is the only known model where these structural relations are clearly defined. Thus, formalisms such as logic, set theory and axiomatic definitions should be applied within the framework of general systems theory to achieve a proper formalization of software design. These formalisms are independent of the software design methods.

The field of software engineering must necessarily draw from several sources which work together to get full coverage of the problem areas being addressed. The unifying discipline is mathematical category theory which is applied to abstract automata. To this is added Klir’s definition of a formal-structural system with its various epistemological levels. The formal components of the formal-structural system may be controlled using various other mathematically-logical formalisms such as logic or set theory. For software systems, the four fundamental perspectives must be added, along with the bridging minimal methods. These perspectives define the substructure upon which the designed system superstructure is built. To these sources must be added domain analysis which adapts the software design methods to a particular domain. Domain analysis is explained by S. Shaler and S. J. Mellor’s article.

184. See Bib#.
Software Systems Meta-methodology

“Recursive Design.” In this article, they identify four types of domain:

1) Application Domain Specialized Architectures

2) Service Domain Database, Operating System, etc.

3) Architecture Domain: Agent, Event, Function, Data

4) Implementation Domain: How architectures are incarnated as programs.

The design methods articulated in this essay describe in detail the architecture domain. Implementation domain includes languages, operating systems, data bases and other tools at various levels of abstraction which allow the who, what, when and where of the design to be expressed as “how.” This is implemented concretely so that it runs as an executing software system. Service domains are various specialized architectures which serve the application which, in turn, has its own architecture. Each of these domains consist of objects that have data and states as well as sets of operations that modify the data and states. Shaler and Mellor suggest using their object-oriented analysis method to delineate these domains. Within the architecture domain, however, the concept of autonomic events is a more powerful organizing construct. Within each domain, various mathematico-logical formalisms may be applied to organize and keep consistent the design information. But these formalisms must remain subservient to the general systems theory concepts which are applied to each domain in order to discern the systems that give the objects within a domain coherence. Object-oriented analysis lacks these basic systems’ theoretical constructs. Architectural design methods must eventually be applied to the various domains in order to render the software design which may then be implemented. The steps of meta-design might appear as follows:

1. Identify domains and their relations

2. Identify objects in domains and their operations

3. Apply formalisms to all domain objects and operations

4. Apply design methods to formally constrained objects

5. Embed meta-structures and meta-models in domains

6. Implement designed system

7. Study implemented design behavior.

Meta-design takes into account the place of the design methods in the world as they are applied to different applications. Design methods, by themselves, are merely an abstract view of how all embedded software must be structured. When combined with domain analysis, formal mathematico-logical methods, and Klir’s definition of the formal structural system, these formal methods find their application within the world.

This essay must finally turn to considering the significance of the substructure of software methods in relation to the superstructure of the designed system. Meta-design posits that these software architectural methods must be understood in relation to other disciplines. But fixing the place of the field of software methods among other disciplines is not enough. We strive here to understand the meta-technical in some fundamental way. It is clear that the meta-technical is the substructure added as a supplement to the technical formal-structural system as defined by Klir. This supplement dovetails with Klir’s definition by fitting into the slot of “backdrops” and “support variables” that ground all measurements within the software source system. Yet, the supplement adds unwanted elements. The perspectives do not all have the same metrical format. The division between fully ordered and partially ordered metrics causes the field of software methods to be distorted. This distortion, when surveyed carefully, reveals a serious gap between the two kinds of metrics. It has been posited that this gap hides the singularity of pure immanence associated with Being 3. Software derives its inherent nature from this kind of Being 3. Software is a strange kind of writing. It is automated writing which shares weird characteristics with the automatic writing of the explorers of the paranormal. In automatic writing, some other force from another realm is said to control what is written. The Ouija board is often used as a tool in the seance. The fingers are moved by spirits from another world to spell fateful words in answer to questions. This phenomenon is sometimes explained as unconscious direction of the motor system. It is usually

185. See Bib#.
dismissed as self delusion by those who are too easily impressed by tales of the supernatural. Whatever the explanation, automatic writing is a strange, borderline phenomenon set outside the boundaries of the domain of order and reason. In automatic writing, the question is “who” or “what” is controlling the writing. Some other agency other than the subject is posited which is speaking to us -- from the unconscious; -- from the spirit world; -- from our need for self delusion.

Within the realm of technology, well within the domain of order and reason, safe from all accusations of pseudo scientific tendencies, there is another kind of automatic writing. In this case, it is the written text of software executed by the central processing unit of a computer. At first glance an altogether different situation. Yet, on further analysis and reflection, software becomes more and more strange. Slowly it is realized that software is not like other mundane things. It has a special kind of presence in the world. That kind of presence has been named Hyper Being. It is the cancellation of Process Being of Heidegger with the Nothingness of Sartre. It indicates the existence of pure immanence, unseeable, behind the showing and hiding of temporalized, i.e. pure transcendence, presence. By analogy with the physics of Blackholes; pure immanence is like the singularity surrounded by an event horizon, i.e., outside the laws of physics and things beyond reasons and normal order. In physics strange things occur at the hypothesized event-horizon. What occurs at the event-horizon where Process Being and Nothingness cancel, becoming crossed out, is described by Derrida as differing/deferring of Differance:

Derridian difference is movement which permits the spacing required by difference while necessarily deferring fixed relations between signifier and signified.\textsuperscript{186}

This event-horizon has been identified with the gap between fully ordered and partially ordered perspectives on software design. The gap between these two pairs of perspectives is a fundamental difference. This difference is opened up by the more fundamental differences between pure immanence and pure transcendence, and between the two cancelling forms of pure transcendence: Process Being and Nothingness. The dynamic of the presence/ non-presence of Process Being, set against the dynamic of presence/absence of Nothingness, gives rise to the phenomenon of cancellation. In cancellation, the present becomes a nonpresent absence. What was present just vanishes through annihilation. Not even a trace is left. This means pure transcendence in its two poles, turns into pure immanence -- that which can never be made present and which is inherently nonpresent and absent. That singularity of pure immanence is then in one sense identical with pure transcendence. In fact, the “purity” of immanence and transcendence slowly becomes suspect. Perhaps these also are nihilistic antimonies of pure reason which also ultimately collapse in the cancellation. What is left after the dust has settled from this cancellation of nihilistic meta-physical opposites is the world -- not even a leaf has rustled as a result of these momentus metaphysical events -- which lies undisturbed before us, wild and primordial, which existed before the edifice of ideation was erected. Difference is the opening up of the fundamental metaphysical differences which creates an opening which all other differences can operate. This opening up of the difference between pure immanence and pure transcendence, which in turn flowers into the difference between Process Being and Nothingness, creates the external/internal horizons of subjectivity. When the subject looks out, it sees the infinite horizon of Process Being as temporalized manifestation of things in the clearing-of-being. When the subject looks inward, it sees the infinite horizon of Nothingness where all its searching for an inward core gets lost in an endless desert of nowhere. The subject gazes on the world from this nowhere. Corlett defines the subject as . . . any being capable of maintaining continuity across time. Accordingly, a (reasonably collected) person can be a subject, but so can a neighborhood or village.\textsuperscript{187}

The subject has been defined by Corlett in a way which separates this concept from the individual, but what is deeply interesting, if not thought provoking (in a Heideggerian sense), is that agency has been projected by computer technology outside the human organism. Computers driven by software can be seen to fall under this definition of subjectivity. They maintain continuity as agents across time/space as CPU cycles and digital memory. Thus, the human organism becomes replaceable by the automation as a locus of experience in terms of a space/time nexus. This new subject does not just maintain continuity passively, but acts as a transformative agent as well. In some way, it is possible to see that the definition of the four fundamental perspectives is some minimal definition of subjectivity, and that the “gap” of the event-horizon of cancellation opens up directly in the middle of this minimal definition of the subject. Whether the subject

\textsuperscript{186} William Corlett, Community Without Unity, Pages 157-158

\textsuperscript{187} Page 22, Community Without Unity.
is a single processor or a cluster (neighborhood or village), is not relevant. Fragmentation of agency is not as important as maintaining the illusion of continuity. The illusion of continuity is precisely what the notion of the subject guards. This illusion is generated by the mechanism of ideation. Ideation repeats the image fast enough so that the gaps in the projection cannot be seen. The differences between repetitions are glossed so that differences within the projected image can catch the eye. Within the human organism pretending to be “subjugated” these images are his simple ideas. Within the automata, these images are the primitive operations of software commands.

Human beings have projected their carefully designed subjectivity out beyond themselves into the world. The ideational mechanism has taken on a life of its own in a nonorganic realm. But the crucial question becomes “who is the designer.” We fashion our robots as we attempt to refashion ourselves through genetic meddling. Strangely, we discover that software is the key in both cases. The genetic software of DNA allows us to reprogram ourselves, while the software that controls the automata allows us to mimic ourselves in silicon representations. The “designer” in both cases is not easily pinned down. The organic foundations are changing, while the ideational locus of subjectivity is projected outward. Suddenly we wonder who is writing the software. It is certainly not the subject, because that is what has been thrown out into the world as automata. It is not the human organism, because that is changing under our feet. When we search for the “designer,” we look into ourselves and see only nothingness. We don’t know who we are. We discover only viewpoints from nowhere. When we look outward, we see an infinite horizon of temporality full of beings we project as we fall towards death. Some of those beings are automats animated by ourselves. Some of those beings are organisms we alter genetically. Suddenly we realize that perhaps the nothingness of our own viewpoints from nowhere is perhaps identical to our projection of manifested beings. These two horizons of transcendence in that moment collapse. The “designer” as subject in a flash disappears. Pure immanence and pure transcendence merge. All at once we are savages again. The ideational edifice has collapsed. Reason has become the same as madness. Order is indistinguishable from disorder, yet the human species has been altered irrevocably. The human species has been redesigned to adapt to the externalized agents that now control it. From the unconscious, or the world of spirits, or the realm of self-delusion the automatic writing has been done. The “designer” is elusive. But that elusive “designer” has used the meta-technology of software to rewrite who we are.

Difference is the name of the movement that opens up the spacing wherein differences are seen. In that spacing -- between pure immanence and pure transcendence -- between Process Being and Nothingness -- between subject and Klir’s object system -- there occurs the deferring of fixed relations between signifier and signified. There, in that spacing, the temporality of nihilistic artificial emergence and erratic change constantly turns over the soil, so the seed of a non-nihilistic distinction may be planted and grow into the tree of genuine emergence. In Old English, the words “tree” and “true” are related by vowel shifts. The world was envisaged as the tree of Yaggrasil. When the Kurgan people roamed the great forest that was Europe, the tree seemed to be the dominant structure in the world connecting heaven and earth. The straight tree did that best, and was called true. In our times, only those genuine emergences rooted in all four kinds of being stand out in the nihilistic landscape that has replaced the great forests of Europe and the Americas. “Software” as a concept is one of those great trees within the nihilistic landscape. Software is a kind of entity whose roots go deep. It reaches down to tap the ground waters of Hyper Being, and becomes a means of reaching even deeper to the level of Wild Being. Then knowledgeware becomes embodied in software. Software reaches through the cracks in the metaphysical bedrocks and taps the primordial writing that expresses difference. The automated/automatic writing of difference is the face of the “designer” who redesigns the subject. The origin of the forms that the “designer” imposes on the subject are always already lost. By definition, the origin cannot be made present. As we look into the face of the meta-technical phenomenon, we see the fragmentation of our subjectivity. As designers of automata, we eventually have to allow them to become nondeterministic (as they embody knowledgeware). That is no longer controlled by us. As designers of ourselves, we begin to evolve as organic artifacts. As designers of the bionic combination of these two evolutions, we become the lost origin of a different kind of creation -- ‘the ubermann’.

When Zarathustra arrived at the nearest of towns lying against the forest, he found in that very place many people assembled in the market square; for it had been announced that a tight-rope walker would be appearing. Zarathustra spoke this to the people:

Kent Palmer
In the Gameplayers of Zan, M. A. Foster imagines the “Ler” as the next step in human evolution. They are totally involved in playing the game of “Zan” which is their interface to the ultimate machine which is an autopoetic spaceship. This wedding between the creatures we may create from our organic basis and the autopoetic automatons we may create from inorganic projections of subjectivity, is brought to life in Foster’s classic science fiction. But this deep possibility owes its potential to the genuine emergence of the archetype of software as a fundamental means of exercising power. Given this tool, our will-to-power enters a new realm in which meta-technology takes over as the dominant theme from technology. Whether Foster’s vision is realized or not, there will be profound changes from the fragmentation of Being. And the intensification of technology as meta-technology is not the end. After the full flowering of meta-technology, comes the final state of proto-technology in which we return to savagery and confront chaos directly.

“Very well. What does the game have to do with piloting? I know the game, thanks to Krisshatem, but I fail to see...”

“The hard question; thus the hard answer. Let me build a dynamic identification series for you: consider vehicles. You make a cart, a wagon, hitch it to a pony, and off you go. Its purpose is to go, but it can be stopped, and it doesn’t change, or stop being a cart. Yes? Now consider a bicycle, which must be in balance to go. Yes? Now an aircraft; it can only be stopped when it is finished being a functional airplane, yes? You can’t stop it just anywhere, and never in the air, unless you have rotary wings, which is just cheating the system. Yes? Just so the leap to the ship. It is a quantum leap into a new concept in machines. If indeed that is the proper word. Before, we had machines that could be turned off. The more complex they became, the harder to turn off. With the ship, we enter the concept-world of machines that can’t be turned off -- at all. They must be on to exist. Once you reach a certain stage in the assembly of it, it’s on, and that’s all there is to it. And when you build it, you are building something very specific; that is the law of multiplexity. The more developed the machine, the more unique it becomes.”

The automata ceases to exist when turned off. Software needs the executing hardware. When we have projected ourselves totally into virtual space via our cybernetic creatures like “knowbots” and the system goes down, what will be left over? The overman will probably be totally locked into his narcissistic interaction with the automata like the Ler. Perhaps he will have already discarded his creator “homo sapiens” when the system goes down. Once genetic engineering has taken place, there is no going back. Little improvements here and there finally lead to something quite different, even barring intentional reengineering. The technical adaptation of the human, so that man no longer confronts technology, occurs through meta-technology. Hardware and organic wetware merged through software and became symbiotically merged. Knowledgeware governs the synthesis. What cannot merge becomes the discarded and the savage -- the proto-Ler or last man.

In software, the destiny of humanity is written. Visions from science fiction are not needed to tell us that important events for the fate of humanity are happening in our times. Software meta-methodology has a pivotal role to play in these events.

188. Nietzsche, Thus Spake Zarathustra, Page 41 See Bib#.
189. Gameplayers of Zan, M. A. Foster, Page 370, See Bib#.
Dedication: This paper was written over the period from September of 1989 to June of 1990. It was just in this period that my son, aged eight, Shuaib Ibrahim Palmer suddenly grew ill and finally died. In November he fell ill and was discovered to have a brain tumor. On January 22 he had his first operation to remove the tumor. The tumor turned out to be an epidermoidal cyst in the third ventrical of the brain attached to the Hypothalmus. This first operation was a success, but within a few months the cyst had grown back. A second operation was attempted on May 18th, and Shuaib died on May 25th from a massive stroke after showing initial signs of recovery. In the brief period between his two operatons, Shuaib spent a great deal of time drawing and painting and attained unusual proficiency for a boy his age. Shuaib, the inventor of many wonderful things, was created with an extremely rare flaw which suddenly manifested and claimed his life. As with all the creations of God, both he and his flaw expressed the perfection of God’s work. This essay, with all its inherent imperfection, is dedicated to my son, whom I love, in death as in life, with all my heart.

KEYWORDS:

ABSTRACT OF SERIES:
Software Engineering exemplifies many of the major issues in Philosophy of Science. This is because of its close relation between a non-representable software theory that is the product of design and the testing of the software. Software Engineering methods are being developed to make various representations of the software design theory. These methods need to be related to each other by a general paradigm. This paper proposes a deep paradigm motivated by recent developments in Western ontology which explains the generally recognized difficulties in developing software. The paradigm situates the underlying field within which methods operate and explains the nature of that field which necessitates the multiple perspectives on the software design theory. Part One deals with new perspectives on the ontological foundations of software engineering based on recent developments in Western philosophy. Part Two deals with the meta-methodology of software engineering by laying out the structural relations between various software architectural methods.

OVERVIEW OF PART TWO:
In this second part, the ontological results of the first part will be extended by a study of specific software engineering methodologies and how they work together in the design of real-time embedded software systems. The fitting together of methods into a coherent whole is the domain of systems meta-methodology as defined by Klir. Software engineering uses general systems theory as a foundation for describing structural systems. Structural systems provide the architectural framework for defining software systems. Software systems design requires the development of four specific points of view. These points of view require methods as bridges that allow movement between viewpoints. This section explores the structural infra-structure of the method bridges between design viewpoints.

[ SEFpart2j/draft6/930908kdp (X90-999/601)]
The Integral Software Engineering Methodology (ISEM) has grown out of our previous studies of the software engineering foundations. The main purpose of this paper is to review the main aspects of a formal language for expressing software designs. That language has many unique features but is based on a set of empirically derived methodologies in current use by advanced software engineering practitioners. The language ‘ISEM’ will be used as an example to show what is necessary for a complete statement of software methodologies for the design of real-time software systems. The language is an experimental formalization of the knowledge structure that software systems engineers must have to design a real-time system abstractly. ISEM may serve as an example for the development of other future languages to fill the void between current partial design methodologies and what the practitioner actually needs.

1. SUMMARY OF THE FOUNDATIONS

The basic lesson to be taken from the previous studies is that there are FOUR viewpoints on Software Design. These are the AGENT (who), FUNCTION (what), DATA (where), and EVENT (when) perspectives. Designs are themselves ultimately unrepresentable as Peter Naur suggests because the reasons why a particular design is the way it is are virtually infinite. Thus all we can expect are partial representations from a variety of viewpoints. A particular design element visible from one perspective may vanish or be transformed as you move to another viewpoint on the design.

Given the viewpoints methods show up as ‘bridges’ between viewpoints. Each viewpoint has its own method for representing the system independent of all other viewpoints, and also each viewpoint has a connection to methods which represent one of the other viewpoints from its own perspective. Thus between each viewpoint on software design there is either a two way bridge or two one way bridging methods that allow transitions between viewpoints on the design. An example of a two way bridge is the Dataflow method. It allows easy transition between the Function and Data viewpoints going in each direction using the same technique. On the other hand between Function and Event are the State Machine and Petrinet methods which are duals of each other and each give a one way transition between these viewpoints.

The four perspectives plus the sixteen minimal methods (4 for the viewpoints independently and 12 bridges between viewpoints) give an overview of the field of all necessary methods on software design. These methods may be embroidered in many ways to allow refinements of design description. But, it is necessary to restrict the methods to their minimal complexity in order to get a view of the entire field of software methods which may be easily understood and applied. The ISEM language attempts to allow this minimal, or near minimal, representation to be captured so that the methods may be studied and their interrelationships to be made explicit.

The four perspectives are not equal. In the process of connecting them with the General Systems Theory of George Klir it was discovered that the Agent and Function viewpoints are only partially ordered whereas the Data and Event viewpoints are fully ordered. This explains the differential complexity of the method bridges associated closely with these viewpoints. Minimal methods themselves are all structurally based on the dual orders (Linear with no distance & Partial with distance). But in real-time systems an unbridgeable gap appears between these two types of perspectives which can only be crossed empirically based on intuition. Thus there is some support for the fact that software engineering will always have
some aspect as an art no matter how strong its scientific rendering. Because of this gap it becomes crucial to map the transformations of design elements between different perspectives on the design. This is the motivation of the creation of the ISEM formal design language.

The minimal methods on which ISEM is based is represented by an entity-relation diagram of core concepts which appears as an appendix to this paper. There is an important difference between ISEM and this conceptual core system. The conceptual core is meant to represent the minimal methods necessary for software design adhering to the principle of Ocham’s Razor. ISEM embroiders this by adding what are strictly unnecessary concepts to make things easier for the practitioner or to express a certain approach to the design of real time systems. The conceptual core of the language may be seen in the glossary by looking at the starred items as opposed to the capitalized which represent ISEM concepts. Where the conceptual core is honed down to the absolute minimal set of concepts necessary to represent all the minimal methods; ISEM on the other hand might be expanded to what ever is useful in the actual design process.

2. FORMAL DESIGN LANGUAGES

Programming languages are one well worked out kind of formal language which allows software implementation. However, as we move up the ladder of abstraction to the level where non-localization no longer appears as it does within the programming language, where all the design element’s descriptions maybe localized, we find that there is a possibility of defining another kind of formalized design language at this level. At the present abstract designs are normally represented by graphical notations in various methodologies. However, these methodologies and their associated enabling graphical tools do not cover all the necessary aspects of design so designers find them less than completely useful. In fact much of design still occurs on the back of envelopes, so to speak. ISEM seeks to create a formal design language at the correct level of abstraction for software designers work which also covers all the necessary aspects of design representation. This language could be rendered into a graphical notation when it has stabilized. However, the first point is to get a complete set of representations, instead of rushing into graphical notations which do not do the job. Also, the formal design language itself could be used to represent designs, in a form like source code, which many software engineers would prefer. All design languages need to offer both linguistic and graphical forms for software design because there are generally different types of people who are more accustom to one rather than the other. I have offered graphical forms taken from the literature for each of the core methods in the second part of this series. However, graphical descriptions lack precision and have a limitation on the complexity they may describe. So even though I am a graphically oriented person, I was driven to create a formal design language in order to cover express everything that needed to be expressed in software designs.

ISEM is not like VDM or Z. Each of these formal languages are generic expressions of the principles of logic and abstract mathematics which may be applied to describe and constrain software specifications. They lack an explicit model of software architecture. ISEM attempts to give that architectural model an so VDM or Z are actually complementary to ISEM and may be used to constrain its architectural designs in the same way they now constrain specifications.

3. THE DESIGN OF ISEM

ISEM was designed with some very different goals from those that normally drive the development of programming languages. First ISEM was not meant to support implementation at all. So it lacks all the constructs that normally appear in a programming language. Instead ISEM would basically allow the definition of design elements and their connection to each other into an architecture. Second, ISEM wants to support incremental definition. So that if the designer know any one fact about the design he can record that without having to know everything about a particular design element. This means if the designer knows about one state of a state machine he should be able to state that the state machine has that state and nothing else. In an implementation language one normally has to know everything about a state machine in order to write its implementation. In design one wants the design facts to build up one by one until one knows enough to restate it in a more concise form. Third, ISEM was to be a language with no deep structure. Thus it should be composed of single statements like those of the BASIC programming language. The recombination of language components using the deep structure of the language would not be allowed. The ISEM language has only surface structure. This simplifies the language and allows it to be used to not just state designs but to analyze design relationships. Each sentence in ISEM is an explicit design relationship. Fourth, the ISEM language is open
ended. It allows new statements to be added by the designer as needed. So the language dealt with in this paper is only the kernel of a language that would be extensively developed and modified in use.

These are some of the ground rules in the development of the ISEM language that makes it very different from a normal programming language. The fact that specialized formal languages can be constructed has not become widely very widely known. People usually think of creating their own programming language but rarely do they think about creating their own formal non-programming language to describe some phenomena. One exception is the biologists who have begun using formal languages to describe biological phenomena. The point is that once a formal description exists using a specialized formal language then it may be accessed as a database or it may be used to drive a simulation even though it cannot be compiled. The descriptive force of formal languages is very great since a few language production rules may express a great range of concrete design possibilities. For instance the AUTOMATA sub-language can express all possible state machines and it only has twenty or thirty statements. Thus the great boon of formal languages is their explicitness and their fund of expressiveness.

The ISEM language is not meant to be compiled instead it is meant to serve as a database of the current design while it is being formed and changed. It may also ultimately serve as the basis for simulation of the design in order to prove certain properties before the design is implemented. In this mode it will serve as a means to explore the design space. And this is the most important function that the ISEM language could fill. Now there is very little exploration of the design space of systems. This is because designs are so complex and they cannot be simulated. However, once we have a complete formalization for design representations it will be possible to simulate many static and dynamic aspects of the design and also change that design so as to find out what other design configurations have to offer for improved satisfaction of crucial required parameters.

So the ultimate aim of ISEM is toward design simulation. However, until we have simulators to support formal design languages ISEM can serve to represent concrete designs and to study the nature of design representations themselves. We need to know what are the minimal set of design statements needed to represent all the aspects of real time systems. Then we need to know what are the set of extensions that designers most often use. These two sets of statements would make up the ISEM language. Then each designer would add their own extensions which reflect their applications area or their own understandings of the correct elements necessary for their designs. But ISEM has another function. It has an educational function in the movement of our craft from the beginnings of the commercial stage into the stage of professional engineering. The ISEM language represents a kernel of design knowledge and a way of organizing that knowledge that new or untutored experienced software engineers can learn from. ISEM represents the formalism of design concepts which every software engineer should be familiar with and be able to apply. It is the foundation on which design heuristics can be built which would allow the neonate to learn from the more experienced engineer. It also contains a conceptual structure that will serve as the a solid center for the learning software engineer to attach the knowledge as he learns how to design different applications. But more than that it is a language that will allow software engineers to express their designs for mutual elucidation and criticism so that using this formalism designs should be more accessible, visible and understandable.

Design formalisms are still in their infancy. This one has its own peculiarities given its origin in concerns about the philosophical and system theoretic foundations of software practice. It is however an example upon which others may build or transform for their own purposes. It is the beginning of a foundation for software engineering as a professional discipline because it makes the basis of software engineering knowledge explicit. What software engineering has to add to Computer Science beyond the concern with the sociological and managerial dimensions of the actual production of software is the knowledge of software design methods that allow systems to be designed at higher levels of abstraction above the level of source code. Software methods are the kernel of this new discipline and ISEM attempts to survey the whole field of necessary methods based on the organizing framework of the four perspectives and the idea that there are 16 minimal methods necessary to represent all aspects of real-time software design.

4. THE STRUCTURE OF ISEM

ISEM doesn’t really have a grammar like we are used to with programming languages. It was consciously designed to be like the BASIC programming language which only has a series of statements with minimal substitution. This will allow the language to be built up incrementally
because separate statements may be added without recompiling the parser like a macro language which are used to extend some applications. Instead multiple levels of substitution ISEM has a template for the formation of design statements. This pattern is used to form all ISEM statements and represents a configuration of basic parts of speech.

STATMENT: AC1
Short Title: PositWrkrDoesActInStateOnEvent
Long Title: PositWorkerDoesActionInStateOnEvent
SUB-LANGUAGE: ACTOR: {which grammar is optional}
OPERATION: POSIT: {statement operator, first noun}
NOUN: WORKER {statement operand}
Identifier: worker_name {instance identifier}
VERB: DOES {statement verb}
PREPOSITION: none {first connector}
NOUN: ACTION {second noun}
Identifier: action_name {instance identifier}
PREPOSITION: IN {second connector}
NOUN: STATE {third noun}
Identifier: state_name {instance identifier}
PREPOSITION: ON {third connector}
NOUN: EVENT {fourth noun}
Identifier: event_name {instance identifier}
PREPOSITION: none {fourth connector}
NOUN: none {fifth noun}
Identifier: none {instance identifier}
Qualifier: none {attribute}
Yindex {the current sort position of all ISEM statements}
Zindex: {sequential statement number, does not change}

Objects are related to the first using prepositions. All the parts of speech in the template are slots that may be empty or full depending on what the designer is attempting to express. So in this case the fifth noun and it’s preposition positions are left empty. The Operator slot is filled with the token ‘POSIT:’ which is optional. Optional words have a colon appended to show that they are not necessary to the sentence. Generally all the action statements are will use the operator-operand form while all the descriptive statements which describe the structure of the design will use the SVO sub-pattern.

EXAMPLE:
WORKER onEvent1 DOES ACTION setClock IN STATE waiting ON EVENT reset.

In a design the pattern will have it’s instance variables filled in as in the example above. Notice that every identifier follows its type so that it is always clear what type any particular token is as opposed to most programming language where you have to remember what type a particular token is. This allows the sentence to function as a database in which the type of a field precedes the contents of a field. One may always search for all ‘ACTIONS’ or all ‘EVENTS’ or what ever one needs to find using standard queries directly on the design text itself. Or the design may be entered into a database set up on this pattern and the database could be queried. The pattern is simple so the designer can make up his own statements with relative ease. The basis idea of the template is to allow a minimal system of design elements to be operated on at one time and connected to any other design element. It turns out that the language as it stands does not use the fifth noun or the fourth preposition. These places are reserved for more complex situations not covered by the language as it stands.

The language itself is divided into sub-languages. Each sub language addresses a particular minimal method or some associated realm of description necessary to represent designs. The following languages have been incorporated to date:

ACTOR: Describes the parallel agents perspective based on Gomaa’s DARTS and Agha’s Actors.
ARCHITECTURE: Describes the generic software architectural elements.
AUTOMATA: Describes State Machines.
DOMAIN: Describes the application domain and re-
quirements analysis extensions to the methods.

INFORMATION: Describes information network within the system.

LIST: An augmented list.

MACHINE: Describes a virtual layered machine as formulated initially by Neilson & Shumate.

PROCESS: Describes dataflow and control flow techniques as defined by Hatley/Pirbhai.

PETRINET: Describes Petri News formalism.

SITUATION: Describes the situational aspects of the active objects design which involves placement of data.

SET: An augmented mathematical set.

SYSTEM: Describes the software system as a whole in relation to the environment and meta-systems.

TEMPORALITY: Describes the time aspect of the system through temporal logic.

TURING: Adds systems mechanization elements that make it possible to unify the design description. Named after Alan Turing.

WORLD: Describes the world line and scenario minimal methods that show how different actors communicate within the relativistic world.

There are currently over seven hundred statements in all the sub-languages taken together. ISEM is merely the sum of its sub-languages and it may be that various other sub-languages are added as their necessity becomes apparent. So ISEM is extensible both within a sub-language and also by adding other sub-languages. For instance a logical sub-language could be added to fill the place of VDM or Z; or those formal languages could be used in conjunction with the ISEM sub-languages. The guiding concept behind ISEM is flexibility. Since ISEM is not tied to a compiler it may grow and change easily.

5. THE ISEM SET AND LIST SUB-LANGUAGES

As an example how the ISEM language is constructed we shall first look at the SET sub-language. The SET sub-language has all the necessary statements to construct any set. For instance:

(01) DEFINE: IDENTIFIER bag IS SET.

(02) BAG SET bag1.

With this statement (01) a SET is created and with the next (02) it is turned into a BAG in which different instance of the same kind may be stored. Here “bag1” is the identifier of an instance of type SET which we now know is a BAG.

(03) POSIT: ELEMENT x, 1, 2, 2 BELONGS TO SET bag1.

Wherever there may be an identifier there may be a list of identifier unless there is a special rule produced to exclude it. So here the elements “x, 1, 2, 2” are placed in “bag1”. Now we know why is in the bag. If ISEM were automated we could query what we had put in the bag.

(04) INQUIRE MEMBERSHIP OF SET bag1.

And presumably the design system would respond something like:

bag1 = x, 1, 2, 2.

So it is with all the aspects of the ISEM language. In fact, all this seems so simple the question at once appears why hasn’t anything like this been done before this. The answer is that there have been many partial attempts to describe designs and some automated tools have been designed to enable these descriptive techniques. ISEM really only brings the idea of using a human readable formal design language to describing all of the pertinent aspects of design. So what is described here is well within our reach technologically. In fact as an industry we have gone to lengths to develop means of depicting designs graphically when we had the tools to develop formal languages already. Since we skipped over the task of developing the formal language that fully expresses the design it is necessary to now go back and do that. The form tools we now need are both graphical and linguistic. An example is the languages that some document production systems have for describing text documents with graphics (for instance, Maker Interchange Format of the Frame Maker document processor available from Frame Technologies). You can draw a picture in Frame Maker then look at the MIF which is more or less human readable. You can change the MIF and see the modified picture when you read the document back into the document processor. The ability to move back and forth between graphical and textual representations is crucial to our ability to do design work. When we are doing some construction graphics is appropriate while for other construction work text is more appropriate. With current systems what we lack is the
formal language to underlie the graphics. In some ways we have the cart before the horse. We solved the exciting technological problem and left unsolved the hard problem of what should be in a design formalism, what are all the crucial aspects of real-time systems at the level of methodological abstraction. It is really only this hard unanswered question that ISEM takes a crack at. It is the first in what will undoubtedly be a series of formal languages to express the essentials of real-time design. Once we have that, and it is more or less agreed on then software engineering as a discipline is going to take a giant step toward becoming a professional engineering discipline. Of the vendors, to my knowledge, it is Interactive Development Environments’s toolset called ‘Software Through Pictures’ that comes closest to this vision. Underlying each of their editors is a data language which mediated between the graphical representation and the underlying database. The real problem they have not addressed yet is that this language must be easily human readable. That the designer should be able to change it directly and see the new pictures, and that all the sub-languages need to interact properly. Now IDE’s editors are all separate languages and graphical notations with minimal mostly behind the scenes connection as dictated by methodologies. Instead I envision a similar tool set on top of a unified design language.

IDE has gone the crucial step of recognizing that formal languages have an important place in their system. Now what I am interested in is starting the process of defining that language.

The ISEM SET sub-language is an example which all the other sub-languages follow the broad outlines of in the way they work. If you look at the SET entity relationship diagram or the list of SET language statements you will see that there are some statements that construct the essential relations and another set of statements that do operations on those relations. The basis steps are always DEFINE the instances; POSIT the relations between instances based on design element types; produce LEMMAs in which specific temporary attributes or relations are set up; and OPERATE on the instances and there configurations. You can also see that there is a statement which will transform a SET into a LIST. Other auxiliary operations will transform design elements in various ways.

There are many ways that a SET language might be designed. This is only one of a myriad styles of formalism. The style is inessential. The essential point is that we need to learn to use these formal languages and that software engineering has a linguistic aspect that goes beyond what logical and mathematical formalisms can offer. This linguistic aspect of software engineering methods is crucial to the development of the discipline. It is introduced with the partially ordered AGENT and FUNCTIONAL perspectives. What is lost with the inability to fully order agency and functional aspects of a system is gained back with the ability to express meaningful relationships between design elements.

In what follows we will take a tour of all the different languages and mention the peculiar features and generally attempt to point out how the structure of the design elements described by the sub-languages fit together. For the most part the sub-languages are self explanatory. If you see something that is missing remember this is just an example and the language is extensible to add relations that I did not think about when making up the individual statements. What I have attempted to do is provide a core set of design constructs that covers all sixteen minimal methods and some other concepts which I deemed necessary to make the language workable for the designer such as the SYSTEM, ARCHITECTURE, and DOMAIN sub-languages.

I will follow the order of the ISEM entity relationship charts. These charts are provided so the reader can get a quick overview of the languages and the relations posited by each statement. The statements often work together and have specific interrelations which show up well in these diagrams. It is difficult for us to read the language directly and see these relationships. But what the language lacks in graphical intuitiveness it make up for by its formal explicitness.

A glossary is provided to give some explanation of the individual design elements. Capitalized words in the following guided tour indicated design elements that may be found in the glossary.

6. SYSTEM

Notice that many times a generic design element like SYSTEM will have three associated statements:

DEFINE: IDENTIFIER id IS SYSTEM.
DEFINE: IDENTIFIER id IS SYSTEM OF TYPE id.
DEFINE: SYSTEM id IS TYPE id.

These allow a SYSTEM to be identified without
specifying its type. Or one may specify the system and its type together. Or one may take a system identifier that already exists and specify it’s type. This shows how the language allows incremental build up of determinations of the design. This aspect of the language is very important as design is an iterative and recursive non-routine enterprise.

The system sub-language describes the relationship between the SYSTEM and it’s ENVIRONMENT and any METASYSTEM that it operates in relation to. Generally current methodologies are poor at describing these relations which are often as crucial as the internal articulation of the system. Systems, metasystems, and environments all have hierarchical decompositions, possibly multiple ones which conflict. Being able to explicitly define any HIERARCHY is of use in defining the envelope of the system and how it interacts with the outside world. Context diagrams are often too simplistic. Since systems, metasystems, and environments are all multiply hierarchical and since interaction may occur between any levels in any hierarchy the context diagram will not give us the necessary depth of representation needed for complex real-time systems. This sub-language allows us to describe any and all situations explicitly. Each system and metasystem also has a specific configuration of artifacts that describe it. The environment contains all the EXTERNALS. The reason metasystems are included is that we are using George Klir’s General Systems Theory as our basis for describing systems and it contains explicit meta-systems and meta-structures. Both of those constructs would be described here in terms of metasystems.

A system has an INFRASTRUCTURE composed of APPLICATION, SERVICE, INTEGRATOR, & BACKPLANE. In most instances it is the Application that is of interest. But applications many times work together and need specific software to integrate them; or they rely on other software to provide services; or they need to talk to each other and cooperate even though they are not integrated. So it is necessary to describe explicitly the background relations of applications to other system components. Each of these elements of the system has an IMPLEMENTATION and that runs on a PLATFORM which is a particular hardware software combination that allows the implementation to execute. All of these parts of the software system may have their own HIERARCHY.

Multiple design elements may have hierarchies. In fact any design element potentially may have a hierarchy although it is not explicitly represented as such in the language as it stands. It is expected that if the designer needs to express the hierarchical nature of another design element he will just extend the language to do so. The HIERARCHY contains levels which may be arranged in relation to each other in terms of which are ON TOP or ON BOTTOM of the other. LEVELs may also be adjacent so the hierarchy is not necessarily strict and may be partially ordered. But the levels of the hierarchy have relations to what is know as the GRID. The GRID is a four-dimensional matrix which is meant to hold the generalized design elements. The grid has four kinds of divisions called LAYER, PARTITION, STRATA, and TIER. Both systems and metasystems may have grids. The grid is a mechanism specifically designed to allow humans cope with complexity. It creates a series of places that may be super close packed to simulate the enmeshed quality of most real systems. The parallelism between levels and the divisions of the grid allow us to move from basic sorting of design elements to the actual placement of design elements in mnemonic places with four dimensional addresses so they may assume a specific spatial configuration that mimics their logical configuration.

All the major systemic features that have been mentioned so far may connect explicitly to a particular application domain. Thus we are viewing not just one application but a class of applications, not one system but a class of systems, etc.

7. DOMAIN

Domain analysis is necessary in order to optimize any class of applications or systems. One looks at the whole class and attempt to create design elements that will be useful over as much of the domain as possible. A domain is made up of GLOSSARY with DEFINITIONS, TAXONOMY, DOCUMENTs, REPRESENTATIONS, PRODUCTS, SPECIFICATIONS containing REQUIREMENTS under CONFIGURATION control. Thus the domain opens up the SYSTEM to a wider analytic perspective on whole classes of systems.

A configuration includes ARTIFACTs which include COMPONENTs that in turn include UNITs. Artifacts may form an AGGREGATION that has a VERSION, DATE, AUTHOR, BUILD or anything else. Aggregations allow the same artifact to appear in multiple contexts during design.

The main part of a DOMAIN is the TAXONOMY which
is a classificatory system of many CATEGORY. Categories contain TAXONs which differentiate elements based on their CHARACTERISTICS. A taxon is a kind of classificational template with the basic set of characteristics that members of the category may have. It is by using the taxonomy that the picture of the systems in a domain is worked up and systems are in turn made reusable and more generic.

8. ARCHITECTURE

ARCHITECTUREs include COMPONENTs that are positioned at INTERSECTIONs in the GRID. Each intersection has its COORDINATES. Architectures are composed of ARCs and NODEs. Nodes correspond to the UNITS within a component.

Nodes may have INTERFACES rather than direct connections with each other. This allows for encapsulation of architectural units. A NODE may be any entity within the ISEM sub-languages. An ARC may be any relation between entities. This allows complete flexibility in defining what is in the architectural design. Different applications may necessitate different types of architectures.

There are a series of four entity relationship charts that describe the relations between the different divisions of the GRID (LAYER, PARTITION, STRATA, & TIER) and the INTERSECTIONS. Basically an intersection is an intersection of any three of these divisions.

9. AUTOMATA

An automata is a Finite State Machine. It is composed of a STATEVAR that holds the current STATE. It also may have an associated meta-state called a MODE that actually changes the state machine’s behavior in some way.

In the state machine STATEs are connected by TRANSITIONS which are associated with EVENTS and ACTIONs. This allows the state machines to be built up incrementally by one transition at a time. Actions are associated with WORKERS that encapsulate PROCEDURES.

10. ACTOR

The autonomous agencies (separate individual processors of some sort) of a system are represented by three different concepts. An AGENT contains an ACTOR which contains a TASK. Basically an agent is an abstract autonomous being. An actor corresponds to Agha’s definition of ‘actor’ which is an concrete autonomous executing unit which is not tied to any one processor. A task on the other hand is tied to a particular processor which it is dependent on for resources. Here the OSSWG Real-time System Reference Model is used to describe the levels of operating systems. The lowest level is the LPOS (Local Processor Operating System) which allocated resources to tasks. The IRAX (Intermediate Resource Allocation eXecutive) allocated resources across a field of LPOS units and it would be at this level that actors would be allocated to different processors and moved. In order to be able to find resources, especially when they are moving it is necessary to have a NAMESPACE. The namespace guarantees unique identifiers that may be used as addresses for messages. The POSTALSERVICE is responsible for delivering these messages and making it appear that all message deliveries are local even if communications networks of various sophistications are involved. At the system level we have the SRAX (System Resource Allocation eXecutive) which allocates various resources to meta-actors called AGENTS. Agents may be a swarm of actors. All actors communicate via messages. But at the lowest level there may be Tasks or Interrupt Sub-routines which have different communications mechanisms. This level is important for real-time system development where performance optimization is constantly a factor. Of course the LPOS is associated with an individual processor, which a typical system different types.

At the task level there is a network of TASKs connected by CONNECTORS linked to specific PORTs. Connectors may be of various types such a QUEUE, FLAG, PIPE, SEMAPHORE, FILE, MAILBOX, RENDEZVOUS for ADA, or just a general multi-party INTERACTION. This is not meant to be an exhaustive list and surely many other communications mechanisms will be added depending on the specifics of the application.

The basic relations between OBJECTs, ATTRIBUTEs and the associated DATASTOREs are also defined in this sub-language. Objects have associated attributes which are kept in datastores. These data constructs may be read or written.

11. INFORMATION

This sub-language provides a generic way of expressing the content of information. There is a hierarchy where data is given context and becomes information which when
absorbed becomes knowledge. At the lowest level we are concerned with the DATUM which may express a myriad of values depending on its type. The generic value representation is the datum which may set the attributes of Entities or Objects based on the values of Expressions. Data values are usually time bound so they are also associated with an interval.

The datum may appear as part of an INFOPACKET which contains many datasets. It is normally the infopacket which surrounds any particular datum that is referenced within a system.

The multiple interrelations between events are portrayed as a series of production rules which based on configurations of events may change other aspects of the system.

12.MACHINE

In this sub-language the correspondences between the different levels of abstraction in the relations between the AGENT and PROCESS viewpoints is made clear. AGENT corresponds to PROCESS. PROCESSOR corresponds to TRANSFORM. TASK corresponds to FUNCTION. The combination of task and function is an EVENTITY. It is the MAPPING of functions onto autonomous individuals which is one of the core minimal methods. The other is the Virtual Layered Machine itself which embodies these mappings.

Both EVENTIES which are seen as the intersection of all four views on the design and PROCESORS (which is to say the LPOS too) has EXECUTIVES associated with them. An executive may either be based on an AUTOMATA or PETRINET. The executive controls a TASK or set of TASKs. The TASK has an associated Virtual Layered MACHINE and a THREAD which contains its allocated resources and keeps track of where it is in case of context switches. The task also has an associated QUEUE which contains MESSAGEs that indicate which OPERATIONS need to be worked on in what order.

The MACHINE is made up of an INSTRUCTIONSET containing INSTRUCTIONS. Instructions may be OPERATIONS associated with OBJECTs. Both operations and instructions directly implement FUNCTIONS perhaps using the general purpose WORKER to do so. A worker is like a macro which may contain a set of PROCEEDURES. Also a MACHINE may be associated with either an AUTOMATA or a PETRINET which fires the INSTRUCTIONS as their ACTIONs or TRANSITs.

There are a whole series of actions which connect OPERATIONS with OBJECTs and ATTRIBUTES that are described. Also the relations between MESSAGEs that invoke INSTRUCTIONs and come in INFOPACKETS is made explicit.

13.PROCESS

The Process sub-language implements the Hatley/Pirbhai methodology. In that methodology controlflow is added to the dataflow diagrams developed by Tom DeMarco. CONTROLFLOWs connect PROCESS (the dataflow functionality bubble) and the CONTROLSPEC. Controlspecs may be either PETRINETs or AUTOMATA which drive an ACTIVATOR that turns process bubbles on and off. It needs to be remembered that in the Hatley methodology dataflow bubbles are on and will be activated to produce their outputs unless explicitly turned off by an ACTIVATOR. Before the automata or petrinet may come a decision table to reduce signals to the minimal set.

A DATAFLOW line connects PROCESSes. The envelope of the system is described by a CONTEXT bubble where dataflows connect to EXTERNALs. The dataflow lines may also connect to DATASTOREs which contain OBJECTs.

14.PETRINET

The PETRINET is a network of PLACEs and TRANSITs in which markers move to simulate controlflow within an application. The petrinet is the dual of the automata (state machine) but it takes many more petrinet transitions to describe a system than a state machine. But in the petrinet the control flow is easily seen ad explicit. There are ENTRY and EXIT places where the markers start out or end up. Since we are describing a colored petrinet it means that markers themselves have structure so that a particular transit must distinguish different patterns of incoming markers. This means that each transit has an associated set of DECISION rules that tell it whether to fire or not. Also the transits are associated with their own transient states.
which are called BANNERs that contain INSIGNIA. When the DECISION allows the TRANSIT to fire because a recognized pattern of markers has arrived in the input PLACES then the TRANSIT may set the BANNER to an INSIGNIA. These terms have been invented to maintain the distinction between the PETRINETs and the AUTOMATA.

A set of connection operations have been specified for building the PETRINET in an incremental way.

15. SITUATION

The situation describes everything that may take place at a SITE which is a staging area for the construction of ENTITIES.

The ENTITY may have both a BEHAVIOR and an ASSEMBLY. The ASSEMBLY contains its PARTs. The BEHAVIOR represents the response of the ENTITY to higher level inputs. Both the ASSEMBLY and the BEHAVIOR may be specified in terms of PETRINETs or AUTOMATA.

An ENTITY belongs to a CLASS in an inheritance hierarchy. It may have a MODE (meta-state) and has a set of ATTRIBUTES. The entity may be mapped to an OBJECT. Entities are distinguished from objects because the concept of object has a certain narrow range of applicability suggesting encapsulation of persistent data. Entity is meant to be a much more general purpose term. Entities have INSTANCES which may in fact be PARTs of an ENTITY. An entity may have a FACE which is a set of it’s attributes presented in a particular manner.

The entity may exist at a particular SITE. The site is a non-located INTERSECTION which may later be connected to an intersection in the GRID. Also the ENTITY may have SLOTS that are ATTRIBUTES connected to EQUATIONS that recalculate when any of their inputs change or periodically. The attribute as it appears in an INSTANCE is called a REPLICA.

16. TEMPORALITY

This sub-language implements Allen’s Interval Logic in a context where it may be connected to Temporal Logic. Temporal Logic treats modalities such as necessity where one event is necessary because a prior event occurred. A TEMPORALITY is a SHEAF of signals. A sheaf of signals may be further broken down into a BUNDLE of SIGNALs. SIGNALs are modulated lines of input which show variation with time. Within a sheaf bundles may BRANCH off from other bundles of signals. These branches which split and join allow one to reason about necessity within the possible worlds of the system. The ATTRIBUTEs of an ENTITY may be connected to a particular signal and receive its modulation. When this connection occurs we have and EVENTITY.

The EVENTITY is an ENTITY plus a LIFECYCLE whether portrayed as an AUTOMATA or PETRINET. EVENTS may have all the relations between each other described in Allen’s Interval Logic such as ‘event is before event’ etc. Events may be associated with INTERVALs that are defined in terms of two TIMEPOINTs or a TIMEPOINT and a TIMESSPAN.

17. TURING

I have named this sub-language in honor of Alan Turing. It contains several key concepts such as ENVIRONMENT, EVENTITY, METASYSTEM, SYSTEM which connect all the other concepts within the sub-languages together. Each of these key concepts have some relation to the SHEAF, BUNDLE, FLAG, EVENTITY, and AUTONETIC. Here we see the confluence of the central concepts of our systems theory. EVENTITYs are active objects that combine ENTITYs and LIFECYCLES. A SYSTEM and a METASYSTEM are composed of EVENTITYs. Or from another point of view they may be seen as special cases of EVENTITYs. The ENVIRONMENT functions in a similar way. In any case an EVENTITY is an ENTITY that has been temporalized, thus the connection to the SHEAF and BUNDLES of signals or to a CLOCK of it’s own. In the case of all these highest level conceptual entities in our systems theory there may be a connection to what is called an AUTONETIC. This word exists as a conflation of AUTOMATIC and CYBERNETIC. The AUTONETIC is a description of the BEHAVIOR, facing upward in the hierarchy, or ASSEMBLY, facing downward in the hierarchy, like one of Arthur Koestler’s ‘holons’.

AUTONETIC’s contain both BEHAVIOR and ASSEMBLY modules and may even have their own meta-states called FLAGs, these are global variables, that may either be STATEVAR or BANNERS.

This sub-language contains all the necessary calls needed
to set the time either directly or by reference clocks. It is assumed that the problems associated with Special Relativity also plague all distributed real-time systems which may not have accurate global reference clocks if the system is too big.

18. WORLD

In this sub-language the relations between AGENTs on different woldlines within a relativistic universe are considered. The general model is Minkowski’s light cones of possible causality. The Microelectronic Computer Consortium (MCC) model called RADDLE which was made into a tool called VERDI is taken as the model of the relation of different agents working together in a distributed fashion. The ability for multi-party interactions is taken for granted. A generic description of the operation of an agent on a worldline is called a ROLE in the RADDLE language. Roles are made up of constructs that may either contain INTERACTIONs with sets of other agents, ITERATORs, SEQUENCEs, or CHOICEs. The agent cycles through the pattern for the ROLE. VERDI the MCC tool allows this to be simulated. Constructs may trigger events which may feed into AUTOMATA or PETRINETS. Each WORLDLINE is generated by a ROLE and may allow the display of multiple SCENARIOs that cut across worldlines that may involve many agents.

19. GENERAL COMMENTS ON ISEM

As you can see, now that we have made a cursory tour of the sets of sub-languages, ISEM is a combination of many existing methodologies. Some of the concepts overlap because as you move from one perspective on design to another the semantics changes. The ideal is to make this overlapping as small as possible. But from our perspective on design it will never be completely gotten rid of so we have interchangeable concepts in the languages with different semantic shading. Also it is somewhat arbitrary in which language some of the statements appear. This is an area where further work needs to be done to determine, given cross links, where any particular statement should appear. The work of narrowing down this language to some minimal and extended set needs to be done, but that must be the work of practitioners as they attempt to apply the language to the design of real-time systems. The language is not meant to be an abstract mathematical image, but rather it is meant to be the tool box of concepts that prove useful to the practitioner. This has an empirical basis. Formalization allows us to narrow down that set to the smallest possible subset of all the possible design statements.

In ISEM there are certain concepts that do not relate to each other at all. There are others that are highly interrelated. The grouping of the statements into sub-languages should work to get all the highly interconnected concepts in as close a proximity to each other as possible but still allow necessary connections that breach the barriers between sub-languages. When the set of statements becomes well formulated then these interconnections will tell us a lot about the deep structure of the design methods. This is because some elements vanish as you move from perspective to perspective while others do not. If we could ever know exactly how any given design concept is transformed by the movement between perspectives then we would know a lot more about the inner structure of real-time systems.

