Applying Software Development Techniques to Statutory Drafting

Hyun G Lee, Boston University

Available at: http://works.bepress.com/hyun_lee/2/
I. Introduction

Statutes are often poorly drafted. Nearly everyone complains about them.1 Countless legal scholars have noted that legislators draft statutes poorly.2 Judges have also commented how

---

1 Alfred F. Conard, New Ways to Write Laws, 56 YALE L.J. 458, 458 (1947).

2 See Layman E. Allen & C. Rudy Engholm, Normalized Legal Drafting and the Query Method, 29 J. LEGAL EDUC. 380 app. A at 408-410 (1978) (listing a number of writings of legal scholars to support the propositions that legal drafting is important and is “now badly done and needs to be improved”).
the “courts are constantly called on to cure a poorly drafted statute.”

Even law school students have complained that some laws leave “many important questions unanswered.” Clearly, there is a broad, general agreement in the overall lack of quality in statutes and statutory drafting.

The cost of poor statutory drafting is high. The public cannot understand poorly drafted statutes that they are suppose to keep, which raises due process concerns. The state cannot justly demand obedience from its citizens of the laws that they cannot know or understand. Empirical data shows that even the highly educated members of society who are non-lawyers fail in understanding the meaning of statutes. Comprehension drops further for those who have only high-school education. This lack of comprehension further leads to high social costs of “uncertainty, unnecessary litigation, and expensive legal research.” Drafting clear, easy to understand statutes should therefore be of vital importance to drafters.

This note will examine three characteristics common in poor statutory drafting: legalese, ambiguity, and poor conceptual organization. Legalese is the arcane, lawyerly language that is

---


5 Building Officials & Code Adm'rs Int'l, Inc. v. Code Tech. Inc., 628 F.2d 730, 734 (1st Cir. 1980) (“Due Process requires people to have notice of what the law requires of them so that they may obey it and avoid its sanctions.”). U.S. CONST. amend. V. Similarly, U.S. CONST. amend XIV, § 1.

6 Banks & Bros. v. West Publ'g Co., 27 F. 50, 57 (C.C.D. Minn. 1886) (“Knowledge is the only just condition of obedience.”).


8 Id. at 544.

9 Allen & Engholm, supra note 2, at 381.
Some common-sense solutions can counter the use of legalese. Ambiguity, which can be either semantic or syntactic, is the uncertainty due to the multiple valid interpretations of the statute. Normalization technique, adapted from the mathematical notation of symbolic logic, can eliminate most ambiguities. Poor conceptual organization results in conceptually related topics physically scattered throughout the statute. Hypertext, legal dialectic, and object-oriented analysis and design are possible solutions to this problem. These characteristics and the solutions are described in more detail in Part II.

Part III examines the parallels between statutory drafting and software development. These parallels suggest that there may be other software development methods that are helpful in statutory drafting. There are, of course, many differences between these two disciplines, so every software development tool or method must be evaluated on whether or not it will work in statutory drafting. Part IV will outline requirements for evaluating the suitability of software development tools and methods for statutory drafting. Part V will evaluate different software development tools based on these requirements, and Part VI will evaluate different methods. Finally, Part VII will summarize the findings and suggest the software development tools and methods that are best suited for statutory drafting.

---


11 Allen & Engholm, *supra* note 2, at 381.


II. Background

This section will examine different characteristics of poor statutory drafting: legalese, ambiguities, and poor conceptual organization. This section will also present possible solutions to these problems, as well as discussing the shortcomings of these solutions.

a. Legalese

The use of “legalese,” the arcane language of lawyers, is perhaps the one widely known and often ridiculed characteristic of confusing statutes.\(^{14}\) The problem with legalese is that it is characterized by features such as passive verbs and idea-stuffed sentences that work against clear understanding.\(^{15}\) Lawyers are accused of using legalese for a variety of avaricious purposes: to protect their own money, power, and prestige by confusing the layperson.\(^{16}\) Despite the fact that lawyers themselves prefer plain English, they continue to use legalese because it is generally not confusing to them.\(^{17}\)

The solution seems simple: use plain English instead of legalese.