As has been noted new sub-languages could be added or new statements could be added to any given sub-language. Unlike a programming language extensibility of the language itself is the key. The set of concepts presented here comes from an analysis of existing design methods separating the fundamental concepts out and adding those necessary to integrate all these methods into a single way of approaching systems design. Other syntheses are possible and encouraged. Designers need to be familiar with all the different methods so they may use them like a tool box in their work. One of the big areas for research is which set of methods is best for a particular application type or design situation.

20. THE FUTURE OF ISEM

ISEM is incomplete, and not fully self-consistent. It needs to be applied to the work of design and rung out in that kind of environment. Once a stable and useful set of statements is produced then there are many uses for such a formalized design language many of which have already been mentioned.

SIMULATION: The language could be used to do dynamic simulations of designs before they are implemented to gain some idea of their performance characteristics.

COMMUNICATION: The language if known by a group of designers may serve as a medium of communication about designs which is sadly lacking in the Software
Engineering discipline today.

**REPRESENTATION:** We lack any complete representation methodologies today. ISEM strives to be complete. A complete design representation will allow us to record designs and changes in them which should be a boon to understanding systems when their designers are long gone.

**EDUCATION:** Neophyte Software Engineers may use ISEM as a core around which to build their knowledge of Software Engineering heuristics and experience.

**REUSE:** ISEM representations may be added directly to the specifications of source code to record the design at a higher level of abstraction. A tool which checks visible identifiers against the ISEM identifiers could ensure that the design and implementations are connected at the most superficial level.

**REVERSE ENGINEERING:** If you have to understand some one else’s design it would certainly help to have a design notation to put what you have learned in so that knowledge would not be lost again and to jog the memory.

**FOUNDATION FOR HEURISTICS:** What is missing from the ISEM language is the Heuristics that tell the designer how to use the methods to get the best possible design. With a standardize language for expressing design formalisms these heuristics may be stated as rules or policies in terms that are easier to understand.

**BASIS FOR GRAPHICAL NOTATION:** Once we have some idea what are all the different concepts we need and have some idea about their interrelationships then we can begin to build a set of graphical notations based on those that already are common which encompass all the aspects of design for real-time systems and not some improper subset.

**ACCESSING DESIGN DATABASES:** Once we have a standard language we can use it to build up design databases that may be accessed using key words and other type of queries.

**DEEPER FORMALIZATION:** By having a formalization of design elements we can added these to the array of specification formalisms to get better control of the relationship between the design and the specification.

These are a few of the uses of the ISEM real-time design language. I am sure there are more that may be discovered it we were to use such a system to describe and design actual real-time systems.

### 21. IMPLEMENTATION OF ISEM

The formal design language presented here is a prototype of an ideal meta-design language. If it were implemented so that it had its own parser it would probably have to change in many ways. An experiment with such an implementation has been attempted by the author. In that implementation it was seen that the BNF of the language has to be structured in a specific way to get the Abstract Syntax Tree to have a certain desirable form. Having parsers is a first step toward meeting the other goals listed in the last section. The value of the prototype ISEM language which has not been fully implemented is that it shows clearly the semantic field which any formal design language should eventually cover. Finding the best possible way to cover this field must be the result of further study.

### 22. CONCLUSIONS

This is a brief introduction to the Integral Software Engineering Methodology (ISEM). An in-depth exposition is really needed in order to show how all the concepts interrelate and to act as a tutorial for those unfamiliar with modern Software Engineering methodologies that have been drawn from to create language that represents a union of many discrete techniques. This union is justified because we need to consider real-time systems from many different perspectives and to move between those perspectives. But all the methodologies are just a tool box for use by the intelligent and creative designer who faces hard problems in the design of real-time systems. There is in real-time software systems an unbridgable gap which can only be crossed in the designers intuition, which is really the gap between linguistic and mathematical types of representations. ISEM is an example of a concrete formal-structural system after the model of those described by George Klir. As such it is a step beyond merely logical or set-theoretical mathematical models into the domain of software systems which has it’s own semantics. We need to specify that semantic field very precisely so we can appreciate that fuzzy, ultimately uncapturable nature of software. It is exactly the fuzzy nature of software, as someone expressed it 'like nailing jelly to a tree’, that we need to comprehend so that we may adequately make use of that nature to adapt and fit to the world the systems we
build.

KEYWORDS:


ABSTRACT OF SERIES:

Software Engineering exemplifies many of the major issues in Philosophy of Science. This is because of its close relation between a non-representable software theory that is the product of design and the testing of the software. Software Engineering methods are being developed to make various representations of the software design theory. These methods need to be related to each other by a general paradigm. This paper proposes a deep paradigm motivated by recent developments in western ontology which explains the generally recognized difficulties developing software. The paradigm situates the underlying field within which methods operate and explains the nature of that field which necessitates the multiple perspectives on the software design theory. Part One deals with new perspectives on the ontological foundations of software engineering based on recent developments in western philosophy. Part two deals with the meta-methodology of software engineering by laying out the structural relations between various software architectural methods. Part three presents the Integral Software Engineering Methodology formal design language.

OVERVIEW OF PART THREE:

In this third part the Integral Software Engineering Methodology will be explained. It is based on the ontological and systems theoretic foundations laid in the previous parts of this series. The methodology is an example of a complete Software Design methodology expressed in a formal language which may be used to express software designs at the proper level of abstraction.
The Future Of Software Process

Kent D. Palmer, Ph.D.
Software Engineering Technologist
PO Box 1632 Orange CA 92856
palmer@kent.name

INTEGRATED PROCESS DEVELOPMENT

This paper will attempt to survey where we are in terms of the relationship between state-of-the-art and state-of-the-practice in the development and application of software engineering processes to actual software engineering practice. From the vantage point gained through this survey, the essay will attempt to peer into the future of software process development. It will draw upon work done at a specific company, but will emphasize the theoretical aspects of process improvement work.

Certainly the initiatives started by the Software Engineering Institute (SEI) have begun to be felt throughout the portion of the industrial community which responds to Department of Defense needs. At the example company we are applying these same SEI driven improvement approaches to all of Engineering. This stemmed from a recent organizational restructuring that moved away from a functional organization toward a team-based organizational structure. Team structures were settled upon as one solution coming out of an organizational excellence exercise. However, we wanted to avoid going from the pitfalls of one organization to the pitfalls of the opposite organization. Thus, when team structuring was applied within the organization, a special process team was set up. The charter of the process team was to focus on and guide the improvement of the product development processes of all the teams within Engineering. Under the auspices of the process team special committees were set up for each Engineering discipline. These committees became recognized centers of process improvement and monitoring. The following committees now exist within our division:

- OSPG Oversight Process Group
- CEPG Cost Estimating Process Group
- SWPG Software Process Group
- HWPG Hardware Process Group
- SAPG Systems Analysis Process Group
- NBPG New Business Process Group
- PSPG Product Support Process Group

This essay will be written from the point of view of the Chair of the Software Engineering Process Group (SWPG). It will address how software process improvement fits into overall organizational improvement. The main point is that the isolation of software engineering from the rest of engineering and the concentration on its improvement without dealing with the overall picture is a fundamental mistake. This only really becomes clear when the whole of the engineering organization is addressed, and the full complexity of the embedding of software within the organization becomes evident. The interfaces of the software process within the overall organization are myriad. If the Engineering process does not treat both sides of those interfaces, then all the improvements gained in software engineering may be easily lost elsewhere. It turns out that many times software is the least of an organization’s problems. In fact, software engineering has garnered so much attention that it is actually more advanced in process understanding than other parts of the organization in many cases. Thus, software process improvement efforts need to act as a catalyst for general organizational process improvement.

The tendency to isolate software and to concentrate on improving it as an island in the organization as per SEI direction is a tragedy from an industry viewpoint. SEI needs to broaden its scope to include general process improvement as its charter; or new organizations like SEI need to be formed for a coordinated push to improve all aspects of the organization. For instance, a Hardware Engineering Institute and a Systems Engineering Institute could be formed, and these could work together with SEI to bring about a balanced improvement program.
Our own approach to the problem of process definition and improvement turned out to be very wide indeed. We eventually embraced the goal of producing an Integrated Product Development (IPD) process which spans not only Engineering but also Logistics and Production. In IPD a concurrent engineering team is made up of members from all these functions who work together in an integrated fashion to develop the end product. The IPD team has responsibility for the entire development of the product.

We decided that this broad cooperation should be mirrored in our IPD oriented process definition. This is another initiative similar to that of SEI which is directed by our customer. We saw it as complementary to our team-based organizational structure. Thus, the Chairpersons of the different process committees gathered and hammered out a hierarchical decomposition of our IPD process using a dataflow diagram analysis technique. That diagram, once finished, gave a picture of the whole of our IPD process which would be implemented by a team drawn from all parts of our organization. The context bubble of this decomposition is called the Integrated Product Development Process which contains the process building blocks shown in the hierarchy diagram.

At the beginning of the process the software process group identified twenty-five kinds of work needed to carry out software development. Over time we reduced these to a core group of software processes which keep evolving as we understand the problem better:

Software Developmental Processes
- software requirements analysis [stand alone]
- software design [stand alone]
- software implementation [stand alone]
- software integration and testing [stand alone]
- software prototyping [stand alone]

Software Support Processes
- software configuration management [absorbed]
- software review [absorbed]
- software quality assurance [absorbed]
- software documentation [absorbed]

Software Control Processes
- software estimating [absorbed]
- software project planning [absorbed]
- software project monitoring and control [absorbed]
- software external liaison [absorbed]

Software Organizational Processes
- software process improvement [absorbed]
- software technology [absorbed]
- software training [absorbed]
- software parametrics [absorbed]
- software reuse [absorbed]

Of these software processes only four were not absorbed into other like processes shared by other functions within the organization. The processes that were not absorbed were the essential software processes that produce a purely software-related transformation in the product. All other supporting, control or organizational processes were common with systems, hardware and other functions. Within these fifty engineering processes the software process group now owns only seven processes. Within definition there is software requirements analysis. Software prototyping, software design, software implementation, software integration and test are visible as separable elements. Software reuse exists within the System Reuse and Element Reuse building blocks.

There is an important lesson in all of this. Software is almost completely embedded in the overall organization. Process descriptions for hardware and systems will cause redundancy unless all engineering and other non-engineering functions such as logistics and production are considered at the same time as software process is being defined. The efficiency of defining only one process for all of engineering far outweighs the costs of considering each function separately. Therefore, the SEI is doing us a disservice by pushing us to define software processes and ignoring the fundamental business processes shared by multiple functions. Fortunately, our Division started down the road of defining processes for all of engineering at the same time as embarking on moving up the SEI maturity levels. Hopefully we will be able to reap the benefits of defining one process for all functions. But I wonder how many other organizations will be faced with reworking their processes completely to incorporate the IPD initiatives and to apply them to other functions. The incremental cost of doing all Engineering is about triple. But the reward is that about three quarters of the processes necessary for software engineering are shared with other functions. So costs can be spread wider to make up for increased costs. Also, the major benefit is that interfaces
o Charter covers all of Engineering.

- Design-to-cost
- Integrated Product Development
- Reuse
- Customer focus
- Risk Management

- Team centered organization.

- Responds to many different initiatives:
  - TQM
  - SEI
  - DOD-STD-2167a-8
  - Air Force SDIP
  - DPRO process focus
with other disciplines are well defined if all engineering is modeled instead of software in isolation.

THE ART OF DEFINING PROCESSES

Following the example of other organizations, we adopted the dataflow diagram technique as our method for defining IPD processes. This dataflow model was hammered out in a series of meetings of the process Chairs over a six month period. From the beginning we were unhappy with this method of defining processes but did not realize until too late how it should have been done. The important lesson from these meetings was that our organization is so big and complex that no one person knows it all. Also there are many different viewpoints on the organization. Basically these are split between those interested in the global picture of the business and those who are narrowly focused on individual disciplines. The dataflow technique concentrates on the global picture and relegates individual disciplines to the lower reaches of the hierarchy. However, by taking a process building block approach, no element gets slighted because each discipline or viewpoint has its own work that is described by some building block. These building blocks are democratic in that all the different kinds of work that needs to get done must be described somewhere.

The dataflow modeling technique proceeds by functional decomposition which means making distinctions between kinds of work that are subdivided again and again until the simplest task is described. There is an inherent problem with this type of description. Its ability to retain information about the organization of process work is poor. As you go about differentiating kinds of work, it becomes clear that they are all enmeshed. The dataflow technique calls for arbitrary decisions that group kinds of work together no matter how enmeshed they are with other kinds of work. Thus the distinctions are always unsatisfactory. As the distinctions are argued back and forth, criteria are derived which usually, by consensus, sway the group to make the distinction in one way rather than another. However, in our process of deriving the model, we had no obvious way of retaining the basis for the distinction which was drawn. Thus we were constantly feeling as if knowledge was being lost, and many times we would have to re-derive why a particular kind of work was segregated and placed where it was in the hierarchy.

It was not until much later that I realized that the dataflow method, which serves software functional analysis so well, was the totally wrong approach to defining kinds of work. The dataflow technique assumes that from a functional point of view the system being described is not enmeshed but is in some sense homogeneous. Where one makes functional distinctions within a software system is more or less arbitrary because the system itself offers no resistance to functional decomposition. It is generally recognized that there are almost an infinite number of partitions for any software system’s functionality. It makes little difference which is chosen since the important distinctions will occur in the architecture, and functional analysis is merely a way to get full and consistent coverage of the requirements of the system. But for sociotechnical work description there is a different problem. Sociotechnical work is totally enmeshed. It is has multiple interpenetrating dimensions. Thus it cannot be easily decomposed functionally because at every level one runs into embedded dimensions of other types of work. For instance, planning occurs in every kind of work to some extent. Even though you have a planning process to produce major planning documents such as the Software Development Plan, there is still an element of planning embedded in every other kind of work. This multidimensional, embedded and enmeshed character of sociotechnical work is violated by the information retention-poor dataflow technique.

The answer to this problem is to use Venn diagrams with overlapping bubbles instead of segregating hierarchical decompositions. Each bubble can then represent a different dimension. So a planning bubble can intersect all the other process building blocks and still have its own space outside all of these. This overlapping or intersecting of kinds of work represents their multidimensional, embedded and enmeshed character very well. When you draw a set of overlapping circles, it is easy for everyone to understand what the different multifaceted sub-processes are and how they relate to each other. Once all the relations are made clear, then it is easy to make the decision as to how these will be grouped at the higher level. The interior lines of demarcation become dotted for those sub-processes that are grouped with a particular super process. The criteria that allowed this grouping to stand can be recorded as a note, and this knowledge is then not lost. Even though bubbles overlap, data flows between overlapping facets can still be added to the diagram in order to make interface surfaces clear.

It is important to recognize the differences that human sociotechnical systems have in relation to other types of
systems and not apply inappropriate modeling techniques when these are known to be inadequate. In this case the multidimensional, embedded, enmeshed character of human work causes pure dataflow modeling to distort the human organization. Because the dataflow assumes the perfect separability of kinds of work, one constantly runs up against the fact that kinds of work overlap or are multidimensionally related. Using Venn diagrams rather than decomposed bubbles recognizes this inherent character of human work and gives a more accurate model. The point is that because grouping criteria are ultimately derived, one ends up eventually with the same result but with a deeper understanding of the relationships than would be accessible otherwise.

Once building blocks are defined within the context of the hierarchical Venn diagram and the Venn diagram has been converted into a standard dataflow diagram, then it is possible to begin to define processes themselves. In our case we decided to have two basic aspects to our description:

**DEFINITION SECTION: WHAT**
- Name
- Number
- Dataflow: Showing inputs and outputs graphically.
- Description: Short description of kind of work.
- Inputs: Products needed to do the work.
- Tasks: Lower level kinds of work that make up the process.
- Verification: Questions asked to assure quality of work.
- Outputs: Products produced by work.
- Guidelines: Process related constraints and criteria.

**METHOD SECTION: HOW**
- Name
- Number
- Tutorial: Explanation of work for novice.
- Procedure: Stepwise ordering of sub-processes.
- Methods & Tools: Standard methods and tools used to do work.
- Metrics: Standard measures applied to work.
- Requirements: External constraints that assure minimal content.
- Policy: Ideal aim of the kind of work.
- Compliance Audit: Required audits of aspects of work.
- Notes: Miscellaneous points such as references.

Every process building block has a document which defines it in terms of a modified ETVX structure and tells how it is to be done. The two sections were separated in order to isolate the parts of the document that we felt would change the most (the how section) and for ease of reference. An exercise was carried out that made sure each of these topics had maximal conceptual separation from each other to reduce redundancy. In addition we listed the inputs and outputs a graphical representation of the process as a bubble with inputs and output lines for quick scan by those who are graphically oriented instead of textually oriented. Descriptions are normally a single paragraph that describe what a particular kind of work.
entails in general terms.

With respect to the Definition, or What, section we elected to have straightforward inputs and outputs instead of entry and exit criteria. It turned out that the criteria were fairly rare, and we subsumed those under the guidelines section when they occurred. In the guidelines section any process-related guidance as to ordering or concurrency of processes is mentioned which describe how the process is meant to be applied. Also, in the guidelines section the quality goals of consumer processes are identified. These then drive the identification of verification questions which will orient the products of a given process toward what their consumers need. This often is different from what the process itself would seek to optimize if left to its own devices. Thus a crucial connection is made between causal links in the process through the guidelines and verification sections.

The Method, or How, section parallels the What section. It contains a tutorial to explain basic concepts and point the novice to literature which will give him a background in the area. The tutorial should introduce any special vocabulary that will be needed to understand the nature of the process. The Procedure section may be either a diagram showing the interconnection and ordering of tasks mentioned in the Definition section or written step-by-step procedures which tell the order in which the work should be done. Procedures may either be optional or mandatory. Procedures may depart from the ordering of the sub-tasks if necessary. It is in this section that all external requirements are met which dictate a specific action be done. The Methods and Tools sections mention the Division standard methodologies that may be selected from to accomplish the work. The Metrics section mentions the metrics which either may or must be collected on this kind of work. The requirements section lists each external or internal requirement that must be satisfied by the process. These are mapped from the requirements document to each process where they are assigned to lower level processes. In those lower level processes the requirements are listed again, but this time it is mentioned where they are satisfied. If they can be satisfied by a specific action, then those actions are placed in the procedures. If the requirement calls for a state of affairs to be attained or is in some other way unclear then a statement of what aspect of the process will satisfy that is given. These unclear requirements are marked by a star to make them obvious to the enactor of the process who refers to the document. Requirements state what we are liable to be audited to in the following of our process. Policy, on the other hand, states our goals for a specific kind of work. There is often a wide discrepancy between requirements and policy mainly because our process goals are set high, and it might not be possible to attain those goals in every case. The Compliance Audit is the actual required process review which must be carried out to verify an ongoing aspect of the process. The SEI originally called this a “mechanism” which was a very confusing term. In the CMM they now use the terms monitoring and verification. The Notes section is for the addition of miscellaneous materials not covered in the format which are deemed necessary to add at some later date.

We are now in the process of writing these process definition documents for all of our process building blocks. The same scheme of topics is applied to all levels of process below the process building block level, and our goal is to have definitions of the building block level and at least the next level by the end of the fiscal year. The application of the same set of topics to each level gives an inner coherence to the process description which is elegant and easy to understand once the pattern is learned. We wrote a “process for defining process” using this same format which we gave out to all writers of process definition and which is expected to be followed by them. Thus, we could get a large number of experts working simultaneously on the process definition project. Due to project priorities constantly conflicting with process definition activities, it is a slow and painful process. The expert who is best suited to define a kind of work is also in high demand to do that work on a project. However, once the format is laid down and the kinds of work are isolated, it is straightforward and easy to write these process definitions for individuals with the right experience. It is merely a matter of them setting down and describing their own best practices in the most practical way possible.

The next task is to define the instantiation and tailoring document that shows how these kinds of work are used on a specific project. This is covered in the CMM as Integrated Project Management. We envision using the Spiral Model from SPC as our example lifecycle. For a particular project instantiation and tailoring of the process may be a complex task which needs to take into account the Work Breakdown Structure, Architecture, Accounting Structure, Tailored Standards, Selected Lifecycle, Selected Processes, and many other project specific details. However, generally we see the final set of work packages as specifying which kinds of work will be needed to carry them out. By just specifying the kinds of work, the entire
process structure is inherited along with the dataflow interconnections between those processes. Thus we see the use of the defined process as a simple addition to the work already done to plan a project which will bring with it a standardized division-wide process description that makes requirements clear and details the best practices which will allow us to claim to have a defined process to satisfy the requirements of the CMM.

This is an overview of where we are in our attempt to define an Engineering-wide process that embeds software processes but at the same time keeps them visible. In the process of creating this process many deficiencies in the techniques of process definition and enactment have made us wonder about the future of software process. The rest of this paper will address some possible elements of that future and how they might fit together into a complete picture. As is made clear in the remainder of the essay, there are some hard problems that need to be solved for software, and Engineering processes in general, to be effective. The solution to these problems will take some radical rethinking of the software processes and how they are used within the organization. In order to get us started looking at things from a more radical perspective, the following sections of the essay are offered as a point of departure.

**FACETED AUTOPOIETIC SOFTWARE PROCESS**

It has been shown by experimental studies that there is a hierarchy in the effort verses size trade-offs between prototyping, waterfall and spiral process models. Prototyping produces the cheapest and smallest product. Rigorous waterfall adherence produces the next highest cost/effort amount. Barry Boehm attempted to come up with a combination of prototyping and waterfall which he called the Spiral Model. It has been taken up by the Software Productivity Consortium which has produced a complete process model based on this lifecycle. In a recent experiment the spiral model surprisingly produced a larger and more effort-consuming product than both prototyping and the waterfall model. Thus though the spiral model is being touted as the next best lifecycle model, it is actually more expensive than merely following the waterfall life-cycle. It really was an additive combination of waterfall and prototyping.

For software process to work, a new approach to process is needed which attempts to approximate the effort/size trade-off of prototyping with added rigor. A new lifecycle model and process model combination is needed to make this happen. The source for this model might be the work done in biology and other subjects on autopoietic systems. Here is a brief sketch of what a software process based on autopoiesis would look like in the spirit of exploring a possible future of process work in general. Autopoiesis means “self-producing” and is generally referred to in terms of self-organization. What we really want in software process is for the software developers to organize themselves so that they produce their products very efficiently and to high standards of quality. Right now the organization that developers left to their own devices imposed on themselves is ad hoc and chaotic. This is an organization which we impose on ourselves, and we could replace it with a different organization. Part of the reason that we do not replace it with a different organization is that all our companies are control oriented so that the practitioners are managed by others who decide the general order and content of the processes that the practitioners will follow in developing products. The problem with this approach is what has been called by R. Ashby “the law of requisite variety” which states that a control channel must be just as complex as the thing it is controlling. Thus, in a control-oriented company a lot of effort is expended to report changes to management and to channel decisions back down to those actually doing the work. This is in fact very inefficient because a lot of information is lost in the transfer to and from the center of control. The other part of the reason is that human beings just naturally produce variety so that each practitioner has his own way of doing things which he defends against all comers. Left to his own devices each software practitioner would do things in his own way. The combination of control orientation and the production of endless variety among practitioners ends up creating a chaotic situation.

Control is always an illusion because of the narrow bandwidth of control channels which cause control to be very gross and prevents refined control past a certain threshold of complexity. There are many pockets in which control cannot be exercised where practitioners are allowed and even encouraged to do things their own way just so they get the job done and meet the milestones that management cares so much about.

At this point in our software maturity most organizations
are imposing a disorganization on themselves which appears to give freedom but is recognized as wasteful if seen in its full implications. Prototyping is merely an excuse for giving the practitioner a means of organizing himself to efficiently produce a product with the minimum interference. The waterfall lifecycle attempts to organize the essential transformations which go on in the practitioner’s head in a way that multiple people can work together toward the production of the necessary products. Thus, the waterfall transformations multiple intermediary products are produced which can be shared between practitioners working together. In prototyping the larger system would be broken up into pieces that individuals would work on together through controlled interfaces. So prototyping may be seen as a type of cooperation in which the work is broken up into interacting products, but that all stages of the essential transformations remain in a single person’s head for a piece. The waterfall model assumes that multiple people have to share intermediary products for the whole system at each essential transformation of the entire system. These two different approaches are antithetical ways of dividing up the work either as separate interacting longitudinal channels (prototyping), or crosswise with the whole group interacting on each stage in the development of the system.

The Spiral model combines these by doing several iterations of prototyping up front, and then in the end doing the waterfall of essential transformations in the final iteration. Thus it is clear why the Spiral model is an additive combination of prototyping and the waterfall. This combination probably produces a better quality product as all the requirements are surfaced by the prototyping stage up front, yet still the internal knowledge embedded in the product is made visible at the last stage of production. However, just doing longitudinal and then crosswise work organization in two different phases of development does not solve the essential problem of software process. The longitudinal work deployment allows the individual practitioner to generate his own way of doing things but gives control through multiple iterations and the addition of risk management techniques. The waterfall model makes the practitioners settle upon a single way of doing things as imposed by the control channel, but it means that all knowledge must be externalized for sharing in a common format. Externalization of intermediary knowledge is inefficient so that the waterfall is more costly.

Now, let us take a completely different look at the problem using the language of autopoiesis instead of control. Autopoiesis means self-organization. The term was invented by F. G. Varela and H. R. Maturana, Chilean Biologists about 1974. Since then it has caused a small revolution in our understanding of the basic assumptions underlying Systems Theory. It offers a new perspective on the way systems are organized which may help us to understand how to produce efficient processes completely different from the way we are trying to produce them today. In effect, our process work today does not question the control orientation of current organizations. Autopoiesis allows us to question this basic assumption and provides us with an alternative model. So the goal is to have the practitioners organize themselves to do the work in the most efficient and quality-instilling way possible that minimizes unnecessary variety but keeps all essential variety. It must work either in a single practitioner’s head or for a group of practitioners working together. It must be a process and lifecycle model that approximates the efficiency of prototyping while adding extra rigor. It must be practically applicable to a wide range of development activities. To accomplish this we will say that software development is not a chain of essential transformational processes, like the waterfall and its variations suggest, but instead is an autopoietic ring. The essential processes in this ring are REQUIREMENTS, DESIGN, IMPLEMENTATION, INTEGRATION, & TEST. Each process in the ring feeds forward and backward to each of the other processes in the

3. Reference?
ring. The ring can be done by a single practitioner where he is organizing his own activities according to these essential processes or by a team of practitioners. As a team each practitioner may be enacting the whole ring or just a subset of the processes in the ring. The way the ring is assigned to the team, members can approximate a waterfall or prototyping divisions of work. The essential point is that the ring is a consistent set of interlocking processes which are a self-organizing unity. This means that there are certain information flows that move around the ring continuously that are independent of interaction with the outside world. There is a different set of information flows in which each of these processes interact with the outside world. It is the interaction of these two kinds of information flows that needs to be considered very carefully in the construction of a software process model.

In this schema there are four flows of inputs and outputs for each node in the ring. First there is the normal flow which is represented by the waterfall in which requirements leads to design which leads to implementation then integration and test back to requirements. Second there is the reverse flow around the ring in which information is feed back to the prior stages in the waterfall. Third there is the cross ring flow in which each node has inputs from the outside and produces products that are given back outside again. In addition there are also the flows that go from each node to support, control, or organizational processes. This fourth type of flow is depicted in the basic model of the self-organizing ring as the flow to the center of the ring where support processes exits. An important point is that each node in the ring is really identified by the interference pattern set up by the four flows. Each node is a confluence of these flows that sets up chaotic interferences between the information flowing in each direction. One of the reasons that it is so hard to model the software process is that each essential transformation is really effected by all four flows so that the transformation itself is constantly changing its nature depending on the situation as presented from all four flows. This donut is really lopsided from the point of view of routine verses non-routine work. The implementation node is the high point of routineness, while as one moves away from that in both directions, it is toward more and more non-routine kinds of work. A single software engineer will bounce around the ring in his head as he goes through the development process. Variety is created because the transformations are themselves constantly transforming under the multiple changes in all four information flows feeding any particular node in the ring. Thus, when we talk about software process, we are really talking about nodes of interference between flows of information. The interference patterns can be very complex. So the variety in the products by different software engineers are a response to the complexity of these interference patterns and in the meta-transformation of the essential transformations themselves. If we visualize the ring as a set of interference patterns between flows of information, it is easier to see why things are difficult to control and why variety is very high.

Above we have shown the autopoietic software process as a ring with feedback and feed forward, which makes it a torus, and as a knot where each node in the ring is an interference pattern between four different streams of information. We can also visualize the ring to be paradoxical like a Mobius strip with global/local differences in structure. Like a Mobius strip that locally looks two sided but globally is realized to be one sided, the software process ring is globally continuous but locally discrete. This means that from a global perspective all the different flows form a single overall process while it locally breaks down into different kinds of interrelated work. It is particularly this problem that appears when multiple people attempt to work on the same thing at the same time. Either the people take different parts of the problem and do the whole ring in parallel, or they attempt to work on a section of the ring together. Both of these ways of dividing up the ring are inefficient. One is inefficient because you are attempting to write down and share all the information flowing through the ring. The other is inefficient because there is no global perspective but merely completely different pieces held together by tight interfaces. In effect, what is needed is a way to stack the rings in a meta-ring.

The meta-ring produces an open system composed of a set of closed systems where a given essential transformation, such as requirements, takes requirements mediated by the whole meta-ring and produces queries for requirements source outside the ring. Or again the design node, for instance, takes technology constraints from the source outside the ring and produces a design consumed by the whole ring. And so on with each of the essential transformations. In such a process the actual work could be divided in either direction. However, wherever the interface is chosen, it must be formalized. Thus, if a different person does all of the requirements nodes in the meta-ring, then an interface needs to be set up with the different designers. Or if several people are doing the sub-rings in parallel, then interfaces need to be set up between

---

**Wild Software Meta-systems**
THE FUTURE OF SOFTWARE PROCESS

**Company PROCESS EVOLUTION**

**Original Process**
- From another division
- Environment centered
- Organizationally Specific
- Lifecycle centered
- Designed to comply with DOD-2167A

**Division ASPM**
- Compromise between lifecycle and process views.
- Process ‘What’ embodied in waterfall.
- Process ‘How’ described separately.
- Outlined only; not finished

**Software Engineering Institute**
- Radice’s ETVX for defining process.
- Capability Maturity Model
- Assessment
- Process emphasis.

**Spiral Lifecycle**
- Developed by Barry Boehm.
- Current Standard for all DOD software.
- Tailorable.
- Setting new directions for standards.
- Developed into full model by SPC

**Division Process**
- Completely Lifecycle independent
- Unified Process Framework; single hierarchy.
- Similar to process work of industry leaders.
- Prepared for future developments

**2167A**
- Example lifecycle built in.
- Current Standard for all DOD software.
- Tailorable.

**IEEE P1074**
- Commercial standard
- Not DOD standard compliant.
- Process separated from Lifecycle.
- Dataflow type representation.

**ISO STANDARD**
- Written by author of 2167A.
- Based on P1074
- Raises process to higher level including operations and acquisition.

**2167B**
- Lifecycle Independent.
- Tuned to allow spiral lifecycle.

**FIGURE 4**
FRAMEWORK

- Gives context to building blocks.
- Shifts to accommodate different focuses.
- Allows different views to be integrated in same structure.
- Has organizational implications.
- Data is collections of artifacts.

BUILDING BLOCK LEVEL

- Fundamental kinds of work.
- For most part unaffected by context.
- Starting point for process definition.

SUB-PROCESSES

- Very concrete jobs with specifiable low level data inputs and outputs.
- Taken as low as necessary to define work completely.

FIGURE 5
Structure of IPD Process

- Interfaces at every level to connect with the rest of engineering.
- Procedures & Requirements decomposed too.

**WHAT**
- Dataflow
- Description
- Entry products
- Tasks (sub-processes)
- Validation
- Exit products
- Guidelines

**HOW**
- Tutorial
- Policy
- Requirements
- Compliance Audit
- Procedure
- Methodologies & Tools
- Metrics
- Notes

**PROCESS Description**

**FIGURE 6**
each practitioner doing the same essential transformation. Because of the global/local paradoxicality, the meta-rings with its subrings are from one viewpoint a single continuous system while from another it is a diverse set of interference patterns in information flows. Working out interfaces between interference patterns is very difficult.

Another completely different way of viewing the rings is as a minimal system\(^4\). Each minimal system must have at least four independent elements and form a tetrahedron of interrelations. Such a system is really a combinatorial set of all the possible interrelations of facets. As we saw, each node has four different sources of input for information. Thus, the points of the tetrahedron are these different sources. These sources are synthesized into a coherent response at each node so all the combinations of the four must be considered.

In a faceted system all the possible combinations need to be looked at. For example, feedforward needs to be pairwise compared with all three other sources of information (ab, ac, ad). Then all the other sources need to be looked at together without feedforward (a) as in bcd. Finally, all the sources need to be looked at together (abcd). The result of all these faceted looks at the information is the correctly synthesized response. That response is the adapted essential transformation which deals with the particular interference pattern located at that node in the ring. Since each node is producing a synthesis perhaps concurrently or under recursive or iterative

\[\text{FACETED RESOLUTION OF INTERFERENCE}\]

\[\begin{align*}
\text{null} & \quad \text{feedforward} \quad \text{feedbackward} \quad \text{external I/O} \quad \text{internal I/O} \\
\text{ab} & \quad \text{ac} \quad \text{abc} \\
\text{ac} & \quad \text{ad} \quad \text{bcd} \\
\text{ad} & \quad \text{bc} \quad \text{cda} \\
\text{bc} & \quad \text{bd} \quad \text{dab} \\
\text{bd} & \quad \text{dc} \\
\text{abc} & \quad \text{bcd} \\
\text{bcd} & \quad \text{cda} \\
\text{cda} & \quad \text{dab} \\
\text{dab} & \quad \text{abcd} \\
\end{align*}\]

\[\text{FIGURE 9}\]

\[\text{Transformed essential transformation}\]

\[\text{Synthesis}\]
refinement, the entire matrix can have very complex patterns of transformational adaptation and transformational operation. Not only is every node faceted, but the whole ring is faceted.

Thus, for each node in the ring there are several levels of syntheses which will allow the whole ring to achieve unity as a self-organizing system. This synthetic root is the source from which all individual software engineering work is coordinated. Each position in the ring is seen as a gap from the point of view of the rest of the ring (e.g. bcde). In order to fill in that gap it is necessary to confront the interference of autonomous intra-ring information with the external information. That is done by seeing what the pairwise relations between that node and the other nodes in the ring are at any given time. Then the relation between that pairwise relation and the rest of the ring is considered. Thus, the faceted process results in a series of complementary twins which relate the given node with the set of the rest of the nodes or the pairwise relation of a

---

### RESOLUTION OF KINDS OF WORK IN RING

<table>
<thead>
<tr>
<th>Essential transformations</th>
<th>transform pairs</th>
<th>the context of transform pairs</th>
<th>UNITY of transforms</th>
</tr>
</thead>
<tbody>
<tr>
<td>Local discreteness</td>
<td>ab</td>
<td>abc</td>
<td>Global continuity</td>
</tr>
<tr>
<td></td>
<td>ac</td>
<td>abd</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ad</td>
<td>ace</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ae</td>
<td>ade</td>
<td></td>
</tr>
<tr>
<td></td>
<td>bc</td>
<td>bce</td>
<td>abcde Synthesis</td>
</tr>
<tr>
<td></td>
<td>bd</td>
<td>bcd</td>
<td></td>
</tr>
<tr>
<td></td>
<td>bc</td>
<td>bce</td>
<td></td>
</tr>
<tr>
<td></td>
<td>bd</td>
<td>bcd</td>
<td></td>
</tr>
<tr>
<td></td>
<td>be</td>
<td>bde</td>
<td></td>
</tr>
<tr>
<td></td>
<td>cd</td>
<td>ade</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ce</td>
<td>abc</td>
<td></td>
</tr>
<tr>
<td></td>
<td>de</td>
<td>abe</td>
<td></td>
</tr>
</tbody>
</table>
The same autopoietic ring is grounding itself and the process grounds the process of developing the product. This means that the process of developing there are multiple people, they are all engaged in the same own process developer as well as a product developer. If concurrently. If there is only one person, he is both his people working both the process and the product traversed by the same essential processes applied to the same process of software development itself. It is the same autopoietic unity.

Nothing can be more efficient than this in terms of process enactment. Setting up autopoietic unities that define their own operation as part of their operation is the task of software management. Thus, management must relinquish control and assume the role of catalyst within large organizations. The point is that every individual or team developing software already designs its own process every time. The trick here is to support this activity with guidelines and templates for generic processes, but to stand back and allow necessary variety. Taking the necessary variety out of the process will cause just as many inefficiencies as the ad hoc processes that have a lot of unnecessary variety in them. Finding a middle ground between these two extremes of not enough variety and too much variety is a very difficult task. This task can only be done by the practitioners themselves as a team effort. This application of the essential processes of software engineering to the process before the product is an overhead of operation that will save money in the long run by eliminating inefficiencies through the reduction of unnecessary variety and the enhancement of necessary variety.

THE IMPACTS OF THE ASSESSMENT OF SOFTWARE PROCESS

Assessment of currently used actual processes on software projects is a completely different problem than defining ideal processes. In the assessment process one is applying the SEI Capability Maturity Model (CMM) and the associated questionnaire. These are now in draft form, and the 87 questionnaire is still the official instrument. However, we decided to use the CMM and its level two questionnaire as the basis of our own internal assessments. We developed a method which is like an audit in which physical evidence for each ‘yes’ answer is submitted by the project and assessed by the assessment team. The auditors reviewed folders containing supporting materials of each yes answer and marked them compliant or non-compliant. If a project gets all its answers accepted as compliant, then a deeper and certification process is applied. The questionnaire for level two only covers about 38% of the bold points in the CMM. So projects must be certified for each Key Process Area directly. This gives a much more thorough view of the actual compliance status than a normal review that only questions exceptions.

We know that software engineering processes are self-organizing because the same essential processes that are applied to producing the software processes can also be applied to defining the software process itself. Thus, the ring of essential processes are recursively entered for process definition. However, we want the practitioners to organize their own processes and thus become truly self-organizing instead of having that process legislated and applied from the outside. The self-organized process may be much more effective and complex than any process applied from the outside. This means that the Mobius strip of product directed essential processes is orthogonally traversed by the same essential processes applied to the process of software development itself. It is the same people working both the process and the product concurrently. If there is only one person, he is both his own process developer as well as a product developer. If there are multiple people, they are all engaged in the same pair of tasks. This means that the process of developing the process grounds the process of developing the product. The same autopoietic ring is grounding itself and organizing itself.

Faceted analysis $\{2^N\}$ is the opposite of relationship analysis. The Lano $N^2$ chart is the technique of choice for analyzing all possible relationships between elements. This, though, does not allow the interpenetration of the elements to be analyzed. This interpenetration and synthesis of sub-components is a forgotten aspect of analysis which is concerned mostly with how things fit together systems that are assumed to be distinguishable and stable. In Autopoietic systems the inverse becomes an issue because of the embedded nature of human work and the fact that the autopoietic ring is a closed unity or realized synthesis.

Faceted analysis $\{2^N\}$ is the opposite of relationship analysis. The Lano $N^2$ chart is the technique of choice for analyzing all possible relationships between elements. This, though, does not allow the interpenetration of the elements to be analyzed. This interpenetration and synthesis of sub-components is a forgotten aspect of analysis which is concerned mostly with how things fit together systems that are assumed to be distinguishable and stable. In Autopoietic systems the inverse becomes an issue because of the embedded nature of human work and the fact that the autopoietic ring is a closed unity or realized synthesis.

We know that software engineering processes are self-organizing because the same essential processes that are applied to producing the software processes can also be applied to defining the software process itself. Thus, the ring of essential processes are recursively entered for process definition. However, we want the practitioners to organize their own processes and thus become truly self-organizing instead of having that process legislated and applied from the outside. The self-organized process may be much more effective and complex than any process applied from the outside. This means that the Mobius strip of product directed essential processes is orthogonally traversed by the same essential processes applied to the process of software development itself. It is the same people working both the process and the product concurrently. If there is only one person, he is both his own process developer as well as a product developer. If there are multiple people, they are all engaged in the same pair of tasks. This means that the process of developing the process grounds the process of developing the product. The same autopoietic ring is grounding itself and organizing itself.

Faceted analysis $\{2^N\}$ is the opposite of relationship analysis. The Lano $N^2$ chart is the technique of choice for analyzing all possible relationships between elements. This, though, does not allow the interpenetration of the elements to be analyzed. This interpenetration and synthesis of sub-components is a forgotten aspect of analysis which is concerned mostly with how things fit together systems that are assumed to be distinguishable and stable. In Autopoietic systems the inverse becomes an issue because of the embedded nature of human work and the fact that the autopoietic ring is a closed unity or realized synthesis.

We know that software engineering processes are self-organizing because the same essential processes that are applied to producing the software processes can also be applied to defining the software process itself. Thus, the ring of essential processes are recursively entered for process definition. However, we want the practitioners to organize their own processes and thus become truly self-organizing instead of having that process legislated and applied from the outside. The self-organized process may be much more effective and complex than any process applied from the outside. This means that the Mobius strip of product directed essential processes is orthogonally traversed by the same essential processes applied to the process of software development itself. It is the same people working both the process and the product concurrently. If there is only one person, he is both his own process developer as well as a product developer. If there are multiple people, they are all engaged in the same pair of tasks. This means that the process of developing the process grounds the process of developing the product. The same autopoietic ring is grounding itself and organizing itself.

Faceted analysis $\{2^N\}$ is the opposite of relationship analysis. The Lano $N^2$ chart is the technique of choice for analyzing all possible relationships between elements. This, though, does not allow the interpenetration of the elements to be analyzed. This interpenetration and synthesis of sub-components is a forgotten aspect of analysis which is concerned mostly with how things fit together systems that are assumed to be distinguishable and stable. In Autopoietic systems the inverse becomes an issue because of the embedded nature of human work and the fact that the autopoietic ring is a closed unity or realized synthesis.
software process.

First, the CMM is very thorough and complete and presents a daunting picture for any organization that contemplates compliance. For us the CMM is treated as a requirements document for our process along with DOD-STD-2167A and DOD-STD-2168. Thus, every point within the CMM (goal, commitment, ability, activity, monitor, verification) is a requirement which is embodied in our process somewhere along with every “shall” from 2167A & 2168. This makes numerous requirements that our process has to fulfill along with actually describing the best practice for doing the work involved. Many people make the mistake of treating these requirements documents as the major driver for the formation of their process. This is a mistake because most of the work actually needed to produce the product is ignored by these standards. Any process must first capture the work that needs to be done to produce the products and then backfit these with the requirements from such sources as the CMM.

If full compliance is ever to be achieved, the CMM needs to become a working operations manual for the projects. When it is applied to the actual workings of a project, a good picture of actual practices is obtained. But what the CMM really requires is that one apply the questionnaire, and if it is satisfied, one goes on to apply the CMM itself to assess full compliance. Thus, the CMM becomes the working criteria for how these aspects of a software project are run. It appears to be a very good model which I am sure will improve over time. However, there are some problems that present themselves that need to be addressed:

1) There is an implied organization in the CMM which is at odds with a team-centered organization such as ours.

2) There is no IPD focus in the CMM so that software functions are isolated. Thus, requirements are levied on software that are not recognized by the rest of the engineering organization. This leads to overemphasis on software to the neglect of other functions.

3) There is an emphasis on documentation and verification in the CMM which may be beyond reason. It is certainly unreasonable to levee such requirements on the software and on no other part of the organization.

4) The fact that there is no process model explicitly called out in the CMM makes compliance more difficult even though it is understandable why SEI attempts to avoid specifying process models. It would help to see a worked example of a process model that embodies the CMM requirements.

Assessment using the CMM and its associated questionnaire makes it clear why a defined process like the one described above is necessary. One clearly sees that many of the problems associated with a project’s compliance could be resolved by having a process description and following it. But the question arises as to how easy is it to enact a defined process. Software is the first industry involving non-routine work to have process descriptions mandated for it. So it quickly becomes relevant to determine whether such a definition and enacting of written processes is appropriate. And here we begin to look at another set of central issues to be addressed in this essay. Can written process descriptions ever be effectively applied to software engineering work, or to engineering work in general? Is this the correct way for us to go about solving our efficiency problems? What are the implications of this for sociotechnical systems in the future?

Assessing one’s organization using the CMM as a basis gives tremendous insight into the way one does business. It makes it clear what the benefits of defining processes are in terms of meeting the goals set by the CMM. But it also raises questions as to the direction all these efforts and initiatives are leading which need to be considered from a global perspective. The rest of this essay will approach these issues.

SOCIOTECHNICAL SYSTEMS, PROCESS AND NON-Routine WORK

The basic question is whether any person who engages in non-routine work can define their own processes and follow those processes to improve their work. This may seem like a trivial point. But there seems to be no research on this point, and the entire edifice of process definition rests on the ability of a person to follow the prescriptive processes defined by others. Yet if a person who is doing non-routine work cannot describe his own personal processes and then follow them to improve what he is doing, then the entire edifice of process definition and enactment collapses.

In order to approach this question it will be necessary to start from square one and trace the development of process
as a way of viewing work in order to anticipate where it is going in the future. The only real process model we have that is published as a standard is IEEE P1074. It identifies software processes as independent of lifecycle. This is the first point that must be appreciated. Process is independent of time. Process is a kind of work that can happen anytime in the lifecycle. What lifecycle you follow is irrelevant. The difference between process and lifecycle follows the analogy of the “set” and “list”. Sets are unordered and can contain only one example of any one thing. Lists are ordered and can contain multiple instances of a certain kind of thing. Thus, process models are more basic than lifecycle models. They are partially ordered by dataflow relations and process constraints, but this partial ordering may be violated as processes may begin without all of their inputs or occur in unexpected points in the lifecycle. Processes allow us to see what work is actually going on between project lifecycle milestones. It allows us to see kinds of work that should not normally occur in a particular portion of the lifecycle when they do actually occur. Processes allow us to portray non-routine work as the iterative and recursive entering of processes which are interweaved and enmeshed in the course of actually performing the development of software products. Processes need to be clearly distinguished from a practitioner’s point of view, so it is obvious to him which kind of work he is doing at any one time.

The whole concept of process actually comes from chemical industry plants where statistical quality control methods were applied using sampling. In such a plant there is a physical envelope which liquid ingredients flow down as they are being mixed and transformed. These techniques were taken by Demming to Japan and applied to production lines which have similar well defined boundaries and channels of flow. Now Humphrey and others are attempting to take these same control techniques and apply them to the non-routine work of software engineering. Software development is actually a mixture of routine and highly non-routine work. It is non-routine toward the ends of the lifecycle and routine in the middle of the lifecycle. Each software process has its own mix of routine and non-routine components. Configuration management is mostly routine. Design is highly non-routine. Documentation is about an even mixture. However, never before has non-routine work been submitted to this sort of discipline by attempting to apply statistical methods suited to production lines. The problem is that there is no natural physical channels of flow. The process definition is supposed to create ideal channels which will allow measurements. But ideal channels are too malleable, and the changes that defeat measurement are constantly happening.

So the problem becomes whether non-routine work can be treated in a manner similar to routine work from the point of view of process control using idealized or imaginary process containment envelopes which are merely written down on paper. This is not an easy question to answer. The work of Walt Scaachi from USC attempts to deal with this problem. “The Web of Computing” article he wrote with Robert Kling of UCI attempted to identify the total environment of computer usage. In that article they borrowed two terms from other researchers which were used to describe “what really goes on.” The term “line of work” is used to describe what a person actually does in his job as separate from any job description he might have. The term “going concern” is used to describe the aggregate of the lines of work. These two terms are very valuable as baselines for process description activities. Process definitions are prescriptive or idealized descriptions of what should go on as the norm of rationalized work. But there is always some variance based on individual behavioral patterns and group dynamics from that ideal. The difference between actual and ideal work process needs to be considered very seriously. The only way that process definition can be of any help is if the people actually doing the work can be convinced to enact processes. This means that they must understand and follow the processes as described within some limits of acceptability. In Walt Scaachi’s work process breakdown is used as a way of understanding the relation between the ideal process and the actual work as carried out by individuals. In this model the process is used to plan activities. The activities are attempted to be carried out. When they encounter obstacles, then these breakdowns are noted and replanning occurs. At this point perhaps the prescribed process is changed, re-instantiated, or re-tailored. Breakdowns are one way of conceiving of the crucial connection between ideal process models and what actually occurs.

Yet is the breakdown model enough? The break down model allows us to adapt the enactment of the process to exigencies of the situation which are unexpected. They assume that our planning is good enough to allow us to carry on a certain distance in the process enactment and
Structure of Process Document

**VOLUME 0**
**ORIENTATION**
Introduction to Process
Overview of Process Architecture.
Guidelines for Process Description.
Guidelines for Process Enaction.
Structure of Process Documents.

**VOLUME 1**
**WHAT**
Dataflow Description
Entry products
Tasks (sub-processes)
Validation
Exit products
Guidelines

**VOLUME 2**
**HOW**
Tutorial
Policy
Requirements
Compliance Audit
Procedure
Methodologies & Tools
Metrics
Notes

**VOLUME 3**
**INSTANTIATION & TAILORING**
Tailor to Standards
Prune Process Trees
Pick Lifecycle
Map Lifecycle to Process
Instantiate Processes
Note: All lifecycle issues dealt with here. Evolutionary Spiral from SPC used as example.

*FIGURE 7*

o A single structure unites two volumes
that when we encounter breakdowns of our plans, then we will replan and remap our processes to the tasks at hand. This is fine for making process models of routine work adaptable. The question arises as to whether non-routine work is similar to continual breakdown. All the models assume a continuity of a specific process for some amount of time. There is, though, a threshold at which this assumption of continuity itself breaks down. In non-routine work there may be an actual fusion of different types of work in which the hopping from one to the other is so quick that there is no actual dwelling in a particular process. The ultimate, is when one is actually performing different kinds of work in one’s head, moving from one to the other is merely the movement between different links in the chains of ideas. This fusion of processes causes boundaries to disappear between types of work and planning to become continuous adaptive behavior which never ceases.

If non-routine work is actually like a continual breakdown state, then there is no way to raise it from the initial level of process maturity. We might characterize this positively as the spontaneous dealing with problems and issues on the fly without time taken to actually work the different issues which inform the problem at hand. A lot of work done by experts and executives is of this type. It is not done in a coherent rational way, but instead is done spontaneously by intuition of the moment or by applying a set of heuristic rules learned by experience. It is based on skill and experience, not on any routinized set of activities. The expert or executive is self-organizing in the sense that what is needed at the moment is produced to the requirements of the present situation. If the circumstances allow, the expert or executive redefines the situation in order to give the right answer for the fulfillment of the perceived need of the moment. Thus, the expert or the executive has the power in many instances to completely redefine the work to be done in accordance with the means of accomplishing that work and prior formulations of that work. Thus, the positive side of continual breakdown is the possibility of creative redefinition of the whole situation in a way completely different from the work’s original formulation. By such a leap the person engaged in non-routine work can change the rules and assumptions that initially defined the work in order to transform what needs to be done. This creative side of working is not covered by any current models of process. However, potentially it is through innovation, creativity, and other forms of non-definable behavior that the greatest gains are to be made in the solution of our problems.

Any definition of process that does not take into account the action of creativity and innovation is flawed. The breakdown metaphor covers it by implication but not explicitly. In fact, what we wish could happen is that all aspects of our process could be the focus of the creativity of our engineers. We would hope that the process is really what Thomas Kuhn called “normal science”8 between the paradigm changes which redefine the whole field of software work. In normal science we work out the implications of a way of seeing things. But paradigm changes precipitate changes to the whole way of defining the work to be done. The point is that some people in the organization are continuously redefining the work to be done. These experts and executives are involved in highly non-routine work which cannot be captured by any process model. This creative center of instability is the epicenter of the process. As we back away from it, we find that there are degrees to which every worker can redefine his work. The problem becomes getting others to buy into his redefinition. This process of negotiation causes the whole field of work to constantly change. But it does not change a piece at a time. Instead, the whole pattern of work and its actual fabric is continually being re-patterned and renewed. Work is a gestalt which is constantly changing in spurts with many starts and stops.

Now this means that simultaneously the ideal patterning of the work is changing along with the “lines of work” within the “going concern.” The work environment can handle this because of the essential freedom of the individuals to redefine their work and respond in real time to the redefinitions of work by others. This aspect of non-routine work can never be captured by process. In fact, process will always be trying to keep up to these changes. The minute you write down a process it is obsolete. To the extent people are not engaged in the creative transformation of their work is exactly the extent that process definitions hold long enough to be applied. Thus, process is in constant tension with the creativity of the individuals who are constructing their world and the work within their world spontaneously as they go. From this view everyone is improvising all the time. The making up of the script as it is being performed has always been recognized as a very important aspect of the preparation of the acting profession. In process enactment it is no different. The enactors are defining the process as they go along. Sometimes they can rely on prepared scripts, and sometimes they must rewrite the script on the spur of the moment or even ad lib. In this sense they are self-

8. See Structure of Scientific Revolutions
organizing, autopoietic, systems. They are producing themselves, projecting what they should be and realizing it simultaneously.

Now there is obviously a spectrum from the extreme non-routine work of self-definition to more mundane types of non-routine work in which known and stable processes are continuously being re-entered iteratively and recursively based on the judgment of the expert practitioner or on interrupts from external sources. However, on the whole people know that they do not want to write work proscriptions down because they are just spending extra time producing something that is obsolete as soon as it is produced. Whatever is captured must be both very important and stable for that to be worthwhile. We are trapped by the fact that writing things down is the only way we have to capture information. What we really need is some intermediary form between action and writing which allows us to record our processes without formulating them into sentences because writing things down is an essentially different kind of activity from doing the action. It is only if the representation actually helps one do the work that it is worth this trouble. One half-way house that exists between writing and action is graphics. If there were some way of diagramming process enactments that record the essence of the enactment without having to write descriptions, then these would be useful. However, they would have to actually help the practitioner improve his process performance. Another half-way house that exists between writing and action is brief notes. If there was a way to describe process which people can keep in note form which is helpful in the enactment of processes, then this may be a way to enable process enactment that makes a bridge between lines of work and proscriptive processes.

Taking the second line of approach to finding a half-way house, I decided that I should perform an experiment in which I attempted to see if a person can record his own process and then follow it. I looked at the planners that are carried by people to help them organize their time, and asked if there was any similar system that could be used for people to apply process to their own work. In this vane I came up with the idea of personal process as the intermediary stage between ideal proscriptive processes and lines of work. The personal process is a way for each individual to describe the kinds of work he engages in each day and attempts to improve that work. The person is spurred to continuously ask himself if this work he is doing can be redefined or reorganized so as to become simpler or eliminated altogether. If the work cannot be redefined, then one is asked to define each kind of work one does and manage the kinds of work as a complete pattern. If the kinds of work are highly non-routine, then the iterative, recursive, or interrupt driven pattern of process enactment is described. If the work is routine, then the sequential pattern of work is defined instead. In either case the personal process can set aside specific time for considering redefinition of work, defining work processes, managing the pattern of personal work processes, or managing work assignments.

PERSONAL WORK PROCESS MANAGEMENT

My experiment with personal process management was conceived to see if process description and enactment is practical. I entertained grave doubts as to the efficacy of process work within organizations. But I hypothesized that if personal process management can be made as useful as time management, then it could actually have some impact within real work settings. Personal process management exists at an intermediate level between the proscriptive work process and the individual’s line of work. It is owned completely by the individual, and is a means of controlling and rendering his non-routine work more efficient. It is modeled after the time planning books that are used almost universally to manage coordination and action assignments by individuals. Personal Process Management is an extension of this system which attempts to give the individual control over their entire work pattern.

The steps of personal process management are fairly simple.

1) Describe the kinds of work you do on the process description sheet. This sheet has areas for notes about inputs, outputs, steps, issues, notes, problems, breakdowns, and the normal phases of any process.

2) Describe the pattern of work processes and how these interrelate, including the routine or non-routine nature of these interactions between processes on the control sheet.

3) Describe the work assignments and action items associated with those work assignments as they appear and relate them to the processes.

4) Consider how the work assignments could be redefined in order to increase efficiency. Do not accept a work assignment without considering how it could be redefined.
5) Consider how action items relate to processes, and again test each action item for the possibility of redefinition.

6) Negotiate redefined action items and work assignments with their organizational source in order to make sure that there is common understanding of the new work to be done.

7) Consider the work source interface carefully. If the work is coming from the work source in some form which is difficult to act upon, then attempt to clarify the inputs, or get redefinition of those inputs.

8) Consider the whole pattern of work to see if it can be streamlined and made more efficient.

9) Consider each personal process within the whole pattern of work, and attempt to make each of these efficient.

10) Attempt to apply personal measures to the work assignments flowing through the processes in order to have a basis for improvement of those assignments.

11) Consider interfaces with the personal processes of others. Model the processes of others that you depend on, and attempt to get them to make changes that will increase overall efficiency of the team.

12) Consider the whole work process within its organizational and prescriptive process setting. Attempt to change the organizational structure and prescriptive process in order to improve the overall efficiency.

Now what these steps show is that work redefinition is the center of personal process. Prescriptive processes must be written in such a way as to allow tremendous variability but still supply standard non-changing definitions everyone can use for a prolonged period of time. Yet personal process represents the “in the trenches” tailoring of prescriptive processes based on the exigencies of the situation. When you start recording your personal process, it is clear that inputs are mostly implicit when any work assignment is given. Those that are explicit are normally things you have to construct yourself or find. The number of explicit handed-over inputs are usually very few, for non-routine work outputs must be made up. They will probably be different for each work assignment, and the steps done will change on a case by case basis. Yet, by thinking through the processes one is doing, it makes them clear and allows one to sort though the recurring problems. This exercise is valuable in itself and is actually what will improve the process overall. We need to think about the processes we perform instead of blindly executing them. In American business culture there is a lack of emphasis on thinking about what we are doing. When it is mentioned, it comes under the rubric of working smart. We need to motivate individuals to think through what they are doing, and personal process analysis facilitates this. It lets the individual step back from what he is doing and get a global perspective based on a record of actual instances focused on the kinds of work he is doing now. The kinds of work people do who engage in non-routine work do will vary widely from month to month. They are called upon to do the hard jobs which demand creativity and intensity of execution. But for a given period the work to be done will fall into a pattern which is amenable to personal process analysis.

The analysis is performed as an ongoing reflection on the kinds of work being engaged in, not as a one-time snapshot. It allows the person to say to him/herself:

- What are the main kinds of work I am engaged in these days?
- How are these related to each other in an overall pattern?
- How could I redefine this work to simplify it and better integrate it with the work of others?
- What are the issues arising in my work assignments for a particular kind of work?
- What are the problems I am having to deal with over and over?
- What are the points where my work assignment plans are breaking down?

The elements of personal process analysis form an integrated system of elements which needs to itself be dealt with in a systematic way.

- Work assignments have tasks.
- Work assignments have priorities.
- Work assignments have resources.
- Work assignments have a due date.
- Work assignments have specified outputs.
- Tasks have action items.
- Tasks embody personal processes.
• Personal processes are the basic pattern of work expected by a series of similar work assignments.
• Work assignments may be self-assignments or externally elicited.
• Personal processes are kinds of work distinguished by the individual.
• Personal processes have inputs which may be explicit, implicit, given or to be found.
• Processes have steps by which the work is normally structured.
• There should be some isomorphism between action items and steps at some level of process definition.
• Working action items through process steps produces issues that need to be resolved.
• Resolution of issues may take deliberation with others.
• Any issue that cannot be resolved within the scope of resources allotted is a risk that must be channeled to the person responsible for risk assessment, avoidance, and abatement.
• Processes when executing a specific work assignment by executing its process steps by action items may encounter problems generic to all similar work assignments.
• Problems when identified must be worked separately from the action items associated with the problem.
• Problems may demand expert assistance.
• Problems are knots of immediately unresolvable issues.
• These knots of issues may generate risks if they go unresolved too long.
• Problems may be wicked in which case there is no optimal solution, wicked solutions require a trade off.
• If enough wicked problems occur, there is a breakdown of the process.
• Breakdowns call for replanning and perhaps redefinition of the work to be done.
• The generic stages of process execution are as follows.
• Formalism selection and validation.
• Exploration based on using the formalism on given content.
• Elaboration once formalism is seen to work.
• Assessment which calls for re-evaluation of issues, risks, & problems.
• Verification that results are actually still true and that changes elsewhere in the system have not rendered them obsolete.
• Inference considers the effects of my results on other work assignments.
• Consistency checking of results.
• Evaluation of results in light of requirements for outputs.
• Deliberation on issues with others.
• Validation by application of available knowledge and criteria to the results of the process.
• Invocation of other processes to deal with outputs.
• Context switches from one process to another cause lags that require time to reacquire the work situation.
• Interrupts cause context switches.
• Personal processes may be entered iteratively or recursively.
• Personal processes may be entered at a particular level of detail or with a specific kind of input which causes different responses.
• Personal processes may instantiate prescribed processes defined generically by the organization.
• Personal processes create an imaginary envelope around segments of an individual’s line of work.
• Personal processes allow an individual to take control of his own work and give him leverage for redefining his work within the organizational context.
• Personal processes revolve around work redefinition and attempt to maximize the efficiency of work and eliminate unnecessary work.

These entity relationship definition propositions define the arena for personal process analysis. Making process a conscious activity for the individual wherein he makes
“objective” certain aspects of his line of work is the key to overall process improvement. Once such an analysis has been performed, sampling from his own ongoing work stream, then it is possible for the individual to relate that analysis to other dimensions of his working world, such as the political, social, organizational, and facilities dimension. Each of these dimensions are incomplete and under construction. An individual’s working life is multidimensional, and once a picture exists of the work stream, it is possible to use it in many ways to improve the working environment of the individual.

My experiment with personal process showed me that it is really difficult to define processes and enact them. These results have strengthened my skepticism regarding the ability of companies to define processes and legislate them on people. The major problems I encountered were as follows:

- It is very difficult in the “heat of battle,” so to speak, to sit down and define one’s processes. What we are really talking about here is thinking about what you are doing. For the most part people do not think through their actions and their implications.
- Even if you force yourself to sit down, it is difficult to isolate kinds of work. It seems that isolatable kinds of work are exceptions rather than the rule. Many times work comes in snippets where one is doing something in an one-off fashion. Or one is called to improvise on the spot. So work appears more fragmentary than the process model of work allows.
- Another problem is that work is almost entirely, in many cases, interrupt driven. The process model does not account for interrupt-driven work. What is needed is some sort of context-saving mechanism for work which allows one to reacquire a particular context more quickly.
- We are all so product driven that the shape of the product almost completely defines the nature of the work to be done. We guide our work by borrowed templates, or worked examples, to a great degree, but unfortunately most of these templates either do not exist or are too different from what is needed in a particular case to be useful. Thus, because the work we do is so product specific, working out a process is redundant in many cases. Processes are by nature generic. Unless the work that is done has some generic elements, it is not clear how processes will be adapted to them.
- Processes in many instances have no inputs. If we are given an assignment, then we must get the inputs needed or invent them ourselves. This means that work does not flow in a dataflow fashion. Whether this is just a symptom of chaotic processes is not clear. Whether dataflows can be set up in human organizations is not clear.
- Many times the person that receives the data produced from a process does not react. Thus, there is the phenomena of output going nowhere. It exists only because one’s boss asked for it, not because anyone else in the organization really needs it. People in the organization do not know each other’s real needs. They are kept isolated by organizational structure. Many times it appears that a control-oriented organization cannot have a process because so much time is spent creating and maintaining organizational barriers, that flows between organizational segments is precluded by necessity.
- What seems most useful is when two people discuss what the two will do together. I will do this, and you do that, and we will produce such and such. It seems that personal process has the most to offer in this context because people actually do get together and outline mutual responsibilities. But this stage is usually not dwelt on, and many times is circumvented by control-oriented structuring of the work by management.

The fact that creating personal process is hard suggests that perhaps the context in which it is done is important. In control-oriented organizations it is clear that one’s influence on the way things are done is limited, so it is difficult to justify doing much to optimize one’s own work. This is especially true if no one else is attempting to
make the same kinds of optimizations. Thus, it appears that autopoietic systems are difficult to create and maintain by oneself. It needs the whole group to participate, and there is really no call for doing this in a control-centered organization. In fact, it is counter to the flow of work set up by power relations in the organization. So the question arises as to how the autopoietic process can differentiate itself in order to become established. Unless the autopoietic system has some basis other than in individual process actions, it can never be put into action. This really says that the autopoietic system needs to have an internal differentiation which is based on something other than just arbitrary distinctions. It needs to have its own naturally arising intersubjectively valid structure which will provide a different context for personal process management.

**GENESIS AND MODELING OF AUTOPOIETIC PROCESSES**

A question that should be addressed is the origin of the autopoietic ring of software engineering essential processes, and once that is understood, we should consider how to model them. It is one thing to say we need to arrange our processes in an autopoietic formation, which is self-organizing, in which people apply personal processes and proscriptive processes to do their work, and quite another to be able to have a formal model of these processes. Industry needs a model which in both proscriptive and can be enacted creatively to produce the requisite variety. That process must be amenable to being modeled. The real challenge of the future is to get these different dimensions correctly balanced in our process models. This necessitates an understanding of the nature of software itself and the unique aspects of the software engineering discipline and its relation to general systems theory.

Here we will provide a short synopsis of work on the nature of software and the relation of software to general systems theory. Software is seen as a new kind of object based on a completely different foundation than other entities we normally encounter in the world. Briefly there is a hierarchy of levels of Being. Software falls at the third meta-level of this hierarchy in a type of Being called by Merleau-Ponty Hyper-Being. The first type of Being that is below this level at which software falls is Pure Presence which is the type of being that applies to everything that can be the focus of our attention in the world. This is the kind of being defined by Aristotle, elaborated by Descartes, and finally formulated by Kant. The second type of Being is called Process Being which is the level of all technology. It is dynamic rather than static like Process Being, and we do not see it explicitly except as technology breaks down and our attention is drawn to the underlying technological infrastructure that supports our activities which are focused on some purely present object of our intentions. Hyper-Being was first identified by Heidegger and later elaborated on by Merleau-Ponty and Jacques Derrida. It is defined as the inner coherence of the technological system which never appears. Michael Henry identified it with what he calls the Essence of Manifestation in his book of the same name. It is like the unconscious of the technological system and like Freud’s unconscious causes distortions in the showing and hiding of manifestation without ever being seen itself. Derrida describes these distortions in terms of what he calls DifferAnce which is the differing and differing of texts. Texts are a field of pure difference which allow us to manipulate them outside the flow of time. In other words, what is written first may be placed last in the sequence of the text, so the display of text allows a different order from its production, unlike speech. Software is a kind of animated text which has all the aspects which Derrida identifies for text, and thus operates at this Hyper-Being level of manifestation. Beyond Hyper-Being is a final level of Being, or level of describing manifestation, which is called Wild Being by Merleau Ponty and which was positively described by Deleuze and Guattari in their books on Capitalism and Schizophrenia. This final level of Being is that at which artificial intelligence operates. It uses software as a virtual machine the way software uses hardware. Computer hardware embodies the first two meta-levels of Being where software embodies the third layer and artificial intelligence techniques embody the last highest level.

This quick tour of the ontological basis of software is meant only to establish that software is a new kind of entity which is based on a kind of Being which is different from what we are normally used to dealing with in our world. Thus, software and its production processes have attributes that are different from other kinds of objects such as hardware. This new kind of entity must be understood in terms of a different paradigm that is adapted to its mode of manifestation in the world. Attempting to understand it as if it were hardware will always fail. One of the salient differences is that software is a theoretical object and exists as an abstraction. These abstractions are described by software methodologies. An analysis of these methodologies has led to a description of the field of
all possible methods. This field is comprised of four
unique viewpoints on software theory: AGENT,
FUNCTION, EVENT, and DATA. The interrelations
between these viewpoints are the basis for software
methods. Each viewpoint has its own specific software
method, plus there is one minimal method for each one-
way bridge between viewpoints. This gives us a set of
sixteen software minimal methods that describe the entire
field of possible software methods. These individual
methods together form a system of representations that
approximate the form of the software theory. Not all the
aspects can be seen at once, but there is always some feature of the software theory that is hidden. The inner
coherence of the system of viewpoints and their associated
methods is the always hidden essence of manifestation, or
the unconscious of the technical system. It only appears
with the introduction of software into the technical system
which at once gives the possibility of integrating various
aspects of the technical system into an autonomous whole,
and at the same time makes the technical system as a
whole unavailable to complete scrutiny. If nothing else,
the technical system as it executes has compiled software
modules which cannot be seen into and remain in their
compiled version totally opaque. This opaque aspect of
the software bound together technical system has been
described elsewhere by the author as the Meta-technology
which has its own emergent aspects not encompassed by
traditional views of technology.

An important discovery has been that the technical
infrastructure that provides the four viewpoints is only
coordinated, by the appearance of methodologies at the
meta-technical level. At the technical level these four
views are not coordinated and also the non-manifesting
aspects of the technical system do not appear. This relates
to the fact that the viewpoints themselves have different
inherent ordering properties that only are recognized when
they are brought together as a coordinated meta-technical
system. In fact, it is seen that the AGENT and
FUNCTION viewpoints are only partially ordered in
relation to the EVENT and DATA viewpoints, that
represent the spacetime continuum, which are fully
ordered. It turns out that the minimal methods are built on
the intermediary positions between these two kinds of
orderings which is “Linear order without distance” and
“Partial order with distance.” The only other type of
ordering is the non-ordering of pure distinctions. The fact
that all minimal methods are duals which are based on
these two types of intermediary orderings explains a great
deal about the structure of the minimal methods. This
connection between the ordering of the viewpoints allows
a coherent connection to be made between the General
Systems Theory of George Klir9 and the software
methods. In that extended general systems theory any
software system may be modeled. But further, it
represents a general set of modeling techniques by which
any discrete dynamic system may be modeled.

This is an important point because these same modeling
techiques that are used to model the software theories
that are implemented into software products can be used to
model the software process itself. That modeling needs to
be augmented by some continuous modeling techniques,
but all discrete aspects of the software process can be
captured by the exact same techniques for modeling
software theories. This reinforces Osterwiel’s point that
doing software process work is analogous to building the
software product10. This recursive nature of software in
which the methods for describing the products and
processes are the same is another indicator that software is
a different kind of entity that immediately produces
paradoxes and recursive self-referential structures.

It is possible to extend these insights into the nature of
software design to generate the software autopoietic ring
of software essential processes out of what we know of the
software methods and their viewpoints. We do this by
realizing that software design has four fundamental
viewpoints. But there is another viewpoint available
which has not been used in describing software design.
This other viewpoint is that which is described as the
source of unordered distinctions. We will call that
viewpoint the Catalyst. The Catalyst viewpoint is the
source of a never ending variety of distinctions and
discriminations which relates the software design theory to
the world. This relation of the software design theory to
the world is the source of the autopoietic ring of essential
software processes. Basically this is done by substitution.
In the other phases one of the fundamental design
viewpoints is hidden, and the Catalyst viewpoint that
makes unordered distinctions is substituted. This
generates five tetrahedral structures out of the single
tetrahedron of design viewpoints. This also connects the
abstract design to the world because the Catalyst
viewpoint makes concrete practical distinctions which
constrain the design.

Requirements

- Catalyst viewpoint (4) (Agent hidden)

9. Reference?

10. Reference?
INTERFACES OF AUTOPOIETIC RING

EXAMPLE DATA FLOW THROUGH RING

- Test software
- Test results
- Unmet requirements
- Derived requirements
- Requirements queries
- Essential model
- Feasibility studies
- Design document
- Design
- Prototype results
- Prototypes
- Language constraints
- Performance constraints
- Hardware constraints
- Coded units
- Builds
- Integrated units
- Build constraints

FIGURE 10

FIGURE 11
INTERACTIONS OF AUTOPOIETIC RING

REQUIREMENTS

DESIGN

SUPPORT PROCESSES

TEST

CONTROL PROCESSES

IMPLEMENTATION

INTEGRATION

ORGANIZATIONAL PROCESSES

ENVIRONMENT

FIGURE 12

INTERFERENCE NODE

non-essential processes

internal to ring

interference

internal to ring

external interface

FIGURE 13
The Future Of Software Process

- Function viewpoint (2)
- Event viewpoint (3)
- Data viewpoint (1)
- This agrees completely with all known requirements analysis methods which are unanimous that the agent viewpoint is suppressed in this essential process. The Catalyst viewpoint is the source of requirements distinctions which become performance thresholds or functional demarcations which are represented as "shall."  

DESIGN
- Agent viewpoint (1)
- Function viewpoint (3)
- Event viewpoint (4)
- Data viewpoint (2)
- The design viewpoint is purely theoretical, and so the Catalyst viewpoint is itself suppressed.

IMPLEMENTATION
- Agent viewpoint (2)
- Function viewpoint (4)
- Catalyst viewpoint (1) (Event hidden)
- Data viewpoint (3)
- In implementation the Catalyst viewpoint provides the pure distinctions between 0s & 1s to which the software ultimately reduces. The source code itself is purely data, and even the compiled code is just data. The implementation deals with the programming language and the embodiment of the design elements in software modules which deal with the effects of delocalization which smears the design elements out in the actual code. This effect of delocalization is another way in which the catalyst viewpoint appears within implementation.

INTEGRATION
- Agent viewpoint (3)
- Catalyst viewpoint (2) (Function hidden)
- Event viewpoint (1)
- Data viewpoint (4)
- In the integration the functional viewpoint is suppressed. The integrator does not care about the functionality of the pieces he is assembling, but only if the components fit together with each other and the hardware environment properly. The Catalyst viewpoint discriminates all the points of proper connection and improper disconnection between pieces and the hardware substrate.

TEST
- Agent viewpoint (4)
- Function viewpoint (1)
- Event viewpoint (2)
- Catalyst viewpoint (3) (Data hidden)
- In test the data viewpoint is hidden. The Catalyst viewpoint discriminated between correct and incorrect performance on tests. The test data is only traces of events that occur when the test software interacts in execution with the product software. Since we continuously think about test data, it is strange to realize that data is really only records of events that occur during testing and that the event perspective is the crucial one to be considered in testing proper.

In this picture of the addition of the Catalyst viewpoint to four fundamental viewpoints on design, we have managed to produce the autopoietic ring of essential software processes. This is an important step in understanding the structure of the ring itself. In the ring there is a differentiation of key transformations of the software product. This differentiation arises when software is done by more than one person. When one person does software by himself, it is called programming. It centers around the production of source code which runs. It only has to satisfy its producer. Many times this isolated programming is called "hacking." Hacking suggests trial and error ad hoc processes. It also suggests working something over and over until it works. The hacking process by a single individual has the different essential transformations fused. The requirements are not distinguished. The design is embedded in the code. The code is perhaps a monolithic mass of highly concentrated interactions analogous to spaghetti. The code is probably
not specifically tested; it is just run until it fails. Since
design is not distinguished, there is no real implementation
process which specifically deals with delocalization.
Everything takes place under the shadow of
delocalization. Thus, all the worst effects of what Derrida
calls DifferAnce appear in the hacking process. If hacking
is carried over into software developed by more than one
person, then these effects escalate exponentially to cause
severe problems. In other words, hacking is good for
creating computer viruses that you don’t want anyone else
to know about; but hacking becomes counter productive
when just one other person is added to the project either as
customer, co-producer, or operator. The autopoietic ring
of essential processes appear whenever two people attempt
to develop software together, whether they divide the work
so each does a separate part in a prototyping mode, or
whether they do each essential process together as the
waterfall model suggests. This means the autopoietic ring
is an intersubjective phenomena. In that cooperative
situation there is a natural differentiation of work which
allows cooperation. The autopoietic ring appears as this
natural differentiation of the fused hacking process into
separate essential kinds of work. It is possible to relate
this to the appearance in linguistics of MOODS or
universal human transactions. We can identify the
following five basic transactions:

• COMMAND -- Requirements

These are commands that the system SHALL do such and
such.

• PROMISE -- Design

The Design explores the different possibilities of the
design landscape and searches for the most promising or
optimal design

• STATEMENT -- Implementation

Implementation results in a series of source statements.

• NEGATION -- Integration

The different pieces when put together may negate each
other, and this contrariness must be overcome to fit the
system together stage by stage.

• QUESTION -- Test

The builds of the system are tested by running it against
test cases and test software which question the software as
to its embodiment of the requirements.

This set of Moods is not undisputed; different linguists
have their own lists. But all lists contain at least
Command, Statement, and Question. But these Moods
may be considered a representative set. They do not have
any force that we might infer if they were the only
possible transactions attested in all known languages. But
Moods are right on the borderline between grammar and
discourse, which is the distinction Saussaire made
between Language and Speaking11. They represent that
area of language where the structure of the sentence
effected the actual speech transaction you are having with
someone. This is interesting because it is exactly where
the autopoietic ring comes into existence. The fact that we
can see a pragmatic dialogic structure which is embedded in
grammar similar in many ways to the qualities of the
phases of software development is intriguing. It allows us
to connect the qualitative differentiation of the software
essential processes to something we all understand and use
every day which is different moods in our discourse. It is
interesting to attempt to think of a world in which there are
no questions, or a world without commands, or without
some of the other moods. This is an exercise that allows
one to see that these moods are indeed crucial to our
existence. If we could not ask questions, our ability to
learn would be severely constrained. And we would
definitely have no notion of testing software. If we could
not give and receive commands, then our ability to
coordinate with others would be curtailed, and we would
also have a hard time coordinating our software
development with the needs of others, as we would have
no notion of requirements. And so it is with the other
moods which exemplify essential transformations of
software, until we realize that each of the essential
transformations exemplifies some aspect of the
intersubjective process of working together which depend
on human cooperative faculties that are deeply embedded
in our beings. These cooperative faculties, such as the
ability to transmit information as statements between
eachother, or the ability to make promises to eachother,
form a ready-to-hand part of our being-in-the-world with
others. This manifests itself in certain linguistic
phenomena, but also it manifests itself in the natural
structuring of the cooperative work process that has
become legitimized across many disciplines.

What is unique to software here is that this basic sequence
of steps in software becomes an autopoietic ring instead of
a linear process with beginning and end. In software the
snake has eaten its own tail in many ways. First, the

11. Reference?
META-RING OF STACKED AUTOPOIETIC RINGS

TWO DIMENSIONS OF PROCESS COORDINATION

FIGURE 16
Spiral Model combines both prototyping and waterfall lifecycles additively.

**FIGURE 17**

**PROTOTYPE LIFECYCLE**

- TEST
- REQUIREMENTS
- DESIGN
- IMPLEMENTATION
- INTEGRATION
- TEST

**FIGURE 18**

**WATERFALL LIFECYCLE**

- TEST
- REQUIREMENTS
- DESIGN
- IMPLEMENTATION
- INTEGRATION
- TEST

**FIGURE 19**

- SORUCE / SINK
- SORUCE / SINK
- SORUCE / SINK
- SORUCE / SINK
- SORUCE / SINK
- SORUCE / SINK

**FIGURE 20**
AUTOPOIETIC HYPER-CYCLE

Keeps subjective autopoietic rings and at same time has intersubjective meta-ring structure.

FIGURE 21
The concept of the autopoietic hypercycle is to allow each individual to operate semi-independently while participating in overall formalized communications. Each autopoietic ring would follow the timewarp analogy, forge ahead with their own work, communicating as much as possible with others via well-defined interfaces. The results of the work would be written out to a global blackboard which would be divided into malleable and frozen areas. The independent agents would read the blackboard and see where the overall design is and what has changed in other areas. The frozen area would be what aspects of the system were intersubjectively agreed upon. This horizon is the actual global present which individual agents may be outstripping. This means they may have to do some rework if the frozen design happens to turn out differently than expected by that agent. The agent would watch the horizon between frozen and malleable aspects of the system, carefully taking risks as to where to go ahead on and where to wait for intersubjective agreement. The outer wheel of blackboards corresponding to each essential process allows intersubjective waterfall-like coordination while the inner hub of stacked autopoietic rings allows the strongest possible coherence throughout the development process. The inner rings and the outer wheel are connected by the concept of mutually accessible blackboards using the time-warp system which allows global time to lag behind the time of independent agents. This may involve some rework, but the process is designed to minimize this and allow for moving ahead on individual pieces of work.
INTERFACES OF AUTOPOIETIC RING

Fig. 23

INTERSUBJECTIVE COORDINATION ENVIRONMENT

REQUIREMENTS
Blackboard

TEST
Blackboard

DESIGN
Blackboard

CONTROL PROCESSES

IMPLEMENTATION
Blackboard

INTEGRATION
Blackboard

ORGANIZATIONAL PROCESSES

SUPPORT PROCESSES
essential processes form a ring which must be broken to stop. Software products keep evolving. They are not etched into hardware and may continue to change in ways the hardware could never change. The reprogramming of Voyager to go on to explore more planets, one after the other, is a case in point. The adaptations of the Voyager spacecraft were made in flight by changing the software, where no hardware changes were possible, other than those caused by deterioration of the platform. Thus, software once tested is merely the basis for a new set of requirements which cause it to be totally revamped. We now artificially separate first time development from maintenance, but actually this is all one process in which the autopoietic ring once established is self feeding. Beyond that there is the fact that software essential steps are the same steps that must be undertaken for the software practitioners to organize their own work. Thus, within software developing the process goes hand in hand with development of the product. In this the autopoietic ring is applied to itself in a way that is unique to this production process. In all other types of production processes it would be absurd for anyone to say that process is software too. But in software it is difficult to distinguish between the product and the organization of the work to produce the product.

Once we accept the augmentation of the four fundamental viewpoints on design with the Catalyst viewpoint, and the fact that this generates the differentiation of the autopoietic ring, then we can go further and look into what this tells us about the autopoietic ring itself. For one thing it means that there is a whole new set of minimal methods that arise which represent the relation of this new viewpoint to the other design viewpoints. In this case the new minimal methods will have a different set of characteristics because they relate to the generation of distinctions within the context of all possible distinctions and relations between things.

- Catalyst alone: Logic of Form
- Catalyst >>> Function: Lano N²

G. Spencer Brown¹² and Francisco Varela¹³ have developed a logic based on the simple act of making a distinction. It is the prototype for all distinctions that are made by the Catalyst.

The Lano N2 chart allows all possible relations between functions to be made visible for analysis.

- Function >>> Catalyst: Facet Analysis 2ᴺ

All possible functional decompositions.

Faceted Analysis and Synthesis allows all possible emeshings of different decompositions of the same thing to be studied.

- Catalyst >>> Event: Temporal Logic¹⁵

All possible event sequences.

Temporal logics studies the relations between necessary and variable event sequences to be studied.

- Event >>> Catalyst: Combinatorics¹⁶

All possible relations between events.

Combinatorics allows the permutation of event sequences to study all possible sequences.

- Catalyst >>> Data: Information Theory¹⁷

All possible relations between data elements.

Information theory studies the coding of data, usually for transmission, but newer forms of information theory consider the building up of patterns of relations between pieces of information.

- Data >>> Catalyst: Fractals¹⁸

All possible data pattern decompositions.

Fractals are a means of seeing the embedded nature of information.

- Catalyst >>> Agent: Network theory¹⁹

All possible relations between agents.

Network theory allows the abstract relations between nodes (that may be thought of as agents) to be analyzed.

- Agent >>> Catalyst: Category theory²⁰

---

¹². See LAWS OF FORM
¹³. Reference?
¹⁴. Reference?
¹⁵. Reference?
¹⁶. Reference?
¹⁷. Reference?
¹⁸. Reference?
¹⁹. Reference?
²⁰. Reference?
All possible embeddings of agents within agents.

Category theory allows the onto and into mappings of embedded agents to be analyzed.

These methods perform a different kind of task than those minimal methods associated with design. They allow the connection of specific distinctions to be analyzed as they relate to the field of all possible distinctions. From this perspective the Catalyst is the means by which a myriad of specific distinctions and decisions are made within the realm of the WHY of the software theory. There are a myriad of reasons why any piece of software is the way it is. This is seen as what Peter Naur\textsuperscript{21} calls the non-representable nature of software theory. All we can really have is partial views which do not deal with WHY but instead with WHO WHAT WHEN & WHERE. But all these myriad reasons why are based on specific distinctions and decisions made somewhere in the history prior to or within the project. The Catalyst is the agent which motivates these decisions within each kind of work other than design. The design has specific decisions as well, but they do not deal with the relation of the theory to the world. The catalyst deals with the specific decisions in each kind of work that connects the software theory to the world. It is called the Catalyst because the viewpoint demands or inspires the requirements analyst, the implementor, the integrator or the tester to make these distinctions and decisions that connect the software to the world and make it more than a theory. The catalyst does not make these decisions. It is a viewpoint that demands that these discriminations be made, not the subjectivity that makes them. The implementor, for instance, is driven by the viewpoint of the Catalyst to create a myriad of ones and zeros using a particular compiler in order to render the design concrete. The Integrator is driven by the viewpoint of the catalyst to put all the pieces of the software together and distinguish those that fit together with the hardware and eachother, and those that do not. The Tester is driven by the Catalyst to distinguish those tests that the software passes and those that it fails. The Requirements Analyst is driven by the Catalyst to distinguish the thresholds of acceptable performance and the limits of desired functionality. The software design itself is an abstraction. Without the work of the Catalyst that connects it to the “real” world it would remain a non-fully representable theory. When the representation is realized in required, implemented, integrated, tested code, then it becomes more than a theory. It achieves full representation in which the gap between the partially ordered and fully ordered viewpoints are jumped over and through the realization that the always flawed embodiment is made complete. Like Zeno’s paradox in which the arrow never reaches the target, without the Catalyst the non-representable theory of the software design would never be realized as a working system.

One way to view the Catalyst is as the positive face of the Essence of Manifestation\textsuperscript{22}, or intersubjective unconscious\textsuperscript{23}, that refuses manifestation at the meta-technical level of manifestation. The Catalyst is a cornucopia of distinctions; it is a source of variety which floods the software practitioner with all the myriad details that have to be just right which make the job of writing software so difficult. Where the essence of manifestation (which Deleuze and Guattari call the Body-without-organs\textsuperscript{24}) is never seen but only its effects are seen as distortions, so the Catalyst for distinctions is seen everywhere. Gregory Bateson called these distinctions “differences that make a difference”\textsuperscript{25}. A software system is just this plenum of minute distinctions which is best epitomized by the ultimate pattern of incomprehensible zeros and ones of the compiled code. The compiled code is the source of the behavior of the software system. Behavior is the manifestation of autonomy in action. The artifact that produces the behavior is opaque or incomprehensible to us. It is exactly the opposite of the transparency of functionality. This artifact of human endeavor which is incomprehensible actually operates when executed, doing many things which are comprehensible as it ties together the technological system and coordinates its actions. Thus, we can see these two as nihilistic opposites which cancel to produce the next higher meta-level of Wild Being, just as Process Being and Nothingness cancelled to produce the meta-technical level of Hyper-Being.

Since the autopoietic ring itself is a natural product of intersubjective cooperative work or discourse, we can see vestiges of it in the history of human civilization. For instance, it shows up in the Chinese archaic sciences as the ring of Five Hsing or transformations. In fact, where the West traditionally produces control-oriented models of systems, we can see now that Chinese archaic science may have been totally preoccupied with producing models of autonomous closed systems. So just as Western

\textsuperscript{21} Reference?

\textsuperscript{22} Michael Henry THE ESSENCE OF MANIFESTATION?

\textsuperscript{23} Carl Jung called this the ‘Collective Unconscious’.

\textsuperscript{24} G. Deleuze & F. Guattari ANTI-OEDIPUS

\textsuperscript{25} See STEPS TO AN ECOLOGY OF THE MIND
perspective lines converge while Chinese perspective lines diverge, so too here the Chinese views of science could perhaps have been diametrically opposed as well. The model of the Five Hsing is very ancient and is the basis of Chinese Acupuncture and Herbal medicine that the West is just now discovering in spite of its lack of Western scientific explanation. Perhaps that lack of explanation results from our not knowing how to describe autonomous closed systems. Now we can see that there are other sources of knowledge of how these autopoietic rings operate which are already available. In a recent book Varela uses the concept from Buddhism of co-dependent arising to explain the relations between the links in the autonomous ring. Co-dependent arising says that these elements of the autopoietic network arise together and are mutually supporting with no outside cause. Each one causes all the others and ultimately itself through the feedback and feed-forward of the network itself. There is no cause outside the network. The origin of the network is always already lost. The elements of the network co-evolve over time arising from a single origin with emergent properties which cannot be reduced to a lower level of structure outside its own self organization. In our case the five essential transformations of software arise from the hacking of the isolated programmer under the need to make that work intersubjectively accessible. This intersubjective alteration of hacking is differentiated under the same constraints that cause the Moods to arise in language, and each transformation takes on some characteristics similar to the moods. But once formed, because we are dealing with software, the essential processes become a self-generating ring which is autopoietic in nature. This means it is a closed unity which has exactly the same form as seen in the five transformations at the basis of the Chinese medical system.

This use of Buddhism to understand the closed autopoietic system suggests we might look at other Eastern sources as well. And the structure of the Five Hsing is an excellent example that fits right in to our study of the five essential software transformations. Strange as this may seem, we would well consider the relation between the production and control processes that the Chinese saw between the Five Hsing.

PRODUCTION CYCLE
- Earth produces Metal
- Metal produces Water
- Water produces Wood
- Wood produces Fire
- Fire produces Earth

CONTROL CYCLE
- Earth controls Water
- Water controls Fire
- Fire controls Metal
- Metal controls Wood
- Wood controls Earth

Notice that true to autopoietic form these two cycles form a ring. Each of these operations has its inverse so there is both feed forward and feed back. The standard image for relating the different elements is a kettle standing on the Earth, over a bunch of wood logs, which are burning, heating the metal container that has water inside. The water boils and lifts the lid with its steam so that the escaping energy is called CHI. The Five Hsing work together to produce the flowing energy within the whole system that has an external effect. This energy has a pattern which is called the LI. These two concepts will be explored in further depth in the next section. Here our interest is on the Five Hsing themselves. They are each transformations that form a closed system. That system is seen as the basis of energy transformation in the body. It is what allows the human body to be an autonomous system. Thus, in the Chinese system it is this set of transformations that is directly responsible for the autonomous behavior of individual creatures.

Now it is interesting to connect the chinese transformations to those that occur in linguistics and software engineering.

<table>
<thead>
<tr>
<th>Element</th>
<th>Category</th>
<th>Order</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Earth</td>
<td>Catalyst</td>
<td>Distinction</td>
<td>No Order</td>
</tr>
<tr>
<td>Metal</td>
<td>Agent</td>
<td>Autonomy</td>
<td>Partial Order</td>
</tr>
<tr>
<td>Water</td>
<td>Data</td>
<td>Space</td>
<td>Full Order</td>
</tr>
<tr>
<td>Wood</td>
<td>Function</td>
<td>Intentionality</td>
<td>Partial Order</td>
</tr>
<tr>
<td>Fire</td>
<td>Event</td>
<td>Time</td>
<td>Full Order</td>
</tr>
</tbody>
</table>

These five viewpoints form the vertices of a pentahedron in four dimensional space. The pentahedron has five points, ten lines, ten sides and five tetrahedral solids. The
The Future Of Software Process

group of rotations of the pentahedron is called by the mathematical group theorists “A5” with 60 different group operations. It shares the same group of rotations with the icosahedron. A pentahedron is actually two intertwined mobius strips. Since the pentahedron has five points or viewpoints and also five tetrahedrons, we can see that the tetrahedrons correspond to the five phases within which the transformations take place. So for this geometrical form the viewpoints and the phase structure is isomorphic. The structure generates by the combination of the viewpoints five phases which we have identified with the linguistic moods and the intersubjective software transformations. The pentahedron gives us a closed form in which we can go back and forth between viewpoints and minimal systems or phases.

As Buckminster Fuller noted in his masterwork SYNERGTICS, it takes four independent co-appearing and overlapping phenomena to produce a minimal system. Thus, the permutation of the five viewpoints taken in sets of four produce the minimal systems of the separate phases. Fuller relates the minimal system to the tetrahedron alone. But actually, the minimal system has four faces that need to be considered which appear as TETRAHEDRON, KNOT, MOBIUS STRIP, & TORUS. Each of these geometrical forms contain 720 degrees of angular momentum. This number is a topologically significant threshold. But these are merely geometrical figures that represent deep structural relations between aspects of the minimal system. The tetrahedron is the structure of the system. It can be taken instead as a lattice of the faceting of the elements that compose the minimal system. The Knot is the self-interference generated by the system. We have already seen this as the interference between the four independent information flows. The torus is the closed ring itself, and the Mobius strip is the paradoxical nature of global versus local features of the system. In the autopoietic configuration all of these faces of the minimal system become relevant to our analysis of the autopoietic system. The pentahedron is composed of five interpenetrating minimal systems. These are generated by the addition of one point and four lines to a normal tetrahedron. Thus, there is heavy redundancy in the reuse of lines and sides between these five tetrahedrons. In this situation the five tetrahedrons are manifestations of a single tetrahedron with a single viewpoint interchanged. This single tetrahedron has its four faces operating within the system of its five projections in each of its projections. So we see the system of projections take on the ring-like structure of the torus. We see within each segment of the ring an interference pattern between four information flows. We see that there is a paradoxical global continuousness to the whole ring while locally it is segmented which is like the global/local paradox of the mobius strip. All these features appear within the autopoietic unity which we can relate to our four dimensional geometry of thought which builds upon B. Fuller’s three dimensional geometry of thought. But this geometrical analogy is merely our way of viewing deep structures that arise naturally out of autopoietic unities as they manifest in spacetime.

The structure of the co-arising co-dependent autopoietic ring as reviewed here using geometric analogies is a static structure, whereas the ring itself is never static. It is, in fact, dynamic and has its own energy structures and pattern forming regimes which will be discussed in the next section. There is a tension set up between the functional and autonomous viewpoints which can be seen as defining the relation between what the Chinese call Chi (flowing energy) and Li (pattern). One way to view this relation between Chi & Li is by seeing that the continuous ring structure is the prerequisite for the production of Soliton waves which move within the ring. Soliton waves do not dissipate energy, and are basically entropically stable. Soliton waves can pass through each other and bounce off the sides of their channel without losing energy. The Chi may be seen as the movement of soliton waves around the ring. This ring itself is a dissipative structure that is negentropic but within the channel of the ring move non-entropic energy patterns as the Chinese have predicted. The Li would then appear as the pattern of standing waves that is set up in the whole ring structure by the flow of the soliton waves around it and their interaction with normal waves that would correspond to the input and output energies of the system. The tension between Chi & Li is adumbrated by the Catalyst which resides within the membrane of the autopoietic unity. The role of the Catalyst has been studied by Zelany. He has produced computer simulations of autopoietic systems which build their own membrane by the action of a catalyst on a flowing substrate that moves through the membrane. These studies show that autopoietic systems can be modeled as dissipative structures that are self maintained through the flow of an artificially maintained non-equilibrium. This is why the set of essential processes are called a WATERFALL model. It is recognized that it is a continuous flow through the stream of essential processes. What was not realized was that this waterfall, like those in Escher drawings, are self feeding and self

27. Reference?
28. Reference?
Another related speculation is that the connection between the five viewpoints is the source of language. Language is behavior coupled with meaning that is dependent on discrimination within the spacetime continuum. This fusion of the Five Hsing into a single structure allows the human being to build up his world. Heidegger identifies three existenials: Discoveredness, Talk, and Understanding. Talk may be seen as the intersection of the Five Hsing within man. Talk is based on a whole series of levels of distinctions which range from phonemes, to words, to clauses, to sentences, to paragraphs, to documents. As has been shown by Conway, these distinctions are built in, and even the newborn baby resonates its whole body to these different levels of linguistic distinctions. These levels of distinctions are patterned by the syntax of the language which is for the most part a behavioral production process based on deep-rooted grammars such as those described by Chomsky in his transformational grammar. But all speech has a semantic dimension as well which corresponds to the functional or intentional aspect of the Hsing. Finally, the dance of language occurs in spacetime with the creation of patterns of sound. Thus, in language all the five viewpoints are represented as major constituents. And language is our major way to understand and even project our world. Thus, it appears that the existential TALK is an intersection of our capabilities to produce everything else. Of course the other existenitals are Discoveredness which is the raw fact of finding ourselves present in the world AND the understanding of that world which is not limited to understanding through language. However, of these the prototype of the creation process itself is embedded in language, and we can see the Five Hsing clearly there in that arena. Those elements get changed when we move from spoken language to written language. The fact that we can understand both is some proof that understanding is not limited to language but is a separate existential. In written language these same elements are present, though, in an altered form. Writing is based on a similar hierarchy of distinctions such as letter, word, clause, paragraph, text. Writing is a behavior which produces something with intended meaning. Writing encodes information in a spacetime process. So all the basic Hsing are interacting here similar to the way they interact in speech. However, there are some major differences. First the distinctions are in something that leaves traces, such as letters on parchment. Second the behavior that produces the communication is separated from the reading process in a way that listening is not separated from speech. The meaning is dependent on a reduced context of the text itself and is not situational. The spacetime process of writing has to do with manipulations by the hands instead of by the mouth and larynx on a different medium. The new medium makes possible the saving of the work for future reference which is the basis of our culture. But also all the effects peculiar to text pointed out by Derrida come into play as well.

M. Zeleny and N. A. Pierre in their article "Simulation of Self-Renewing Systems" enumerates the principles of management of human systems that he draws from his computer models. It might be well to review these in preparation for attempting to return to the problem of workflow moving through the autopoietic cycle.

- "Complex and dynamical human systems are to be managed rather than be analyzed or designed. Human systems management is not systems analysis or design."

Although we talk about applying the essential processes of requirements, design, implementation, integration, and test to the development of a process for doing those same things to develop software, we do not mean that one person designs the process and another person executes it. In this case there is a split in the autopoietic unity, and the whole thing falls apart into an allopoietic (other produced) system. At that point the law of requisite variety (R. Ashby and S. Beer) comes into effect. That law says that a control channel must be as complex as the thing controlled to exert complete control. Software processes are so complex that no communication channel that complex is practical. Only if the people enacting the process are the designers of the process, can they have any hope of controlling it. All external control is in fact illusory. Thus when Zeleny says that human systems are managed, not designed, it is clear that he means that management does not design the autopoietic processes but merely seeks to influence them indirectly by acting as a Catalyst through a narrow communications channel. This breaks the illusion of complete control and makes management a resource (instead of some grand designer divorced from the work being done) used by the enactors of their self-defined process.

29. Reference?
30. OF GRAMMATOLOGY ?
32. Reference?
33. Reference?
“Human systems management is a process of catalytic reinforcement of a dynamic self-organization and bonding of human components. It does not design a managerial hierarchy of command and control.”

Managers are catalysts which attempt to get an autopoietic sociotechnical formation instituted and then stand back to act as a resource for that once it is in operation. But as we have seen, management is not the only catalyst. In fact, the catalytic perspective is important in each of the other phases outside of design. The point is that there needs to be many catalytic reactants in order to keep the system in non-equilibrium. This is the main functional role of the catalyst. So in requirements we see that the requirements are always changing. This is a natural formation that occurs. In fact, it appears in all the non-design phases. In Implementation there are always changes in the underlying development system or the target system; in Integration there are always changes in the set of builds and the way the system is linked and configured; in Test there are always undiscovered bugs in the system to be routed out. All these points of continual production of variety must be managed. But they are essentially the generators of non-equilibrium that the autopoietic unity feeds off to keep itself going. We continually talk of getting rid of these nuisances. But, in fact, it is these producers of variety that are what keep the autopoietic system going. The autopoietic system is set in motion by the catalytic action of management. But the unity itself is a series of catalytic components which form a minimal system and keep the unity in existence as a waterfall of produced variety which comes about as a reaction to the catalytic viewpoint.

“Components of human systems are humans. As such, they differ significantly from other components, mechanistic or biological, in their ability to anticipate the future, to formulate their objectives, to plan for their attainment, and to make decisions. These properties are sufficient to make human systems quite distinct from all other systems.”

It is important in our understanding of software process to keep this point in mind. We are designing human processes. We need to design them in such a way that they take advantage of human creativity and the ability to distinguish the correct way of doing things. Many process designers seem to forget this and treat the practitioners on the same level as computer hardware, seeing process only from control perspective. In the development of personal process and the recognition that human autopoietic systems must have enactor-process designers, we have attempted to build in this important precept.

“The integral complexity of human systems can be lost in the process of its simplifying reinterpretation by the rigor of mathematical mechanics. Human systems can be described and studied through a relatively simple set of linguistic, fuzzy, and semantic rules, governing the self-creation of its complex organization. Human systems management is not operations research, econometrics, or applied mathematics.”

Zelney sees process from the point of view of cellular automata. Each actor uses a set of rules to produce his own variations on the process model. In fact, we see each phase of an autopoietic unity as such a cell. This is closer to Varela’s cellular automata models of autopoietic rings. Thus, it is the phase of the autopoietic unity that is the “cell” not the human being. Given that change in emphasis, it is clear that the autopoietic ring can generate much more variety than any control-oriented structure. The human that is enacting a particular kind of work executes the rules associated with a particular process. The fact that these processes should be described in terms of fuzzy or linguistic variables is fully supported. In fact, we see an interference phenomena at the center of each phase in the ring where internal and external information flows come together. This knot of interference needs to be described using fuzzy techniques because the complexity of the interference pattern itself is far beyond what any human being can handle. Thus, a fuzzy interpretation occurs in every phase of the ring by which the interfering information flows are reduced to some common denominator which the human can handle.

“Interactions between components are not those of electronic circuitry, communication channels, or feedback loop mechanisms. Rather, they are organic and dynamical manifestations of organizational autopoiesis. Human systems management is not cybernetics or the information theory of communications.”

This point is just a statement that autopoietic units are emergent phenomena. They cannot be reduced to other

Kent Palmer
phenomena without doing violence to the autopoietic unity itself. In fact, if we reduce them, they become invisible. This is why they are not seen by control-oriented Western science. Unless you are looking for closed unities, they do not appear. In fact, the two viewpoints are probably mutually exclusive. The Chinese did not see control-based systems because they saw everything as closed unity. This goes along with the fact that their worldview did not have transcendentials but was entirely immanent from our perspective. It is our dualistic tradition which sees every relationship as a power relation which cannot “see” closed unities. In the non-dualistic Chinese tradition complementary opposites are seen instead of power relations. Probably there are many closed unities right in front of our very eyes if we could but see them. An example that springs to mind is the family. The family is a self-differentiating unity which is having trouble surviving in our present society. But internal to the different types of family there is probably another different set of autopoietic phases. It is not the members that are the autopoietic elements, but different sets of behaviors which the members process in sequence. To recognize autopoietic unities we must shift from looking at the “entities” to the processes, and then we must look for circular intersubjective behavioral processes. In this light it is important to recognize the special place Symbolic Interactionism social theories have in being in tune with the autopoietic viewpoint on sociotechnical systems. Symbolic Interactionism focuses on behavior and thus treats autonomy without the reductionism of pure behaviorism. For Symbolic Interactionism it is exactly the self that becomes the center of attention when considering what controls behavior. In autopoietic systems the nature of the self becomes the crucial question. The self for G.H. Mead, the founder of Symbolic Interactionism, is an internal reflection of how we think others see us. It is this reflection that controls behavior of the individual. We would expect in every social context where some group organizes itself that these autopoietic entities may arise. But not all human social groups are autopoietic unities. Autopoietic systems are a special type of social formation which is perhaps rare. A textbook example of an autopoietic system is given by Plato in his LAWS. Despite the criticism that he has received for the totalitarian aspect of his “closed society,” it is clear that he had the ideal of creating autopoietic social unities in mind.

• “Dynamical order of human systems organizations is maintained through a continuous renewal of certain nonequilibrium conditions. Both nonequilibrium and instability are essential for self-organization or higher complexity. Human systems management is not a theory of general equilibrium.”

Autopoietic systems are Dissipative Structures in the sense defined by Jantsch35. A dissipative structure produces a membrane which represents an negentropic gradient which creates organization despite overall entropic dissipation. The autopoietic system is exactly the same thing. It produces an organization which is negentropic in spite of entropy at the component structural level which is constantly being replenished in order to fight the continuing always already lost battle against entropy. Thus, we do not think of autopoietic systems as things. They are by definition processes only, and as soon as the process stops, they cease to exist. Entropy disperses their structural components. An excellent example is the cult which has become such a popular religious form in American society. These closed social systems with their own behavior patterns and beliefs many times greatly at odds with predominate values flourish in the overall materialistically-oriented dominate society. The membrane of the cult itself must be continuously maintained, and once the members stop maintaining their difference from the social environment, the cult vanishes. All autopoietic systems are a little like cults in this sense of being a closed intersubjective system. In software we call them the team. They develop their own sets of values, specialized vocabulary, group memory, etc. which mark them off from everyone else. Everyone knows who is in the team and who is outside. The boundary of the team is continuously being maintained and augmented as a spin-off of all the activities of the team, such as meetings that include and exclude certain people within the overall organization.

• “The concepts of optimization and optimal control are not meaningful in a general theory of human systems. Human aspirations and objectives are dynamical, multiple, and in conflict, as are those of human organizations. This conflict is the very source of their creative evolutionary unfolding. Human systems management is not optimal

34. See Patrick Baert and Jan De Schampaetheire “Autopoiesis, Self-organization and Symbolic Interactionism: Some Convergences” in Kybernetes 17, 1 p60-69

Human beings are variety producers, as Stanford Beer has said. Once the variety is produced, then it becomes the field for interpersonal conflict. But the focus on conflict is probably a throwback to the old control-oriented way of viewing human systems management. Rather, it is important to take the view exposed by Deleuze and Guattari in their books on Capitalism and Schizophrenia that are a study in the organization Wild Being, which sees human society as at root schizophrenic. This realization that the process of variety production is endemic and will never go away is very important. From the old control-oriented viewpoint this looks like the natural appearance of conflict at the root of all social structures. In the LAWS Plato calls this the war of the ALL AGAINST THE ALL. Plato goes on to show that the laws must seek to instill all the virtues, not just courage, but also justice, moderation, and wisdom. Thus, we must learn to accept that human beings will produce variety, and we cannot ever control that effectively with any process we might invent. Therefore, we must seek to only reduce unnecessary variety and do that by getting the practitioner to define his own process via a set of high level guidelines. In extremes the variety production goes out of control and is seen as a sickness like schizophrenia. The ad hoc initial maturity level software processes suffer from this syndrome. All we are doing is attempting to channel the variety to a certain degree. If we can get an autopoietic system set up, then we expect it to organize itself creating the requisite variety for its own problems but reducing the wasted effort of creating unnecessary variety which in extremes becomes schizophrenic. It takes more energy to create new variety instead of reusing and following old trails. In fact, in process work we are attempting to produce “channeling” or what Waddington calls “cheords” which are paths of least resistance to serve as the river bed of the autopoietic unity. The autopoietic social unity organizes itself in a channel of least resistance which is laid down for it to follow. If it needs to deviate, then it decides to spend the resources to produce the necessary variations and perhaps develop a new channel. Other than that the autopoietic unity would be expected to reign in its own variety production and as part of its own self-organization produce only the requisite variety of the task at hand.

* “The inquiry of human systems is transdisciplinary by necessity. Human systems encompass the whole hierarchy of natural systems: physical, biological, social, and spiritual. Human systems management is not interdisciplinary or multidisciplinary; it does not attempt to unify scientific disciplines, but transcends them.”

Those engaged in the development of software processes have a lot to learn from other disciplines in the humanities. So the interdisciplinary and multi-disciplinary aspect of software process definition will increase in the future under the breakdown of the purely technical solutions being proposed now. However, what is not well recognized is the transdisciplinary nature of autopoietic unities. They are trans-disciplinary because they are rooted in ontological structures that determine how manifestation occurs in our worldview. They are rooted in the structures of Being which has fragmented into a series of meta-levels with new entities entering our world founded on specific kinds of Being. Software is one of these new entities. Artificial intelligence techniques is another source of new entities from an even higher meta-level of Being from that of software. Whenever we see specific structures such as the autopoietic software development ring that are representative of the structuring of our worldview itself, then we have entered into the transdiscipline arena. All disciplines that recognize and study autopoietic systems will eventually cross pollinate as those that study chaotic systems now do. But it is yet another step to recognize that we are acting out structures that articulate the general structure of our worldview and that our exploration of human sociotechnical systems has implications for our development of philosophies underlying our work structures.

THE WORK FLOW AND PROCESS WISDOM

With the definition of personal process and the recognition of the nature of autopoietic systems we are in a unique position to differentiate the meta-levels of process refinement. The foundation of all process work is the generic process definition. There are a myriad of ways to define generic organizational processes. Some researchers want to treat people as machines and program them like computers, saying that process may be captured in a formalism like a programming language. Other industry people implementing process in their own organization think we need to provide desk instructions

36. See TOOLS FOR THOUGHT ?
that tell their people every move to make. There are others that merely want something that passes inspection by SEI assessors as a workable process definition. In our own case we are concerned with actually improving our work, and so we see process descriptions as an uneasy marriage between a description of what already goes on and a description of our best practices with a sprinkling of mandated improvements. However, all prescriptive process improvements assume a continuity of underlying work that can be ordered according to process guidelines. The envelope of continuity is different for different teams, individuals and projects. But each envelope is construed as an ongoing context within the work situation which can be entered and re-entered in midstream to keep some aspect of the work to be accomplished going. This continuity is socially constructed. It is a product of negotiations for resources and agreements as to what work needs to be done to accomplish the final ends of the project. It is the wool that runs through the warp of the schedule of the project. But in some way the prescriptive process is assuming a worker (subject) who is continuously putting effort into that context maintaining the forward momentum of that aspect of the project. This forward momentum is something of an illusion as is the abstract description of processes that assume ideal workers under ideal conditions. This is, of course, the level at which SEI sanctioned process definition occurs. We can see that this level of prescriptive process corresponds to the ontological level of manifestation represented by Pure Presence.

In execution projects always turn out differently than they were planned. Continuous effort is dependent on a myriad of constraints holding steady, and other activities going well and producing the necessary outputs on which immediate work is based. These constraints are continuously being trespassed so that the plans are always breaking down and having to be reformulated. This adaptation of the project in midstream is necessary to maintain the illusion of continuity. In reality the continuity is constantly changing and adapting as the project is moving forward. This level of adaptation is where the negative form of breakdown is aperiodically occurring as the plans attempted to be turned into actualities. Probabilities reign here as in every case where determinate plans are actualized. This is the level at which statistical process measurements are taken and used as a basis for global optimization of the overall process on particular actual projects. We can see that this level of statistically enacted process corresponds to the ontological level of manifestation represented by Process Being.

The next deeper layer of process is that which personal process addresses. It addresses the positive aspect of breakdown where all work is redefined on the basis of someone’s understanding it in a new way. Personal process is directed at getting a picture of the work from one’s own perspective in order to understand it well enough to redefine it. Whereas the top layer assumes work has a long lasting form, and the second layer is just making local fixes to get back on track, this third layer is actually actively trying to transform the work into a new pattern by understanding it thoroughly from a particular perspective. Every kind of work is susceptible to redefinition. This is the arena in which all real process improvement must take place. It is the exact arena ignored by SEI. The statistical analysis performed by process improvement specialists see individuals as random variations. However, it is individual initiative and insight which can really make the big difference in redefining process creatively to streamline it. And this is also the level where our culture has its greatest strengths in providing individuals who are able to take initiative and make changes that make a difference in the way things are done. It is ironic that American business culture has emphasized monolithic organizational structures and multiple tiers of control that have mitigated against the very forces of innovation spawned by our culture. In many instances creative individuals have either had to start their own companies, go elsewhere to work where they were given the power to make changes, or keep quiet and let things remain bad. We can see that this level of personal innovative process corresponds to the ontological level of manifestation represented by what Merleau-Ponty calls Hyper-Being.

New team organizations offer the potential of supporting individual initiative. The layers of stultifying management have disappeared. However, the problem in the new organization is that if your idea for improving things is beyond the scope of your project, then there are no resources to draw upon to make it happen and no one with broader concerns with the power to make it happen. Yet the resource that the team organization has to offer is the close knit group with a common goal so that if your idea for improvement falls within their scope, then it is more likely to get implemented if they can be convinced.
of its efficacy.

The final and deepest layer of process is that of the line of work and going concern. This is the layer which addresses what actually happens in all its gory detail and irrationality. It is the land of political decisions, wrong goals, personal conflicts, and everything that is counter productive. It is also the layer in which the person’s own motives and behavior patterns exist; where social alliances and personal networks have real effects on the way things actually get done. It is many times referred to as under the rubric of the Jungle. Deleuze and Guattari call this the schizophrenic level of society. At this level there are aspects of personal and social behavior that will never be rationalized and that should never be attempted. It is the level at which personal freedom of action reserves the right to reign unfettered. Yet this is also a level under attack by automated systems which can track personal productivity keystroke by keystroke in automated routinized jobs. SEI only implicitly recognize this level in their classification of initial maturity as ad hoc processes. We can see that this ground beneath all processes corresponds to what Merleau-Ponty calls Wild Being.

You will notice that the SEI model completely ignores the third layer at which personal process appears. Their model is not culturally adapted. It does not take into account the creativeness of the individual that has the highest potential impact on changing the process for the better. It, in fact, assumes the old control structure oriented organizations that have a major incentive to stifle creativity. Culturally this tension between control orientation and the natural percolation of innovation from self-directed individuals represents an untapped resource for process improvement. The connection of this resource with the team-centered environment is potentially a means of deriving great benefits in efficiency and quality improvements. By giving the practitioners control over their own processes and encouraging adoption of creative solutions, we can anticipate great gains. But this mitigates against legislated proscriptive processes imposed from above in the old control oriented style.

Processes are streams of predefined actions called tasks, streams of information such as communications, streams of thought and self-reflection, streams of materials that are worked upon, streams of representations that are transformed. As streams they represent the combined energy of groups of individuals working in concert. As such they appear as a dimension of our work lives that have been hidden by the static control structures that attempt to govern these streams. Organizational structures appear as static. Facilities structures appear as static. People who do the work are considered as replaceable roles. The old control structure has taken the assembly line as its model with not only replaceable parts, but replaceable workers with limited specialized skills. In a shrinking workforce the emphasis is shifting to multi-skilled experts and cross training. The functional organization is slowly disappearing. With this shift the previously invisible realm of process streams are becoming visible. Teams own their processes. The work is shared, and the complete product is a visible measure of team performance. The workers are multi-skilled and enmeshed in a compact sociotechnical system where workflow is more important than concrete reified products because it has finally been realized that the product is only as good as the process used to develop it.

When we enter into the realm of process, we are really in an arena in which we do not know how to deal with things. We are so used to concentrating on the final result that it is difficult to even see the flow of incremental actions and thoughts that results in the concrete product. Here we need a different non-object centered vocabulary such as that developed by the process philosopher Whitehead, or some other that allows us to deal with streams rather than the reified in products of streams. When we look around for such a viewpoint that will allow us to understand the streams of work and describe them directly, the model that first comes to mind is that developed by the Chinese. Slowly we are realizing that the Western worldview is inadequate for dealing with evolutionary and continuously changing fluctuations in the world. So we are led to look further and further afield for a way to get a handle on these phenomena. The Chinese had in their civilization a whole way of looking at the world that dealt with it in terms of fluctuations instead of reified objects. Here we will introduce a few concepts from archaic Chinese science to show how they apply to our level of personal process articulation.

With every consideration of process the basic question is how to differentiate sub-processes within the overall flux of life. That flux of life will be called here primary process, and the question is how we distinguish other secondary processes that are embedded in primary process. The Chinese had three concepts that they applied to this problem, and we might well learn something from them in our own exploration of process. The three
concepts are SHU (Countability), LI (Principle), and CHI (Flowing Energy). At the level of primary process or the flux of life these three concepts are fused and indistinguishable. The secondary autonomous processes cannot be counted or distinguished, and Principle is the same as the Flowing Energy of the process. The secondary processes appear when a differentiation occurs under the action of the Catalyst. In our case it is the differentiation of kinds of work. This differentiation is subjective and tied to the observer. Each observer can see a different set of process classes in operation depending on his viewpoint. However, once this differentiation occurs and different kinds of workflows are distinguished by an observer, then LI (Principle) becomes distinguished from CHI (Flowing Energy). Note here that it is not a matter of distinguishing process from products. Products are not as crucial in this way of looking at things as it has been in our Western worldview that continuously attempts to reify processes into things. Instead the major differentiation is between the flows of energy and their patterning. The patterning, according to the Chinese, always manifests a principle called LI. In a view of things that sees process as manifestation or presencing, it is the inner coherence of the interweaving processes that is important, not the reification of streams into things.

Let’s take the example of the growth of a tree. A tree grows by manufacturing energy and using it to build up the structure of the tree at a cellular level that hardens into the dead core of the tree. Only the outermost layer of the tree is alive. This process of energy production and transformation into structure using photosynthesis and materials taken from the soil is an ongoing process. But as the process goes, it lays down the rings of the tree layer by layer in a certain specific pattern. This patterning specific to each tree, to each species of tree, to trees in general displays the principle that the tree manifests. Seeing this pattern is of the utmost importance in the Chinese way of looking at things. One is looking not at the products per se, but instead at the total pattern of the interplay of subprocesses as they lay down a pattern in their products. One is focusing on the traces of the intertwining streams of activities, not at individual products as entities separate from each other. The LI or principle is implicit in the patterning of the relation of secondary processes to each other. It is observed in the traces left by these streams in the total pattern of the products. The LI can only be separated out from the Energy Flow of the secondary processes by looking at their interrelationships. The LI can also be inferred by the patterning of all the secondary processes products. If you cannot see secondary processes as countable and separate, then the CHI and LI are indistinguishable. CHI is the energy flows themselves. These energy flows each have their own quality that can be distinguished in the envelope of countability or distinguishability. The quality of the process is something we intuit directly. It is completely subjective even if different people can distinguish the same processes. But it is a very important element of the analysis of processes that we are not used to dealing with. Normally if something is not objective, it is dismissed. However, for process improvement what feels right or looks right is very important. Quality of process is an intangible, and we need to have a way of talking about the differences between these intangibles if we are going to improve process. The idea that metrics is the only way to improve process is completely wrong. It takes into account quantity but not quality. We need a balanced approach to process improvement that equally emphasizes quality and quantity. CHI is merely the evolution or fluctuations of energy that has different qualities within the area demarked as the providence of a particular secondary autonomous process.

Finally we should mention tertiary processes. A tertiary process is embedded in a secondary process in a manner similar to the way secondary is embedded in the primary process. The tertiary processes are important for all technical work since they represent the spinoffs set in motion by the secondary process. For instance, in design process one might start some automated consistency checking tool working on a design representation. The process of consistency checking is done independently by the computer but set in motion and guided by the secondary process. If the secondary process does not set it in motion, guide it and integrate its results, then it merely dies without being able to contribute anything to the overall process. Tertiary processes may be represented by other people. When you get someone to give you a partial solution to help you in your work, then you have set off a tertiary process that must be guided by you and the results of which must be re-assimilated. To that other person though, what you considered a tertiary process may be a secondary process because to them that is their main line of work. Tertiary processes, like secondary processes, are countable, have flowing energy and exhibit the inner coherence of principle, both in the patterning of the interaction of streams and in the patterning of the products. The major difference from the point of view of the observer is whether this is the major ongoing kind of work being done or whether these are spinoffs that represent invocations of other processors.
The Chinese ideas about process are very sophisticated, and this is just a brief introduction. It helps us to see process as a different way of looking at things. We could also use process-based philosophy to make similar points. But the Chinese ideas are simple and practical, whereas within the Western worldview starting from the position of assumed reification (NOUN CENTERED) we need some work to get to a non-reified process view of things (VERB CENTERED). But there is a considerable wisdom in the Chinese view of process. It tells us that we should be looking for the inner coherence of streams of work and not at the products. It says that each stream of work will have its own principle which we are approximating when we isolate it and its own quality of flowing energy. We can expect each kind of work to be experienced quite differently by the practitioners, and that is how they know one from the others. In kinds of work there are no clear cut boundaries. The boundaries are imaginary. But these imaginary boundaries are reinforced by the countable difference between the differentiated kinds of work and the qualitative differences between the streams as well as the different principles being manifested in the patterning of the interaction of the work streams.

Much work needs to be done to come to a full appreciation of the Chinese concepts and their application within our study of processes. Some cultural translation is definitely in order. But the fact that other cultures have succeeded in looking at the world in a non-reified way is encouraging. Our efforts must be toward attempting to find non-reified ways to describe and consider the enaction of processes. This is a difficult task. But it is central to the improvement of process centered working environments. As long as we look at processes from a product orientation, we will be stuck with terms and concepts inapplicable to our process centered way of looking at things. Process centered ways of seeing work is a paradigm shift. It calls for a major reformulation of the way we conceive and do work. If we cannot make this shift, then we will merely trip over our own shoelaces as we apply the old way of doing things to the new tasks before us. This transition to the process centered paradigm will take time and will not be painless, but it signals a major reformulation in the way we construct our sociotechnical system and the working world.

THE PROCESS FUTURE

Process is actually a combination of behavior and functionality within the sociotechnical system. Functionality is really intentionality. All the basic views which apply to software systems may be applied to the work of constructing software. These views are AGENT, FUNCTIONALITY, EVENT, DATA. The traditional life-cycle view of process is event and data centered. Events are milestones, and data is products. The new way of looking at process emphasizes teams of multiple agents and functional (intentional) goals. Kinds of work are merely the functions necessary to produce a product within a certain timeframe. The process paradigm shift now emphasizes teams and work functions rather than spacetime ordering that the old control centered organizations emphasized. To them the process of getting to milestones and producing products was a black box. Now we are emphasizing close cooperation of a team which takes responsibility for all the kinds of work necessary to produce a product using an adaptable spiral lifecycle.

It is important to realize that the agent and functional perspectives are very different. In Maturna’s description of autopoietic systems he clearly distinguishes between organization and structure and between the closed and open aspects of living systems. In his work it is clear that the autonomous individual that expresses self organization (autopoiesis) must be considered as a closed organization; that the openness of systems where that individual reacts to inputs and responds with outputs is only loosely coupled to the closed system of the individual who is organizing itself and maintaining its organization. In such an individual the structural components might change, but the interrelation between those components is maintained. Here an example from work organization might be the relation between work assignments that are different for each individual and the pattern he maintains between his processes for dealing with those work assignments. Clearly the person who is governing his behavior in order to maintain a specific disequilibrium that allows him to continually respond to new work assignments is self organizing. That disequilibrium is the pattern of his personal work processes. The maintenance of the disequilibrium between functions is related to the autonomy of the individual. The individual may be seen as the carrier of this pattern of disequilibrium. The functions themselves are the channels of response to the outside world. The maintenance of the disequilibrium is a closed hysteresis loop which continually attempts to maintain itself or seek a new stable disequilibrium point at which a new pattern of processes would appear.

Thus, there is an inherent tension between the autonomy
of the individual as a closed system and the openness and response to the environment. This tension must be addressed by any theory that attempts to show how process works in the sociotechnical system. The tension between open and closed, between function and autonomy, between process and team structure, defines the interface across which SHU (countableness), LI (principle), and CHI (qualitative flowing energy) show up. We are constantly giving different kinds of work to different people based on skills, and the same kind of work to different people who perform differently. This constant redefinition of who does what and how well is exactly where we see primary process being differentiated from secondary process. When the work is divided and assigned, then the different energies and qualities of the people interact with the different types of work to be done. This generates the differentiations of the qualitative flowing energy of the people doing the different kinds of work in concert. It also manifests in their interaction the combined LI of the people and the LI of the work. If you reassigned the same work to the different people of the same group, you would get a different result.

The basic principle announced by Lo Ch’in-shun in his book Knowledge Painfully Acquired (K’un-chih chi)\(^2\) is that “Principle is one; its particularizations are diverse”. This means that the LI of the people and the LI of the work to be done is ultimately one from the view of primary process. However, as particulars are produced in the stream of process by the agency of specific individuals, diversity is produced which is like the shattering of glass. So give the same work to different people of the same team, and you will get different results. The manifestation of the LI of the people in the team and the team itself and the organization in which the team is embedded is unified. In other words, the manifestation of fused LI and CHI in the whole of the organization is unified by the nature of manifestation. So any one thing you look at mirrors the whole. It is only when we differentiate from our own perspective that we see different secondary processes producing myriad diverse particularizations. The work of secondary process is to attempt to give the individual control over his portion of the process he is embedded in and give him a way to resonate with the rest of the team and organization so that the pattern of the LI is clearer, and the CHI or qualitative energy flows faster and clearer.

The differentiation of LI, CHI, and SHU occurs in the tension between autonomy and intentionality (functionality). The Chinese developed this process view of their world centuries ago. We are just now learning how to look at the world in a similar light. In the future of process development we have a long way to go as ours is a young civilization. But the way is cleared for us to dig deep into the process-oriented way of looking at the world and learn something about our world from the way we envisage our work. This says that the process view of work is the start of a fundamentally different way of conceiving work which will ultimately effect the whole economy and transform our whole culture. Process centered philosophies such as that of Whitehead are not that old, but already we are considering our work differently. Eventually we will realize that we are our work and that our products are not separate from ourselves. And the quality of the products we produce will become an issue of the quality of our lives.

This transformation of our concern for the quality of our products into a concern with the quality of ourselves is an essential change. A change that must begin with the overhaul of the educational system, and end with the revamping of the corporate structures. The changes that are needed to keep America competitive are too numerous to detail here. But the major point is that one of the results of any planned restructuring is the integration and harmonization of the various levels of process identified in this essay. Chang\(^3\) has identified several levels of harmony:

- Logical Consistency
- Interaction
- Mutual Support
- Interpenetration

Proscriptive Process
Breakdown Adaptive Spirals
Personal Process & Autopoietic Rings
Self identification with own process.

The definition of proscriptive processes supplies us only the lowest level of harmony in terms of the logical consistency of the processes with each other and internally. As we apply these processes to the work we do, we see that they interact in complex ways which are many times unpredictable. Thus, adaptive processes are necessary that take into account breakdowns and necessary changes in direction. The Spiral Lifecycle model of Barry Boehm is such an adaptive model for software development. But unless the processes become not only adaptive but also mutually supportive, then there is no real robustness.

---

42. Reference?
43. Reference?
within the processes as they are enacted. I think personal process comes in here as a means of creating dynamic interdependencies between the work which different team members are carrying out asynchronously but in harmony with each other. If each member is looking to redefine the work to make it more efficient, higher quality, with greater impact on the achievement of mutual goals, then through mutual negotiation a work structure will coevolve into something unexpected by everyone. Work which is laid out in advance as immutable is a roadblock to coevolution. Coevolution assumes ongoing real-time co-adaptation by people redefining the work together.

Redefining work is in someway a redefinition of ourselves. Thus, the motivational layer which appears in the lines of work and going concern ultimately needs to be integrated into our view of processes. They are OUR processes, not something imposed from without, but something we are creating ourselves as the means of defining our world. We are the world we define. So it is necessary to align our work processes with our own goals. This is difficult to do as long as they are imposed on us. But many companies are discovering that encouraging internal entrepreneurs allows them to maximize the relation between their employees, goals and the company goals. Employees are seen as owners, not as an external resource. Employees assume ownership and the rewards and risks are assumed by empowered individuals. This fosters an in-depth commitment beyond what is normally possible to achieve in control-oriented companies which view their employees as expendable resources.

The harmonization of all the levels of process is the ultimate goal of process. Process can be optimized as an external or objective feature of the organization only to a certain extent. In order to achieve the greatest possible optimization, the view of process as an external or objective thing must be replaced with the view that process goes to the very heart of who we are and that our work is the creative expression of who we are as individuals, teams, and organizations. This means changing how we see ourselves and each other. The utilitarian approach to others now dominant in our corporate culture can only reach the interaction level of harmony. In order to go beyond that we must be able to become mutually supportive and finally see each other as completely implicated in what each of us does separately. Interpenetration means that what I do is not just supporting what you do, but that it is recognized as totally intertwined. At that point the line of work and going concern itself becomes the process which is completely enmeshed, but not as a hidden undergrowth. Rather the going concern and line of work is the organization and the individuals creating themselves anew over and over again as their work adapts and is completely transformed by redefinition. When we think in terms of continual redefinition of work as a result of personal process, it is clear that we are ourselves a product of the work we define for ourselves. Thus, redefinition of our work gives us the power over our situation we normally lack in the workplace. Empowerment of individuals to control their own processes makes them able to express themselves through their work. They are more creative, and are thus more fulfilled. There is less loss of productivity because of lack of motivation.

The future of process is a fundamental transformation in how we view ourselves and work. It is our challenge to navigate that transformation and arrive at a process that is harmonized. Process harmony requires a depth which is uncommon. However, unless we attempt to attain that goal, we will never succeed, and our efforts in the process arena will remain hollow and superficial. The development of process wisdom is the ultimate result of process harmony. Process wisdom comes from the continual attempt to attain process harmony. It is the kind of wisdom that the Chinese were able to develop before us. The age of their civilization attests to the value of that wisdom. The fact that we were able to destroy that civilization does not mean we Westerners are capable of producing a long living civilization ourselves. The process initiative is the first step toward understanding the ecosystem of the sociotechnical system. Whether that ecological view will allow sociotechnical systems to survive is a question still to be answered. But the future looks bleak unless it is possible to develop some process wisdom along the way.

So many people when they hear about process say, “We talked about all this before and nothing has ever changed.” They say, “We are still inventing our processes anew with every project.” They also say, “We never have enough money to do it right but always enough to do it over and over.” Software process improvement needs to address these comments directly. It can only be done by just saying, “You own your process.” No one can fix someone else’s process, except superficially. Invent your process anew if it saves money, time, or improves quality. But do not reinvent when it costs more for no benefit. Then go beyond that to say, “You ARE your process.” This is the only way that real progress will be made in the trade-off between quality, and productivity. Optimization from
outside is only a half measure. Optimization from inside is the key to real efficiency.

**ABSTRACT:**

This paper begins with example company division initiatives and goes on to explore the central issues in software process development with a view to uncovering possible future solutions.

**KEYWORDS:**

Key words: Software Engineering, Software Process, Sociology of Technology, Autopoiesis, Software Methodology, Sociotechnical Systems, Software Assessment
**Personal Process Notes**

**Process Name:** A-process  

**Guidance:**

*Formal process description, if any.*

---

**Inputs:**

*Identify known inputs to process.*

**Outputs:**

*Identify known outputs to process.*

**Steps:**

*Identify known steps in process.*

**Issues:**

*Identify known issues that arise in process.*

**Notes:**
Problems:
problems encountered

Breakdown:
What breakdown occurred and when?

Are the results in line with what is already known?
Apply identified criteria

What issues need to be considered with others?

What is the quality of the final product?

Is representation and method use self-consistent?

Given results of this work and other previous work, what new has been learned about the whole?

Make sure representation reflects reality.

Look at completed elaboration for fulfilment of initial goals.

Use method to capture discoveries and complete the coverage of the territory.

Apply method in an initial discovery pass.

Select & test method.

* formalism

* exploration

* elaboration

* assessment

* verification

* inference

* evaluation

* deliberation

* validation

Back
Personal Process Planning

Concurrent Processes:

- A-process
- B-process
- C-process
- D-process

Interrupts:

*source -- subject*

Work Assignments:

1 -- assignment
2 -- assignment
3 -- assignment
Five phases derived from design viewpoints plus one.

FIGURE 28
Five viewpoints:

Each has its own independent lifecycle. One would expect a different modality of Being to be emphasised in each phase of these mini-lifecycles.

<table>
<thead>
<tr>
<th>KIND OF BEING</th>
<th>MODALITY</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pure Presence</td>
<td>Present-at-hand</td>
</tr>
<tr>
<td>Process Being</td>
<td>Ready-to-hand</td>
</tr>
<tr>
<td>Hyper-Being</td>
<td>In-hand</td>
</tr>
<tr>
<td>Wild Being</td>
<td>Out-of-hand</td>
</tr>
</tbody>
</table>

FIGURE 29

Each has its own independent lifecycle. One would expect a different modality of Being to be emphasised in each phase of these mini-lifecycles.
Possible Minimal Methods associated with the Catalytic viewpoint

- **Agent**
  - Category Theory
  - All possible agent embeddings
  - Network Theory
  - All possible relations between agents

- **Catalyst**
  - Logic of Form

- **Function**
  - Lano N2
  - All possible relations between functions
  - Facet Analysis 2N
  - All possible decompositions of functions

- **Data**
  - Fractals
  - All possible data decompositions
  - Information Theory
  - All possible relations between data

- **Event**
  - Temporal Logic
  - All possible relations between events
  - Combinatorics
  - All possible event decompositions

**FIGURE 30**
FIVE HSING

Catalyst

Earth

Water

Metal

Event

Fire

Full order

Partial order

Partial order

Full order

Control cycle

Production cycle

Wood

Partial order

Intentionaliy

Function

Agent

Time

Space

Intentionality

Production cycle

Full order

Figure 31
FOUR FACES OF EVERY MINIMAL SYSTEM

- LATTICE STRUCTURE
- TETRAHEDRON
- KNOT
- LOCAL DIFFERENCE / GLOBAL CONTINUITY
- MOBIUS STRIP
- RING STRUCTURE
- TORUS
- INTERFERENCE

FIGURE 32
APPENDIX 1

Graphical Depiction of

for Software Design Minimal Methods
Agent ~ Data minimal methods and conceptual core

- Tasking Architecture minimal method defined by Gomaa in DARTS and elaborated by SPC/Gomaa in ADARTS.

- Monitor minimal method defined by Hoare.

- These two minimal methods form a two way bridge.
Function ~ Agent minimal methods and conceptual core

- Mapping minimal method defined by Mellor/Ward in STRUCTURED DEVELOPMENT FOR REAL-TIME SYSTEMS Volume 3

- Virtual Layered Machine minimal method defined by Nielsen/Shumate in DESIGNING LARGE REAL-TIME SYSTEMS WITH ADA. Method called VLM/OOD. [See also OOSD by Constantine/Wasserman.]

- Methods do not combine to form a two way bridge.

- Tasking structure mediates between these two minimal methods.

- Lowest level virtual machine instructions are methods of objects.

---

![Diagram: FUNCTION ~ AGENT conceptual core](image)

**FUNCTION what**
function views agent as vehicles for system functionality

**AGENT who**
agent views function as virtual machine instructions performed

**IMPLEMENTATION MODEL**

- mapping of function task

- Tasking Structure

- Instruction

- Call Structure

- nested tasks

- lower level

- virtual machine

- method

- object

- object

**FIGURE 9**

**FIGURE 10**

The diagram illustrates the conceptual core of functionality allocation, where an agent is considered as a vehicle for functionality. The virtual machine instruction embodies the functionality considered in terms of implementation.
Function ~ Data minimal methods and conceptual core

- Dataflow minimal method defined by Yourdon/DeMarco
- Object transform minimal method defined by Guttag/Liskov
- Transform between these two methods demonstrated by Vaclav Rajlich
- Two minimal methods form a conceptual bridge.

**FUNCTION ~ DATA conceptual core**

tracking data movement between storage and through transforming functions

Dataflow Arrow

Function Bubble

Data Store

Inhibition Signal (mode)

Function considered externally

Data considered externally

D backbox

K backbox

W backbox

tracking data movement and through transforming functions

data views function as transforming method operating on persistent data

where

DATA

FIGURE 11

FIGURE 12
Event ~ Function minimal methods and conceptual core

- State machine minimal method defined by Ward/ Mellor and Hatley/ Pribhai

- PetriNet method defined by Petri and has been developed by many others.

- Two methods do not form a two way bridge, but they are closely related by the presence of states and actions in both methods.
Event ~ Data minimal methods and conceptual core

- Design element flow diagram sees design elements as flowing through the states of the application instead of as a clockwork mechanism. Not defined as a standard method. Allows coordination of state machines at different system levels.

- Data mutation sees the changes in individual data items as the program executes. This is the most basic method for all computer programming. Can either track information flow through the system or compare cross sections of different information flows.

- These minimal methods do not form a two way bridge.

EVENT ~ DATA conceptual core

- the content of the data structure that changes

  $D_{whitebox}$

  Data Content

  $Y$

  Trigger (exciting transience)

  Persistence State
  what lasts between transients

  $E_{link}$

  event as pure transience

**FIGURE 15**

EVENT
when
digital view

event views data as flowing design elements

DESIGN ELEMENT FLOW

observer watching data will raise trigger when predefined threshold crossed

increment counter

counter static

nil counter

s1

set counter

s2

observer

trigger

print changes in variable each iteration

DATA MUTATION

data views event as changes in data value

analog view

where

**FIGURE 16**
Event ~ Agent minimal methods and conceptual core

- Worldline minimal method defined by Agha in ACTORS.
- Scenario minimal method used in practice but not explicitly defined as a formal method.
- These two minimal methods form a two way bridge.

**FIGURE 17**

**FIGURE 18**

EVENT ~ AGENT

conceptual core

illusionary continuity

Continuity of Agents

Sequence of Events

Relativity in networked systems

Ender relational concepts in networked systems

Alink Communication Channel between Agents

whitebox

may be decomposed into smaller and smaller events

EVENT when

agent views events as worldline

scenario

worldline

agent1 agent2 agent3 agent4 agent5

who

AGENT

event views agent as scenarios
APPENDIX 2a

Entity Relationship statements
for Core Concepts in the
Software Design Minimal Methods

Appendix 2b: Core ER Diagrams

Appendix 2c: Core ER Concordance
Method has Name (Variable). [n001]
Method has FromViewPoint (Variable). [n002]
Method has ToViewPoint (Variable). [n003]
Method may have Dual (Variable). [n004]

StateMachine is_a Method. [n005]
StateMachine has CurrentState (Variable). [n006]
StateMachine has StateVectors (List). [n007]
StateMachine has Mode (Variable). [n008]
StateMachine has SMInput (Parameter). [n009]
StateMachine has SMOutput (Parameter). [n010]
StateVector has Variables (Set). [n011]
StateVector has Event (Variable). [n012]
StateVector has NowCurrentState (Variable). [n013]
StateVector has NextState (Variable). [n014]
StateVector has Action (Variable). [n015]
Action is_a FunctionName.
StateVector is_a Trigger for Function. [n017]
NextState maps_to CurrentState. [n018]
CurrentState maps_to NowCurrentState. [n019]
Event maps_to SMInput. [n020]
FunctionName maps_to SMOutput. [n021]
Mode maps_to StateVectors. [n022]
StateMachine has Operations (Function List). [n023]
GetCurrentState(OUT CurrentState) is_a Operation. [n024]
SetInitialState(IN CurrentState) is_a Operation. [n025]
SetInitialVectorList(IN StateVectors) is_a Operation. [n026]
GetAction(IN SMInput-> Event; OUT SMOutput-> Action) is_a Operation. [n027]

Petrinet is_a Method. [n028]
Petrinet has PetriMatrix. [n029]
PetriMatrix has Places (List). [n030]
PetriMatrix has Transits (List). [n031]
PetriMatrix has PetriArcs (List). [n032]
Petrinet has Markers (List). [n033]
Places has Place (Variable). [n034]
Transits has Transit (Function). [n035]
PetriArcs has PetriArc (Relation) from Place to Transit. [n036]
Markers is_a Tokens (Type). [n037]
PetriArcs maps_to Places. [n038]
PetriArcs maps_to Transits. [n039]
Markers propagate_along PetriArcs. [n040]
Markers has Color (Type). [n041]
Transit has PetriRules (List). [n042]
Transit has InputPlaces (List). [n043]
PetriRule has LeftHandSide. [n044]
PetriRule has RightHandSide. [n045]
LeftHandSide has Marker Colors in InputPlaces. [n046]
RightHandSide triggers Transit. [n047]
Petrinet has Operations (List). [n048]
FirePetriNet(null) is_a Operation. [n049]

Decomposition is_a Method. [n050]
Decomposition has Mode (Type). [n051]
Decomposition has Externals (source or sink). [n052]
Decomposition has Context (Bubble). [n053]
Context has Bubbles (Function). [n054]
Decomposition has DataArcs (Relation). [n055]
Decomposition has DataStore (Holder). [n056]
Bubbles decompose_into Bubbles (List). [n057]
Bubble has Mode. [n058]
DataItems has DataItem (Variable). [n059]
Bubble has ControlArc (Relation). [n060]
Bubble has ControlSpec (Holder). [n061]
DataArcs maps_to Bubbles. [n062]
DataArcs maps_to DataStore. [n063]
DataArcs maps_to Externals. [n064]
Bubble has InputDataArcs (Parameters). [n065]
Bubble has OutputDataArcs (Parameters). [n066]
When InputDataArcs present Bubble is Triggered. [n067]
ControlSpec may_have StateMachine. [n068]
ControlSpec may_have DecisionTable. [n069]
ControlSpec may_have ProcessActivationTable. [n070]
ControlSpec may_have PetriNet. [n071]
ControlSpec establishes Mode. [n072]
Function may_have Loop. [n073]
Function may have Selector. [n074]
Function may have Equation. [n075]
Function may have Rule. [n076]
Function has atleast one Assignment.

Articulation is a Method. [n078]
Articulation has Situation. [n079]
Situation has Entities with RelationArcs. [n080]
Articulation has DataDictionary (List). [n081]
DataDictionary has Entity. [n082]
DataDictionary has RelationArc (Relation). [n083]
RelationArcs map to Entities from Entities. [n084]
Entity decomposes into Entities (List). [n085]
Entity has DataStore (Holder). [n086]
Entity has Operations (Function). [n087]
DataStores has DataItems (List). [n088]
Operations modify DataItems. [n089]
DataItems has DataTypes (Type). [n090]
RelationArc has RelationAttributes (Variable). [n091]

DARTS is a method. [n092]
DARTS has DistributedDesign. [n093]
DARTS has ConcurrentDesign. [n094]
DARTS has CommunicationsChannels (List). [n094]
DistributedDesign has ProcessingElements. [n096]
ConcurrentDesign has ProcessingElements. [n097]
ProcessorArrays is a ProcessingElement. [n098]
ProcessorArrays decompose into ProcessorArrays. [n099]
ProcessorArrays has Processors. [n100]
Task is a ProcessingElement. [n101]
Processors has Tasks. [n102]
Tasks decompose into Tasks. [n103]
CommunicationChannel has CommunicationMechanism. [n104]
CommunicationMechanism may be Queue. [n105]
CommunicationMechanism may be Rendezvous. [n106]
CommunicationMechanism may be Semaphore. [n107]
CommunicationMechanism may be Flag. [n108]
CommunicationMechanism may be Variable. [n109]
CommunicationChannel has Protocol. [n110]
CommunicationChannel has DataArcs. [n111]
Protocol has Messages. [n112]
Message has Datitems. [n113]
Protocol has Sender StateMachine. [n114]
Protocol has Receiver StateMachine. [n115]
DARTS has Monitors. [n116]
Monitor has DataStore. [n117]
Monitor has Semaphore. [n118]
Task receives Message from CommunicationChannel. [n119]
Task has Function. [n120]
Function maps to ProcessingElements. [n121]
Task has Selector of Function. [n122]
Task has ExecutiveLoop. [n123]

Allocation is_a Method. [n124]
Allocation has FunctionalMappings (List). [n125]
FunctionalMapping depends_on SystemMode (Variable). [n126]
FunctionalMapping has Functions (List). [n127]
FunctionalMapping has ProcessingElements (List). [n128]
FunctionalMapping has FunctionalArc (List of Relations). [n129]
FunctionalArc maps_to ProcessingElement from Function. [n130]

VirtualMachine is_a Method. [n131]
VirtualMachine decomposes_into VirtualMachines. [n132]
VirtualMachine has Instructions. [n133]
VirtualMachine may_have StateMachine. [n134]
Instruction is_a Function. [n135]

WorldLine is_a Method. [n136]
Worldline has Messages (List) associated with one ProcessingElement. [n137]

Scenario is_a Method. [n138]
Scenario has causally related Messages (List) between ProcessingElements. [n39]

DesignElementFlow is_a Method. [n140]
DesignElementFlow has DesignElements. [n141]
DesignElementFlow has System. [n142]
System has System. [n143]
System has StateMachine. [n144]
DesignElement has StateMachine. [n145]
System States maps_to DesignElement States. [n146]
System Actions maps_to DesignElement Actions. [n147]
DesignElement Actions maps_to System States. [n148]
System Actions maps_to DesignElement States. [n149]

InformationFlow is_a Method. [n150]
InformationFlow has Variables. [n151]
InformationFlow has Datum. [n152]
InformationFlow has SynchronicMapping. [n153]
InformationFlow has DiachronicMapping. [n154]
In Synchronic mapping the values in a set of variables relate to a single timespan. [n155]
In Diachronic mapping the datum over time moves through a set of variables. [n156]
Datum maps_to Variables. [n157]
Variables maps_to Variables via Datum. [n158]

Temporality is_a Method. [n159]
Temporality has a Sheaf (Holder). [n160]
Sheaf has Bundles (Holder). [n161]
Sheaf has SignalArc (Relation). [n162]
SignalArc maps_to Signal from Signal. [n163]
SignalArc has IntervalConstraints. [n164]
IntervalConstraint may_be Before. [n165]
IntervalConstraint may_be After. [n166]
IntervalConstraint may_be During. [n167]
IntervalConstraint may_be Startings. [n168]
IntervalConstraint may_be Finishes. [n169]
IntervalConstraint may_be Overlapping. [n170]
IntervalConstraint may_be Meets. [n171]
IntervalConstraint may_be Equals. [n172]
Bundle has Signals (List). [n173]
Signal has Interval (List). [n174]
Signal has Lacune (List). [n175]
Bunch has Intervals (List). [n176]
Interval has Duration (Variable). [n177]
Interval has Event (String). [n178]
Interval has State (String). [n179]
Interval decomposes into Intervals (List). [n180]
APPENDIX 2b

Entity Relationship diagrams
for Core Concepts in the
Software Design Minimal Methods
One minimal method looks at event from point of view of process.
One minimal method looks at process from point of view of event.
Three minimal methods represented:
Two minimal methods looks at data from viewpoint of process and vice versa. Basic structure is dataflow diagram. Control added by Hatley to interface with control structures. Decomposition of bubbles constitutes a third minimal method that looks at function alone.
One minimal method looks data in isolation.
Three minimal methods represented:
Two minimal methods looks at data from viewpoint of agent and vice versa. Decomposition of tasks constitutes a third minimal method that looks at agent alone.
One minimal method looks at agent from point of view of process.
One minimal method looks at process from point of view of agent.
One minimal method looks at an event from the point of view of an agent.
One minimal method looks at agent from point of view of event.
One minimal method looks at data from point of view of event.
One minimal method looks at event from point of view of data.
Temporality \(\text{is}_a\) Method

\[
\begin{align*}
\text{has} \\
\text{Sheaf} \\
\text{Bundle(s)} \\
\text{SignalArc(s)} \\
\text{Signal} \\
\text{Relation} \\
\text{Interval(s)} \\
\text{IntervalConstraint} \\
\text{Event} \\
\text{State} \\
\text{Duration}
\end{align*}
\]

One minimal method looks at event alone.
APPENDIX 2c

Entity Relationship concordance

for Core Concepts in the

Software Design Minimal Methods
Action (Variable). [n015]; StateVector has . . .
Action is_a FunctionName. [n16]
After. [n166]; IntervalConstraint may_be . . .
Allocation has FunctionalMappings (List). [n125]
Allocation is_a Method. [n124]
Articulation has DataDictionary (List). [n081]
Articulation has Situation. [n079]
Articulation is_a Method. [n078]
Assignment. ; Function has atleast one . . .
Before. [n165]; IntervalConstraint may_be . . .
Bubble has ControlArc (Relation). [n060]
Bubble has ControlSpec (Holder). [n061]
Bubble has InputDataArcs (Parameters). [n065]
Bubble has Mode. [n058]
Bubble has OutputDataArcs (Parameters). [n066]
Bubble is Triggered. [n067]; When InputDataArcs present . . .
Bubbles (Function). [n054]; Context has . . .
Bubbles (List). [n057]; Bubbles decompose_into . . .
Bubbles decompose_into Bubbles (List). [n057]
Bubbles. [n062]; DataArcs maps_to . . .
Bunch has Intervals (List). [n176]
Bundle has Signals (List). [n173]
Bundles (Holder). [n161]; Sheaf has . . .
Color (Type). [n041]; Markers has . . .
CommunicationChannel has CommunicationMechanism. [n104]
CommunicaitonChannel has DataArcs. [n111]
CommunicationChannel has Protocol. [n110]
CommunicationChannel. [n119]; Task receives . . . ; Message from . . CommunicationChannel
CommunicationMechanism may_be Flag. [n108]
CommunicationMechanism may_be Queue. [n105]
CommunicationMechanism may_be Rendezvous. [n106]
CommunicationMechanism may_be Semaphore. [n107]
CommunicationMechanism may_be Variable. [n109]
CommunicationMechanism. [n104]; CommunicationChannel has . . .
CommunicationsChannels (List). [n094]; DARTS has . . .
ConcurrentDesign has ProcessingElements. [n097]
ConcurrentDesign. [n094]; DARTS has . . .
Context (Bubble). [n053]; Decomposition has . . .
Context has Bubbles (Function). [n054]
ControlArc (Relation). [n060]; Bubble has . . .
ControlSpec (Holder). [n061]; Bubble has . . .
ControlSpec establishes Mode. [n072]
ControlSpec may_have DecisionTable. [n069]
ControlSpec may_have PetriNet. [n071]
ControlSpec may_have ProcessActivationTable. [n070]
ControlSpec may_have StateMachine. [n068]
CurrentState (Variable). [n006]; StateMachine has . . .
CurrentState maps_to NowCurrentState. [n019]
CurrentState. [n018]; NextState maps_to . . .
DARTS has CommunicationsChannels (List). [n094]
DARTS has ConcurrentDesign. [n094]
DARTS has DistributedDesign. [n093]
DARTS has Monitors. [n116]
DARTS is_a method. [n092]
DataArcs (Relation). [n055]; Decomposition has . . .
DataArcs maps_to Bubbles. [n062]
DataArcs maps_to DataStore. [n063]
DataArcs maps_to Externals. [n064]
DataArcs. [n111]; CommunicationChannel has . . .
DataDictionary (List). [n081]; Articulation has . . .
DataDictionary has Entity. [n082]
DataDictionary has RelationArc (Relation). [n083]
DataItem (Variable). [n059]; DataItems has . . .
DataItems (List). [n088]; DataStores has . . .
DataItems has DataItem (Variable). [n059]
DataItems has DataTypes (Type). [n090]
DataItems. [n089]; Operations modify . . .
DataItems. [n113]; Message has . . .
DataStore (Holder). [n056]; Decomposition has . . .
DataStore (Holder). [n086]; Entity has . . .
DataStore. [n063]; DataArcs maps_to . . .
DataStore. [n117]; Monitor has . . .
DataStores has DataItems (List). [n088]
DataTypes (Type). [n090]; DataItems has . . .
Datum maps_to Variables. [n157]
Datum. [n152]; InformationFlow has . . .
DecisionTable. [n069]; ControlSpec may_have . . .
Decomposition has Context (Bubble). [n053]
Decomposition has DataArcs (Relation). [n055]
Decomposition has DataStore (Holder). [n056]
Decomposition has Externals (source or sink). [n052]
Decomposition has Mode (Type). [n051]
Decomposition is_a Method. [n050]
DesignElement Actions. [n147]; System Actions maps_to . . .
DesignElement Actions maps_to System States. [n148]
DesignElement States. [n149]; System Actions maps_to . . .
DesignElement.States. [n146]; System.States maps_to . . .
DesignElementFlow has DesignElements. [n141]
DesignElementFlow has System. [n142]
DesignElementFlow is_a Method. [n140]
DesignElements. [n141]; DesignElementFlow has . . .
DesignElment has StateMachine. [n145]
DiachronicMapping. [n154]; InformationFlow has . . .
DistributedDesign has ProcessingElements. [n096]
DistributedDesign. [n093]; DARTS has . . .
Dual (Variable). [n004]; Method may_have . . .
Duration (Variable). [n177]; Interval has . . .
During. [n167]; IntervalConstraint may_be . . .
Entities (List). [n085]; Entity decomposes_into . . .
Entities from Entities. [n084]; RelationArcs map_to . . .
Entities with RelationArcs. [n080]; Situation has . . .
Entity decomposes_into Entities (List). [n085]
Entity has DataStore (Holder). [n086]
Entity has Operations (Function). [n087]
Entity. [n082]; DataDictionary has . . .
Equals. [n172]; IntervalConstraint may_be . . .
Equation. [n075]; Function may_have . . .
Event (String). [n178]; Interval has . . .
Event (Variable). [n012]; StateVector has . . .
Event maps_to SMInput. [n020]
ExecutiveLoop. [n123]; Task has . . .
Externals (source or sink). [n052]; Decomposition has . . .
Externals. [n064]; DataArcs maps_to . . .
Finishes. [n169]; IntervalConstraint may_be . . .
FirePetriNet(null) is_a Operation. [n049]
Flag. [n108]; CommunicationMechanism may_be . . .
FromViewPoint (Variable). [n002]; Method has . . .
Function has atleast one Assignment.
Function maps_to ProcessingElements. [n121]
Function may_have Equation. [n075]
Function may_have Loop. [n073]
Function may_have Rule. [n076]
Function may_have Selector. [n074]
Function. [n120]; Task has . . .
Function. [n122]; Task has . . .; Selector of . .
Function. [n135]; Instruction is_a . . .
FunctionName maps_to SMOutput. [n021]
FunctionName. ; Action is_a . . .
FunctionalArc (List of Relations). [n129]; FunctionalMapping has . . .
FunctionalArc maps_to ProcessingElement from Function. [n130]
FunctionalMapping depends_on SystemMode (Variable). [n126]
FunctionalMapping has FunctionalArc (List of Relations). [n129]
FunctionalMapping has Functions (List). [n127]
FunctionalMapping has ProcessingElements (List). [n128]
FunctionalMappings (List). [n125]; Allocation has . . .
Functions (List). [n127]; FunctionalMapping has . . .
GetAction(IN.SMInput->.Event;.OUT.SMOutput->.Action) is_a Operation. [n027]
GetCurrentState(OUT CurrentState) is_a Operation. [n024]
In Diachronic mapping the datum over time moves through a set of variables. [n156]
In Synchronic mapping the values in a set of variables relate to a single timespan. [n155]
InformationFlow has Datum. [n152]
InformationFlow has DiachronicMapping. [n154]
InformationFlow has SynchronicMapping. [n153]
InformationFlow has Variables. [n151]
InformationFlow is_a Method. [n150]
InputDataArcs (Parameters). [n065]; Bubble has . . .
InputPlaces (List). [n043]; Transit has . . .
Instruction is_a Function. [n135]
Instructions. [n133]; VirtualMachine has . . .
Interval (List). [n174]; Signal has . . .
Interval decomposes_into Intervals (List). [n180]
Interval has Duration (Variable). [n177]
Interval has Event (String). [n178]
Interval has State (String). [n179]
IntervalConstraint may_be After. [n166]
IntervalConstraint may_be Before. [n165]
IntervalConstraint may_be During. [n167]
IntervalConstraint may_be Equals. [n172]
IntervalConstraint may_be Finishes. [n169]
IntervalConstraint may_be Meets. [n171]
IntervalConstraint may_be Overlapping. [n170]
IntervalConstraint may_be Starts. [n168]
IntervalConstraints. [n164]; SignalArc has . . .
Intervals (List). [n176]; Bunch has . . .
Intervals (List). [n180]; Interval decomposes_into . . .
Lacune (List). [n175]; Signal has . . .
LeftHandSide has Marker Colors in InputPlaces. [n046]
LeftHandSide. [n044]; PetriRule has . . .
Loop. [n073]; Function may_have . . .
Marker Colors in InputPlaces. [n046]; LeftHandSide has . . .
Markers (List). [n033]; Petrinet has . . .
Markers has Color (Type). [n041]
Markers is_a Tokens (Type). [n037]
Markers propagate_along PetriArcs. [n040]
Meets. [n171]; IntervalConstraint may_be . . .
Message from CommunicationChannel. [n119]; Task receives . . .
Message has DataItems. [n113]
Messages. [n112]; Protocol has . . .
Messages.(List) associated_with_one ProcessingElement. [n137]; Worldline has . . .
Messages.(List) between ProcessingElements. zn39]; Scenario has_causally_related . . .
Method has FromViewPoint (Variable). [n002]
Method has Name (Variable). [n001]
Method has ToViewPoint (Variable). [n003]
Method may_have Dual (Variable). [n004]
Method. [n092]; DARTS is_a . . .
Method. [n005]; StateMachine is_a . . .
Method. [n028]; Petrinet is_a . . .
Method. [n050]; Decomposition is_a . . .
Method. [n078]; Articulation is_a . . .
Method. [n124]; Allocation is_a . . .
Method. [n131]; VirtualMachine is_a . . .
Method. [n136]; WorldLine is_a . . .
Method. [n138]; Scenario is_a . . .
Method. [n140]; DesignElementFlow is_a . . .
Method. [n150]; InformationFlow is_a . . .
Method. [n159]; Temporality is_a . . .
Mode (Type). [n051]; Decomposition has . . .
Mode (Variable). [n008]; StateMachine has . . .
Mode maps_to StateVectors. [n022]
Mode. [n058]; Bubble has . . .
Mode. [n072]; ControlSpec establishes . . .
Monitor has DataStore. [n117]
Monitor has Semaphore. [n118]
Monitors. [n116]; DARTS has . . .
Name (Variable). [n001]; Method has . . .
NextState (Variable). [n014]; StateVector has . . .
NextState maps_to CurrentState. [n018]
NowCurrentState (Variable). [n013]; StateVector has . . .
NowCurrentState. [n019]; CurrentState maps_to . . .
Operation. [n024]; GetCurrentState(OUT CurrentState) is_a . . .
Operation. [n025]; SetInitialState(IN.CurrentState) is_a . . .
Operation. [n026]; SetInitialVectorList(IN.StateVectors) is_a . . .
Operation. [n049]; FirePetriNet(null) is_a . . .
Operations (Function List). [n023]; StateMachine has . . .
Operations (Function). [n087]; Entity has . . .
Operations (List). [n048]; Petrinet has . . .
Operations modify DataItems. [n089]
OutputDataArcs (Parameters). [n066]; Bubble has . . .
Overlapping. [n170]; IntervalConstraint may_be . . .
PetriArc (Relation) from Place to Transit. [n036]; PetriArcs has . . .
PetriArcs (List). [n032]; PetriMatrix has . . .
PetriArcs has PetriArc (Relation) from Place to Transit. [n036]
PetriArcs maps_to Places. [n038]
PetriArcs maps_to Transits. [n039]
PetriArcs. [n040]; Markers propagate_along . . .
PetriMatrix has PetriArcs (List). [n032]
PetriMatrix has Places (List). [n030]
PetriMatrix has Transits (List). [n031]
PetriMatrix. [n029]; Petrinet has . . .
PetriNet. [n071]; ControlSpec may_have . . .
PetriRule has LeftHandSide. [n044]
PetriRule has RightHandSide. [n045]
PetriRules (List). [n042]; Transit has . . .
Petrinet has Markers (List). [n033]
Petrinet has Operations (List). [n048]
Petrinet has PetriMatrix. [n029]
Petrinet is_a Method. [n028]
Place (Variable). [n034]; Places has . . .
Places (List). [n030]; PetriMatrix has . . .
Places has Place (Variable). [n034]
Places. [n038]; PetriArcs maps_to . . .
ProcessActivationTable. [n070]; ControlSpec may_have . . .
ProcessingElement from Function. [n130]; FunctionalArc maps_to . . .
ProcessingElement. [n098]; ProcessorArrays is_a . . .
ProcessingElement. [n101]; Task is_a . . .
ProcessingElement. [n137]; Worldline has . . .; Messages.(List) associated_with_one . . .
ProcessingElements (List). [n128]; FunctionalMapping has . . .
ProcessingElements. [n096]; DistributedDesign has . . .
ProcessingElements. [n097]; ConcurrentDesign has . . .
ProcessingElements. [n121]; Function maps_to . . .
ProcessingElements. zn39]; Scenario has_causally_related . . .; Messages.(List) between . . .
ProcessorArrays decompose_into ProcessorArrays. [n099]
ProcessorArrays has Processors. [n100]
ProcessorArrays is_a ProcessingElement. [n098]
ProcessorArrays. [n099]; ProcessorArrays decompose_into . . .
Processors has Tasks. [n102]
Processors. [n100]; ProcessorArrays has . . .
Protocol has Messages. [n112]
Protocol has Receiver StateMachine. [n115]
Protocol has Sender StateMachine. [n114]
Protocol. [n110]; CommunicationChannel has . . .
Queue. [n105]; CommunicationMechanism may_be . . .
Receiver StateMachine. [n115]; Protocol has . . .
RelationArc (Relation). [n083]; DataDictionary has . . .
RelationArc has RelationAttributes (Variable). [n091]
RelationArcs map_to Entities from Entities. [n084]
RelationArcs. [n080]; Situation has . . .; Entities with . . .
RelationAttributes (Variable). [n091]; RelationArc has . . .
Rendezvous. [n106]; CommunicationMechanism may_be . . .
RightHandSide triggers Transit. [n047]
RightHandSide. [n045]; PetriRule has . . .
Rule. [n076]; Function may_have . . .
SMInput (Parameter). [n009]; StateMachine has . . .
SMInput. [n020]; Event maps_to . . .
SMOutput (Parameter). [n010]; StateMachine has . . .
SMOutput. [n021]; FunctionName maps_to . . .
Scenario has_causally_related Messages (List) between ProcessingElements.
Scenario is_a Method. [n138]
Selector of Function. [n122]; Task has . . .
Selector. [n074]; Function may_have . . .
Semaphore. [n107]; CommunicationMechanism may_be . . .
Semaphore. [n118]; Monitor has . . .
Sender StateMachine. [n114]; Protocol has . . .
SetInitialState(IN.CurrentState) is_a Operation. [n025]
SetInitialVectorList(IN.StateVectors) is_a Operation. [n026]
Sheaf (Holder). [n160]; Temporality has . . .
Sheaf has Bundles (Holder). [n161]
Sheaf has SignalArc (Relation). [n162]
Signal from Signal. [n163]; SignalArc maps_to . . .
Signal has Interval (List). [n174]
Signal has Lacune (List). [n175]
SignalArc (Relation). [n162]; Sheaf has . . .
SignalArc has IntervalConstraints. [n164]
SignalArc maps_to Signal from Signal. [n163]
Signals (List). [n173]; Bundle has . . .
Situation has Entities with RelationArcs. [n080]
Situation. [n079]; Articulation has . . .
Starts. [n168]; IntervalConstraint may_be . . .
State (String). [n179]; Interval has . . .
StateMachine has CurrentState (Variable). [n006]
StateMachine has Mode (Variable). [n008]
StateMachine has Operations (Function List). [n023]
StateMachine has SMInput (Parameter). [n009]
StateMachine has SMOutput (Parameter). [n010]
StateMachine has StateVectors (List). [n007]
StateMachine is_a Method. [n005]
StateMachine. [n068]; ControlSpec may_have . . .
StateMachine. [n134]; VirtualMachine may_have . . .
StateMachine. [n144]; System has . . .
StateMachine. [n145]; DesignElement has . . .
StateMachine has Action (Variable). [n015]
StateMachine has Event (Variable). [n012]
StateMachine has NextState (Variable). [n014]
StateMachine has NowCurrentState (Variable). [n013]
StateMachine has Variables (Set). [n011]
StateMachine is_a Trigger for Function. [n017]
StateVectors (List). [n007]; StateMachine has . . .
StateVectors. [n022]; Mode maps_to . . .
SynchronicMapping. [n153]; InformationFlow has . . .
System Actions maps_to DesignElement Actions. [n147]
System Actions maps_to DesignElement States. [n149]
System has StateMachine. [n144]
System has System. [n143]
System. [n142]; DesignElementFlow has . . .
System. [n143]; System has . . .
System States. [n148]; DesignElement Actions maps_to . . .
System.States maps_to DesignElement.States. [n146]
SystemMode (Variable). [n126]; FunctionalMapping depends_on . . .
Task has ExecutiveLoop. [n123]
Task has Function. [n120]
Task has Selector of Function. [n122]
Task is_a ProcessingElement. [n101]
Task receives Message from CommunicationChannel. [n119]
Tasks decompose_into Tasks. [n103]
Tasks. [n102]; Processors has . . .
Tasks. [n103]; Tasks decompose_into . . .
Temporality has Sheaf (Holder). [n160]
Temporality is_a Method. [n159]
ToViewPoint (Variable). [n003]; Method has . . .
Tokens (Type). [n037]; Markers is_a . . .
Transit (Function). [n035]; Transits has . . .
Transit has InputPlaces (List). [n043]
Transit has PetriRules (List). [n042]
Transit. [n047]; RightHandSide triggers . . .
Transits (List). [n031]; PetriMatrix has . . .
Transits has Transit (Function). [n035]
Transits. [n039]; PetriArcs maps_to . . .
Trigger for Function. [n017]; StateVector is_a . . .
Variable. [n109]; CommunicationMechanism may_be . . .
Variables (Set). [n011]; StateVector has . . .
Variables maps_to Variables via Datum. [n158]
Variables via Datum. [n158]; Variables maps_to . . .
Variables. [n151]; InformationFlow has . . .
Variables. [n157]; Datum maps_to . . .
VirtualMachine decomposes_into VirtualMachines. [n132]
VirtualMachine has Instructions. [n133]
VirtualMachine is_a Method. [n131]
VirtualMachine may_have StateMachine. [n134]
VirtualMachines. [n132]; VirtualMachine decomposes_into . . .
When InputDataArcs present Bubble is Triggered. [n067]
WorldLine is_a Method. [n136]
Worldline has Messages (List) associated with one ProcessingElement. [n137]
APPENDIX 3

Description of

for Software Design Minimal Methods
STATE MACHINE METHOD

DESCRIPTION:

The State Machine method is fundamental to understanding all software systems. It is the canonical way to represent software and other types of systems. It represents the reactive ability of the system to stimuli.

CONCEPTS:

Tutorial:

The State Machine is sometimes called Finite State Machines (FSM), Finite Automata, or just AUTOMATA. It is a control structure that associates an ACTION with an input EVENT and a AUTOMATA STATE. When the automata reacts to an input it resets its own new state. All computing structures may be reduced to an automata or an automata with a pushdown stack which is equivalent to a Turing Machine.

For actual use an automata needs to allow nested states and have modes. Nested states, or sub-automata allow for a variety of responses depending on the situation as do modes. In fact a mode is normally a sub-state of a higher level state machine.

The automata is the dual of the PetriNet. The PetriNet displays control explicitly where as the automata displays control implicitly. This is to say that in a state machine the control structure is coded into tables of stateVectors where as it is a direct link from one design element to the next. Sometimes the control path of the system may be very involved and may jump from state to state in a way that is difficult to understand. In those cases the number of PetriNets needed to represent a single state machine would be very large. In fact there is normally a combinatorial explosion between the representation of a system as a PetriNet rather than an Automata.

Principles:

1. Discrimination of System States into a finite set.
2. Association of Actions with Events and the System State.
3. The System may only be in one state at a time.

Properties:

1. Transitions connect two states and are associated with an event and action.
2. An automata may have multiple modes or meta-states.
3. Each state may be decomposed into sub-states and be associated with a lower level automata.
4. Automata may appear in the CONTROLSPEC of the Controlflow Method flanked by decisionTables and processActivationTables.

Criteria:

1. Automata should be constructed to be as compact and simple as possible.
2. Unnecessary states should be eliminated.

3. It is better to have multiple small state machines than one large state machine.

4. System modes should be minimized. Normally there is a Error or Fault Handling modes, Initialization mode, and normal running mode.

Guidelines:

1. Each entity within the system may have its own special lifecycle which should be modeled with an Automata.

2. Automata at different levels in the hierarchy need to interact properly and special care should be taken to design and test that interaction.

Measures:

1. Number of Automata

2. Number of States and Events per Automata

3. Number of Modes

Notes:

1. See STATEMATE from ILOGICS for an example of a design method based on embedded state machines.

ACTIONS:

1. Define input events.

2. Define output actions.

3. Define system states.

4. Define sub-states.

5. Define Modes.

6. Connect states by transitions.

7. Connect Actions and Events to transitions.

REPRESENTATION:

Elements:


2. State

3. Event
4. Action
5. Mode

Relations:
1. Transition from State to State
2. Between Event, Action, and Transition
3. Between Automata and sub-Automata.

Notation:
1. There are two different state machine notations. In one actions happen along the transitions where as in the other actions happen on entering a state. We have selected the notation where the actions happen along the transition.

Artifacts:
1. Automata may be represented as Tables or graphically as state diagrams.

EXAMPLE:
1. The structure of an automata.
PETRINET METHOD

DESCRIPTION:

The Petri Net is a formalism that shows control flow through the system. Instead of being reactive like the state machine it is self-activating. It replaces the old method of using flowcharts to understand system control flow. Colored Petrinets allow a great deal of diversity in the simulation of active control structures.

CONCEPTS:

Tutorial:

Petrinets are networks connecting PLACEs and TRANSITs. Each transit has a set of input places and a set of output places. Tokens or markers move through the places. When the correct configuration of markers appear in the input places as determined by a DECISION rule then the transit will fire and place an arrangement of tokens in the output places. In the original petrinet all the tokens were the same. In the more modern version called colored petrinets different kinds of tokens may flow through the network. A transit may react differently to different sets of tokens in its input places. When the transit fires it is simulating the accomplishment of functional work.

Principles:

1. Control flow lines are distinguished explicitly instead of implicitly.
2. The system is self activating.

Properties:

1. Places are connected to transits.
2. Colored markers move through the network of places and transits.
3. Rules govern the firing of the transit.
4. When the transit fires functional work is accomplished.
5. A transit may contain a sub-Petrinet.
6. The petrinet transit may be associated with a variable called a Banner whose contents is called an Insignia. When the transit fires it may set the banner and further firing many depend on the value of the banner.

Criteria:

1. Petrinets should only be used when control flow is problematic and needs to be studied, otherwise automata are preferred because it only takes one automata to represent the same data as many petrinets.
2. Transit decision rules should be minimized.
3. The number of colored markers should be minimized.
Guidelines:
1. Keep petrinets as simple as possible and use sub-nets to hide detail.

Measures:
1. Number of places and transits in a net.
2. Number of sub-nets

Notes:
1. See the book on Petrinets by Peterson.

ACTIONS:
1. Define Places
2. Define Transits
3. Define Banners
5. Link places to transits.
6. Decide on starting positions for tokens.
7. Link transits to functions.

REPRESENTATION:

Elements:
1. Place
2. Transit
3. Banner

Relations:
1. Banner to transit
2. Decision rule to transit
3. Place to Transit

Notation:
1. Petrinets are normally drawn but they may be represented as a matrix of places by transits.

Artifacts:
1. Petri net diagrams or tables.

**EXAMPLE:**

1. Structure of Colored Petrinet
DATAFLOW DECOMPOSITION METHOD

DESCRIPTION:

The dataflow method is a means of functionally decomposing a system which takes into account the relations of functions to data. It is a two way bridge between the data and functional views of a software system. Functionality concerns what the system does and data concerns where the system does it in memory. The dataflow diagram allows one to determine what is done to data and how data connects to what is done in the system in general.

CONCEPTS:

Tutorial:

The dataflow method is used to get a coherent picture of the functionality of the software system as a whole. The dataflow method is employed in either requirements analysis or the early stages of software design. The model used in requirements analysis is called the essential model. It is employed to get a functional view of the entire design space without implementation constraints. In the design process these implementation constraints are introduced and the essential model is transformed into the implementation model that allows the design to be traced back to the requirements at every stage of the design.

One always begins the functional decomposition of the system by identifying the externals of the system and thus the system boundary. The context diagram is the end result of this analysis. Then the context diagram is successively decomposed into lower and lower levels of functionality until the system is sufficiently defined. That definition is done in terms of the data interfaces of functional subsets of the system. The identification of the data inputs and outputs is the major means of defining the functionality of a particular part of the system. Data decomposition goes hand in hand with functional decomposition. The decomposed data is kept in a data dictionary where sub-relations between data-items is maintained along with data definitions. data-stores are identified where groups of persistent data-items are kept. Some attempt to use normal form database methods should be applied to defining data-stores.

Principles:

1. Top-down decomposition of both functions and data.
2. Identification of the external interface with a context bubble.
3. Definition of functional and data components by identifying inputs and outputs.
4. Functions are separated by the kind/type relations: i.e. similar functions are grouped together and different kinds of actions are separated systematically by the functional decomposition process.
5. A firm distinction between essential and implementation dataflow models.

Properties:

1. The data flow diagram is a series of nested networks of dataflow bubbles, externals, and data-stores connected by dataflow arcs.
2. Any dataflow bubble can be decomposed into a lower level dataflow.
3. Any data-store or dataflow arc can be decomposed into data subcomponents

4. Dataflow bubbles, arcs, and data-stores are strictly hierarchical.

5. The essential dataflow model does not contain any implementation constraints and assumes perfect or ideal conditions like infinite speed, infinite storage capacity, perfect transmission along data-lines, etc.

6. The implementation dataflow model is impure in that it contains alterations that take into account implementation constraints.

7. The dataflow diagrams of the system is accompanied by the data dictionary which describes all the data relations in the system.

Criteria:

1. The dataflow representation of the system should all be developed to the same level of detail.

2. No agent or event characteristics of the system should appear in the dataflow diagrams.

3. A given dataflow bubble should contain no more that nine sub-bubbles

4. There is no stopping rule for deciding when the dataflow decomposition is finished. Normally the first few levels, say at least three, should be done for every system. Too many levels of decomposition quickly becomes unwieldy.

5. The dataflow is meant to give a clear idea of the functional design space as a whole. Once this objective is achieved then further work on this system view is at the discretion of the designer.

Guidelines:

1. The dataflow may either represent the existing system or the proposed system.

2. The dataflow identifies the system boundary and represents the whole system.

3. The essential model should be developed first so that first versions should not contain implementation constraints details.

4. The dataflow essential model is used as a test for the coherence of system requirements.

5. Data items should as much as possible be grouped into tables to prevent unnecessary data expansion in the design of the system.

Measures:

1. Number of levels in functional decomposition.

2. Number of data-items.

Notes:

ACTIONS:

1. Functional decomposition
2. Interface Identification of inputs and outputs.

3. Identification of external entities

4. Data decomposition

5. Leveling the hierarchy of dataflows.

6. Following datalines to assure path coherence.

REPRESENTATION:

Elements:

1. Dataflow bubble
   - Name
   - Indented Number
   - Sub-bubbles

2. Dataflow arc
   - Name
   - Sub-data-items

3. Data-store
   - Name
   - Sub-data-items

4. External
   - Name
   - Sub-externals

Relations:

1. Dataflow bubbles have input and output data-arcs.

2. Data-stores have input and output data-arcs.

3. Externals have input and output data-arcs.

4. Dataflow bubbles contain sub-bubbles

5. Data-stores contain data-items

6. Data arcs are data-items

7. Externals are only outside the context bubble.
8. All other bubbles are within the context bubble.

9. Externals to any particular lower level dataflow may or may not be shown.

10. Externals may or may not be decomposed.

Notation:

1. Yordon/DeMarco notation
2. Gane/Sarenson notation
3. Hatley/Pribahi notation

Artifacts:

1. Context diagram
2. Hierarchically ordered dataflow diagrams
3. Data dictionary
4. Functional descriptions
EXAMPLE:

1. Context Diagram

2. Functional Decomposition
3. Dataflow with data-store

4. data-store and data-item decomposition
CONTROLFLOW DECOMPOSITION METHOD

DESCRIPTION:

The controlflow is an addition to the dataflow diagram which makes explicit control information missing from the dataflow diagrams as originally conceived. This addition was made in order to adapt the dataflow method to the functional analysis of real-time systems.

CONCEPTS:

Tutorial:

Principles:

1. Control information flows through software systems along lies parallel to but distinct from data.
2. Control information may originate from functional bubbles, externally or from datastores and flows to control specifications.
3. Control information flows from control specifications to activate functional bubbles.
4. Control may appear as interrupting triggers that make things happen in the system.

Properties:

1. Control diagrams may or may not have dataflow lines on them.
2. Control lines always flow to control specs on the same diagram never between dataflow diagrams. This means that control is always one level higher in the hierarchy of functional decomposition than what is being controlled.
3. Control specs may contain a variety of control structures such as state machines, process activation tables, decision tables, petri nets, etc.
4. Dataflows fire when all their data inputs are available unless they are deactivated by a process activation table in which case it is as if they were taken out of the system all together.
5. Thus control exerts what is really equivalent to modal changes in the system. Specific control sequences are not modeled with control flows but should instead be modeled with petri nets or state machines.
6. Triggers may be introduced to show response to external stimuli (this is a weakness of the Hatley Pirbhai model and is taken from ESML)

Criteria:

1. Control should always be minimized. No control flows should appear where a dataflow line would do.
2. The distinction between data and control is hazy and so control should only appear where it is clearly called for and is definitely distinct from dataflow.
3. Only as many control flow and control specs should be introduced as necessary to specify all the
modes in the system.

Guidelines:

1. Control specs may connect different levels of the dataflow diagram but as much as possible control should be limited to the level at which the controlspec resides.

2. Triggers should be avoided unless absolutely necessary as these are a deviation from the Hatley/Pribhai methodology.

Measures:

1. Number of control specs.

2. Number of triggers.

Notes:

ACTIONS:

1. Define control lines for a particular dataflow diagram and represent them going to the controlspec.

2. Identify all triggers within the system.

3. Design the control structure within the control specification.

4. Design the relation of the control structure back to the processes using a process activation scheme.

5. Make sure the modalities of the system are coherent.

REPRESENTATION:

Elements:

1. Control arc

2. Controlspec

3. Decision Table

4. State Machine

5. Petri net


7. Triggers

Relations:

1. The control arc goes from a data bubble, store, or external to a controlspec.
2. The controlspec may contain any specific combination of control structures including decision table, state machine, petrinet, process activation tables.

3. The process activation table may specify control relations between the controlspec and the functional bubbles which are shown by control arcs but this is normally omitted.

4. The functional bubbles, actions of state machines, and transitions of petrinets may be made equivalent.

5. Triggers may cause transitions in state machines.

Notation:

1. Hatley/Pirbhai Notation
2. Ward/Mellor Notation
3. ESML Notation of Jensen et al.

Artifacts:

1. Control flow diagrams or Combination control and dataflow diagrams
2. Control specifications.
3. Decision tables (optional)
4. State machines (must be at least one or petrinet substituted)
5. Petri nets (optional)
6. Process Activation Table (optional)
EXAMPLE:

1. Control flow diagram

2. Controlspec structure
ENTITY-RELATIONSHIP ARTICULATION METHOD

DESCRIPTION:

The Entity Relation technique allows us to specify any number of entities and their relations. Both entities and relations may have attributes. This allows us to see all the static elements in the system and their synchronic relations to each other.

CONCEPTS:

Tutorial:

Principles:

1. Entities are the persistent elements of a system.
2. Entities may have relations to each other.
3. Entities and relations may have types or belong to classes.
4. Entities and relations have attributes.
5. Entities represent the structural decomposition of the components of the system.
6. All entities should be encapsulated.
7. Entities are kinds and may have multiple instances.

Properties:

1. Entities may be decomposed into lower level entities.
2. Relations may be decomposed into sub-relations.
3. Attributes may be shared.
4. Classes have attributes shared by all the entities of a type.
5. Classes for an inheritance hierarchy.
6. The fundamental relations are IS_A signifying the belonging to a class and HAS_A which signifies ownership.
7. All the instances of an entity have the same general form.

Criteria:

1. The entity relation diagram should be as simple as possible.
2. The hierarchical nature of the entity relation diagram should be used to hide unnecessary detail.

Guidelines:
Measures:

1. Number of classes.
2. Number of entities
3. Average number of Attributes per entity plus maximum for all.

Notes:

1. See Chen for the origin of this technique.

ACTIONS:

1. Define Entities.
2. Define Relations between Entities
3. Sub-divide entities if necessary.
4. Define attributes for entities.
5. Group into Classes.
6. Define Class attributes.
7. Define Instances.

REPRESENTATION:

Elements:

1. Entity
2. Relation
3. Attribute
4. Classes
5. Instances

Relations:

1. Two entities are related.
2. Entities have attributes
3. Relations have attributes.
4. Classes have entities.
5. Classes have classes.
Notation:

1. The inheritance hierarchy is show as a tree structure.
2. Entity relation diagrams have many notations. It normally appears as a network where the entities are nodes.

Artifacts:

1. Data Dictionary listing all entities and their attributes.
2. Entity-Relationship Diagram.
3. Inheritance Hierarchy

EXAMPLE:

1. 
DARTS METHOD

DESCRIPTION:

DARTS means Design & Analysis of Real-time Systems. It is a methodology developed by Hassan Gomma which describes the agency aspects of embedded software systems. Basically the methodology identifies Tasks and the communication mechanisms which allow them to relate to each other and their operating environment. The methodology was taken up by the Software Productivity Consortium who developed ADARTS which is an elaboration of the DARTS methodology.

CONCEPTS:

Tutorial:

The actual architecture of a real-time embedded system must be expressed in terms of cooperating independent processing entities. At the lowest level these are tasks that share the resources of a processor. Tasks normally make it appear to the program that they contain as if it were running without obstruction whereas actually it is starting and stopping as it is allocated resources by some executive which controls the processor and its resources. Of course, there are units of processing below the task level called interrupt subroutines, but these are special cases which will not be considered here except to say that they may be considered special cases of the task scheme.

Tasks would left to their own run their programs without any reference to what other parts of the system is doing. In actual systems tasks normally need to coordinate their actions with other tasks and with the external environment of the platform and Local Processor Operating System (LPOS). The NGCR OSSWG Real-time System Reference Model gives a good picture of the normal environment for the development of embedded real-time systems defining all the pertinent interfaces. In the case of this method there are two types of important interfaces: between tasks and between tasks and shared resources.

There are many types of communications between tasks. These range from the Multiparty interaction as defined in MCC RADDLE & VERDI, to the Rendezvous in the Ada Programming Language, to the pipe that is used in UNIX, to the Queue which is normally used in real-time operating systems, to flags and semaphores.

If resources in a multi-tasking system are not controlled then deadlock and race conditions may occur. Therefore, Hoar developed what is called the monitor which is basically a control mechanism using a semaphore to allow tasks to fight over tokens and not the resources themselves. This solves most of the deadlock and race conditions that may occur.

Principles:

1. The system needs to be divided into a hierarchy of agents, normally called tasks.
2. Agents cooperate with each other through communication channels which carry messages.
3. Agents get their resources from controlled resource pools.
4. Distributed design needs to be distinguished from Tasking design.

Properties:

1. The tasking and monitor methods are analogous to the dataflow method where tasks are like func-
tion bubbles and monitored resources are like datastructures. Though there is an analogy it is ex-
actly in the transformation of functions into task groupings that all the difficulty of designing real-
time systems arises.

2. The methodology begins with the essential model of the functional decomposition of the system as a given.

3. Functions are grouped into tasks.

4. Tasks are connected by communications mechanisms.

5. Tasks are connected to their resources using monitors.

6. Distributed design deals with the aspects of system performance that accrue from distribution in space of the system.

7. Tasking design proper deals with the aspects of the design that have to do with time and the shar-
ing of processing and other resources within a processor.

8. Tasks have ports that receive communications from a particular mechanism.

Criteria:

1. There is normally no one to one mapping between function-bubbles and tasks. Unless a system is very simple the cross mapping between functions and tasks will expected to be complex.

2. Tasks are constructed by taking lowest level functions and grouping them according to multiple cri-
teria. This grouping may occur in many ways and different ways should be tried by giving the grouping criteria different criteria. In ADARTS these criteria are enumerated.

3. Tasking should be minimized in a system as context switch time may degrade performance. But where performance is not a consideration then tasks may be used to segregate groups of func-
tions in a logical manner. Between these tow extremes there may be many ways to segregate functions into a task within any particular software system.

4. Good tasking first meets performance goals and secondly cuts down on the use of the communi-
cations channels.

5. Architectural design is done when all distributed and tasking design issues have been solved for the system.

Guidelines:

1. Do distributed design before tasking design.

2. Distributed design may occur on many levels depending on how processors are connected to each other. Deal with each level of distributed design separately.

3. Tasking design depends on the hierarchy of tasks. Do not use a tasking hierarchy as allowed in Ada unless absolutely necessary.

Measures:

1. Number of tasks.
2. Number and type of communication channels.

3. Number of levels in hierarchy of tasks.

Notes:

1. See Hassan Gomma on DARTS.

2. See SPC on ADARTS.

3. See RADDLE and VERDI from MCC.

4. See DURRA TASK DESCRIPTION LANGUAGE by Mario Barbacci of SEI/CMU.

ACTIONS:

1. Distributed Design

2. Define Tasks by grouping functions.

3. Define Tasking Hierarchy.


6. Define internal structure of tasks.

REPRESENTATION:

Elements:

1. Task

2. Ports

3. Communication Mechanism

4. Monitor

5. Distributed Communication Channel.

Relations:

1. Task to task via ports using communications mechanism.

2. Task to monitor using ports.

Notation:

1. Ready Systems had a task design tool that has a notation for showing the relations between tasks. It uses communications mechanisms and other features specific to their real-time operating system.
2. The DURRA Tasks Description Language has a formal language that describes tasks and their relations via ports. The statements in this language is used to generate source code for the tasking structure.

3. TRW NAS is a tool that allows tasking diagrams to be drawn and then these are used to generate simulations.

**Artifacts:**

1. Distributed Design
2. Tasking Design
3. Global System Design
EXAMPLE:

1. Structure of tasking method.
ALLOCATION METHOD

DESCRIPTION:

The allocation method actually connects the functional and tasking views of the software design. It allows requirements traceability to occur throughout the design process thought the transformation of the essential model into an implementation model. This process is described in detail in the third volume of Mellor & Ward's Structured Real-time System Design.

CONCEPTS:

Tutorial:

The allocation of functionality to tasks and the trace back from tasks to functionality is a fundamental part of software design that is often not recognized as important. Allocation is a two way street which moves back and forth between tasking design and functional decomposition in order to continually assure the compliance of the design with the constraints of the requirements. This is done by starting design with an essential model and using the lowest level functions grouping them into tasks and then by modifying the essential model iteratively to embed implementation decisions within it.

Principles:

1. Must start design with an essential model.
2. Use the essential model to derive tasks.
3. Map back implementation decisions into the essential model transforming it into an implementation model.
4. The implementation model reflects the mapping of requirements to design at every stage of the design process.

Properties:

1. The essential model and implementation use the Hattley/Pribhai or Ward/Mellor methodologies.
2. The mapping is essentially a cross reference between tasks and functions.

Criteria:

1. The essential model should express functionality in an ideal environment of infinite computing speed, infinite memory, instantaneous communication, no distribution, and no tasking.
2. The implementation model should track the distributed and tasking design modifying the essential design.

Guidelines:

Measures:

1. Complexity of mapping.

Notes:

**ACTIONS:**

1. Bring Implementation model up to date.

**REPRESENTATION:**

**Elements:**

1. Same as Data & Control Flow Methods.

**Relations:**

1. Same as Data & Control Flow Methods.

**Notation:**

1. Same as Data & Control Flow Methods

**Artifacts:**

1. Implementation Model

**EXAMPLE:**

1. None
VIRTUAL MACHINE METHOD

DESCRIPTION:

The Virtual Layered Machine methodology was developed by Nielsen and Shumate in their book *Designing Large Real-time Systems in Ada*. It is an elaboration of the structure chart method which is a traditional way of describing software structure.

CONCEPTS:

Tutorial:

The basic concept of the Virtual Layered Machine method is that for any given problem it may be solved at some level of abstraction by a set of instructions which are used to solve that particular design problem. These instructions form a virtual machine and any particular instruction may in turn be considered such a machine until the problem is completely decomposed and taken down to its most concrete level of description. At any level Instructions may be composed of operations on objects, pure transformations of non-persistent data, or sub-instructions.

Principles:

1. Every solution to a software design problem may be expressed as set of instructions which work together to solve the problem.
2. Instructions may be decomposed into sub-instructions.
3. Sets of instructions may operate as an Automata or Petrinet.
4. Lowest level instructions are operations of the persistent data of objects in most cases.

Properties:

1. Instructions may contain instructions, operations, or functions.
2. Sets of instructions form a whole called a Virtual Machine.
3. Machines may be associated with Automata or Petrinets.
4. The virtual machines form a hierarchy.

Criteria:

1. Any

Guidelines:

1. One may build virtual machines from bottom up or top down or any combination that makes sense.
2. Any level of abstraction may be picked as the starting point.
3. The virtual machines mapping to the structural decomposition hierarchy of the essential model should be maintained.
Measures:

1. Number of levels of virtual machines.
2. Number of instructions.
3. Number of Objects and Operations.
4. Number of Automata or Petrinets.

Notes:

1. See Nielsen and Shumate’s *Designing Large Real-time Systems in Ada*.

ACTIONS:

1. Select level of abstraction.
2. Produce instructions set to solve design problem.
3. Decompose Instructions to get next lowest level.
4. If you know about Objects and their operations design them as you go along.
5. Attach Automata or Petrinets as appropriate.

REPRESENTATION:

Elements:

1. Instruction
2. InstructionSet
3. VirtualMachine
4. Object
5. Operation
6. Action
7. Transit

Relations:

1. Instructions to Instruction
2. Instruction to VirtualMachine
3. VirtualMachine to Automata or Petrinet
4. Instruction to Object Operation
5. Instruction to Automata Action
6. Instruction to Petrinet Transit

**Notation:**

1. Structure Charts may be used.
2. Nielsen & Shumate suggest a graphical notation.
3. Tony Wasserman of IDE has suggested a generic Object Notation that has many of the necessary features.

**Artifacts:**

1. Structure Chart.

**EXAMPLE:**

1. Structure of a hierarchy of Virtual Machines.

![Diagram of Virtual Machines and Objects](image-url)
WORLDLINE & SCENARIO METHOD

DESCRIPTION:

This pair of minimal methods allow the interaction of agents to be traced and interrelated through time even though each is traveling its own world line the agents interact in scenarios which allow causal waves to be determined.

CONCEPTS:

Tutorial:

This pair of methods invokes Special Relativity which appears in distributed systems as the impossibility of a reliable global clock. Each agent appears to have its own independent timeline called a world-line which represents its view of everything that is happening inside and outside the system. Each agent has its own point of view and things may happen at a very different time than they appear to happen from the point of view of another agent. What we are really interested in developing here is how the events of the rest of the system and environment appear from the point of view of every agent and also what are the possible chains of causality which link these appearances together.

Principles:

1. Every agent has their own clock.
2. Clocks may be set to reference clocks but that takes time.
3. There is the possibility of time warp where an individual agent can go ahead and make calculations given what appears to it as good data as long as it maintains a history and can reset if it finds out some of its data is wrong.
4. The wave of finalized causality reaches each agent at different times and it is not possible to determine when that is exactly. It is basically when no more corrections to a message arrive.

Properties:

1. Agents have their own time line.
2. Communication channels connect timelines.
3. Causality travels via communications channels.
4. Agents may interact with groups of other agents in a single INTEACTION.
5. Worldlines are generated from role templates.

Criteria: 

1.

Guidelines:

1.
**Measures:**

1. Number of Agents.
2. Number of Messages
3. Number of roll-backs

**Notes:**

1. See Time-Warp Simulation systems.
2. See Einstein’s explanation of Special Relativity.
3. See MCC RADDLE and VERDI

**ACTIONS:**

1. Trace message paths.
2. Trace actions of each agent through time.
3. Produce Roles that generate timelines of each Agent.

**REPRESENTATION:**

**Elements:**

1. Agent.
2. Timeline
3. Scenario
4. Role
5. Message

**Relations:**

1. Agents receive Messages
2. Agent has sequence of Events.
3. Scenario contains Events across worldlines.

**Notation:**

4. See MCC RADDLE & VERDI
5. Agha ACTORS

Artifacts:
1. Scenario descriptions
2. Role diagrams
3. Timing diagrams

EXAMPLE:

1. Worldline
2. Finalized Causality
3. ROLL BACK
4. SCENARIO
5. Event
DESIGN ELEMENT AND INFORMATION FLOW METHODS

DESCRIPTION:

The fundamental methods related to spacetime gage the flow of information and design elements through the system. Information flows between variables and can be looked at in terms of all the data that are in all the variables at one time (synchronously) or by following one or more pieces of data as they flow through the system and are transformed (diachronically). Design Elements also flow through the software systems whose structural elements are malleable and even self modifying. Design elements states may be compared to system states and design element actions may be compared to system actions.

CONCEPTS:

Tutorial:

Information systems may be looked at either as state machines or information networks. As state machines they take inputs and produce outputs so one may look at the discrete state of variables at successive points in time. As information networks it is possible to instead look at the way data flows through the network and is transformed in the process. From the information network point of view the information is malleable whereas from the state machine it is the states of the variables that are malleable.

Information systems have data that flows through them and also data that compose them. The compositional data are called design elements. Design elements are all little state machines that must relate to the overall state machine of the system as a whole or some higher level aggregate. Design elements flow through the system in the same way information does except at each state there is a correspondence maintained between overall states and lower level design element states. This correspondence may either be looked at by looking at systems states and design element transitions or system transitions and design element states.

Principles:

1. Every software system has a set of variables.
2. You can look at the contents of all the variables at once or a single piece of information flowing through several variables.
3. You can look at variables as the state variables of automata.
4. You can look at the coordination between overall system states and design element states.
5. Either you look at system states and design element transitions or design element states and system transitions.

Properties:

1. Variables have values.
2. Sets of values in all system variables at one time is the aggregate system state.
3. Information flows through variables represented as values.
4. Variables may be considered to be individual automata.

Criteria:

1. Overall the goal is to make the design element states coherent as possible with the overall system states.
2. Only necessary information should be kept in the system variables at any one item.

Guidelines:

1. Information flow and information network are the basic debugging information for software systems.
2. All variables should be treated as state machines and explicitly coordinated with overall system states.

Measures:

1. Number of Variables
2. Number of Information flows.
3. Number of Design Elements
4. Number of states and transitions per design element
5. Number of System states and transitions.

Notes:

1. See David Gelernter Programming Linguistics for an excellent generalized model of these space-time relations as an abstract programming language model.

ACTIONS:

1. Design information variables for design elements.
2. Describe information network that connects variables.
3. Describe information flows through network
4. Describe coordination of overall system states and design elements states.

REPRESENTATION:

Elements:

1. Variable
2. State

Relations:
1. Relation between Variables.
2. Sets of variable relations represent Information network.
3. Information flowing through network through channels
4. Relation between states which are values of variables
5. Actions of Automata or Petrinets cause changes in values.

Notation:
1. Gelernter PROGRAMMING LINGUISTICS
2. SOFTWARE DIAGRAMMING

Artifacts:
1. Information flow diagram
2. Information network
3. List of Variables in System
4. Design element states and actions
5. System states and actions
EXAMPLE:

1. Information network and information flows

2. Design Element Flows in system context
TEMPORALITY METHOD

DESCRIPTION:
In every system time must be described. It is normally described in terms of events and duration.

CONCEPTS:

Tutorial:
Allen has provided us with an interval logic for the description of time through constraints on the ordering of events.

Principles:
1. Events are fully ordered (linear with distance) in a software system.
2. Constraints operating between events may be described using a temporal logic.
3. Events are fluctuations in signals.

Properties:
1. Events may have the following relations:
   - Before
   - After
   - During
   - Starting
   - Finishes
   - Overlaps
   - Meets
   - Equals
2. Signals come in bundles which in turn form sheaves.
3. A single signal by itself is meaningless.
4. Bundles split and join describing branching points in possible system timelines.
5. The fundamental signal is the clock pulse.

Criteria:
1. All temporal constraints should be satisfied by the system.

Guidelines:
1. Timing diagrams should be as simple as possible with only the most crucial signals portrayed.

Measures:

1. Number of Events

Notes:

1. See Temporal logic for relations of causality and necessity in sheaves and between bundles.
2. See Allen’s Interval Logic

ACTIONS:

1. Define events.
2. Produce timing diagram.
3. Work out constraints between events using Interval Logic

REPRESENTATION:

Elements:

1. Signal
2. Bundle
3. Sheaf
4. Event
5. Interval
6. Duration

Relations:

1. Signals contain Bundles
2. Bundles contain signals
3. Signals have events as fluctuations.
4. Gaps can only be measured by comparing signals.

Notation:

1. Allen’s temporal logic.

Artifacts:
1. Timing Diagram

EXAMPLE:

1.
INTEGRAL SOFTWARE ENGINEERING METHODOLOGY (ISEM) FORMAL DESIGN LANGUAGE

by Kent D. Palmer

SOFTWARE ENGINEERING FOUNDATIONS: Part III

Appendix 4

Entity Relationship diagrams for Integral Software Engineering Methodology

PO Box 1632
Orange CA 92856
714-633-9508
kent@palmer.name
Intergral Software Engineering Methods (ISEM) --- Entity Relation Diagram For ISEM Language

```
LAYER

PARTITION

INTERSECTION

ARCHITECTURE

COMPONENT

DeftIdentIsPartit

PositPartitIsAdjPartit

PositPartitIsLeftOfPartit

PositPartitIsRightOfPartit

PositPartitIssectsLayerAtIsection

PositPartitIssectsTierAtIsection

PositPartitIssectsStrataAtIsection

PositComponInPartit

DeflIdentIsPartit

z125

z126

z127

z128

z129

z133

z134

z135
```

Copyright 1992 Kent D. Palmer. All Rights Reserved.
**ACTOR**
**SITUATION**

- **SITUATION**
  - `SI DefIdentIsSituation z922`
  - `SI z923 PositSituationHasConstellation`

- **CONSTELLATION**
  - `SI DefIdentHasConstellation z924`
  - `SI z925 PositConstellationHasBag`

- **SET**
  - `z921`

- **BAG**
  - `SI DefIdentHasBag z926`
  - `SI z927 PositBagHasEntity`

- **ENTITY**
Intergral Software Engineering Methods (ISEM) --- Entity Relation Diagram For ISEM Language

Copyright 1992 Kent D. Palmer. All Rights Reserved.
TURING

Integral Software Engineering Methods (ISEM) --- Entity Relation Diagram For ISEM Language

STATEVAR

FLAG

BANNER

AUTONETIC

ATTRIBUTE

BEHAVIOR

ASSEMBLY

Copyright 1992 Kent D. Palmer. All Rights Reserved.
INTEGRAL SOFTWARE ENGINEERING METHODOLOGY (ISEM) FORMAL DESIGN LANGUAGE

by Kent D. Palmer

SOFTWARE ENGINEERING FOUNDATIONS: Part III

Appendix 5

The ISEM Language

PO Box 1632
Orange CA 92856
714-633-9508
kent@palmer.name
AC1 acquireSemaphore................ ->  
ACTOR: ACQUIRE SEMAPHORE id [z666]

AC2 CommCnectrFromPortToPort....... ->  
ACTOR: COMMUNICATE CONNECTOR id FROM PORT id TO PORT id [z678]

AC3 CommCnectrFromTaskToTask....... ->  
ACTOR: COMMUNICATE CONNECTOR id FROM TASK id TO TASK id [z668]

AC4 CommCnectrFromPort.............. ->  
ACTOR: COMMUNICATE CONNECTOR id FROM PORT id [z697]

AC5 CommCnectrToPort............... ->  
ACTOR: COMMUNICATE CONNECTOR id TO PORT id [z698]

AC6 CommCnectrFromTask.............. ->  
ACTOR: COMMUNICATE CONNECTOR id FROM TASK id [z699]

AC7 CommCnectrToTask............... ->  
ACTOR: COMMUNICATE CONNECTOR id TO TASK id [z700]

AC8 DefIdentIsActor................ ->  
ACTOR: DEFINE IDENTIFIER id IS ACTOR [z671]

AC9 DefIdenfifierIsIrax........... ->  
ACTOR: DEFINE: IDENTIFIER id IS IRAX [z683]

AC10 DefIdenfifierIsLpos.......... ->  
ACTOR: DEFINE: IDENTIFIER id IS LPOS [z682]

AC11 DefIdenfifierIsSrax.......... ->  
ACTOR: DEFINE: IDENTIFIER id IS SRAX [z681]

AC12 DefIdentiferIsNamespace..... ->  
ACTOR: DEFINE: IDENTIFIER id IS NAMESPACE [z679]

AC13 DefIdentIsCnectr.............. ->  
ACTOR: DEFINE: IDENTIFIER id IS CONNECTOR [z658]

AC14 DefIdentIsFile............... ->  
ACTOR: DEFINE: IDENTIFIER id IS FILE [z659]

AC15 DefIdentIsInteraction........ ->
`ACTOR: DEFINE: IDENTIFIER id IS INTERACTION` [z660]

AC16 DefIdentIsMailbox........... ->

`ACTOR: DEFINE: IDENTIFIER id IS MAILBOX` [z661]

AC17 DefIdentIsPipe.............. ->

`ACTOR: DEFINE: IDENTIFIER id IS PIPE` [z662]

AC18 DefIdentIsPort............... ->

`ACTOR: DEFINE: IDENTIFIER id IS PORT` [z674]

AC19 DefIdentIsPostal............. ->

`ACTOR: DEFINE: IDENTIFIER id IS POSTALSERVICE` [z680]

AC20 DefIdentIsRendezvous......... ->

`ACTOR: DEFINE: IDENTIFIER id IS RENDEZVOUS` [z663]

AC21 DefIdentIsSemaphore.......... ->

`ACTOR: DEFINE: IDENTIFIER id IS SEMAPHORE` [z664]

AC22 PositActorInhabitsIrax....... ->

`ACTOR: POSIT: ACTOR id INHABITS IRAX id` [z688]

AC23 PositActorHasTask............. ->

`ACTOR: POSIT: ACTOR id HAS TASK id` [z672]

AC24 PositAgentHasActor............ ->

`ACTOR: POSIT: AGENT id HAS ACTOR id` [z673]

AC25 PositAgentInhabitsSrax....... ->

`ACTOR: POSIT: AGENT id INHABITS SRAX id` [z687]

AC26 PositCnectrIsFile............. ->

`ACTOR: POSIT: CONNECTOR id IS FILE id` [z654]

AC27 PositCnectrIsFlag............. ->

`ACTOR: POSIT: CONNECTOR id IS FLAG id` [z657]

AC28 PositCnectrIsInteraction..... ->

`ACTOR: POSIT: CONNECTOR id IS INTERACTION id` [z650]

AC29 PositCnectrIsMailbox......... ->

`ACTOR: POSIT: CONNECTOR id IS MAILBOX id` [z652]
AC30 PositCnctrlsPipe............. ->
   **ACTOR: POSIT: CONNECTOR id IS PIPE id** [z653]

AC31 PositCnctrlsQueue............. ->
   **ACTOR: POSIT: CONNECTOR id IS QUEUE id** [z651]

AC32 PositCnctrlsRendezvous........ ->
   **ACTOR: POSIT: CONNECTOR id IS RENDEVOUS id** [z655]

AC33 PositCnctrlsSemaphore......... ->
   **ACTOR: POSIT: CONNECTOR id IS SEMAPHORE id** [z656]

AC34 PositInteractionIncludesTask.. ->
   **ACTOR: POSIT: INTERACTION id INCLUDES TASK id** [z665]

AC35 PositIraxHasNamespace......... ->
   **ACTOR: POSIT: IRAX id HAS NAMESPACE id** [z691]

AC36 PositIraxHasPostal............. ->
   **ACTOR: POSIT: IRAX id HAS POSTALSERVICE id** [z694]

AC37 PositIraxIsWithinSrax......... ->
   **ACTOR: POSIT: IRAX id IS WITHIN SRAX id** [z701]

AC38 PositLposIsWithinIrax......... ->
   **ACTOR: POSIT: LPOS id IS WITHIN IRAX id** [z685]

AC39 PositLposHasNamespace......... ->
   **ACTOR: POSIT: LPOS id HAS NAMESPACE id** [z692]

AC40 PositLposHasPostal............. ->
   **ACTOR: POSIT: LPOS id HAS POSTALSERVICE id** [z695]

AC41 PositLposManagesProcessor..... ->
   **ACTOR: POSIT: LPOS id MANAGES PROCESSOR id** [z684]

AC42 PositSraxIsWithinSrax......... ->
   **ACTOR: POSIT: LPOS id IS WITHIN SRAX id** [z686]

AC43 PositSraxHasNamespace......... ->
   **ACTOR: POSIT: SRAX id HAS NAMESPACE id** [z690]

AC44 PositSraxHasPostal............. ->
ACTOR: POSIT: SRAX id HAS POSTALSERVICE id [z693]

AC45 PositTaskInhabitsLpos.......... ->

ACTOR: POSIT: TASK id INHABITS LPOS id [z689]

AC46 PositTaskHasPortIn............ ->

ACTOR: POSIT: TASK id HAS PORT id IN [z675]

AC47 PositTaskHasPortInout........ ->

ACTOR: POSIT: TASK id HAS PORT id INOUT [z677]

AC48 PositTaskHasPortOut........... ->

ACTOR: POSIT: TASK id HAS PORT id OUT [z676]

AC49 ReadAttribute................ ->

ACTOR: READ ATTRIBUTE id [z902]

AC50 ReadDatastoreObjectAttr....... ->

ACTOR: READ DATASTORE id OBJECT id ATTRIBUTE id [z669]

ACX51 ReadObjectAttribute......... ->

ACTOR: READ OBJECT id ATTRIBUTE id [z903]

AC52 ReleaseSemaphore.............. ->

ACTOR: RELEASE SEMAPHORE id [z667]

AC53 WriteAttribute................ ->

ACTOR: WRITE ATTRIBUTE id [z901]

AC54 WriteDatastoreObjectAttr...... ->

ACTOR: WRITE DATASTORE id OBJECT id ATTRIBUTE id [z670]

AC55 WriteObjectAttribute......... ->

ACTOR: WRITE OBJECT id ATTRIBUTE id [z900]

AR1 CnectArcFrmIfaceToIface....... ->

ARCH: CONNECT ARC id FROM INTERFACE id TO INTERFACE id [z85]

AR2 CnectArcFrmNodeToNodeDirless.. ->

ARCH: CONNECT ARC id FROM NODE id TO NODE id DIRECTIONLESS [z83]

AR3 CnectArcFrmNodeToNodeDiral.... ->

ARCH: CONNECT ARC id FROM NODE id TO NODE id DIRECTIONAL [z82]
AR4 CnectArcFrmNodeToNodeBidiral.. ->

ARCH: CONNECT ARC id FROM NODE id TO NODE id BIDIRECTIONAL [z55]

AR5 CnectArcToNodeAsSink.......... ->

ARCH: CONNECT ARC id TO NODE id AS SINK [z45]

AR6 CnectArcFrmNodeAsSource..... ->

ARCH: CONNECT ARC id FROM NODE id AS SOURCE [z46]

AR7 DefArcIsBundle............... ->

ARCH: DEFINE: ARC id IS BUNDLE id [z41]

AR8 DefArcIsCnector.............. ->

ARCH: DEFINE: ARC id IS CONNECTOR id [z42]

AR9 DefArcIsControlflow......... ->

ARCH: DEFINE: ARC id IS CONTROLFLOW id [z43]

AR10 DefArcIsDataflow............ ->

ARCH: DEFINE: ARC id IS DATAFLOW id [z44]

AR11 DefArcIsMapping............. ->

ARCH: DEFINE: ARC id IS MAPPING id [z50]

AR12 DefArcIsRel.................. ->

ARCH: DEFINE: ARC id IS RELATION id [z52]

AR13 DefArcIsSheaf................. ->

ARCH: DEFINE: ARC id IS SHEAF id [z53]

AR14 DefArcIsSignal............... ->

ARCH: DEFINE: ARC id IS SIGNAL id [z54]

AR15 DefArcIsTransit............... ->

ARCH: DEFINE: ARC id IS TRANSIT id [z57]

AR16 DefArcIsType.................. ->

ARCH: DEFINE: ARC id IS TYPE id [z56]

AR17 DefArtifactHasArch.......... ->

ARCH: DEFINE: ARTIFACT id HAS ARCHITECTURE id [z73]

AR18 DefArtifactIsOfType......... ->

Copyright 1990 Kent D. Palmer. All Rights Reserved. Page: 416
ARCH: DEFINE: ARTIFACT id IS OF TYPE id [z72]

ARCH: DEFINE: COMPONENT id IS OF TYPE id [z78]

ARCH: DEFINE: IDENTIFIER id IS APPLICATION [z40]

ARCH: DEFINE: IDENTIFIER id IS ARC OF TYPE id [z47]

ARCH: DEFINE: IDENTIFIER id IS ARC id [z48]

ARCH: DEFINE: IDENTIFIER id IS ARCHITECTURE [z68]

ARCH: DEFINE: IDENTIFIER id IS ARTIFACT OF TYPE id [z70]

ARCH: DEFINE: IDENTIFIER id IS ARTIFACT [z71]

ARCH: DEFINE: IDENTIFIER id IS COMPONENT OF TYPE id [z76]

ARCH: DEFINE: IDENTIFIER id IS COMPONENT id [z75]

ARCH: DEFINE: IDENTIFIER id IS CONFIGURATION [z81]

ARCH: DEFINE: IDENTIFIER id IS IMPLEMENTATION [z84]

ARCH: DEFINE: IDENTIFIER id IS INTERFACE id OF TYPE id [z87]

ARCH: DEFINE: IDENTIFIER id IS INTERFACE [z86]

ARCH: DEFINE: IDENTIFIER id IS INTERSECTION [z91]
AR33 DefIdentIsLayer................ ->
ARCH: DEFINE: IDENTIFIER id IS LAYER [z97]

AR34 DefIdentIsNodeOfType......... ->
ARCH: DEFINE: IDENTIFIER id IS NODE id OF TYPE id [z104]

AR35 DefIdentIsNode............... ->
ARCH: DEFINE: IDENTIFIER id IS NODE [z105]

AR36 DefIdentIsPartit.............. ->
ARCH: DEFINE: IDENTIFIER id IS PARTITION [z126]

AR37 DefIdentIsService............. ->
ARCH: DEFINE: IDENTIFIER id IS SERVICE id [z142]

AR38 DefIdentIsStrata.............. ->
ARCH: DEFINE: IDENTIFIER id IS STRATA [z146]

AR39 DefIdentIsTier................ ->
ARCH: DEFINE: IDENTIFIER id IS TIER [z151]

AR40 DefIdentIsUnitOfType........... ->
ARCH: DEFINE: IDENTIFIER id IS UNIT OF TYPE id [z154]

AR41 DefIdentIsUnit............... ->
ARCH: DEFINE: IDENTIFIER id IS UNIT [z153]

AR42 DefIfaceIsOfType.............. ->
ARCH: DEFINE: INTERFACE id IS OF TYPE id [z88]

AR43 DefNodeIsAgent................ ->
ARCH: DEFINE: NODE id IS AGENT id [z100]

AR44 DefNodeHasAnyid.............. ->
ARCH: DEFINE: NODE id HAS ANYID id [z124]

AR45 DefNodeIsAnyid............... ->
ARCH: DEFINE: NODE id IS ANYID id [z123]

AR46 DefNodeIsAmata................. ->
ARCH: DEFINE: NODE id IS AUTOMATA id [z101]

AR47 DefNodeIsAutonetic.......... ->
ARCH: DEFINE: NODE id IS AUTONETIC id [z116]

ARCH: DEFINE: NODE id IS BRANCH id [z115]

ARCH: DEFINE: NODE id IS CONTEXT id [z122]

ARCH: DEFINE: NODE id IS DATASTORE id [z119]

ARCH: DEFINE: NODE id IS ENTITY id [z102]

ARCH: DEFINE: NODE id IS EVENTITY id [z103]

ARCH: DEFINE: NODE id IS EXTERNAL id [z120]

ARCH: DEFINE: NODE id IS FILE id [z121]

ARCH: DEFINE: NODE id IS MACHINE id [z118]

ARCH: DEFINE: NODE id IS OBJECT id [z106]

ARCH: DEFINE: NODE id IS PETRINET id [z107]

ARCH: DEFINE: NODE id IS PLACE id [z114]

ARCH: DEFINE: NODE id IS PROCESS id [z108]

ARCH: DEFINE: NODE id IS PROCESSOR id [z117]

ARCH: DEFINE: NODE id IS TEMPORALITY id [z109]
AR62 DefNodeIsType.................. ->
ARCH: DEFINE: NODE id IS TYPE id [z110]

AR63 DefNodeIsWorld................. ->
ARCH: DEFINE: NODE id IS WORLD id [z113]

AR64 DefUnitIsOfType............... ->
ARCH: DEFINE: UNIT id IS OF TYPE id [z155]

AR65 PositArcHasAnyid............... ->
ARCH: POSIT: ARC id HAS ANYID id [z330]

AR66 PositArcIsAnyid............... ->
ARCH: POSIT: ARC id IS ANYID id [z329]

AR67 PositArcFrmIfaceToIface....... ->
ARCH: POSIT: ARC id FROM INTERFACE id TO INTERFACE id [z49]

AR68 PositArcFrmIfaceToNode........ ->
ARCH: POSIT: ARC id FROM INTERFACE id TO NODE id [z156]

AR69 PositArcHasIface............... ->
ARCH: POSIT: ARC id HAS INTERFACE id [z69]

AR70 PositArcFrmNodeToIface........ ->
ARCH: POSIT: ARC id FROM NODE id TO INTERFACE id [z51]

AR71 PositArcIsTransition.......... ->
ARCH: POSIT: ARC id IS TRANSITION id [z340]

AR72 PositArchIncludesArc.......... ->
ARCH: POSIT: ARCHITECTURE id INCLUDES ARC id [z60]

AR73 PositArchIncludesComponent.... ->
ARCH: POSIT: ARCHITECTURE id INCLUDES COMPONENT id [z61]

AR74 PositArchIncludesIsect........ ->
ARCH: POSIT: ARCHITECTURE id INCLUDES INTERSECTION id [z62]

AR75 PositArchIncludesLayer........ ->
ARCH: POSIT: ARCHITECTURE id INCLUDES LAYER id [z63]

AR76 PositArchIncludesNode.......... ->
ARCH: POSIT: ARCHITECTURE id INCLUDES NODE id [z64]

AR77 PositArchIncludesPartit....... ->

ARCH: POSIT: ARCHITECTURE id INCLUDES PARTITION id [z65]

AR78 PositArchIncludesStrata....... ->

ARCH: POSIT: ARCHITECTURE id INCLUDES STRATA id [z66]

AR79 PositArchIncludesTier......... ->

ARCH: POSIT: ARCHITECTURE id INCLUDES TIER id [z67]

AR80 PositArtifactIncludesComponent ->

ARCH: POSIT: ARTIFACT id INCLUDES COMPONENT id [z74]

AR81 PositComponentHasIface........ ->

ARCH: POSIT: COMPONENT id HAS INTERFACE id [z77]

AR82 PositComponentIsInLayer....... ->

ARCH: POSIT: COMPONENT id IS IN LAYER id [z99]

AR83 PositComponentIsInPartit...... ->

ARCH: POSIT: COMPONENT id IS IN PARTITION id [z129]

AR84 PositComponentIsInStrata...... ->

ARCH: POSIT: COMPONENT id IS IN STRATA id [z144]

AR85 PositComponentIsInTier........ ->

ARCH: POSIT: COMPONENT id IS IN TIER id [z152]

AR86 PositComponentIncludesUnit.... ->

ARCH: POSIT: COMPONENT id INCLUDES UNIT id [z79]

AR87 PositConfigIncludesArtifact... ->

ARCH: POSIT: CONFIGURATION id INCLUDES ARTIFACT id [z80]

AR88 PositCoordinateBelongsToIsect. ->

ARCH: POSIT: COORDINATE coord BELONGS TO INTERSECTION id [z90]

AR89 PositIsectHasArtifact......... ->

ARCH: POSIT: INTERSECTION id HAS ARTIFACT id [z92]

AR90 PositIsectHasComponent......... ->

ARCH: POSIT: INTERSECTION id HAS COMPONENT id [z93]
AR91 PositIsectHasCoordinate...... ->

ARCH: POSIT: INTERSECTION id HAS COORDINATE coord [z89]

AR92 PositIsectHasUnit.......... ->

ARCH: POSIT: INTERSECTION id HAS UNIT id [z94]

AR93 PositLayerIsAdjLayer......... ->

ARCH: POSIT: LAYER id IS ADJACENT LAYER id [z95]

AR94 PositLayerIsBelowLayer....... ->

ARCH: POSIT: LAYER id IS BELOW LAYER id [z96]

AR95 PositLayerIsAboveLayer....... ->

ARCH: POSIT: LAYER id IS ABOVE LAYER id [z98]

AR96 PositLayerIsectsPartitAtIsect. ->

ARCH: POSIT: LAYER id INTERSECTS PARTITION id AT INTERSECTION id [z130]

AR97 PositLayerIsectsStrataAtIsect. ->

ARCH: POSIT: LAYER id INTERSECTS STRATA id AT INTERSECTION id [z131]

AR98 PositLayerIsectsTierAtIsect... ->

ARCH: POSIT: LAYER id INTERSECTS TIER id AT INTERSECTION id [z132]

AR99 PositNodeHasComponent......... ->

ARCH: POSIT: NODE id HAS COMPONENT id [z111]

AR100 PositNodeHasIsect........... ->

ARCH: POSIT: NODE id HAS INTERSECTION id [z112]

AR101 PositNodeHasUnit............. ->

ARCH: POSIT: NODE id HAS UNIT id [z328]

AR102 PositPartitIsectsLayerAtIsect. ->

ARCH: POSIT: PARTITION id INTERSECTS LAYER id AT INTERSECTION id [z133]

AR103 PositPartitIsOnLeftOfPartit... ->

ARCH: POSIT: PARTITION id IS ON LEFT OF PARTITION id [z127]

AR104 PositPartitIsAdjPartit........ ->

ARCH: POSIT: PARTITION id IS ADJACENT PARTITION id [z125]

AR105 PositPartitIsOnRightOfPartit.. ->
ARCH: POSIT: PARTITION id IS ON RIGHT OF PARTITION id

AR106 PositPartitIsectsStrataAtIsect ->

ARCH: POSIT: PARTITION id INTERSECTS STRATA id AT INTERSECTION id

AR107 PositPartitIsectsTierAtIsect.. ->

ARCH: POSIT: PARTITION id INTERSECTS TIER id AT INTERSECTION id

AR108 PositStrataIsectsLayerAtIsect. ->

ARCH: POSIT: STRATA id INTERSECTS LAYER id AT INTERSECTION id

AR109 PositStrataIsectsPartitAtIsect ->

ARCH: POSIT: STRATA id INTERSECTS PARTITION id AT INTERSECTION id

AR110 PositStratalSectsAdjStrata....... ->

ARCH: POSIT: STRATA id IS ADJACENT STRATA id

AR111 PositStratalSectsHitherStrata..... ->

ARCH: POSIT: STRATA id IS HITHER STRATA id

AR112 PositStratalSectsThitherStrata.... ->

ARCH: POSIT: STRATA id IS THITHER STRATA id

AR113 PositStratalSectsTierAtIsect.. ->

ARCH: POSIT: STRATA id INTERSECTS TIER id AT INTERSECTION id

AR114 PositTierIsectsLayerAtIsect... ->

ARCH: POSIT: TIER id INTERSECTS LAYER id AT INTERSECTION id

AR115 PositTierIsectsPartitAtIsect.. ->

ARCH: POSIT: TIER id INTERSECTS PARTITION id AT INTERSECTION id

AR116 PositTierIsectsStrataAtIsect.. ->

ARCH: POSIT: TIER id INTERSECTS STRATA id AT INTERSECTION id

AR117 PositTierIsectsTierAtIsect.. ->

ARCH: POSIT: TIER id IS ADJACENT TIER id

AR118 PositTierIsectsFrontTier........ ->

ARCH: POSIT: TIER id IS FRONT TIER id

AR119 PositTierIsectsBehindTier........ ->

ARCH: POSIT: TIER id IS BEHIND TIER id
AT1 AssignAmataStateToSvar........ ->

AUTOMATA: ASSIGN AUTOMATA id STATE id TO STATEVAR id [z299]

AT2 CnectTransitionFrmStateToState ->

AUTOMATA: CONNECT TRANSITION id FROM STATE id TO STATE id [z296]

AT3 DefIdentIsAction............... ->

AUTOMATA: DEFINE: IDENTIFIER id IS ACTION [z292]

AT4 DefIdentIsAmata................ ->

AUTOMATA: DEFINE: IDENTIFIER id IS AUTOMATA [z300]

AT5 DefIdentIsMode................ ->

AUTOMATA: DEFINE: IDENTIFIER id IS MODE [z308]

AT6 DefIdentIsStateA............... ->

AUTOMATA: DEFINE: IDENTIFIER id IS STATE [z322]

AT7 DefIdentIsSvar................ ->

AUTOMATA: DEFINE: IDENTIFIER id IS STATEVAR id [z311]

AT8 DefIdentIsTransition.......... ->

AUTOMATA: DEFINE: IDENTIFIER id IS TRANSITION id [z314]

AT9 DefIdentIsWorker............... ->

AUTOMATA: DEFINE: IDENTIFIER id IS WORKER [z320]

AT10 DoActionWhenEventInState...... ->

AUTOMATA: DO ACTION id WHEN EVENT id IN STATE id [z304]

AT11 DoActionOnEventForTransition.. ->

AUTOMATA: DO ACTION id ON EVENT id FOR TRANSITION id [z316]

AT12 AssertEventOccurs............... ->

AUTOMATA: ASSERT: EVENT id OCCURS [z317]

AT13 AssertModeHasTransitionDisabled ->

AUTOMATA: ASSERT: MODE id HAS TRANSITION id DISABLED [z307]

AT14 AssertModelsOn................ ->

AUTOMATA: ASSERT: MODE id IS ON [z305]

AT15 AssertModelsOff............... ->

Copyright 1990 Kent D. Palmer. All Rights Reserved. Page: 424
AUTOMATA: ASSERT: MODE id IS OFF [z306]

AT16 AssertStateActivated......... ->

AUTOMATA: ASSERT: STATE id ACTIVATED [z293]

AT17 AssertStateDeactivated......... ->

AUTOMATA: ASSERT: STATE id DEACTIVATED [z294]

AT18 OnEventDoActionForTransition.. ->

AUTOMATA: ON EVENT id DO ACTION id FOR TRANSITION id [z315]

AT19 PositAmataIncludesAmata....... ->

AUTOMATA: POSIT: AUTOMATA id INCLUDES AUTOMATA id [z301]

AT20 PositAmataHasState............ ->

AUTOMATA: POSIT: AUTOMATA id HAS STATE id [z297]

AT21 PositAmataHasSvar............. ->

AUTOMATA: POSIT: AUTOMATA id HAS STATEVAR id [z298]

AT22 PositAmataHasTransition....... ->

AUTOMATA: POSIT: AUTOMATA id HAS TRANSITION id [z295]

AT23 PositTransitionHasAction...... ->

AUTOMATA: POSIT: TRANSITION id HAS ACTION id [z312]

AT24 PositTransitionHasEvent....... ->

AUTOMATA: POSIT: TRANSITION id HAS EVENT id [z313]

AT25 PositWorkerDoesActionInState.. ->

AUTOMATA: POSIT: WORKER id DOES ACTION id IN STATE id [z318]

AT26 PositWorkerDoesAction......... ->

AUTOMATA: POSIT: WORKER id DOES ACTION id [z319]

AT27 PositWorkerIsProcedure........ ->

AUTOMATA: POSIT: WORKER id IS PROCEDURE id [z321]

AT28 ReadSvarIntoVariable.......... ->

AUTOMATA: READ STATEVAR id INTO VARIABLE id [z309]

AT29 SetSvarToState................. ->

AUTOMATA: SET STATEVAR id TO STATE id [z303]
AT30 WhenEventDoActionInState...... ->

AUTOMATA: WHEN EVENT id DO ACTION id IN STATE id [z302]

AT31 WriteSvarFrmVariable........... ->

AUTOMATA: WRITE STATEVAR id FROM VARIABLE id [z310]

DM1 DefDocumentIsType.............. ->

DOMAIN: DEFINE: DOCUMENT id IS TYPE id [z176]

DM2 DefIdentIsCharacteristic...... ->

DOMAIN: DEFINE: IDENTIFIER id IS CHARACTERISTIC [z201]

DM3 DefIdentIsDocumentOfType...... ->

DOMAIN: DEFINE: IDENTIFIER id IS DOCUMENT OF TYPE id [z172]

DM4 DefIdentIsProductOfType....... ->

DOMAIN: DEFINE: IDENTIFIER id IS PRODUCT OF TYPE id [z173]

DM5 DefIdentIsRepresentationOfType ->

DOMAIN: DEFINE: IDENTIFIER id IS REPRESENTATION OF TYPE id [z171]

DM6 DefIdentIsSpecificationOfType. ->

DOMAIN: DEFINE: IDENTIFIER id IS SPECIFICATION OF TYPE id [z174]

DM7 DefProductIsType............... ->

DOMAIN: DEFINE: PRODUCT id IS TYPE id [z177]

DM8 DefRepresentationIsType....... ->

DOMAIN: DEFINE: REPRESENTATION id IS TYPE id [z175]

DM9 DefSpecificationIsType........ ->

DOMAIN: DEFINE: SPECIFICATION id IS TYPE id [z178]

DM10 DefDomainIsOfType............. ->

DOMAIN: DEFINE: DOMAIN id IS OF TYPE id [z159]

DM11 DefIdentIsAggreg.............. ->

DOMAIN: DEFINE: IDENTIFIER id IS AGGREGATION [z333]

DM12 DefIdentIsAuthor............. ->

DOMAIN: DEFINE: IDENTIFIER id IS AUTHOR [z191]

DM13 DefIdentIsBuild.............. ->
DOMAIN: DEFINE: IDENTIFIER id IS BUILD [z192]

DM14 DefIdentIsCategory......... ->

DOMAIN: DEFINE: IDENTIFIER id IS CATEGORY [z198]

DM15 DefIdentIsDate.............. ->

DOMAIN: DEFINE: IDENTIFIER id IS DATE [z190]

DM16 DefIdentIsDocument.......... ->

DOMAIN: DEFINE: IDENTIFIER id IS DOCUMENT [z168]

DM17 DefIdentIsDomainOfType..... ->

DOMAIN: DEFINE: IDENTIFIER id IS DOMAIN OF TYPE id [z158]

DM18 DefIdentIsDomain............. ->

DOMAIN: DEFINE: IDENTIFIER id IS DOMAIN [z157]

DM19 DefIdentIsGlossary.......... ->

DOMAIN: DEFINE: IDENTIFIER id IS GLOSSARY [z167]

DM20 DefIdentIsProduct.......... ->

DOMAIN: DEFINE: IDENTIFIER id IS PRODUCT [z169]

DM21 DefIdentIsRepresentation... ->

DOMAIN: DEFINE: IDENTIFIER id IS REPRESENTATION [z165]

DM22 DefIdentIsReqOfType......... ->

DOMAIN: DEFINE: IDENTIFIER id IS REQUIREMENT OF TYPE id [z211]

DM23 DefIdentIsReq................ ->

DOMAIN: DEFINE: IDENTIFIER id IS REQUIREMENT [z210]

DM24 DefIdentIsSpecification..... ->

DOMAIN: DEFINE: IDENTIFIER id IS SPECIFICATION [z170]

DM25 DefIdentIsTaxon............. ->

DOMAIN: DEFINE: IDENTIFIER id IS TAXON [z197]

DM26 DefIdentIsTaxonomy.......... ->

DOMAIN: DEFINE: IDENTIFIER id IS TAXONOMY [z166]

DM27 DefIdentIsVersion.......... ->

DOMAIN: DEFINE: IDENTIFIER id IS VERSION [z189]
DM28 DefReqIsType.................. ->
  DOMAIN: DEFINE: REQUIREMENT id IS TYPE id [z212]

DM29 PositAnyidRelatesToReq......... ->
  DOMAIN: POSIT: ANYID id RELATES TO REQUIREMENT id [z213]

DM30 PositCategoryRelatesToCategory->
  DOMAIN: POSIT: CATEGORY id RELATES TO CATEGORY id [z203]

DM31 PositDomainHasSpec................ ->
  DOMAIN: POSIT: DOMAIN id HAS SPECIFICATION id [z331]

DM32 PositReqHasText.................. ->
  DOMAIN: POSIT: REQUIREMENT id HAS TEXT st [z332]

DM33 PositAggregHasAnyid............... ->
  DOMAIN: POSIT: AGGREGATION id HAS ANYID id [z336]

DM34 PositAggregHasArtifact........... ->
  DOMAIN: POSIT: AGGREGATION id HAS ARTIFACT id [z335]

DM35 PositConfigHasAuthor............... ->
  DOMAIN: POSIT: AGGREGATION id HAS AUTHOR id [z187]

DM36 PositConfigHasBuild............... ->
  DOMAIN: POSIT: AGGREGATION id HAS BUILD id [z188]

DM37 PositConfigHasDate................ ->
  DOMAIN: POSIT: AGGREGATION id HAS DATE id [z186]

DM38 PositConfigHasVersion............. ->
  DOMAIN: POSIT: AGGREGATION id HAS VERSION id [z185]

DM39 PositAnyidTracesToReq............. ->
  DOMAIN: POSIT: ANYID id TRACES TO REQUIREMENT id [z214]

DM40 PositAnyidTracesFrmReq............ ->
  DOMAIN: POSIT: ANYID id TRACES FROM REQUIREMENT id [z215]

DM41 PositCategoryIncludesCategory. ->
  DOMAIN: POSIT: CATEGORY id INCLUDES CATEGORY id [z202]

DM42 PositCategoryIsSet............... ->
DOMAIN: POSIT: CATEGORY id IS SET id [z199]

DM43 PositCategoryHasTaxon.... ->

DOMAIN: POSIT: CATEGORY id HAS TAXON id [z196]

DM44 PositConfigHasAggreg......... ->

DOMAIN: POSIT: CONFIGURATION id HAS AGGREGATION id [z334]

DM45 PositDefinitionHasText........ ->

DOMAIN: POSIT: DEFINITION id HAS TEXT st [z208]

DM46 PositDocumentHasConfig......... ->

DOMAIN: POSIT: DOCUMENT id HAS CONFIGURATION id [z182]

DM47 PositDomainHasDocument......... ->

DOMAIN: POSIT: DOMAIN id HAS DOCUMENT id [z163]

DM48 PositDomainHasGlossary......... ->

DOMAIN: POSIT: DOMAIN id HAS GLOSSARY id [z161]

DM49 PositDomainHasProduct......... ->

DOMAIN: POSIT: DOMAIN id HAS PRODUCT id [z164]

DM50 PositDomainHasRepresentation.. ->

DOMAIN: POSIT: DOMAIN id HAS REPRESENTATION id [z162]

DM51 PositDomainHasTaxonomy......... ->

DOMAIN: POSIT: DOMAIN id HAS TAXONOMY id [z160]

DM52 PositGlossaryHasConfig......... ->

DOMAIN: POSIT: GLOSSARY id HAS CONFIGURATION id [z180]

DM53 PositGlossaryHasDefinition.... ->

DOMAIN: POSIT: GLOSSARY id HAS DEFINITION id [z206]

DM54 PositIdentIsDefinition......... ->

DOMAIN: POSIT: IDENTIFIER id IS DEFINITION [z207]

DM55 PositProductHasConfig......... ->

DOMAIN: POSIT: PRODUCT id HAS CONFIGURATION id [z183]

DM56 PositRepresentationHasConfig.. ->

DOMAIN: POSIT: REPRESENTATION id HAS CONFIGURATION id [z181]
DM57 PositSpecificationHasConfig... ->

DOMAIN: POSIT: SPECIFICATION id HAS CONFIGURATION id [z184]

DM58 PositSpecificationHasReq...... ->

DOMAIN: POSIT: SPECIFICATION id HAS REQUIREMENT id [z209]

DM59 PositTaxonHasCharacteristic... ->

DOMAIN: POSIT: TAXON id HAS CHARACTERISTIC id [z200]

DM60 PositTaxonIsSet................. ->

DOMAIN: POSIT: TAXON id IS SET id [z195]

DM61 PositTaxonIncludesTaxon....... ->

DOMAIN: POSIT: TAXON id INCLUDES TAXON id [z204]

DM62 PositTaxonRelatesToTaxon...... ->

DOMAIN: POSIT: TAXON id RELATES TO TAXON id [z205]

DM63 PositTaxonomyHasCategory...... ->

DOMAIN: POSIT: TAXONOMY id HAS CATEGORY id [z193]

DM64 PositTaxonomyHasConfig........ ->

DOMAIN: POSIT: TAXONOMY id HAS CONFIGURATION id [z179]

DM65 PositTaxonomyHasTaxon......... ->

DOMAIN: POSIT: TAXONOMY id HAS TAXON id [z194]

IN1 DefDatumIsType................ ->

INFORMATION: DEFINE: DATUM id IS TYPE id [z744]

IN2 DefIdenierIsInfopacket...... ->

INFORMATION: DEFINE: IDENTIFIER id IS INFOPACKET id [z538]

IN3 DefIdentIsDatumOfType......... ->

INFORMATION: DEFINE: IDENTIFIER id IS DATUM OF TYPE id [z742]

IN4 DefIdentIsDatum............... ->

INFORMATION: DEFINE: IDENTIFIER id IS DATUM I [z534]

IN5 DefIdentIsInfopOfType........... ->

INFORMATION: DEFINE: IDENTIFIER id IS INFOPACKET OF TYPE id [z741]

IN6 DefInfopacketIsType.......... ->
INFORMATION: DEFINE: INFOPACKET id IS TYPE id [z743]

IN7 IfEventSetBanToInsignia....... ->

INFORMATION: IF EVENT id SET BANNER id TO INSIGNIA id [z529]

IN8 IfEventChangeEntityToMode..... ->

INFORMATION: IF EVENT id CHANGE ENTITY id TO MODE id [z528]

IN9 IfEventAndEventThenEvent...... ->

INFORMATION: IF EVENT id AND EVENT id THEN EVENT id [z524]

IN10 IfEventOrNotEventThenEvent.... ->

INFORMATION: IF EVENT id OR NOT EVENT id THEN EVENT id [z527]

IN11 IfEventOrEventThenEvent....... ->

INFORMATION: IF EVENT id OR EVENT id THEN EVENT id [z525]

IN12 IfEventAndNotEventThenEvent... ->

INFORMATION: IF EVENT id AND NOT EVENT id THEN EVENT id [z526]

IN13 IfEventResetEntityAutonetic. ->

INFORMATION: IF EVENT id RESET EVENTITY id AUTONETIC id [z533]

IN14 IfEventResetEntityFlag....... ->

INFORMATION: IF EVENT id RESET EVENTITY id FLAG id [z532]

IN15 IfEventSetPetrinetToPlace..... ->

INFORMATION: IF EVENT id SET PETRINET id TO PLACE id [z530]

IN16 IfEventSetSvarToState......... ->

INFORMATION: IF EVENT id SET STATEVAR id TO STATE id [z531]

IN17 InqAttrValueIntoAttrForEntity. ->

INFORMATION: INQUIRE ATTRIBUTE id VALUE INTO ATTRIBUTE id FOR ENTITY id [z543]

IN18 InqAttrValueIntoAttrForObject. ->

INFORMATION: INQUIRE ATTRIBUTE id VALUE INTO ATTRIBUTE id FOR OBJECT id [z544]

IN19 InqAttrValueIntoAttr......... ->

INFORMATION: INQUIRE ATTRIBUTE id VALUE INTO ATTRIBUTE id [z542]

IN20 InqAttrGenChangeEvent......... ->

INFORMATION: INQUIRE ATTRIBUTE id GENERATE CHANGE EVENT id [z539]
IN21 InqDatumGenMovementEvent...... ->

INFORMATION: INQUIRE DATUM id GENERATE MOVEMENT EVENT id [z540]

IN22 InqDatumLocation................. ->

INFORMATION: INQUIRE DATUM id LOCATION [z541]

IN23 InqInfopacketGenMovementEvent. ->

INFORMATION: INQUIRE INFOPACKET id GENERATE MOVEMENT EVENT id [z545]

IN24 InqInfopacketLocation.......... ->

INFORMATION: INQUIRE INFOPACKET id LOCATION [z546]

IN25 PositDatumHasValueExp......... ->

INFORMATION: POSIT: DATUM id HAS VALUE EXPRESSION ex [z536]

IN26 PositDatumIsInInfop............ ->

INFORMATION: POSIT: DATUM id IS IN INFOPACKET id [z739]

IN27 PositDatumHasInterval......... ->

INFORMATION: POSIT: DATUM id HAS INTERVAL id [z535]

IN28 PositInfopIsInCnectr.......... ->

INFORMATION: POSIT: INFOPACKET id IS IN CONNECTOR id [z696]

IN29 PositInfopacketHasSeqdDatum... ->

INFORMATION: POSIT: INFOPACKET id HAS SEQUENCED DATUM id [z537]

IN30 PositInfopHasSizeExp......... ->

INFORMATION: POSIT: INFOPACKET id HAS SIZE EXPRESSION ex [z740]

IN31 ReadAttrFromInfopInCnectr.... ->

INFORMATION: READ ATTRIBUTE id FROM INFOPACKET id IN CONNECTOR id [z738]

IN32 ReadAttrFromInfopAtInterval.. ->

INFORMATION: READ ATTRIBUTE id FROM INFOPACKET id AT INTERVAL id [z746]

IN33 ReadInfopacketByAttr.......... ->

INFORMATION: READ INFOPACKET id BY ATTRIBUTE id [z520]

IN34 SetAttrToDatumForEntity....... ->

INFORMATION: SET ATTRIBUTE id TO DATUM id FOR ENTITY id [z521]

IN35 SetAttrToDatumForObject....... ->
INFORMATION: SET ATTRIBUTE id TO DATUM id FOR OBJECT id [z523]

IN36 SetAttrToDatum................... ->

INFORMATION: SET ATTRIBUTE id TO DATUM id [z522]

IN37 ThreshExpGenHigherEventForAttr ->

INFORMATION: THRESHOLD EXPRESSION ex GENERATE HIGHER EVENT id FOR ATTRIBUTE [z548]

IN38 ThreshExpGenLowerEventForAttr. ->

INFORMATION: THRESHOLD EXPRESSION ex GENERATE LOWER EVENT id FOR ATTRIBUTE [z547]

IN39 WriteAttrInEntityToInfop...... ->

INFORMATION: WRITE ATTRIBUTE id IN ENTITY id TO INFOPACKET id [z735]

IN40 WriteAttrToInfopInCnectr...... ->

INFORMATION: WRITE ATTRIBUTE id TO INFOPACKET id IN CONNECTOR id [z737]

IN41 WriteAttrToInfop.............. ->

INFORMATION: WRITE ATTRIBUTE id TO INFOPACKET id [z736]

IN42 WriteAttrInObjToInfop........... ->

INFORMATION: WRITE ATTRIBUTE id IN OBJECT id TO INFOPACKET id [z734]

IN43 WriteAttrToInfopAtInterval.... ->

INFORMATION: WRITE ATTRIBUTUTE id TO INFOPACKET id AT INTERVAL id [z745]

LS1 ChangeListBecomesSet.......... ->

LIST: CHANGE: LIST id BECOMES SET id [z26]

LS2 DefIdentIsListWith{List}...... ->

LIST: DEFINE IDENTIFIER id IS LIST id WITH { list } [z29]

LS3 DefIdentItsList................... ->

LIST: DEFINE: IDENTIFIER id IS LIST [z28]

LS4 DefMasterlistContainsList..... ->

LIST: DEFINE: MASTERLIST id CONTAINS LIST id [z30]

LS5 EmptyList...................... ->

LIST: EMPTY LIST id [z15]

LS6 ExcludeElmFrmList............. ->

LIST: EXCLUDE ELEMENT id FROM LIST id [z14]
Integral Software Engineering Methodology: LANGUAGE

LS7 FindIdentInListForAll........ ->

LIST: FIND IDENTIFIER id IN LIST FOR ALL [z16]

LS8 FindIdentInList................. ->

LIST: FIND IDENTIFIER id IN LIST id [z17]

LS9 InqHeadOfList.................... ->

LIST: INQUIRE HEAD OF LIST id [z18]

LS10 InqMembershipOfList............. ->

LIST: INQUIRE MEMBERSHIP OF LIST id [z19]

LS11 InqPositionInList............... ->

LIST: INQUIRE POSITION nm IN LIST id [z20]

LS12 InqRestOfList.................... ->

LIST: INQUIRE REST OF LIST id [z21]

LS13 InqSetInList..................... ->

LIST: INQUIRE SET IN LIST id [z22]

LS14 InqTailOfList.................... ->

LIST: INQUIRE TAIL OF LIST id [z23]

LS15 IsectListWithListIntoList..... ->

LIST: INTERSECTION LIST id WITH LIST id INTO LIST id [z24]

LS16 JamElmIntoList.................. ->

LIST: JAM ELEMENT id INTO LIST id [z25]

LS17 NullList........................ ->

LIST: NULL LIST id [z31]

LS18 PositListContains{List}....... ->

LIST: POSIT: LIST id CONTAINS { list } [z27]

LS19 PutElmAfterElmInList.......... ->

LIST: PUT ELEMENT id AFTER ELEMENT id IN LIST id [z32]

LS20 PutElmBeforeElmInList......... ->

LIST: PUT ELEMENT id BEFORE ELEMENT id IN LIST id [z33]

LS21 PutElmAtHeadOfList............. ->
LIST: PUT ELEMENT id AT HEAD OF LIST id [z34]

LS22 PutElmAtPositionOfList......... ->

LIST: PUT ELEMENT id AT POSITION nm OF LIST id [z35]

LS23 PutElmAtTailOfList............. ->

LIST: PUT ELEMENT id AT TAIL OF LIST id [z36]

LS24 ReorderListFrmSeqToSeq......... ->

LIST: REORDER LIST id FROM SEQUENCE sr TO SEQUENCE sr [z37]

LS25 ReplaceElmWithElmInTheList..... ->

LIST: REPLACE ELEMENT id WITH ELEMENT id IN LIST id [z38]

LS26 UnionListWithListIntoList..... ->

LIST: UNION LIST id WITH LIST id INTO LIST id [z39]

MC1 DefIdentIsExecutive............. ->

    MACHINE: DEFINE: IDENTIFIER id IS EXECUTIVE id [z568]

MC2 DefIdentIsFunction............... ->

    MACHINE: DEFINE: IDENTIFIER id IS FUNCTION id [z569]

MC3 DefIdentIsAgent.................. ->

    MACHINE: DEFINE: IDENTIFIER id IS AGENT [z550]

MC4 DefIdentIsInstr.................. ->

    MACHINE: DEFINE: IDENTIFIER id IS INSTRUCTION [z583]

MC5 DefIdentIsInstrset............... ->

    MACHINE: DEFINE: IDENTIFIER id IS INSTRUCTIONSET [z581]

MC6 DefIdentIsMachine................. ->

    MACHINE: DEFINE: IDENTIFIER id IS MACHINE [z586]

MC7 DefIdentIsMapping............... ->

    MACHINE: DEFINE: IDENTIFIER id IS MAPPING [z590]

MC8 DefIdentIsMessage............... ->

    MACHINE: DEFINE: IDENTIFIER id IS MESSAGE [z591]

MC9 DefIdentIsOp..................... ->

    MACHINE: DEFINE: IDENTIFIER id IS OPERATION [z599]
MC10 DefIdentIsProcess.............. ->
  MACHINE: DEFINE: IDENTIFIER id IS PROCESS [z600]
MC11 DefIdentIsProcessor............. ->
  MACHINE: DEFINE: IDENTIFIER id IS PROCESSOR id [z603]
MC12 DefIdentIsQueue............... ->
  MACHINE: DEFINE: IDENTIFIER id IS QUEUE [z609]
MC13 DefIdentIsTask................ ->
  MACHINE: DEFINE: IDENTIFIER id IS TASK [z610]
MC14 DefIdentIsThread............... ->
  MACHINE: DEFINE: IDENTIFIER id IS THREAD [z615]
MC15 DefIdentIsTransform............ ->
  MACHINE: DEFINE: IDENTIFIER id IS TRANSFORM [z616]
MC16 FiringInstrSetAmataStateActive ->
  MACHINE: FIRING INSTRUCTION id SET AUTOMATA id STATE id ACTIVE [z557]
MC17 FiringInstrSetBanToInsignia... ->
  MACHINE: FIRING INSTRUCTION id SET BANNER id TO INSIGNIA id [z560]
MC18 FiringInstrGenerateEvent....... ->
  MACHINE: FIRING INSTRUCTION id GENERATE EVENT id [z561]
MC19 FiringInstrMarkPetrinetPlace.. ->
  MACHINE: FIRING INSTRUCTION id MARK PETRINET id PLACE id [z559]
MC20 FiringInstrSetSvarToState..... ->
  MACHINE: FIRING INSTRUCTION id SET STATEVAR id TO STATE id [z558]
MC21 InqInfopackNextDatum.......... ->
  MACHINE: INQUIRE INFOPACK id NEXT DATUM [z571]
MC22 InqQueueNextMessage.......... ->
  MACHINE: INQUIRE QUEUE id NEXT MESSAGE [z593]
MC23 InvokeObjectOpWithParameters.. ->
  MACHINE: INVOKE OBJECT id OPERATION id WITH PARAMETERS pr [z584]
MACHINE: INVOKE OBJECT id OPERATION id [z585]
MC25 InvokeQueueNextMessage........ ->

MACHINE: INVOKE QUEUE id NEXT MESSAGE [z594]
MC26 MappingFunctionToTask......... ->

MACHINE: MAPPING FUNCTION id TO TASK id [z555]
MC27 MappingProcessToAgent......... ->

MACHINE: MAPPING PROCESS id TO AGENT id [z554]
MC28 MappingTransformToProcessor... ->

MACHINE: MAPPING TRANSFORM id TO PROCESSOR id [z556]
MC29 ModifyAttrWithObjectOp........ ->

MACHINE: MODIFY ATTRIBUTE id WITH OBJECT id OPERATION id [z596]
MC30 PositEventityHasExecutive..... ->

MACHINE: POSIT: EVENTITY id HAS EXECUTIVE id [z566]
MC31 PositEventityIsFunction........ ->

MACHINE: POSIT: EVENTITY id IS FUNCTION id [z563]
MC32 PositEventityIsTask............ ->

MACHINE: POSIT: EVENTITY id IS TASK id [z565]
MC33 PositExecutiveIsPetrinet....... ->

MACHINE: POSIT: EXECUTIVE id IS PETRINET id [z567]
MC34 ExecutiveHasAmata.............. ->

MACHINE: POSIT: EXECUTIVE id HAS AUTOMATA id [z905]
MC35 PositExecutiveHasTask.......... ->

MACHINE: POSIT: EXECUTIVE id HAS TASK id [z608]
MC36 PositFunctionAccessesOp........ ->

MACHINE: POSIT: FUNCTION id ACCESSES OPERATION id [z570]
MC37 PositImplContainsInstr......... ->

MACHINE: POSIT: IMPLEMENTATION id CONTAINS INSTRUCTION id [z574]
MC38 PositImplContainsOp............ ->

MACHINE: POSIT: IMPLEMENTATION id CONTAINS OPERATION id [z575]
MC39 PositInstrIsAmataAction....... ->

MACHINE: POSIT: INSTRUCTION id IS AUTOMATA id ACTION id [z572]

MC40 PositInstrImplFunction.. ->

MACHINE: POSIT: INSTRUCTION id IMPLEMENTS FUNCTION id [z573]

MC41 PositInstrHasMachine......... ->

MACHINE: POSIT: INSTRUCTION id HAS MACHINE id [z576]

MC42 PositInstrIsOp............... ->

MACHINE: POSIT: INSTRUCTION id IS OPERATION id [z578]

MC43 PositInstrIsPetrinetTransit... ->

MACHINE: POSIT: INSTRUCTION id IS PETRINET id TRANSIT id [z579]

MC44 PositInstrIsWorker............ ->

MACHINE: POSIT: INSTRUCTION id IS WORKER id [z580]

MC45 PositInstrSendsMessage........ ->

MACHINE: POSIT: INSTRUCTION id SENDS MESSAGE id [z911]

MC46 PositInstrsetHasInstr......... ->

MACHINE: POSIT: INSTRUCTIONSET id HAS INSTRUCTION id [z582]

MC47 PositMachineHasAmata.......... ->

MACHINE: POSIT: MACHINE id HAS AUTOMATA id [z587]

MC48 PositMachineIsInstr.......... ->

MACHINE: POSIT: MACHINE id IS INSTRUCTION id [z577]

MC49 PositMachineHasInstrset....... ->

MACHINE: POSIT: MACHINE id HAS INSTRUCTIONSET id [z588]

MC50 PositMachineHasPetrinet....... ->

MACHINE: POSIT: MACHINE id HAS PETRINET id [z589]

MC51 PositMessageContainsOp........ ->

MACHINE: POSIT: MESSAGE id CONTAINS OPERATION id [z908]

MC52 PositMessageInvokesInstr...... ->

MACHINE: POSIT: MESSAGE id INVOKES INSTRUCTION id [z910]
MACHINE: POSIT: MESSAGE id IS INFOPACKET id [z592]

MC54 PositObjectHasMessage......... ->

MACHINE: POSIT: OBJECT id HAS MESSAGE id [z906]

MC55 PositObjectHasOp............... ->

MACHINE: POSIT: OBJECT id HAS OPERATION id [z595]

MC56 PositOpIsWorker............... ->

MACHINE: POSIT: OPERATION id IS WORKER id [z598]

MC57 PositProcessIncludesProcess... ->

MACHINE: POSIT: PROCESS id INCLUDES PROCESS id [z601]

MC58 PositProcessHasTransform...... ->

MACHINE: POSIT: PROCESS id HAS TRANSFORM id [z602]

MC59 PositProcessorHasExecutive.... ->

MACHINE: POSIT: PROCESSOR id HAS EXECUTIVE id [z607]

MC60 PositProcessorHasMemory....... ->

MACHINE: POSIT: PROCESSOR id HAS MEMORY nm [z604]

MC61 PositProcessorHasSpeed........ ->

MACHINE: POSIT: PROCESSOR id HAS SPEED nm [z605]

MC62 PositProcessorHasTask......... ->

MACHINE: POSIT: PROCESSOR id HAS TASK id [z904]

MC63 PositProcessorHasThroughput... ->

MACHINE: POSIT: PROCESSOR id HAS THROUGHPUT nm [z606]

MC64 PositQueueContainsMessage..... ->

MACHINE: POSIT: QUEUE id CONTAINS MESSAGE id [z907]

MC65 PositTaskHasMachine.......... ->

MACHINE: POSIT: TASK id HAS MACHINE id [z611]

MC66 PositTaskAccessesOp.......... ->

MACHINE: POSIT: TASK id ACCESSES OPERATION id [z564]

MC67 PositTaskHasPriority......... ->

MACHINE: POSIT: TASK id HAS PRIORITY id [z612]
MC68 PositTaskHasQueue.......... ->
MACHINE: POSIT: TASK id HAS QUEUE id [z613]

MC69 PositTaskHasThread......... ->
MACHINE: POSIT: TASK id HAS THREAD id [z614]

MC70 PositTransformHasFunction..... ->
MACHINE: POSIT: TRANSFORM id HAS FUNCTION id [z617]

MC71 PositTransformImplFunction.... ->
MACHINE: POSIT: TRANSFORM id IMPLEMENTS FUNCTION id [z909]

M72 PositWorkerImplFunction. ->
MACHINE: POSIT: WORKER id IMPLEMENTS FUNCTION id [z907]

MC73 ReadAttrWithObjectOp......... ->
MACHINE: READ ATTRIBUTE id WITH OBJECT id OPERATION id [z597]

MC74 SetDatumValueByInstr.......... ->
MACHINE: SET DATUM id VALUE BY INSTRUCTION id [z562]

MC75 PositAgentIncludesAgent....... ->
MACHINE: POSIT: AGENT id INCLUDES AGENT id [z551]

MC76 PositAgentHasProcessorOnPlat.. ->
MACHINE: POSIT: AGENT id HAS PROCESSOR id ON PLATFORM id [z553]

MC77 PositAgentHasProcessor......... ->
MACHINE: POSIT: AGENT id HAS PROCESSOR id [z552]

PR1 CnectDflowFromDstoreToProcess. ->
PROCESS: CONNECT DATAFLOW id FROM DATASTORE id TO PROCESS id [z618]

PR2 CnectDflowFromExtToProcess.... ->
PROCESS: CONNECT DATAFLOW id FROM EXTERNAL id TO PROCESS id [z619]

PR3 CnectDflowFromProcessToDstore. ->
PROCESS: CONNECT DATAFLOW id FROM PROCESS id TO DATASTORE id [z620]

PR4 CnectDflowFromProcessToExt.... ->
PROCESS: CONNECT DATAFLOW id FROM PROCESS id TO EXTERNAL id [z621]

PR5 CnectDflowFromProcessToProcess ->
PROCESS: CONNECT DATAFLOW id FROM PROCESS id TO PROCESS id [z622]

PR6 DefIdentIsActivator........... ->

PROCESS: DEFINE: IDENTIFIER id IS ACTIVATOR [z623]

PR7 DefIdentIsCflow.............. ->

PROCESS: DEFINE: IDENTIFIER id IS CONTROLFLOW [z624]

PR8 DefIdentIsControlspec........ ->

PROCESS: DEFINE: IDENTIFIER id IS CONTROLSPEC [z625]

PR9 DefIdentIsDflow............... ->

PROCESS: DEFINE: IDENTIFIER id IS DATAFLOW [z626]

PR10 DefIdentIsDstore............. ->

PROCESS: DEFINE: IDENTIFIER id IS DATASTORE [z627]

PR11 DefIdentIsDtab................ ->

PROCESS: DEFINE: IDENTIFIER id IS DECISIONTABLE id [z628]

PR12 DefIdentIsExt................ ->

PROCESS: DEFINE: IDENTIFIER id IS EXTERNAL [z629]

PR13 PositActivatorActivatesProcess ->

PROCESS: POSIT: ACTIVATOR id ACTIVATES PROCESS id P[z631]

PR14 PositActivatorDactsProcess.... ->

PROCESS: POSIT: ACTIVATOR id DEACTIVATES PROCESS id [z632]

PR15 PositAmataDrivesActivator..... ->

PROCESS: POSIT: AUTOMATA id DRIVES ACTIVATOR id [z633]

PR16 PositContextHasProcess........ ->

PROCESS: POSIT: CONTEXT id HAS PROCESS id [z634]

PR17 PositCflowLeadsToControlspec.. ->

PROCESS: POSIT: CONTROLFLOW id LEADS TO CONTROLSPEC id [z635]

PR18 PositCflowIsEvent............ ->

PROCESS: POSIT: CONTROLFLOW id IS EVENT id [z636]

PR19 PositCflowAttachedToProcess... ->

PROCESS: POSIT: CONTROLFLOW id ATTACHED TO PROCESS id [z637]
PR20 PositControlspecHasActivator.. ->

PROCESS: POSIT: CONTROLSPEC id HAS ACTIVATOR id [z638]

PR21 PositControlspecHasAmata...... ->

PROCESS: POSIT: CONTROLSPEC id HAS AUTOMATA id [z639]

PR22 PositControlspecHasDtab....... ->

PROCESS: POSIT: CONTROLSPEC id HAS DECISIONTABLE id [z640]

PR23 PositControlspecHasPetrinet... ->

PROCESS: POSIT: CONTROLSPEC id HAS PETRINET id [z641]

PR24 PositDflowIncludesDflow....... ->

PROCESS: POSIT: DATAFLOW id INCLUDES DATAFLOW id [z642]

PR25 PositDstoreIncludesObject..... ->

PROCESS: POSIT: DATASTORE id INCLUDES OBJECT id [z643]

PR26 PositDtabDrivesAmata.......... ->

PROCESS: POSIT: DECISIONTABLE id DRIVES AUTOMATA id [z644]

PR27 PositDtablIncludesDecision..... ->

PROCESS: POSIT: DECISIONTABLE id INCLUDES DECISION id [z645]

PR28 PositPetrinetDrivesActivator.. ->

PROCESS: POSIT: PETRINET id DRIVES ACTIVATOR id [z646]

PR29 PositProcessOriginatesCflow... ->

PROCESS: POSIT: PROCESS id ORIGINATES CONTROLFLOW id [z647]

PR30 PositProcessReceivesCflow..... ->

PROCESS: POSIT: PROCESS id RECEIVES CONTROLFLOW id [z648]

PR31 PositProcessIncludesProcess... ->

PROCESS: POSIT: PROCESS id INCLUDES PROCESS id [z649]

PT1 CnectDecToPlace................. ->

PETRINET: CONNECT DECISION id TO PLACE id [z341]

PT2 CnectTransitFromDecToDec...... ->

PETRINET: CONNECT TRANSIT id FROM DECISION id TO DECISION id [z347]

PT3 CnectTransitFromDecToPlace.... ->
PETRINET: CONNECT TRANSIT id FROM DECISION id TO PLACE id [z348]

PT4 CnectTransitToDec............. ->

PETRINET: CONNECT TRANSIT id TO DECISION id [z365]

PT5 CnectTransitFromPlaceToDec.... ->

PETRINET: CONNECT TRANSIT id FROM PLACE id TO DECISION id [z349]

PT6 CnectTransitFromPlaceToEntry.. ->

PETRINET: CONNECT TRANSIT id FROM PLACE id TO ENTRY id [z913]

PT7 CnectTransitFromPlaceToExit... ->

PETRINET: CONNECT TRANSIT id FROM PLACE id TO EXIT id [z350]

PT8 CnectTransitFromPlaceToPlace.. ->

PETRINET: CONNECT TRANSIT id FROM PLACE id TO PLACE id [z351]

PT9 DefIdentIsDec.................. ->

PETRINET: DEFINE: IDENTIFIER id IS DECISION id [z352]

PT10 DefIdentIsEntry.............. ->

PETRINET: DEFINE: IDENTIFIER id IS ENTRY [z912]

PT11 DefIdentIsExit............... ->

PETRINET: DEFINE: IDENTIFIER id IS EXIT [z356]

PT12 DefIdentIsInsignia.......... ->

PETRINET: DEFINE: IDENTIFIER id IS INSIGNIA [z357]

PT13 DefIdentIsPetrinet.......... ->

PETRINET: DEFINE: IDENTIFIER id IS PETRINET [z363]

PT14 DefIdentIsPlace............. ->

PETRINET: DEFINE: IDENTIFIER id IS PLACE [z364]

PT15 DefIdentIsTransit.......... ->

PETRINET: DEFINE: IDENTIFIER id IS TRANSIT [z369]

PT16 DefIdnetifierIsBan......... ->

PETRINET: DEFINE: IDENTIFIER id IS BANNER [z342]

PT17 IfDecIsInsigniaThenPlaceNext.. ->

PETRINET: IF DECISION id IS INSIGNIA id THEN PLACE id NEXT [z353]
PT18 AssertPlaceIsMarked............ ->

PETRINET: ASSERT: PLACE id IS MARKED [z358]

PT19 AssertPlaceIsUnmarked........... ->

PETRINET: ASSERT: PLACE id IS UNMARKED [z359]

PT20 PositPlaceIsEntry............. ->

PETRINET: POSIT: PLACE id IS ENTRY [z354]

PT21 PositBanHasInsignia.......... ->

PETRINET: POSIT: BANNER id HAS INSIGNIA id [z343]

PT22 PositPetrinetHasDec........... ->

PETRINET: POSIT: PETRINET id HAS DECISION id [z360]

PT23 PositPetrinetHasPlace......... ->

PETRINET: POSIT: PETRINET id HAS PLACE id [z361]

PT24 PositPetrinetHasSubnet......... ->

PETRINET: POSIT: PETRINET id HAS SUBNET id [z345]

PT25 PositPetrinetHasTransit....... ->

PETRINET: POSIT: PETRINET id HAS TRANSIT id [z362]

PT26 PositTransitSetsBanToInsignia. ->

PETRINET: POSIT: TRANSIT id SETS BANNER id TO INSIGNIA id [z368]

PT27 PositTransitsIsWorker.......... ->

PETRINET: POSIT: TRANSIT id IS WORKER id [z355]

PT28 ReenterPlaceOnExit............. ->

PETRINET: REENTER PLACE id ON EXIT id [z366]

PT29 RunPetrinet................... ->

PETRINET: RUN PETRINET id [z367]

PT30 SetBanToInsignia............... ->

PETRINET: SET BANNER id TO INSIGNIA id [z344]

PT31 TakeExitToPlaceEntry.......... ->

PETRINET: TAKE EXIT id TO PLACE id ENTRY [z346]
SITUATION: ASSIGN ATTRIBUTE id VALUE OF EXPRESSION ex FOR ENTITY id [z428]
SI2 AssignRplValueOfExpForInstRpl. ->

SITUATION: ASSIGN REPLICA id VALUE OF EXPRESSION ex FOR INSTANCE id [z432]
SI3 DefAttrHasRplInEntity. ->

SITUATION: DEFINE: ATTRIBUTE id HAS REPLICA id IN ENTITY id [z434]
SI4 DefAttrIsType. ->

SITUATION: DEFINE: ATTRIBUTE id IS TYPE id [z408]
SI5 DefIdentIsAssembly. ->

SITUATION: DEFINE: IDENTIFIER id IS ASSEMBLY [z405]
SI6 DefIdentIsAttr. ->

SITUATION: DEFINE: IDENTIFIER id IS ATTRIBUTE [z409]
SI7 DefIdentIsBag. ->

SITUATION: DEFINE: IDENTIFIER id IS BAG [z926]
SI8 DefIdentIsBehavior. ->

SITUATION: DEFINE: IDENTIFIER id IS BEHAVIOR [z411]
SI9 DefIdentIsConstellation. ->

SITUATION: DEFINE: IDENTIFIER id IS CONSTELLATION [z924]
SI10 DefIdentIsClass. ->

SITUATION: DEFINE: IDENTIFIER id IS CLASS [z413]
SI11 DefIdentIsEntity. ->

SITUATION: DEFINE: IDENTIFIER id IS ENTITY [z425]
SI12 DefIdentIsEq. ->

SITUATION: DEFINE: IDENTIFIER id IS EQUATION id [z427]
SI13 DefIdentIsFace. ->

SITUATION: DEFINE: IDENTIFIER id IS FACE [z429]
SI14 DefIdentIsInst. ->

SITUATION: DEFINE: IDENTIFIER id IS INSTANCE [z431]
SI15 DefIdentIsObject. ->

SITUATION: DEFINE: IDENTIFIER id IS OBJECT [z435]
SI16 DefIdentIsPart................ ->

SITUATION: DEFINE: IDENTIFIER id IS PART [z438]

SI17 DefIdentIsRel............... ->

SITUATION: DEFINE: IDENTIFIER id IS RELATION [z440]

SI18 DefIdentIsRpl............... ->

SITUATION: DEFINE: IDENTIFIER id IS REPLICA [z433]

SI19 DefIdentIsSite............... ->

SITUATION: DEFINE: IDENTIFIER id IS SITE [z459]

SI20 DefIdentIsSituation......... ->

SITUATION: DEFINE: IDENTIFIER id IS SITUATION [z922]

SI21 DefIdentIsSlot............... ->

SITUATION: DEFINE: IDENTIFIER id IS SLOT [z456]

SI22 PositAssemblyHasAmata........ ->

SITUATION: POSIT: ASSEMBLY id HAS AUTOMATA id [z404]

SI23 PositAssemblyHasPart.......... ->

SITUATION: POSIT: ASSEMBLY id HAS PART id [z403]

SI24 PositAssemblyHasPetrinet..... ->

SITUATION: POSIT: ASSEMBLY id HAS PETRINET id [z406]

SI25 PositBagHasEntity............ ->

SITUATION: POSIT: BAG id HAS ENTITY id [z927]

SI26 PositBehaviorHasAmata........ ->

SITUATION: POSIT: BEHAVIOR id HAS AUTOMATA id [z410]

SI27 PositBehaviorHasPetrinet..... ->

SITUATION: POSIT: BEHAVIOR id HAS PETRINET id [z412]

SI28 PositConstellationHasBag..... ->

SITUATION: POSIT: CONSTELLATION id HAS BAG id [z925]

SI29 PositEntityHasAssembly....... ->

SITUATION: POSIT: ENTITY id HAS ASSEMBLY id [z414]

SI30 PositEntityHasAttr........... ->
SITUATION: POSIT: ENTITY id HAS ATTRIBUTE id [z415]

SITUATION: POSIT: ENTITY id HAS BEHAVIOR id [z416]

SITUATION: POSIT: ENTITY id IS OF CLASS id [z417]

SITUATION: POSIT: ENTITY id HAS FACE id [z418]

SITUATION: POSIT: ENTITY id HAS INSTANCE id [z420]

SITUATION: POSIT: ENTITY id HAS MODE id [z426]

SITUATION: POSIT: ENTITY id HAS NUMBER nm INSTANCES [z419]

SITUATION: POSIT: ENTITY id HAS OBJECT id [z421]

SITUATION: POSIT: ENTITY id HAS SLOT id [z423]

SITUATION: POSIT: EQUATION id IS EXPRESSION ex [z458]

SITUATION: POSIT: FACE id HAS ATTRIBUTE id [z430]

SITUATION: POSIT: INSTANCE id HAS NUMBER nm FOR ENTITY id [z407]

SITUATION: POSIT: OBJECT id HAS ATTRIBUTE id IN POSITION nm [z436]

SITUATION: POSIT: OBJECT id HAS ATTRIBUTE id [z437]

SITUATION: POSIT: PART id IS ENTITY id [z439]
SI45 PositRelFromAnyidToAnyid...... ->

**SITUATION: POSIT: RELATION id FROM ANYID id TO ANYID id [z449]**

SI46 PositRelFromAssemblyToBehavior ->

**SITUATION: POSIT: RELATION id FROM ASSEMBLY id TO BEHAVIOR id [z446]**

SI47 PositRelFromAssemblyToPart.... ->

**SITUATION: POSIT: RELATION id FROM ASSEMBLY id TO PART id [z447]**

SI48 PositRelHasAttr.................. ->

**SITUATION: POSIT: RELATION id HAS ATTRIBUTE id [z441]**

SI49 PositRelFromEntityToAssembly.. ->

**SITUATION: POSIT: RELATION id FROM ENTITY id TO ASSEMBLY id [z450]**

SI50 PositRelFromEntityToBehavior.. ->

**SITUATION: POSIT: RELATION id FROM ENTITY id TO BEHAVIOR id [z451]**

SI51 PositRelFromEntityToEntity.... ->

**SITUATION: POSIT: RELATION id FROM ENTITY id TO ENTITY id [z422]**

SI52 PositRelFromEntityToFace....... ->

**SITUATION: POSIT: RELATION id FROM ENTITY id TO FACE id [z452]**

SI53 PositRelFromEntityToObject.... ->

**SITUATION: POSIT: RELATION id FROM ENTITY id TO OBJECT id [z444]**

SI54 PositRelFromInstToFace........ ->

**SITUATION: POSIT: RELATION id FROM INSTANCE id TO FACE id [z454]**

SI55 PositRelFromObjectToObject.... ->

**SITUATION: POSIT: RELATION id FROM OBJECT id TO OBJECT id [z443]**

SI56 PositRelFromPartToFace......... ->

**SITUATION: POSIT: RELATION id FROM PART id TO FACE id [z453]**

SI57 PositRelFromPartToInst......... ->

**SITUATION: POSIT: RELATION id FROM PART id TO INSTANCE id [z448]**

SI58 PositRelFromPartToPart......... ->

**SITUATION: POSIT: RELATION id FROM PART id TO PART id [z445]**

SI59 PositSiteIsIsect............... ->
SITUATION: POSIT: SITE id IS INTERSECTION id [z466]
SI60 PositSituationHasConstellation ->

SITUATION: POSIT: SITUATION id HAS CONSTELLATION id [z923]
SI61 PositSlotEqualsEqForEntity.... ->

SITUATION: POSIT: SLOT id EQUALSEQUATION st FOR ENTITY id [z424]
SI62 PositSlotHasEq................ ->

SITUATION: POSIT: SLOT id HAS EQUATION id [z457]
SI63 PositSiteHasEntity.......... ->

SITUATION: POSIT: SITE id HAS ENTITY id [z460]
SI64 SelectEntityInstNmbr......... ->

SITUATION: SELECT ENTITY id INSTANCE NUMBER nm [z455]
ST1 ChangeSetBecomesList........ ->

SET: CHANGE: SET id BECOMES LIST id [z10]
ST2 BagSet......................... ->

SET: BAG SET id [z921]
ST3 DefIdentIsElm.................. ->

SET: DEFINE: IDENTIFIER id IS ELEMENT [z1]
ST4 DefIdentIsSet.................. ->

SET: DEFINE: IDENTIFIER id IS SET [z12]
ST5 DefMastersetContainsSet.... ->

SET: DEFINE: MASTERSET id CONTAINS SET id [z8]
ST6 EmptySet....................... ->

SET: EMPTY SET id [z3]
ST7 ExcludeElmFrmSet............. ->

SET: EXCLUDE ELEMENT id FROM SET id [z4]
ST8 InqMembershipOfSet.......... ->

SET: INQUIRE MEMBERSHIP OF SET id [z6]
ST9 InqPowersetOfSet.......... ->

SET: INQUIRE POWERSET OF SET id [z5]
ST10 IntersectionSetWithSetIntoSet.... ->

\textbf{SET: INTERSECTION SET} \textit{id WITH SET} \textit{id INTO SET} \textit{id} [z7]

ST11 NullSet.......................... ->

\textbf{SET: NULL SET} \textit{id} [z9]

ST12 PositiveElemBelongsToSet......... ->

\textbf{SET: POSIT: ELEMENT} \textit{id BELONGS TO SET} \textit{id} [z22]

ST13 PositiveSetContains{Set}......... ->

\textbf{SET: POSIT: SET} \textit{id CONTAINS} \{ \textit{set} \} [z11]

ST14 UnionSetWithSetIntoSet.......... ->

\textbf{SET: UNION SET} \textit{id WITH SET} \textit{id INTO SET} \textit{id} [z13]

SY1 DefineEnvironIsType............. ->

\textbf{SYSTEM: DEFINE: ENVIRONMENT} \textit{id IS TYPE} \textit{id} [z224]

SY2 DefineIdentIsBackplane.......... ->

\textbf{SYSTEM: DEFINE: IDENTIFIER} \textit{id IS BACKPLANE} [z239]

SY3 DefineIdentIsEnvironOfType...... ->

\textbf{SYSTEM: DEFINE: IDENTIFIER} \textit{id IS ENVIRONMENT OF TYPE} \textit{id} [z223]

SY4 DefineIdentIsEnviren............. ->

\textbf{SYSTEM: DEFINE: IDENTIFIER} \textit{id IS ENVIRONMENT} [z222]

SY5 DefineIdentIsGrid.............. ->

\textbf{SYSTEM: DEFINE: IDENTIFIER} \textit{id IS GRID} [z287]

SY6 DefineIdentIsHeir.............. ->

\textbf{SYSTEM: DEFINE: IDENTIFIER} \textit{id IS HIERARCHY} \textit{id} [z260]

SY7 DefineIdentIsInfraOfType......... ->

\textbf{SYSTEM: DEFINE: IDENTIFIER} \textit{id IS INFRASTRUCTURE OF TYPE} \textit{id} [z229]

SY8 DefineIdentIsInfra............... ->

\textbf{SYSTEM: DEFINE: IDENTIFIER} \textit{id IS INFRASTRUCTURE} [z228]

SY9 DefineIdentIsInteg............... ->

\textbf{SYSTEM: DEFINE: IDENTIFIER} \textit{id IS INTEGRATOR} [z238]

SY10 DefineIdentIsLevel............. ->
SYSTEM: DEFINE: IDENTIFIER id IS LEVEL  [z262]

SY11 DefIdentIsMetasystemOfType.... ->

SYSTEM: DEFINE: IDENTIFIER id IS METASYSTEM OF TYPE id [z220]

SY12 DefIdentIsMetasystem.......... ->

SYSTEM: DEFINE: IDENTIFIER id IS METASYSTEM [z219]

SY13 DefIdentIsPlat............... ->

SYSTEM: DEFINE: IDENTIFIER id IS PLATFORM [z237]

SY14 DefIdentIsSystem............. ->

SYSTEM: DEFINE: IDENTIFIER id IS SYSTEM OF TYPE id [z217]

SY15 DefIdentIsSystem............... ->

SYSTEM: DEFINE: IDENTIFIER id IS SYSTEM [z216]

SY16 DefInfraIsType.............. ->

SYSTEM: DEFINE: INFRASTRUCTURE id IS TYPE id [z230]

SY17 DefMetasystemIsType.......... ->

SYSTEM: DEFINE: METASYSTEM id IS TYPE id [z221]

SY18 DefSystemIsType............... ->

SYSTEM: DEFINE: SYSTEM id IS TYPE id [z218]

SY19 PositApplHasArch............... ->

SYSTEM: POSIT: APPLICATION id HAS ARCHITECTURE id [z245]

SY20 PositApplPlugsIntoBackplane... ->

SYSTEM: POSIT: APPLICATION id PLUGS INTO BACKPLANE id [z323]

SY21 PositApplEmbodiesDomain...... ->

SYSTEM: POSIT: APPLICATION id EMBODIES DOMAIN id [z272]

SY22 PositApplHasHeir............... ->

SYSTEM: POSIT: APPLICATION id HAS HIERARCHY id [z254]

SY23 PositApplHasImpl............... ->

SYSTEM: POSIT: APPLICATION id HAS IMPLEMENTATION id [z241]

SY24 PositApplUsesService.......... ->

SYSTEM: POSIT: APPLICATION id USES SERVICE id [z240]
SY25 PositBackplaneHasArch......... ->

SYSTEM: POSIT: BACKPLANE id HAS ARCHITECTURE id [z248]

SY26 PositBackplaneEmbodiesDomain.. ->

SYSTEM: POSIT: BACKPLANE id EMBODIES DOMAIN id [z276]

SY27 PositBackplaneHasHeir......... ->

SYSTEM: POSIT: BACKPLANE id HAS HIERARCHY id [z258]

SY28 PositBackplaneHasImpl......... ->

SYSTEM: POSIT: BACKPLANE id HAS IMPLEMENTATION id [z244]

SY29 PositEnvironEmbodiesDomain.... ->

SYSTEM: POSIT: ENVIRONMENT id EMBODIES DOMAIN id [z271]

SY30 PositEnvironHasExternal....... ->

SYSTEM: POSIT: ENVIRONMENT id HAS EXTERNAL id [z324]

SY31 PositEnvironHasHeir.......... ->

SYSTEM: POSIT: ENVIRONMENT id HAS HIERARCHY id [z253]

SY32 PositEnvironImpingesOnSystem.. ->

SYSTEM: POSIT: ENVIRONMENT id IMPINGES ON SYSTEM id [z325]

SY33 PositGridHasLayer............. ->

SYSTEM: POSIT: GRID id HAS LAYER id [z288]

SY34 PositGridHasPartit............ ->

SYSTEM: POSIT: GRID id HAS PARTITION id [z289]

SY35 PositGridHasStrata............ ->

SYSTEM: POSIT: GRID id HAS STRATA id [z290]

SY36 PositGridHasTier.............. ->

SYSTEM: POSIT: GRID id HAS TIER id [z291]

SY37 PositHeirContainsLevel....... ->

SYSTEM: POSIT: HIERARCHY id CONTAINS LEVEL id [z261]

SY38 PositImplEmbodiesDomain..... ->

SYSTEM: POSIT: IMPLEMENTATION id EMBODIES DOMAIN id [z274]

SY39 PositImplHasHeir............. ->
SYSTEM: POSIT: IMPLEMENTATION id HAS HIERARCHY id [z256]

SY40 PositImplRunsOnPlat........... ->

SYSTEM: POSIT: IMPLEMENTATION id RUNS ON PLATFORM id [z249]

SY41 PositInfraHasAppl............. ->

SYSTEM: POSIT: INFRASTRUCTURE id HAS APPLICATION id [z231]

SY42 PositInfraHasBackplane........ ->

SYSTEM: POSIT: INFRASTRUCTURE id HAS BACKPLANE id [z236]

SY43 PositInfraEmbodiesDomain...... ->

SYSTEM: POSIT: INFRASTRUCTURE id EMBODIES DOMAIN id [z268]

SY44 PositInfraHasHeir............. ->

SYSTEM: POSIT: INFRASTRUCTURE id HAS HIERARCHY id [z250]

SY45 PositInfraHasImpl............. ->

SYSTEM: POSIT: INFRASTRUCTURE id HAS IMPLEMENTATION id [z233]

SY46 PositInfraHasInteg............. ->

SYSTEM: POSIT: INFRASTRUCTURE id HAS INTEGRATOR id [z235]

SY47 PositInfraHasPlat............. ->

SYSTEM: POSIT: INFRASTRUCTURE id HAS PLATFORM id [z234]

SY48 PositInfraHasService.......... ->

SYSTEM: POSIT: INFRASTRUCTURE id HAS SERVICE id [z232]

SY49 PositIntegHasArch............. ->

SYSTEM: POSIT: INTEGRATOR id HAS ARCHITECTURE id [z247]

SY50 PositIntegHasImpl............. ->

SYSTEM: POSIT: INTEGRATOR id HAS IMPLEMENTATION id [z243]

SY51 PositIntegPlugsIntoBackplane.. ->

SYSTEM: POSIT: INTEGRATOR id PLUGS INTO BACKPLANE id [z339]

SY52 PositIntegEmbodiesDomain...... ->

SYSTEM: POSIT: INTEGRATOR id EMBODIES DOMAIN id [z277]

SY53 PositIntegHasHeir............. ->

SYSTEM: POSIT: INTEGRATOR id HAS HIERARCHY id [z259]
SY54 PositLevelRelatesToGrid....... ->

SYSTEM: POSIT: LEVEL id RELATES TO GRID id [z278]

SY55 PositLevelContigToLayerInGrid. ->

SYSTEM: POSIT: LEVEL id CONTIGUOUS TO LAYER id IN GRID id [z279]

SY56 PositLevelRelatesToLevelInHeir ->

SYSTEM: POSIT: LEVEL id RELATES TO LEVEL id IN HIERARCHY id [z267]

SY57 PositLevelContigToLevelInHeir. ->

SYSTEM: POSIT: LEVEL id CONTIGUOUS TO LEVEL id IN HIERARCHY id [z266]

SY58 PositLevelIsAdjLevel.......... ->

SYSTEM: POSIT: LEVEL id IS ADJACENT LEVEL id [z265]

SY59 PositLevelIsOn_TopLevel....... ->

SYSTEM: POSIT: LEVEL id IS ON_TOP LEVEL id [z263]

SY60 PositLevelIsOn_BottomLevel.... ->

SYSTEM: POSIT: LEVEL id IS ON_BOTTOM LEVEL id [z264]

SY61 PositLevelContigToPartitInGrid ->

SYSTEM: POSIT: LEVEL id CONTIGUOUS TO PARTITION id IN GRID id [z280]

SY62 PositLevelContigToStrataInGrid ->

SYSTEM: POSIT: LEVEL id CONTIGUOUS TO STRATA id IN GRID id [z281]

SY63 PositLevelContigToTierInGrid.. ->

SYSTEM: POSIT: LEVEL id CONTIGUOUS TO TIER id IN GRID id [z282]

SY64 PositMetasystemHasConfig...... ->

SYSTEM: POSIT: METASYSTEM id HAS CONFIGURATION id [z284]

SY65 PositMetasystemEmbodiesDomain. ->

SYSTEM: POSIT: METASYSTEM id EMBODIES DOMAIN id [z269]

SY66 PositMetasystemContainsEnviron ->

SYSTEM: POSIT: METASYSTEM id CONTAINS ENVIRONMENT id [z226]

SY67 PositMetasystemHasGrid........ ->

SYSTEM: POSIT: METASYSTEM id HAS GRID id [z286]

SY68 PositMetasystemHasHeir........ ->
SYSTEM: POSIT: METASYSTEM id HAS HIERARCHY id \[z251\]

SY69 PositMetasystemContainsSystem. ->

SYSTEM: POSIT: METASYSTEM id CONTAINS SYSTEM id \[z225\]

SY70 PositPlatEmbodiesDomain....... ->

SYSTEM: POSIT: PLATFORM id EMBODIES DOMAIN id \[z275\]

SY71 PositPlatHasHeir............. ->

SYSTEM: POSIT: PLATFORM id HAS HIERARCHY id \[z257\]

SY72 PositServiceHasArch.......... ->

SYSTEM: POSIT: SERVICE id HAS ARCHITECTURE id \[z246\]

SY73 PositServicePlugsIntoBackplane ->

SYSTEM: POSIT: SERVICE id PLUGS INTO BACKPLANE id \[z237\]

SY74 PositServiceEmbodiesDomain.... ->

SYSTEM: POSIT: SERVICE id EMBODIES DOMAIN id \[z273\]

SY75 PositServiceHasHeir......... ->

SYSTEM: POSIT: SERVICE id HAS HIERARCHY id \[z255\]

SY76 PositServiceHasImpl.......... ->

SYSTEM: POSIT: SERVICE id HAS IMPLEMENTATION id \[z242\]

SY77 PositServiceUsesInteg........ ->

SYSTEM: POSIT: SERVICE id USES INTEGRATOR id \[z338\]

SY78 PositSystemHasConfig......... ->

SYSTEM: POSIT: SYSTEM id HAS CONFIGURATION id \[z283\]

SY79 PositSystemEmbodiesDomain..... ->

SYSTEM: POSIT: SYSTEM id EMBODIES DOMAIN id \[z270\]

SY80 PositSystemHasGrid............ ->

SYSTEM: POSIT: SYSTEM id HAS GRID id \[z285\]

SY81 PositSystemHasHeir............ ->

SYSTEM: POSIT: SYSTEM id HAS HIERARCHY id \[z252\]

SY82 PositSystemHasInfra.......... ->

SYSTEM: POSIT: SYSTEM id HAS INFRASTRUCTURE id \[z227\]
SY83 PositSystemIsMetasystem...... ->

  SYSTEM: POSIT: SYSTEM id IS METASYSTEM id [z327]

SY84 PositSystemContainsSystem..... ->

  SYSTEM: POSIT: SYSTEM id CONTAINS SYSTEM id [z326]

TM1 AttachAtribOfEntityToSignal... ->

  TEMPORALITY: ATTACH ATTRIBUTE id OF ENTITY id TO SIGNAL id [z393]

TM2 DefIdentiferIsSignal......... ->

  TEMPORALITY: DEFINE: IDENTIFIER id IS SIGNAL [z391]

TM3 DefIdentlsBranch............. ->

  TEMPORALITY: DEFINE: IDENTIFIER id IS BRANCH [z370]

TM4 DefIdentlsBundle............. ->

  TEMPORALITY: DEFINE: IDENTIFIER id IS BUNDLE [z371]

TM5 DefIdentlsEvent............... ->

  TEMPORALITY: DEFINE: IDENTIFIER id IS EVENT [z383]

TM6 DefIdentlsInterval........... ->

  TEMPORALITY: DEFINE: IDENTIFIER id IS INTERVAL [z387]

TM7 DefIdentlsLcyc............... ->

  TEMPORALITY: DEFINE: IDENTIFIER id IS LIFECYCLE id [z461]

TM8 DefIdentlsSheaf............... ->

  TEMPORALITY: DEFINE: IDENTIFIER id IS SHEAF [z389]

TM9 DefIdentlsTemporality........ ->

  TEMPORALITY: DEFINE: IDENTIFIER id IS TEMPORALITY [z395]

TM10 DefIdentlsTimepoint......... ->

  TEMPORALITY: DEFINE: IDENTIFIER id IS TIMEPOINT [z914]

TM11 DefIdentlsTimespan.......... ->

  TEMPORALITY: DEFINE: IDENTIFIER id IS TIMESPAN [z915]

TM12 DefIntervalDurFromTptToTpt.... ->

  TEMPORALITY: DEFINE: INTERVAL id DURATION FROM TIMEPOINT tp TO TIMEPOINT id [z385]

TM13 DefIntervalDurFromTptForTspan. ->
TEMPORALITY: DEFINE: INTERVAL id DURATION FROM TIMEPOINT tp FOR TIMESPAN ts [z396]

TM14 DefIntervalEndsTpt............ ->

TEMPORALITY: DEFINE: INTERVAL id ENDS TIMEPOINT tp [z386]

TM15 DefIntervalEndsAtTpt........... ->

TEMPORALITY: DEFINE: INTERVAL id ENDS AT TIMEPOINT tp [z400]

TM16 DefIntervalEnclosesTpt......... ->

TEMPORALITY: DEFINE: INTERVAL id ENCLOSES TIMEPOINT tp [z401]

TM17 DefIntervalBeginsAtTpt......... ->

TEMPORALITY: DEFINE: INTERVAL id BEGINS AT TIMEPOINT tp [z399]

TM18 DefIntervalExcludesTpt......... ->

TEMPORALITY: DEFINE: INTERVAL id EXCLUDES TIMEPOINT tp [z402]

TM19 DefIntervalBeginsTpt............ ->

TEMPORALITY: DEFINE: INTERVAL id BEGINS TIMEPOINT tp [z388]

TM20 DefIntervalDurForTspanToTpt... ->

TEMPORALITY: DEFINE: INTERVAL id DURATION FOR TIMESPAN ts TO TIMEPOINT tp [z397]

TM21 DefIntervalDurForTspan......... ->

TEMPORALITY: DEFINE: INTERVAL id DURATION FOR TIMESPAN ts [z398]

TM22 IntervalEventFinishesEvent.... ->

TEMPORALITY: INTERVAL: EVENT id FINISHES EVENT id [z378]

TM23 IntervalEventBeforeEvent...... ->

TEMPORALITY: INTERVAL: EVENT id BEFORE EVENT id [z375]

TM24 IntervalEventOverlapsEvent.... ->

TEMPORALITY: INTERVAL: EVENT id OVERLAPS EVENT id [z380]

TM25 IntervalEventEqualsEvent...... ->

TEMPORALITY: INTERVAL: EVENT id EQUALSEVENT id [z377]

TM26 IntervalEventMeetsEvent....... ->

TEMPORALITY: INTERVAL: EVENT id MEETS EVENT id [z379]

TM27 IntervalEventDuringEvent....... ->

TEMPORALITY: INTERVAL: EVENT id DURING EVENT id [z376]
TM28 IntervalEventAfterEvent...... ->

TEMPORALITY: INTERVAL: EVENT id AFTER EVENT id [z382]

TM29 IntervalEventStartsEvent...... ->

TEMPORALITY: INTERVAL: EVENT id STARTS EVENT id [z381]

TM30 JoinBundleWithBundleAtEvent... ->

TEMPORALITY: JOIN BUNDLE id WITH BUNDLE id AT EVENT id [z372]

TM31 PositBranchContainsBundle...... ->

TEMPORALITY: POSIT: BRANCH id CONTAINS BUNDLE id [z918]

TM32 PositBundleHasSignal.......... ->

TEMPORALITY: POSIT: BUNDLE id HAS SIGNAL id [z373]

TM33 PositEventHasInterval........... ->

TEMPORALITY: POSIT: EVENT id HAS INTERVAL id [z384]

TM34 PositLcycHasAmata.............. ->

TEMPORALITY: POSIT: LIFECYCLE id HAS AUTOMATA id [z463]

TM35 PositLcycHasEvent.............. ->

TEMPORALITY: POSIT: LIFECYCLE id HAS EVENT id [z462]

TM36 PositLcycHasPetrinet.......... ->

TEMPORALITY: POSIT: LIFECYCLE id HAS PETRINET id [z465]

TM37 PositLcycHasSignalWithEvent... ->

TEMPORALITY: POSIT: LIFECYCLE id HAS SIGNAL id WITH EVENT id [z464]

TM38 PositRelFromSignalToSignal.... ->

TEMPORALITY: POSIT: RELATION id FROM SIGNAL id TO SIGNAL id [z392]

TM39 PositSheafHasBranch............ ->

TEMPORALITY: POSIT: SHEAF id HAS BRANCH id [z917]

TM40 PositSheafHasBundle............ ->

TEMPORALITY: POSIT: SHEAF id HAS BUNDLE id [z390]

TM41 PositSignalHasEvent............ ->

TEMPORALITY: POSIT: SIGNAL id HAS EVENT id [z394]

TM42 PositTemporalityHasSheaf...... ->
TEMPORALITY: POSIT: TEMPORALITY id HAS SHEAF id [z916]

TM43 SplitBundleWithBundleAtEvent.. ->

TEMPORALITY: SPLIT BUNDLE id WITH BUNDLE id AT EVENT id [z374]

TR1 CheckClockReference........... ->

TURING: CHECK CLOCK id REFERENCE [z479]

TR2 CheckReferenceOnEvent......... ->

TURING: CHECK REFERENCE id ON EVENT id [z475]

TR3 DefIdentIsAnetic............... ->

TURING: DEFINE: IDENTIFIER id IS AUTONETIC [z474]

TR4 DefIdentIsClock................

TURING: DEFINE: IDENTIFIER id IS CLOCK id [z477]

TR5 DefIdentIsEventity.............

TURING: DEFINE: IDENTIFIER id IS EVENTITY [z493]

TR6 DefIdentIsFlag................

TURING: DEFINE: IDENTIFIER id IS FLAG [z494]

TR7 DefIdentIsReference............

TURING: DEFINE: IDENTIFIER id IS REFERENCE [z497]

TR8 DefIdentIsSpot................

TURING: DEFINE: IDENTIFIER id IS SPOT [z498]

TR9 GlobalClockForReferenceGlobal.

TURING: GLOBAL CLOCK id FOR REFERENCE id [z476]

TR10 InitializeClockToTimepoint....

TURING: INITIALIZE CLOCK id TO TIMEPOINT tp [z478]

TR11 PositAneticHasAssembly........

TURING: POSIT: AUTONETIC id HAS ASSEMBLY id [z468]

TR12 PositAneticHasInputAttr......

TURING: POSIT: AUTONETIC id HAS INPUT ATTRIBUTE id [z471]

TR13 PositAneticHasOutputAttr....

TURING: POSIT: AUTONETIC id HAS OUTPUT ATTRIBUTE id [z472]
Integral Software Engineering Methodology: LANGUAGE

TR14 PositAneticHasReactionAttr.... ->
TURING: POSIT: AUTONENTIC id HAS REACTION ATTRIBUTE id [z473]

TR15 PositAneticHasActivationAttr.. ->
TURING: POSIT: AUTONENTIC id HAS ACTIVATION ATTRIBUTE id [z467]

TR16 PositAneticHasBehavior........ ->
TURING: POSIT: AUTONENTIC id HAS BEHAVIOR id [z469]

TR17 PositAneticHasFlag............ ->
TURING: POSIT: AUTONENTIC id HAS FLAG id [z470]

TR18 PositClockIsInReference....... ->
TURING: POSIT: CLOCK id IS IN REFERENCE id [z482]

TR19 PositEnvironmentHasAnetic..... ->
TURING: POSIT: ENVIRONMENT id HAS AUTONENTIC id [z514]

TR20 PositEnvironmentHasBundle..... ->
TURING: POSIT: ENVIRONMENT id HAS BUNDLE id [z515]

TR21 PositEnvironmentHasClock...... ->
TURING: POSIT: ENVIRONMENT id HAS CLOCK id [z516]

TR22 PositEnvironmentHasEventity... ->
TURING: POSIT: ENVIRONMENT id HAS EVENTITY id [z517]

TR23 PositEnvironmentHasFlag....... ->
TURING: POSIT: ENVIRONMENT id HAS FLAG id [z518]

TR24 PositEnvironmentHasSheaf...... ->
TURING: POSIT: ENVIRONMENT id HAS SHEAF id [z519]

TR25 PositEventityHasAutonentic.... ->
TURING: POSIT: EVENTITY id HAS AUTONENTIC id [z487]

TR26 PositEventityHasBundle........ ->
TURING: POSIT: EVENTITY id HAS BUNDLE id [z488]

TR27 PositEventityHasClock......... ->
TURING: POSIT: EVENTITY id HAS CLOCK id [z489]

TR28 PositEventityHasSheaf...... ->
TURING: POSIT: EVENTITY id HAS SHEAF id [Z919]

TR29 PositEventityIsEntityWithLcyc. ->

TURING: POSIT: EVENTITY id IS ENTITY id WITH LIFECYCLE id [Z490]

TR30 PositEventityHasFlag......... ->

TURING: POSIT: EVENTITY id HAS FLAG id [Z491]

TR31 PositEventityIsAtSpot......... ->

TURING: POSIT: EVENTITY id IS AT SPOT id [Z486]

TR32 PositFlagHasBan............... ->

TURING: POSIT: FLAG id HAS BANNER id [Z492]

TR33 PositFlagHasSvar.............. ->

TURING: POSIT: FLAG id HAS STATEVAR id [Z496]

TR34 PositMetasystemHasAnetic...... ->

TURING: POSIT: METASYSTEM id HAS AUTONETIC id [Z508]

TR35 PositMetasystemHasBundle...... ->

TURING: POSIT: METASYSTEM id HAS BUNDLE id [Z509]

TR36 PositMetasystemHasClock....... ->

TURING: POSIT: METASYSTEM id HAS CLOCK id [Z510]

TR37 PositMetasystemHasEventity.... ->

TURING: POSIT: METASYSTEM id HAS EVENTITY id [Z511]

TR38 PositMetasystemHasFlag.......... ->

TURING: POSIT: METASYSTEM id HAS FLAG id [Z512]

TR39 PositMetasystemHasSheaf....... ->

TURING: POSIT: METASYSTEM id HAS SHEAF id [Z513]

TR40 PositReferenceHasClock........ ->

TURING: POSIT: REFERENCE id HAS CLOCK id [Z481]

TR41 PositSpotIsIsect.............. ->

TURING: POSIT: SPOT id IS INTERSECTION id [Z500]

TR42 PositSpotIsSite............... ->

TURING: POSIT: SPOT id IS SITE id [Z499]
TR43 PositSystemHasAnetic........... ->

TURING: POSIT: SYSTEM id HAS AUTONETIC id [z502]

TR44 PositSystemHasBundle.......... ->

TURING: POSIT: SYSTEM id HAS BUNDLE id [z503]

TR45 PositSystemHasClock........... ->

TURING: POSIT: SYSTEM id HAS CLOCK id [z504]

TR46 PositSystemHasEventity........ ->

TURING: POSIT: SYSTEM id HAS EVENTITY id [z505]

TR47 PositSystemHasFlag............ ->

TURING: POSIT: SYSTEM id HAS FLAG id [z506]

TR48 PositSystemHasSheaf.......... ->

TURING: POSIT: SYSTEM id HAS SHEAF id [z507]

TR49 ReadFlag........................ ->

TURING: READ FLAG id [z495]

TR50 ReadClockIntoClock.......... ->

TURING: READ CLOCK id INTO CLOCK id [z485]

TR51 ReadClockReference.......... ->

TURING: READ CLOCK id REFERENCE [z480]

TR52 ResetClock.................... ->

TURING: RESET CLOCK id [z483]

TR53 SetClockToTimepoint.......... ->

TURING: SET CLOCK id TO TIMEPOINT tp [z484]

TR54 SynchronizeReferenceOnEvent... ->

TURING: SYNCHRONIZE REFERENCE id ON EVENT id [z501]

WR1 DefIdentIsChoice............... ->

WORLD: DEFINE: IDENTIFIER id IS CHOICE [z702]

WR2 DefIdentIsConstruct.......... ->

WORLD: DEFINE: IDENTIFIER id IS CONSTRUCT [z703]

WR3 DefIdentIsIterator............ ->
WORLD: DEFINE: IDENTIFIER id IS ITERATOR [z920]

WR4 DefIdentIsRole............... ->

WORLD: DEFINE: IDENTIFIER id IS ROLE [z704]

WR5 DefIdentIsScenario.......... ->

WORLD: DEFINE: IDENTIFIER id IS SCENARIO [z705]

WR6 DefIdentIsSeq............... ->

WORLD: DEFINE: IDENTIFIER id IS SEQUENCE [z706]

WR7 DefIdentIsWorld............. ->

WORLD: DEFINE: IDENTIFIER id IS WORLD [z707]

WR8 DefIdentIsWorldline........... ->

WORLD: DEFINE: IDENTIFIER id IS WORLDLINE [z708]

WR9 PositActionCauzEvent.......... ->

WORLD: POSIT: ACTION id CAUSES EVENT id [z709]

WR10 PositAgentHasRole............. ->

WORLD: POSIT: AGENT id HAS ROLE id [z710]

WR11 PositAgentHasWorldline....... ->

WORLD: POSIT: AGENT id HAS WORLDLINE id [z711]

WR12 PositChoiceHasSeq............. ->

WORLD: POSIT: CHOICE id HAS SEQUENCE id [z712]

WR13 PositConstructTriggersEvent... ->

WORLD: POSIT: CONSTRUCT id TRIGGERS EVENT id [z713]

WR14 PositConstructHasInstr........ ->

WORLD: POSIT: CONSTRUCT id HAS INSTRUCTION id [z714]

WR15 PositConstructHasSeq.......... ->

WORLD: POSIT: CONSTRUCT id HAS SEQUENCE id [z715]

WR16 PositConstructUsesWorker...... ->

WORLD: POSIT: CONSTRUCT id USES WORKER id [z716]

WR17 PositConstructIsChoice....... ->

WORLD: POSIT: CONSTRUCT id IS CHOICE [z717]
WORLD: POSIT: CONSTRUCT id IS ITERATOR [z718]

WORLD: POSIT: CONSTRUCT id IS INTERACTION [z719]

WORLD: POSIT: EVENT id CAUSES EVENT id [z720]

WORLD: POSIT: EVENT id CAUSED BY STATE id IN STATEVAR id [z721]

WORLD: POSIT: EVENT id CAUSED BY STATE id INACTIVE [z722]

WORLD: POSIT: EVENT id CAUSED BY STATE id ACTIVE [z723]

WORLD: POSIT: INSIGNIA id CAUSES EVENT id ON BANNER id [z724]

WORLD: POSIT: ITERATOR id HAS SEQUENCE id [z725]

WORLD: POSIT: PLACE id CAUSES EVENT id UNMARKED [z726]

WORLD: POSIT: PLACE id CAUSES EVENT id MARKED [z727]

WORLD: POSIT: ROLE id HAS SEQUENCE id [z728]

WORLD: POSIT: SCENARIO id HAS SEQUENCE id [z729]

WORLD: POSIT: SEQUENCE id ORDERED CONSTRUCT id [z730]

WORLD: POSIT: TRANSITION id CAUSES EVENT id [z731]
WORLD: POSIT: WORLD id CONTAINS WORLDLINE id [z732]

WR33 PositWorldlineHasSeq......... ->

WORLD: POSIT: WORLDLINE id HAS SEQUENCE id [z733]

LAST Z# USED =927
APPENDIX 6

GLOSSARY

for Integral Software Engineering Methodology

(ISEM) Language
In this glossary the words that appear as entities in the ISEM sub-languages are always capitalized. Words that form part of the conceptual core of the minimal methods have asterisk appended. Words that are part of the conceptual core but not used in ISEM are not capitalized.

**ACTION** -- The output of an AUTOMATA (State Machine).

**ACTIVATOR** -- Turns on and off a PROCESS when activated by AUTOMATA or PETRINET.

**ACTOR** -- An independent processing entity which may move from processor to processor. See Agha ACTORS.

**AGENT** -- An executing or processing entity with autonomy. Identified with a viewpoint on software design represented in ISEM by AGENT, ACTOR, & TASK which execute on PROCESSORs.

**AGGREGATION** -- A set of ARTIFACTs that describe a design which are grouped for the purposes of identification with AUTHOR, BUILD, DATE, and VERSION.

**Allocation** -- Called in ISEM a MAPPING of PROCESS to AGENT.

**ANYID** -- Any Identifier. Used in language to make it possible to connect a sentence to any noun.

**APPLICATION** -- A complete set of components that provide a specific utility. Distinguished from a SERVICE, BACKPLANE, and other elements of a complete system.

**ARC** -- A generic connector in a ARCHITECTURE which are combined with NODEs.

**ARCHITECTURE** -- The structural configuration of a design made up of NODEs and ARCs.

**Articulation** -- The declaration of a RELATION between ENTITYs and specification of their ATTRIBUTEs or TYPEs called a SITUATION in ISEM.

**ARTIFACT** -- A document or graphical representation that records a design.

**ASSEMBLY** -- A sub-component of an AUTONETIC that contains various PARTs as well as AUTOMATA or PETRINET describing internal reaction of the whole to the PARTs.

**Assignment** -- Placing a VALUE in a VARIABLE, FLAG, BANNER, STATEVAR or some other memory location with a name and a data type.

**ATTRIBUTE** -- A feature of an ENTITY which may be assigned a definite value.

**AUTHOR** -- The designer of a particular AGGREGATE of ARTIFACTs.

**AUTOMATA** -- A Finite State Machine [FSM] which has a set of STATEs which on receipt of EVENTs will invoke ACTIONs. The current STATE of an instance of an AUTOMATA is saved in the STATEVAR. The connection between STATEs, which are activated by an EVENT and elicits
an ACTION is called a TRANSITION. [z300]

AUTONETIC -- A sub-component of an EVENTITY that contains either an AUTOMATA or PETRINET. The word is a combination of automatic and cybernetic. It is the active control element within a design. [z474]

BACKPLANE -- A software communication mechanism which allows APPLICATIONs to cooperate and obtain SERVICES. [z474]

BAG -- A SET that can have more than one of the same kind of object.

BANNER -- The equivalent of a STATEVAR for a PETRINET PLACE which contains an INSIGNIA which is the equivalent of a STATE. Banners and Insignia allow a regular PETRINET to become a colored PETRINET. [z342]

BEHAVIOR -- A sub-component of an AUTONETIC which expresses its reaction to external stimuli or higher AUTONETICS in the hierarchy of design elements. May contain either an AUTOMATA or a PETRINET. [z411]

BRANCH -- A portion of a SHEAF composed of a BUNDLE of SIGNALs. This concept allows one to talk of a set of BUNDLES within a SHEAF. [z370]

BUILD -- A subset of the total SYSTEM with some core set of functionality which is designed or implemented before the whole system. The system is created in stages called builds. [z192]

Bubble* -- Generic name for any PROCESS, TRANSFORM or FUNCTION as they appear usually as circles in a Structural Decomposition.

BUNDLE* -- A subset of a SHEAF of SIGNALs. [z371]

CATEGORY -- A broad conceptual classification bin for sorting items by specific features. [z198]

CHARACTERISTIC -- A feature of a categorized element. [z201]

CHOICE -- A selection in which one of the possible alternatives must be picked. [z702]

CLASS -- An abstract datatype to which an entity belongs in an inheritance hierarchy. [z413]

CLOCK -- A timepiece which gives the time or counts down an interval of time. [z477]

CommChannel -- The real-time data link between TASKS called in ISEM a CONNECTOR.

CommMechanism -- The specific type of communication between TASKs: QUEUE, SEMAPHORE, FLAG, RENDEZVOUS, etc.

COMPONENT -- An element of an ARCHITECTURE. [z075]

CONFIGURATION -- A controlled version of an AGGREGATION of ARTIFACTs. [z081]

CONNECTOR -- A communications channel that connects PORTs of TASKs, or just simply tasks to each other. [z658]
CONSTRUCT -- Either a CHOICE, SEQUENCE, or INTEACTION which may have an associated 
INSTRUCTION or use a WORKER. [z703] [See the RADDLE language and the VERDI tool 
from MCC.]

CONTEXT -- The top level PROCESS bubble expressing the functional boundary of a SYSTEM. [z122]

ControlArc* -- Analogous to a DATAFLOW arc called in ISEM a CONTROLFLOW.

CONTROLFLOW -- A line of transmission of control information within a functional view of a system. 
[z624] [See Hatley/Pribhai methodology.]

CONTROLSPEC* -- The terminus of a CONTROLFLOW which may contain DECISIONTABLE, 
AUTOMATA (or PETRINET), or ACTIVATOR. [z625] [See Hatley/Pirbhai methodology]

COORDINATE -- The abstract location in a four (or less) dimensional matrix. It is a location of an 
INTERSECTION.

CurrentState*: The state now in the STATEVAR of the STATEMACHINE.

DataArc* -- Analogous to ControlArc called a DATAFLOW in ISEM.

DATAFLOW -- A channel through which data flows within a system. It may be decomposed into lower level 
data items. The atomic level data item is called a DATUM. [z626]

DATASTORE* -- A repository of data within a system. [z627]

DATA -- A fundamental viewpoint on software design represented in ISEM by SITUATION. The data itself 
is called DATUM.

DataDictionary* -- Ordered collection of all DataItems.

DATE -- The calendar date.

DataItem* -- A piece of data stored in a DATASTORE called in ISEM a DATUM.

DATUM* -- An atomic level of information within a system. Has been called an INFON. [z534]

DECISION* -- A rule in a colored PETRINET. Decides how to react to different color tokens in the input 
places of transitions. [z352]

DECISIONTABLE* -- Takes input control signals and reduces them logically to the logically smallest set of 
necessary control signals. May be the first element in a CONTROLSPEC which reduces the 
signals which are needed by an AUTOMATA or PETRINET. [z628] [See Hatley/Pirbhai 
methodology]

Declaration* -- A statement of fact normally by positing a RELATION between any two ENTITYs.

Decomposition* -- The hierarchy of PROCESS, TRANSFORM, or FUNCTIONS produced by Structured 
Decomposition in either the Hatley/Pirbhai or Tom DeMarco methodologies.

DEFINITION -- The definition of a GLOSSARY item (word). [z207]
DesignElement* -- The conceptual building block of a software design. In ISEM these are EVENTITYs or EVENTITYs.

DesignElementFlow* -- A view of systems that relates StateMachines at different levels in the hierarchy of the system. This is accomplished in ISEM by ASSEMBLYs and BEHAVIORs.

DiachronicSlice* -- A picture of the system as it changes through time.

DOCUMENT -- A set of artifacts published together with a specific format. [z168]

DOMAIN -- A range of variations for a particular APPLICATION. [z157]

DURATION* -- A length of time. See also INTERVAL.

ELEMENT -- An item within a SET or LIST. [z001]

ENTITY* -- An Object or Thing which has ATTRIBUTEs and can be instantiated. [z425]

ENTRY -- The first place in a PETRINET where the tokens begin at start-up.

ENVIRONMENT -- The context of a SYSTEM. [z222]

EQUATION* -- Produces a value in a SLOT for an ENTITY. [z427] Generically an equations appear as Expressions in programming languages.

EVENT* -- An occurrence or happening which has a discrete duration or point in time. [z383] Also a fundamental viewpoint on software design that considers time represented in ISEM by TEMPORALITY.

EVENTITY -- A combination of an EVENT with a LIFECYCLE. This concept allow us to have active objects within our design. [z493]

EXECUTIVE* -- The control loop within and EVENTITY. [z568]

EXIT -- The final position within a PETRINET which is not completely cyclical. [z356]

EXPRESSION -- A mathematical statement that may be evaluated to produce an answer.

EXTERNAL* -- Entities outside the system with which the SYSTEM must interface. [z629]

FACE -- A display of some of the ATTRIBUTEs of an ENTITY. [z429]

FLAG* -- A flag is a global variable. It may be attached to a BANNER or STATEVAR. [z494]

FUNCTION* -- A function is a subset of a TRANSFORM which is a subset of a PROCESS. It is the lowest level grouping of functionality. [z569] Also signifies a fundamental viewpoint on software design that are represented in ISEM by PROCESS.

FunctionArc* -- The mapping relation of a FUNCTION to a ProcessingElement.
GLOBALCLOCK -- A timepiece used to coordinate other CLOCKS. [z476]

GLOSSARY -- A list of standard or specialized terms and their DEFINITIONs.

GRID -- The GRID is a four dimensional matrix within which design elements or a ARCHITECTURE are placed, generally consisting of various types of NODEs and ARCs. [z287]

HEAD -- The head of a LIST. [z18, 34]

HIERARCHY -- A tree structure for the placement of components within an ENVIRONMENT, IMPLEMENTATION, INFRASTRUCTURE, INTEGRATOR, METASYSTEM, PLATFORM, SERVICE, or SYSTEM. [z260]

HITHER -- A way of expressing order in four dimensional space corresponding to FRONT, ABOVE, RIGHT in the other dimensions. See STRATA.

IDENTIFIER -- A label or symbol denoting an instance of a type.

IMPLEMENTATION -- The working model of a design of a APPLICATION, BACKPLANE, INFRASTRUCTURE, INTEGRATOR, or SERVICE. [z084]

INFOPACKET -- A grouping of DATUMs, or 'infons', which flows into, through or out of a SYSTEM. Usually contains MESSAGEs. [z538]

InformationFlow -- The picture of how information units, so called 'infons' or DATUM, flow within the SYSTEM.

INFRASTRUCTURE -- The Infrastructure contains the basic major components of the SYSTEM which are APPLICATION, SERVICE, INTEGRATOR, BACKPLANE, along with their IMPLEMENTATIONs running on a PLATFORM. [z228]

Input* -- The data handed to a FUNCTION as parameters.

InputPlace* -- The input PLACE for a PETRINET TRANSIT.

INSIGNIA -- The insignia is the value contained within a BANNER. It corresponds to the STATEVAR of the AUTOMATA but instead does the same function within the realm of the colored PETRINET. [z357]

INSTANCE -- An instantiation of an ENTITY which is an independent copy with it's own values for each of the ATTRIBUTES. [z431]

INSTRUCTION* -- A component in a virtual MACHINE which performs an OPERATION. It may be associated with an AUTOMATA ACTION, PETRINET TRANSIT, OR general WORKER. [z583]

INSTRUCTIONSET -- A set of INSTRUCTIONs which may or may not form a MACHINE or a part of a MACHINE. [z581]

INTEGRATOR -- A part of a SYSTEM that brings together common functions shared by APPLICATIONs and does them in one place for the sake of efficiency. An example of an integrator might be
an Event Manager, or Object Manager. [z238]

INTERACTION -- An interaction is a point of synchronization between multiple AGENTs, ACTORs or TASKs. [z238]

INTERFACE -- The point of contact between two ARCHITECTURAL components (NODEs) which is indirect rather than direct in order to preserve uniformity and standardization of connection between architectural components. [z086]

INTERSECTION -- Within a GRID the intersections are the places which are indexed by the coordinates WXYZ. ARCHITECTURAL NODEs occupy these SITEs or SPOTs. [z091]

INTERVAL* -- A segment of time within a Temporal Logic extending by a DURATION from an EVENT or between EVENTs. [z387]

IntervalConstraint* -- In Allen’s Interval Logic these are descriptions of relations between events such as before, after, during, starting, finishes, overlapping, meets, equals.

IRAX -- Taken from the Naval Operating System Services Working Group (OSSWG) reference model for real-time systems. It is an acronym which stands for Intermediate Resource Allocation Executive. It manages resources for a group of PROCESSORs. See LPOS and SRAX. [z683]

ISEM -- Integral Software Engineering Methodology which takes into account all four views on software designs (AGENT, PROCESS, EVENT, DATA) and all sixteen methods that act as bridges between these viewpoints or represent the viewpoint itself.

ITERATOR -- A loop CONSTRUCT which exercises a SEQUENCE, INTERACTION, or CHOICE a number of times.

Lacune* -- A gap between pulses in a SIGNAL.

LAYER -- One of several ways of dividing a SYSTEM ARCHITECTURE within a GRID. Different layers are ABOVE and BELOW or ADJACENT each other. SEE STRATA, TIER, and PARTITION. [z097]

LeftHandSide* -- The set of conditions for a DECISION rule in a PETRINET.

LEVEL -- A segment of a HIERARCHY. Levels may be contiguous to LAYERs, STRATAs, PARTITIONs, and TIERs within the GRID. Levels may be ON_TOP on ON_BOTTOM, or ADJACENT to other levels within the HIERARCHY. [z262]

LIFECYCLE* -- An AUTOMATA or PETRINET for an ENTITY. [z461]

LIST -- An ordered bag of elements with a HEAD and TAIL. [z028]

LOCATION -- A segment in an INFOPACKET where a DATUM resides.

Loop* -- A statement that causes other statements to be repeated. See ITERATOR.

LPOS -- Taken from the OSSWG Real-time system reference model. The acronym stands for the Local Processor Operating System. It is the executive resident in a processor which manages the
resources used by that processor. See IRAX and SRAX. [z682]

MACHINE -- A Virtual Layered Machine in the sense advocated by Neilson & Shumate in DESIGNING REAL-TIME SYSTEMS IN ADA. It is composed of INSTRUCTIONs and is composed of LAYERS of INSTRUCTIONSETs which form a HIERARCHY. Each layer may be associated with an AUTOMATA or PETRINET. The basic point is that a Virtual Machine solves a design or implementation problem at a certain level of abstraction. At its lowest levels the INSTRUCTIONs of the Virtual Machine may map into OPERATIONS of ENTITIES or some other FUNCTION which does not have persistent data. [z586]

Marker **-- The activation of a PLACE in a PETRINET. In a Colored Petri Net markers are differentiated and may be any symbol or string.

MASTERLIST -- A masterlist is a type of LIST.

MASTERSET -- A masterset is a type of SET.

MESSAGE* -- A request for action or data to an OBJECT. The message travels in an INFOPACKET and is delivered into a QUEUE usually associated with a TASK. [z591]

METASYSTEM -- A SYSTEM that operates on or controls another system. [z219]

Method* -- ‘meta - hodos’ means ‘a way after’. A method is a means or technique which if you follow it will allow you to get to the same place as someone else who blazed the trail to that place. There is no ‘logic of discovery’ or method which will assure discovery of the best design; all that can be offered is methods and heuristics to guide intuition.

MODE* -- A meta-state of an AUTOMATA. When a mode of a state machine changes then its response to stimuli may differ dramatically. [z308]

Monitor* -- A data protection scheme in an environment with multiple agents going after the same data using SEMAPHOREs.

NAMESPACE -- A virtual space in which labels for entities are guaranteed unique so that by having the name it is possible to find the entity. [z679]

NextState* -- Part of the StateVector that indicates which STATE would be next if a certain TRANSITION fired.

NODE -- A generic ARCHITECTURE component which are connected by ARCs to form network of design elements. [z105]

NowCurrentState* -- Part of the StateVector which indicates which state the AUTOMATA should now be in for a transition to be elicited.

OBJECT -- An ENTITY which holds persistent data in a DATASTORE and encapsulates that data and is associated with OPERATIONS that operate on that protected data. [z435]

OPERATION* -- A function that changes persistent data in a DATASTORE of an OBJECT. [z599]

Output* -- The data returned from a FUNCTION either as parameters or in some other way.
OutputPlace* -- The output PLACE for a TRANSIT in a PETRINET.

PART -- A segment of an ASSEMBLY which allows part/whole relations to be expressed. [z438]

PARTITION -- A way of dividing a SYSTEM using the GRID. Partitions are RIGHT, LEFT, and ADJACENT of each other. See STRATA, TIER, and LAYER. [z126]

PetriArc* -- The connection between a PLACE and a TRANSIT in a PETRINET.

PetriMatrix* -- The matrix of PLACEs by TRANSITs that represents the PETRINET network. Each cell in the matrix is a connection either to or from between a PLACE and a TRANSIT.

PETRINET* -- A way to represent control flow invented by Petri. It is a network of PLACEs and TRANSITs through which colored MARKERs move to express control flowing within a SYSTEM. It is the dual of an AUTOMATA.

PIPE -- A communications mechanism between AGENTs, ACTORs, or TASKs which forms the primary connecting element in Unix. [z662]

PLACE* -- A collection point for MARKERs within a PETRINET network. [z114]

PLATFORM* -- The collection of equipment necessary to execute an IMPLEMENTATION. [z237] Platforms may contain groups of PROCESSORs.

PORT -- The input or output sockets of a TASK. [z674]

POSITION -- A segment of a LIST in which a certain element is located.

POSTALSSERVICE -- A means of delivering INFOPACKETS containing MESSAGES between PROCESSORs. This is a SERVICE that interacts with the communications network and makes all deliveries of messages appear to be local. [z680]

POWERSET -- The set of all possible sets within a SET.

PRIORITY -- The status of a particular TASK, ACTOR or AGENT when resources are being distributed at execution time.

PROCEDURE -- A programming language construct that implements a WORKER.

PROCESS -- A functional unit within a system. The Process contains TRANSFORMs which in turn contain FUNCTIONs. The dataflow diagramming method bubbles are called processes here. The meaning assigned to process in UNIX and elsewhere to denote a combination of agency and function does not apply in this case. Process as used here denotes pure functionality at the highest level. [z600] Represents a fundamental viewpoint on design.

ProcessActivationTable* -- Activates and Deactivates PROCESS, TRANSFORM, or FUNCTION in the Hatley/Pirbhai methodology called an ACTIVATOR in ISEM.

ProcessingElement* -- A combination of a TASK and a PROCESSOR.

PROCESSOR* -- The part of the hardware PLATFORM that contains the Central Processing Unit (CPU).
PRODUCT -- A deliverable which is given to a customer which may be composed of an IMPLEMENTATION of an APPLICATION or some other parts of a SYSTEM and other AGGREGATES of ARTIFACTS.

Protocol* -- MESSAGES conform to protocols set up by OBJECTS which determine which messages they will accept and their format must be.

QUEUE *-- A communications mechanism that allows MESSAGES to accumulate and be worked on one at a time.

REFERENCE -- A CLOCK to which another clock compares its own time.

RELATION* -- A generic mapping from one design element to another. Relations have their own ATTRIBUTEs.

RelationArc* -- The connector that represents a RELATION.

RENDEZVOUS* - A two way two party INTERACTION between TASKs as appears in the Ada Programming Language (Trademarked by DOD).

REPLICA -- The representation of an ATTRIBUTE in an INSTANCE.

REPRESENTATION -- A depiction of an APPLICATION or SYSTEM design within a DOMAIN.

REQUIREMENT -- An expression of need by a Customer which constrains design.

RightHandSide* -- The consequences of a DECISION rule in a PETRINET.

ROLE -- The abstraction of a WORLDLINE for an AGENT composed of CONSTRUCTS which include INTERACTIONs, SEQUENCE, CHOICE, and ITERATORS.

Rule* -- A statement that has conditions which if satisfied will lead to consequences. See DECISION.

SCENARIO* -- The abstraction of a SEQUENCE of CONSTRUCTs (usually EVENTs) cutting across multiple WORLDLINEs.

Selector* -- Part of Structured Programming a Selector chooses between alternatives and is generically represented by an IF statement or CASE statement in programming languages. See also CHOICE.

SEMAPHORE* -- A special type of FLAG which protects DATASTOREs from arbitrary access by multiple ACTORs or TASKs.

SEQUENCE -- A series of CONSTRUCTs performed in a certain linear order.

SERVICE -- A provider of certain functionality shared by APPLICATIONs.

SET -- A grouping of ELEMENTs which only allows one of each kind.
SHEAF* -- A grouping of SIGNALs which may be subdivided into BUNDLES. [z053]

SIGNAL* -- A pulse or stream of information which expresses temporal ordering. Signals must appear in BUNDLES which ultimately are grouped into SHEAFs. Unless signals are grouped then lacunae in signals cannot be detected. Normally a REFERENCE CLOCK produces the fundamental pulse that coordinates the processing on a specific PLATFORM. Signals may appear in continuous streams of pulses, in cyclical oscillations of pulse, or in arbitrary pulses. It is through signals that actors coordinate their operations across spacetime. The paradoxes of Special Relativity may apply to these attempts at coordination if it takes too long for a pulse to reach a Reference clock from the GLOBAL CLOCK. [z391]. A signal is a set of pulses that occur ordered in time. [z391]

SignalArc* -- A SIGNAL expressed as a RELATION.

SINK* -- A terminal NODE in an ARCHITECTURE network composed of NODEs and ARCs.

SITE -- A position for an ENTITY that may or may not be connected to an INTERSECTION within the GRID. See SPOT. [z459]

SITUATION* -- The network of ENTITYs and their RELATIONS. The SITUATION contains CONSTELLATIONS which contain BAGs which contain ENTITYs.

SLOT -- An ATTRIBUTE that has an EQUATION that produces a VALUE whenever any of it's input parameters change or on a SIGNAL. [z456]

SMInput* -- The VARIABLE or ATTRIBUTE in which the EVENT being handed an AUTOMATA should be placed in order to elicit a response.

SMOutput* -- The VARIABLE or ATTRIBUTE in which the response of an AUTOMATA (i.e. an ACTION) appears.

SOURCE* -- The terminal NODE in an ARCHITECTURE network composed of NODEs and ARCs.

SPECIFICATION -- A set of REQUIREMENTs for a design ARCHITECTURE. [z170]

SPOT -- Another design element parallel to the SITE of an ENTITY. Spots and sites are places to put entities until they are entered into the GRID. [z498]

SRAX -- This concept is taken from OSSWG Real-time system reference model. The acronym stands for System Resource Allocation Executive. It coordinates the usage of resources by all the ACTORS, AGENTS, or TASKS of the system. See IRAX and LPOS. [z681]

STATE* -- A discrete moment in the operation of an AUTOMATA. An Automata is a Finite State Machine. Given an EVENT it looks at its internal state which exists in its STATEVAR and determines which ACTION was elicited by the input. It then resets its own internal state if necessary. [z322]

StateMachine* -- A Finite State Machine or Automata called in ISEM an AUTOMATA. Given an event and the currentState it uses the StateVector(s) to yield an Action and a NextState.

STATVAR -- The holder of the current STATE of an AUTOMATA. [z311]
StateVector* -- A set of arrays that hold the Event, NowCurrentState, NextState, and Action.

STRATA -- A division of a SYSTEM within the GRID. Strata may be HITHER, THITHER, or ADJACENT each other as they extend into four dimensional space. See PARTITION, LAYER, and TIER. [z146]

SUBNET -- A portion of a PETRINET.

SynchronicSlice* -- A snapshot of the SYSTEM at a moment in time.

SYSTEM* -- System is defined by KLIR in his ARCHITECTURE OF SYSTEMS PROBLEM SOLVING as occurring on several epistemological levels. Here system is used in each of those senses. It is the Object of investigation or design activities, it is the Source of data, it is the set of instrumented Data channels, it is the set of Generators that simulate the information flowing within data channels, it is the various levels of Meta-systems and Structural systems that explain different patterns of the data or temporal relations within the data stream. [z216]

SystemMode* -- The MODE of a SYSTEM.

TAIL -- The end of a LIST.

TASK* -- Within a PROCESSOR the computing resources, and especially processing clock cycles, are divided up by the LPOS so that each sub-AGENT gets the correct share according to their PRIORITY. Tasks are virtual agents since they do not have a whole processor under their command only appear as if they were autonomous within the time meted out to them by the executive. [z610] Also signifies a fundamental viewpoint on software design that in ISEM is represented by AGENT.

TAXON -- A bin which organizes the CHARACTERISTICS of ENTITIES being classified in a TAXONOMY according to their ATTRIBUTES. [z197]

TAXONOMY -- A classification system composed of CATEGORIEs and TAXONs for ENTITIES based on their ATTRIBUTES. [z166]

TEMPORALITY* -- A set of timing constraints described in terms of Temporal Logic based on the work of Allen. See SHEAF, BUNDLE, SIGNAL, INTERVAL, and EVENT. [z395]

TEXT -- The written form of a REQUIREMENT or ARTIFACT.

THITHER -- A way of expressing order in four dimensional space corresponding to BACK, BELOW, LEFT in the other dimensions. See STRATA

TIER -- A division of a SYSTEM within the GRID. Strata may be FRONT, BEHIND, or ADJACENT each other. See PARTITION, LAYER, and STRATA. [z146]

TIMEPOINT -- A specific point in time at which an EVENT may occur such as a CLOCK tick.

TIMESPAN -- A duration in time extending from at least one TIMEPOINT. See INTERVAL.

TRANSFORM -- A PROCESS contains a transform that in turn contains a FUNCTION. It is a set of transforming Functions which embody the higher level Process. [z616] Also signifies a
fundamental viewpoint on software design which in ISEM is represented by PROCESS.

TRANSIT* -- A NODE in a PETRINET that signifies work done. The transit is connected in some cases to a WORKER or may be seen to map to a PROCESS, TRANSFORM, or FUNCTION. When all the input PLACEs are filled with correctly colored markers then a DECISION calls the transit to fire placing output markers in the necessary places. [z369]

TRANSITION -- A link between STATES in an AUTOMATA associated with an EVENT and ACTION as well as a new STATE for the automata. [z314]

Trigger*-- An interrupt that activates a FUNCTION.

TYPE* -- A ‘datatype’ or classification label of an ENTITY, DATUM, OBJECT or other design element. All elements with that classification label have the same properties.

UNIT -- A subdivision of an ARCHITECTURE COMPONENT. [z155]

VALUE -- The contents of a STATEVAR, BANNER, FLAG, ATTRIBUTE, or VARIABLE.

VARIABLE* -- A memory location of a certain TYPE that can be assigned a VALUE.

VERSION -- The particular copy of CONFIGURATION which has been controlled in order to baseline changes as the Configuration Item changes. [z189]

VirtualMachine -- Called in ISEM a MACHINE.

WORKER -- A general purpose construct that allows different IMPLEMENTATION structures to be connected such as ACTION, INSTRUCTION, TRANSIT, OPERATION and be operationalized as PROCEDURES. [z320]

WORLD -- The causal nexus of an AGENT, ACTOR or TASK. All causal lines that can get to an agent within his own WORLDLINE or across communications mechanisms through a SCENARIO are in the WORLD of that agent. [z707]

WORLDLINE* -- The temporal unfolding of an AGENT, ACTOR, or TASK. [z708]
AC10 DefIdenfierIsLpos...........  
[688] AC22 PositActorInhabitsSrax.......  
AC11 DefIdenfierIsSrax............  
[689] AC45 PositTaskInhabitsLpos......  
AC19 DefIdentIsPostal.............  
[690] AC43 PositSraxHasNamespace......  
AC12 DefIdenfiferIsNamespace.....  
[691] AC35 PositIraxHasNamespace......  
AC20 DefIdenfierIsRendezvous.....  
[692] AC39 PositLposHasNamespace......  
AC21 DefIdenfierIsSemaphore.....  
[693] AC44 PositSraxHasPostal.........  
AC23 DefIdenfierIsMailbox.......  
[694] AC36 PositIraxHasPostal.........  
AC24 DefIdenfierIsActor.........  
[695] AC40 PositLposHasPostal.........  
AC25 DefIdenfierIsCnectr...........  
[696] IN28 PositInfopInCncetr.........  
AC26 DefIdenfierIsInteraction....  
[697] AC4 CommCnctrFromPort.........  
AC27 DefIdenfierIsProcConectr...  
[698] AC5 CommCnctrToPort..........  
AC28 DefIdenfierIsExtension.....  
[699] AC6 CommCnctrFromTask......  
AC29 DefIdenfierIsProcess.......  
[700] AC7 CommCnctrToTask.........  
AC30 DefIdenfierIsDstore.........  
[701] AC37 PositIraxisWithinSrax....  
AC31 DefIdenfierIsDflow.........  
[702] WR1 DefIdentIsChoice..........  
AC32 DefIdenfierIsControlspec...  
[703] WR2 DefIdentIsConstruct.......  
AC33 DefIdenfierIsDtab...........  
[704] WR4 DefIdentIsRole............  
AC35 DefIdenfierIsRendezvous...  
[705] WR5 DefIdentIsScenario.......  
AC36 DefIdenfierIsLpos..........  
[706] WR6 DefIdentIsSeq.............  
AC37 DefIdenfierIsPort..........  
[707] WR7 DefIdentIsWorld.........  
AC38 DefIdenfierIsSrax..........  
[708] WR8 DefIdentIsWorldline......  
AC39 DefIdenfierIsSemaphore....  
[709] WR9 PositActionCauzEvent......  
AC40 DefIdenfierIsDtab.........  
[710] WR10 PositAgentHasRole........  
AC41 DefIdenfierIsProcess......  
[711] WR11 PositAgentHasWorldline...  
AC42 DefIdenfierIsControlspec..  
[712] WR12 PositChoiceHasSeq.........  
AC43 DefIdenfierIsData.........  
[713] WR13 PositConstructTriggersEvent..  
AC44 DefIdenfierIsInteraction...  
[714] WR14 PositConstructHasInstr....  
AC45 DefIdenfierIsProcess......  
[715] WR15 PositConstructHasSeq......  
AC46 DefIdenfierIsProcess......  
[716] WR16 PositConstructUsesWorker...  
AC47 DefIdenfierIsDflow........  
[717] WR17 PositConstructIsChoice......  
AC48 DefIdenfierIsDtab..........  
[718] WR18 PositConstructIsIterator...  
AC49 DefIdenfierIsProcess......  
[719] WR19 PositConstructIsInteraction...  
AC50 DefIdenfierIsProcess......  
[720] WR20 PositEventCauzEvent.......  
AC51 DefIdenfierIsProcess......  
[721] WR21 PositEventCauzedByStateInSvar.  
AC52 DefIdenfierIsProcess......  
[722] WR22 PositEventCauzedByStateInAct..  
AC53 DefIdenfierIsProcess......  
[723] WR23 PositEventCauzedByStateActive.  
AC54 DefIdenfierIsProcess......  
[724] WR24 PositInsigniaCauzEventOnBan..  
AC55 DefIdenfierIsProcess......  
[725] WR25 PositIteratorHasSeq.......  
AC56 DefIdenfierIsProcess......  
[726] WR26 PositPlaceCauzEventUnmark..  
AC57 DefIdenfierIsProcess......  
[727] WR27 PositPlaceCauzEventMarked....  
AC58 DefIdenfierIsProcess......  
[728] WR28 PositRoleHasSeq.........  
AC59 DefIdenfierIsProcess......  
[729] WR29 PositScenarioHasSeq.......  
AC60 DefIdenfierIsProcess......  
[730] WR30 PositSeqOrderedConstructid...  
AC61 DefIdenfierIsProcess......  
[731] WR31 PositTransitionCauzEvent....  
AC62 DefIdenfierIsProcess......  
[732] WR32 PositWorldContainsWorldline...  
AC63 DefIdenfierIsProcess......  
[733] IN42 WriteAttrInObjToInfop....  
AC64 DefIdenfierIsProcess......  
[734] IN39 WriteAttrInEntityToInfop...  
AC65 DefIdenfierIsProcess......  
[735] IN41 WriteAttrToInfop............  
AC66 DefIdenfierIsProcess......  
[736] IN40 WriteAttrToInfop ..........  
AC67 DefIdenfierIsProcess......  
[737] IN40 WriteAttrToInfopInCncetr...  
AC68 DefIdenfierIsProcess......  
[738] IN31 ReadAttrFromInfopInCncetr...  
AC69 DefIdenfierIsCflow.........  
[739] IN26 PositDumInInfop..........  
AC70 DefIdenfierIsDflow........  
[740] IN30 PositInfopHasSizeExp......  
AC71 DefIdenfierIsDflow........  
[741] IN5 DefIdentIsInfopOfType......  
AC72 DefIdenfierIsDflow........  
[742] IN3 DefIdentIsDataOfType....  
AC73 DefIdenfierIsDflow.......  
[743] IN6 DefInfopacketsOfType.......  
AC74 DefIdenfierIsDflow.......  
[744] IN1 DefDatumIsType........  
AC75 DefIdenfierIsDflow.......  
[745] IN43 WriteAttrToInfopAtInterval...  
AC76 DefIdenfierIsDflow......  
[746] IN32 ReadAttrFromInfopAtInterval...  
AC77 DefIdenfierIsDflow......  
[747] z900 AC55 WriteObjectAttribute.

[z904] MC62 PositProcessorHasTask
[z905] MC34 ExecutiveHasAmata
[z906] MC54 PositObjectHasMessage
[z907] M72 PositWorkerImplFunction
[z907] MC64 PositQueueContainsMessage
[z908] MC51 PositMessageContainsOp
[z909] MC71 PositTransformImplFunction
[z910] MC52 PositMessageInvokesInstr
[z911] MC45 PostiInstrSendsMessage
[z912] PT10 DefIdentIsEntry
[z913] PT6 CnectTransitFromPlaceToEntry
[z914] TM10 DefIdentIsTimepoint
[z915] TM11 DefIdentIsTimespan
[z916] TM42 PositTemporalityHasSheaf
[z917] TM39 PositSheafHasBranch
[z918] TM31 PositBranchContainsBundle
[z919] TR28 PositEventityHasSheaf
[z920] WR3 DefIdentIsIterator
[z921] ST2 BagSet
[z922] SI20 DefIdentIsSituation
[z923] SI60 PositSituationHasConstellation
[z924] SI9 DefIdentIsConstellation
[z925] SI28 PositConstellationHasBag
[z926] SI7 DefIdentIsBag
[z927] SI25 PositBagHasEntity
ACKNOWLEDGEMENTS:

I would like to thank Larry Daubert for his insightful criticisms during many hours of conversation over the last four years concerning software engineering and our application of it to specific projects. Also, I thank Leonard Woo and Brian McNeese for their close reading and helpful comments concerning format and comprehensibility of this document. Don Baurschmidt, who is manager of Autonetics Marine Systems Software Engineering Department, deserves special credit for fostering the exploration of innovative technology within our department in general, and on specific projects. Finally, I would like to thank my mentor, Ian Dallas, who encouraged me to finish my studies in England and who taught me what was worth thinking about. Little did I know at that time that I would be led from the study of the philosophical and theoretical foundations of science and technology to the practice of software engineering. That this strange blend of philosophical study and practical work as a technologist might lead to a synthesis of the sort presented here, is truly unexpected. I thank God for the insights that have been granted to me along the way.

I would also like to thank Stuart LeVine and Alasdair Mullarney for their comments on the second part of this series of essays.

I would like to thank Bob Cummings for many useful discussions on these and other related topics.

FOOTNOTES:

Footnotes appear on the same page as cited, yet numbered sequentially from the beginning of the paper.

Footnotes refer to the Bibliography by the sequential number of the bibliographic item in the form "See Bib#xxx".

Incomplete footnotes have '???' question marks in them as do incomplete bibliographical items.

BIBLIOGRAPHY:

Abt, T.; Progress Without Loss of Soul; (Chiron Pub.; Wilmette IL, 1989)


Adorno, T.W.; Negative Dialectics; (Routledge, Kegan and Paul; London, 1973)


Agha, G.; "Semantic Considerations in the Actor Paradigm"; Lecture Notes in Computer Science #197; Seminar on Concurrency Carnegie-Mellon University Pittsburgh PA July 9-11, 1984; Springer-Verlag 1985

Akin, O.; "Exploration of the Design Process"; Design Methods and Theories; Volume 13, Number 3/4, pp 115-119

Alexander, C.; A Timeless Way of Building; (???)


Ampt, G.J.; "On the Integration of Software Design Information"; Information and Software Technology; Volume 29, Number 6, July/August 1987

Andrews, G.R. and Schneider, F.B.; "Concepts and Notations of Concurrent Programming"; Computing Surveys; Volume 15; Number 1, March 1983; pp3-43


Ashmore, M.; Reflexive Thesis; (Chicago U.P.; Chicago, 1989)


Bainbridge, E.S.;"Addressed Machines and Duality"; Lecture Notes in Computer Science; 25(1975)93-98; Category Theory Applied to Computation and Control


Barker, R.; CASE*METHOD; (Addison Wesley; 1990)

Basalla, G.; The Evolution of Technology; Cambridge U.P.; UK, 1988)

Bateson, G.; Steps Toward and Ecology of the Mind; (Paladin; London, 1973)

Batille, G.; The Accursed Share; (???)

Beam; Command and Control Systems; (???)

Berger, P; Social Construction of Reality; (???)

Berman, M.; Coming To Our Senses; (Simon and Schuster; NY, 1989)

Berry, W.; The Unsettling of America; (Sierra Club; San Fran, 1977)


Bloom, A. The Closing of the American Mind; (Simon and Schuster; NY,1987)

Blum, Alan; THEORIZING; (Gower Pub. Co.: 1974)


Bohm, D.; Wholeness and the Implicate Order; (Routledge, Kegan and Paul; London, 1980)


Brennan, A.; Thinking About Nature; (U. Of Georgia Press; Athens, 1988)

Briot, J.P. and Yonezaw a, A.; "Inheritence and Synchronization in Concurrent OOP"; Lectures in Computing; 276(1987)???,???,???

Broadbent, G.; "Design and Theory Building"; Design Methods and Theory; Volume 13, Number 3/4, pp103-107

Brooks, F.; "No Silver Bullet"; ???
Brown, G. Spencer-; Laws of Form; (Allen and Unwin; London, 1969)


Buhr, R.J.A.; SYSTEM DESIGN WITH ADA; (Prentice-Hall: Englewood Cliffs NJ; 1984; 83-13673)


Camus, A. The Myth of Sysiphus; (Hamilton; London, 1955)

Cannetti, E.; Crowds and Power; (Penguine; London, 1965)

Carroll, J.M., Thomas, J.C., and Malhotra, A.; "Clinical-Experimental Analysis of Design Problem Solving"; Design Studies; Volume 1, Number 2, Oct 1979


CASE OUTLOOK; "Case Tools for Real-Time Analysis and Design"; CASE OUTLOOK Newsletter, Volume 2, Number 1, January 1988; Case Consulting Group, 224 SW First Avenue, Portland, OR 97204

Castenada, C.; Teaching of Don Juan; (Ballentine; NY, 1968)

Chandy, K.M. and Misra, J.; Parallel Program Design; (Addison Wesley; NY, 1988)


Chen, Giming; "A Rule Based Object/Task Modeling Approach"; SIGMOD REC; Volume 15, Number 2, 6/86, pp281-292; ACM 0-89791-191-1/86/0500/0281S00.75

Chen, Giming; "An Object/Task Modeling Approach Based on Domain Knowledge and Control Knowledge Representation"; Fall Joint Computer Conference; CH2345-7/86/0000/0200S01.00; 1986 IEEE

Cherniak, C.; Minimal Rationality; (MIT Press; Cambridge MA, 1986, 86-7497)


Cohen, B.; The Specification of Complex Systems; (Addison-Wesley; Menlo Park CA, 1986, 85-30702)

Collard, A. and Contrucci, J.; Rape of the Wild; (Indiana U.P.; Bloomington, 1989)


Connelley, D.M.; Traditional Accupuncture; (Center for Traditional Accupuncture; Columbia Maryland, 1975)


Copi, I.M.; The Theory of Logical Types; (Routledge Kegan Paul; London, 1971)

Corlett, ?;Community Without Unity; (???)

Coulter, H.L.; Homeopathic Science and Modern Medicine; (North Atlantic Books; Berkeley, 1980)

Coulter, H.L.; Divided Legacy; 3 Volumes; (North Atlantic Books; Berkeley, 1973)


Creemer, David, Garrison, N., Marks, P.; The Germ 3.1 Primer; MCC Technical Report STP-086-90(Q); Feb 1990

Cumont, F. The Mysteries of Mithra; (Dover Pub.; NY, 1956)


DeMarco, T.; STRUCTURED ANALYSIS AND SYSTEM SPECIFICATION; (Prentice-Hall: Englewood Cliffs, NJ; 1979; 79-14655)


Derrida, J.; Speech and Phenomena; (Northwestern U.P.; 1973)

Derrida, J.; Of Gramatology; (Johns Hopkins U.P.; Baltimore, 1974)

Descartes, R.; Discourse on Method; in Descartes Selections translated by R. M. Eatorn; (Charles Scribner’s Sons; NY, 1955, 1927)

d’Espagnat, B.; In Search of Reality; (Springer Verlag; NY, 1983)


Devaney, R.L.; An Introduction to Chaotic Dynamical Systems; (Bengamin/Cummings Pub Co; Menlo Park CA, 1986, 85-15801)

Dickenson, Brian; Developing Quality Systems; (McGraw Hill Software Engineering Series, 1989, 88-12869); Note: A software process model.


Dubin, R.; THEORY BUILDING; (Free Press: NY; 1969)

Dubois, M., Scheurich, C. and Briggs, F.A.; "Synchronization, Coherence, and Event Ordering in Multiprocessors"; IEEE Computer; February 1988; pp9-21

Duerr, H.; Dreamtime; (Basil Blackwell; NY, 1985)

Eaton, S.B.; The Paleolithic Perscription; (Harper and Row; NY, 1988)

Eco, U. and Sebeok, T.A.; The Sign of Three; (Indiana U.P.; Bloomington, 1983)


Elbert, T.F.; Embedded Programming in Ada; (Van Nostrand Reinhold CO; NY, 1986, 85-29464)

Elliot, T.S.; Four Quartets; (Harcourt Brace World; NY, 1943)

Elluel, J.; Technological Society; (Vintage Books; NY, 1974)


Fandozi, P.R.; Nihilism and Technology; A Heideggerian Investigation; (U.P. of America; Wash. DC, 1982, 82-17337)


Feyerabend, P.; AGAINST METHOD; (New Library Books - Atlantic Highlands Humanities Press: London;


Fleishman, E.A. and Quaintance, M.K.; Taxonomies of Human Performance; The Description of Human Tasks; (Academic Press; NY, 1984, 84-70594)

Foster, ?; Game Players of Zan; (DAW; NY, 1977)

Foucault, M.; The Order of Things; (Travistock; London, 1970)

Foucault, M.; The Archeology of Knowledge; (Travistock; London, 1972)


Freeman, Peter; Software Perspectives: The System is the Message; (Addison-Wesley Pub Co.; Menlo Park, CA, 1987, 86-32135)

Fuchs, W.W.; Phenomenology and the Metaphysics of Presence; (Nijhoff; The Hague, 1976)

Fuller, B.; Synergetics; Two Volumes; (Macmillan; NY, 1975 and 1979)

Gadamer, H. J.; Truth and Method; (Sheed and Ward; London, 1975)


Gasche, R.; The Tain of the Mirror; (Harvard U.P.; Boston, 1986, 86-4673)


Girardot, N.J.; Myth and Meaning in Early Taoism; (U. Calif. Press; Berkeley, 1983)

Gleick, J.; CHAOS; Making a New Science; (Viking; NY, 1987, 87-40025)

Gomaa, H.; "Software Development of Real-Time Systems"; Communications of the ACM; July 1986; Volume 29; Number 7; pp657-668; ACM0001-0782/86/0700-0657

Gomaa, H.; "Structuring Criteria for Real-time System Design"; Proceedings 11th International Conference on Software Engineering; Pittsburgh, PA; May 1989


Graham, R.L., Kunth, D.E., Patashnik, I.O.; Concrete Mathematics; (Addison Wesley;?,1989)

Grathoff, R.H.; The Structure of Social Inconsistencies; (Nijhoff; The Hague, 1970)

Grossinger, R.; Planet Medicine; (North Atlantic Books; Berkeley, 1985)

Groth, M.; Preparational Thinking; (Routledge, Kegan, Paul, London, 1985, 84-24893)

Hahn, R.; Kant’s Newtonian Revolution in Philosophy; (Southern Illinois U.P.; Carbondale, 1988, 87-17618)


Hatley, D.J. and Pirbhai, I.A.; STRATEGIES FOR REAL-TIME SYSTEMS DEVELOPMENT; (Dorset House Pub. Co.; 1986)
Wild Software

Bibliography

Hawking, Stephen; A Brief History of Time; (Bantam; NY, 1988)

Heidegger, M.; On Time and Being; (Harper and Row; London, 19720

Heidegger, M.; Identity and Difference; (Harper and Row; London, 1969)

Heidegger, M.; Introduction to Metaphysics; (Yale U.P.; London, 1968)

Heidegger, M.; What is Called Thinking; (Harper and Row; London, 1968)

Heidegger, M.; Being and Time; (Harper and Row; London, 1962)

Heidegger, M.; The End of Philosophy; (Harper and Row; NY, 1973)

Heller, P.; Dialectics and Nihilism; (U. Mass. Press; 1966)

Henry, M.; The Essence of Manifestation; (Nijhoff; The Hague, 1973)


HighWater, J.; The Primal Mind; (New American Library; NY, 1981)

Hills, Fred; A Domain Analysis Process; SPC Technical Report DOMAIN_ANALYSIS_90001-N; Version01.00.02; Jan 1990; Herndon VA


Hofstadler, D.R.; Godel, Escher, Bach: Eternal Golden Braid; (Harvester Press; 1979)

Holbrook, B.; The Stone Monkey; (Wm Morrow; NY, 1981)

Horning, J.J. and Randell, B.; "Process Structuring"; Computing Surveys; 5-1(1973)?-???, March

Huang, J.P.; "Modeling of Software Partition for Distributed Real-Time Applications; IEEE Transactions on Software Engineering; Volume SE-11, Number 10, October 1985, pp1113 - 1126

Humphries, Watts; Managing the Software Process; (Addison-Weseley, 1989, 88-34453)

Idhe, D.; Technology And The Lifeworld; (Indiana U. P.; Bloomington, 1990)

Husserl, E.; Crisis of European Sciences and Transcendental Phenomenology; (Northwestern U.P.; Evanston, 1970)

Husserl, E.; Logical Investigations; (Routledge, Kegan, Paul; London, 1970)

Husserl, E.; Ideas; (George, Allen, Unwin; London, 1931)

Idhe, D.; Technology And The Lifeworld; (Indiana U. P.; Bloomington, 1990)

Jahn, R.G. and Dunne, B.T.; Margins of Reality; (Harcourt Brace Jovanovich Pub; NY, 1987)


Kalman, R.E.; "On the General Theory of Control Systems; Proc 1st IFAC Congress Moscow 1960; pp 481-492

Kaneko, K.; Collapse of Tori and genesis of Chaos in Dissipative Systems; (World Scientific Pub Co; Singapore, 1986)

Kant, I; Critique of Pure Reason; (Dutton; NY, 1934)
Kaufmann, A. and Gupta, M.M.; Introduction to Fuzzy Arithmetic; Theory and Applications; (Van Nostrand Reinhold; NY, 1985)


Kendal, A.; Fuzzy Mathematical Techniques with Applications; (Addison-Wesley; Reading MA, 1986, 85-3892)

Kerola, P. and Freeman, P.; "A Comparison of Life Cycle Models"; Fifth International Conference Proceedings on Software Engineering; CH1627-9/81/0000/0090S00.75 1981 IEEE

Khoshafian, S.N. and Copeland, G.P.; "Object Identity"; OOPSLA Proceedings; 1986

Klir, G.; Architecture of Systems Problem Solving; (Plenum Press; NY, 1985, 85-9283)

Klir, G.; "The Role of Methodological Paradigms in Systems Design"; SUNY at Binghampton; Working Paper

Klir, G.; "Reconstructability Analysis: Aims, Results and Open Problems"; Systems Research; 2-2(1985)141-163

Klir, G.; Fuzzy Sets, Uncertainty, and Information; (Prentice Hall; Englewood Cliffs, NJ, 1988)

Klir, G.; "Methodological Principles of Uncertainty in Information Systems Modelling"; SUNY at Binghampton; Working Paper


Klir, G.; "The Emergence of Two Dimensional Science in the Information Society"; Systems Research; 2-1(1985)131-140


Koehler, G.; The Handbook of Homeopathy; (Healing Arts Press, Rochester VT, 1989)


Kringe, J.; Science, Revolution, and Discontinuity; (Humanities; NY, 1980)

Kubler, G.; The Shape of Time; (Yale U.P.; London, 1962)


Lao Tzu; Tao Te Ching; translated in Chen, E. M. (Paragon House; NY, 1989); also translated in Henricks, R.G. (Ballantine Books; NY, 1989)

Lakatos, I; Science and Pseudo-science; (BBC/Open University; 1973)

Leonard, G.; The Silent Pulse; (Dutton; NY, 1986)

Levi-Strauss, C.; The Savage Mind; (Weidenfeld and Nicolsen; London, 1966)

Bibliography

Cliffs NJ, 1981, 80-21293)
Lieberman, H.; "Reversible Object-Oriented Interpreters; Lect. in Computing; 276(1887)??-???
Liskov, B and Guttag, J.; ABSTRACTION and SPECIFICATION IN PROGRAM DEVELOPMENT (MIT Press: 1986)
Liskov, B. and Zilles, S.; "An Approach to Abstraction"; Massachusetts Institute of Technology; PROJECT MAC; Computation Structures Group Memo 88; September 1973
Lowen, W.; Dichotomies of the Mind; (John Wiley; NY, 1982, 82-2611)
Lubars, M.D.; A General Design Representation; MCC Technical Report STP-066-89; Feb 1989
Lukacher, N.; Primal Scenes; (Cornell U.P.; Ithica, 1986)
Mallory, J.P.; In Search of the Indo-Greeks; (Thames and Hudson; London, 1989)
Manning, C.A.; Bioenergetic Medicine East and West; (North Atlantic Books; Berkeley, 1988)
Martin, J. and McClure, C.; Diagramming Techniques for Analysts and Programmers; (Prentice Hall; Englewood Cliffs, NJ, 1985)
May, R.; Love and Will; (Dell Pub.; NY, 1969)
McLaughlin, M. P.; "Simulated Annealing"; Dr. Dobb’s Journal; September 1989; pp 26-37
Mead, G.H.; Mind, Self, and Society; (???)
Mellor, Stephen J.; Cf. Project Technology Inc. and Shlaer, S.
Merleau-Ponty, M.; The Phenomenology of Perception; (1966)
Merleau-Ponty, M.; The Visible and The Invisible (Northwestern U.P.; Evanston, 1968)
Merchant, C.; Ecological Revolutions; (U.North Carolina Press; 1989)
Mittal, S. et al; "Virtual Copies: At the Boundary between Classes and Instances"; OOPSLA Proceedings; 1986
Monod, J.; Chance and Necessity; (Collins- Fontana; London, 1974)
Moore, Cristopher; "Unpredictability and Undecidability in Dynamical Systems"; Physical Review Letters; Vol.64 No.20; May14, 1990; pp 2354-2357
Narayanan, A. On Being A Machine; Vol 1 and 2; (Wiley: NY, 1988, 88-23020)
Naur, P. "Programming as Theory Building"; in Microprocessing and Microprogramming; Volume 15 (1985), pp 253-216 (North Holland Pub Co.); Keynote address EuroMicro’84 Copenhagen, Denmark
Negoita, C.V.; Fuzzy Systems; (Abacus Press; Turnbridge Wells, Kent, 1981)
Negoita, C.V.; Expert Systems and Fuzzy Systems; (Benjamin/Cummings; Menlo Park, CA, 1985)
Neilson, K.W. and Shumate, K.; "Designing Large Real-Time Systems With Ada"; Communications of the ACM; August 1987; Volume 30; Number 8; ACM 0001-0782/87/0800-0695; pp 695-715


Nietzsche, F.; Thus Spoke Zarathustra; (Penguin; London, 1969)

Nietzsche, F.; The Will To Power; (Random House; NY, 1967)

NGCR OSSWG Reference Model; Version 1.03 Dec 1989; Next Generation Computer Resources, Operating Systems Standards Working Group, This is a US Navy Committee.

Norman, R.J. and Nunamaker, J.F.; "CASE Productivity Perceptions of Software Engineering Professionals"; Comm of ACM; Sept 1989 No. 9; pp 1102-1108

O'Malley, J.B.; Sociology of Meaning; (Human Context; London, 1971)

Palmer, K. D.; The Structure of Theoretical Systems in Relation to Emergence; (Dissertation, London School of Economics, U.K., 1982)


Partridge, D.; Artificial Intelligence: Applications in the Future of Software Engineering; (John Wiley and Sons: NY; 1986; 86-7171)

Patton, H.J.; Kant's Metaphysic of Experience; (Macmillan; NY, 1936)

Pava, Clavin; Managing New Office Technology: An Organizational Strategy; Free Press 1983; Chapter 5 on Nonroutine Office Work

Peirce, C.S.; Collected Papers; (Harvard U.P.; Cambridge Mass, 1931 - 1966; 8 Volumes

Persig, R.M.; Zen and The Art of Motorcycle Maintainance; (Bantam; NY, 1974)

Peterson, I.; The Mathematical Tourist; (W.H. Freeman; NY, 1988, 87-33078)

Phillips, D.C.; Holistic Thought in Social Science; (Stanford U.P., 1973)


Popper, Karl; LOGIC OF SCIENTIFIC DISCOVERY (Harper and Row)

Porkert, M.; Chinese Medicine; (Wm Morrow; NY, 1988)

Potter, W.D. and Trueblood, R.P.; "Traditional, Semantic, and Hyper-Semantic Approaches to Data Modeling; IEEE COMPUTER; June 1988; pp 53-63


Pratt, Vaughan; "Modeling Concurrency with Partial Orders; International Journal of Parallel Programming; Vol.15 No.1; 1986; pp 33-71; Plenum Pub. Corp.;

PROJECT TECHNOLOGY INC.; "Overview of Object-Oriented Analysis" Presentation Materials; Professional Development Seminar of Orange County ACM 2/20/88; Given by Stephen J. Mellor; Condensed from workshop materials of Project Technology, Inc., 2560 Ninth Street, Suite 214, Berkeley, CA 94710


Rajlich, Vaclav and Silva, J.; "Making Functional Decomposition Methodologies Object Oriented"; Dept. Computer Science, Wayne State University, Detroit, MI

Reisig, W.; Petri Nets: An Introduction; (Springer-Verlag; NY, 1985, 84-26700); ISBN 0-387-13723-8

Riddle, W.E.; "Procedural Approaches To Software Design Modeling"; in SOFTWARE DEVELOPMENT TOOLS; (Springer-Verlag; NY, 1980)

Rifkin, J.; Declaration of a Heretic; (Routledge Kegan Paul; London, 1988)


Roman, Gruia-Catalin; "A Taxonomy of Current Issues in Requirements Engineering"; IEEE Computer; April 1985; 0018-9162/85/0400-0014/$01.00

Romanyszun, R.D.; Technology as Symptom and Dream; (Routledge; NY, 1989)

Rosen, S.; Nihilism; (Yale U.P.; London, 1969)

Rucker, R.; Mind Tools; (Houghton-Mifflin; Boston, 1987, 86-27790)

Russell, B. and Whitehead, A.N.; Principia Mathematica; (???)

Sallis, J.; Being and Logos; (Duquesne U.P.; Pittsburgh, 1975)

Salthe, Stanley; Evolving Hierarchical Systems; (Columbia Univ. Press; NY, 1985)

Sanden, Bo; "Entity-Life Modeling and Structured Analysis in Real-Time Software Design -- A Comparison"; Comm of ACM; Vol.32 No.12; Dec 1989; pp 1458-1466

Sartre, J.P.; Critique of Dialectical Reason; Volume 1; (New Left Books; London, 1969)

Sartre, J.P.; Being and Nothingness; (Methuen; London, 1969)

Scarry, E.; The Body In Pain; (Oxford U.P.; UK, 1985)

Schmookler, A.B.; Parable of the Tribes; (Houghton Mifflin Co.; Boston, 1984)

Schumacher, E.F.; Good Work; (Harper and Row; NY, 1979)

Schumacher, E.F.; Small Is Beautiful; (Harper and Row; NY, 1973)

Schutz, A.; Structures of the Lifeworld; Two Volumes; (???)

Schutz, A.; Reflections on the Problem of Relevance; (Yale U.P.; New Haven, 1970)

SEI SEPG; Second Annual SEPG Workshop "Leading the Way to Process Improvement"; Software Engineering Institute; Carnegie Mellon University; June 21-22, 1989; Pittsburgh, PA

Sheldrake, R.; The Presence of the Past; (Vintage Random House; NY, 1988)


Shlaer, S. and Mellor, S.J. OBJECT-ORIENTED SYSTEMS ANALYSIS: Modeling the World in Data; (Prentice-Hall: Englewood Cliffs, NJ; 1988; 87-34049);


Shlaer, S. and Mellor, S.J.; "Recursive Design"; Computer Language; March 1990; pp 53-65

SIGADA AREWG; A Framework for Describing Ada Runtime Environments; by Ada Runtime Environment Working Group of the Association of Computing Machinery; 87 Sept. 23; No publication data.

Simon, H.A.; "The Structure of Ill Structured Problems"; Artificial Intelligence; Volume 4 (1973), pp 181-201


Soloway, E. et al; "Designing Documentation to Compensate for Delocalized Plans"; Comm of ACM
Vol.31 No.11; Nov. 1988; pp1259-1258

Stankovic, J.A.; A Serious Problem for Next-Generation Systems; IEEE COMPUTER; Oct 1988; pp 10-19

Stavely, A.M. et al; "Inference from Models of Software Systems"; Journal of Systems and Software; Volume 5, 1985, pp185-191


Stewart, I.; Does God Play Dice; (???)

Stuart, I.; Does God Play Dice?: The Mathematics of Chaos; (Blackwell; Oxford UK, 1989)

Sussare, F. de; Course in General Linguistics; (Philosophical Library; NY, 1959)

Theunissen, M.; The Other; (MIT Press, Cambridge, MA, 1986)

Thomas, J.C. and Carroll, J.M.; "The Psychological Study of Design"; Design Studies; Volume 1, Number 1, 1979

Thorin, M.; SOFTWARE ENGINEERING; (Butterworths: London; 1985; 84-23031)

Thorsson, E.; Rune Lore; (S. Weiser; York Beach, Maine, 1988)


Unschuld, P.U.; Medicine In China; (U. Calif. Press.; Berkeley, 1985)

Vail, M.; Heigegger and Ontological Difference; (Penn State U.P.; London, 1972)

Verhaar, J.W.M.; The Verb 'BE' and its Synonyms; Philosophical and Grammatical Studies; D. Reidel Pub. Co.; Dordrecht-Holland, 1967 and 1968; Four Volumes

Waiz, Diane; A Longitudinal Study of Group Design of Computer Systems; MCC Technical Report STP-110-89; March 1989; Ph.D. Dissertation University of Texas at Austin

Ward, P.T. and Mellor, S.J.; STRUCTURED DEVELOPMENT FOR REAL-TIME SYSTEMS; Three volumes; (Yourdon Press(Prentice HALL): Englewood Cliffs, NJ; 1985; 85-050815)


Wasserman, Anthony I. , Pircher, P. A., Muller, Robert J.; "The Object-Oriented Structured Design Notation for Software Design Representation"; IEEE COMPUTER; March 1990; pp50-63

Watanabe, Satoshi; "Creative Learning and Propensity Automation"; IEEE Transactions on Systems, Man, and Cybernetics; Vol. SMC-5, No. 6, November 1975, pp 603-609

Webster, D.E.; "Mapping the Design Information Representation Terrain"; IEEE COMPUTER; Dec 1988; pp 8-23

Wegner, P.; "Object-Oriented Concept Hierarchies"; Brown University; Working Paper presented at CASE88

Welke, R.J.; "Meta Systems on Meta Models; CASE Outlook 89 No. 4; pp 35-44

Wells, G. C.; "The application of real-time design techniques to simulation"; Software Engineering Journal; November 1989

Whitehead, A.N.; Process and Reality; (Cambridge U.P.; London, 1929)

Wilber, K.; The Holographic Paradigm; (Shambala; Boulder, 1982)

Wilczek, F. and Devine, B.; Longing for the Harmonies; (W.W.Norton; NY, 1988, 87-7653)

Wilden, A.; SYSTEM and STRUCTURE; ( Travistock:
BIBLIOGRAPHY:

Kent Palmer has a degree from University of Kansas with majors in Sociology and East Asian Studies. In 1982 he received a Ph.D. degree in Philosophy of Science / Sociology from the London School of Economics, University of London. The title of his dissertation was The Structure of Theoretical Systems in Relation to Emergence and concerned the construction and unanticipated changes in theoretical systems. He has been working in the field of real-time software development since returning to the United States, first as a systems analyst, then as software engineer designing a real-time embedded systems, and now as a software engineering technologist for Rockwell Autonetic Electronic Systems Division. As an S.E. Technologist he has been engaged with cost estimation models, software development process modeling, the exploration of the use of software methodologies, evaluation and deployment of software tools, integration of software environments, software and systems engineering process development and other related activities, as well as normal software engineering design and development work. He has recently undergone training at the Software Engineering Institute in Process Assessment.

CODA:

After these papers we written Kent Palmer went on to write the following books

The Fragmentation of Being and the Path Beyond the Void. (Copyright Number TXu-884-397)

Reflexive Autopoietic Systems Theory (Copyright Number TXu-909-504)

Primal Ontology and Archaic Existentiality (Copyright Number TXu-1-040-820)


He also engaged in research toward a second Ph.D. in Systems Engineering. See http://holonomic.net for further details and associated works.
Bibliography

Apeiron Press

PO Box 1632
Orange,
California 92856

714-633-9508

palmer@exo.com
palmer@think.net
palmer@dialog.net
kent@palmer.name

Library of Congress
Cataloging in Publication Data

Palmer, Kent Duane
born 1951

Copyright 1996 by Kent Duane Palmer
Copyright Number TXu-885-046

Draft #1 950710 Editorial Copy.
Draft #2 070220 Small Changes

Not for distribution. Personal copies can be
made for scholarly study.

All rights reserved. No part of this book may be
reproduced in any form or by any means,
electronic or mechanical, including
photocopying, recording, or by any information
storage and retrieval system, without permission
in writing from the publisher.

This book was set using Framemaker document
publishing software by the author.

Electronic Version in Adobe Acrobat PDF
available at URL http://archonic.net/apeiron.htm

See also URL http://dialog.net

Electronic Version C published 980514.
Electronic Version D published 070220

1. Philosophy-- Ontology
2. Software Engineering
3. Software Design Methods
4. Software Process

Keywords:

Software, Design Methods, Ontology, Integral
Software Engineering Methodology, Systems
Theory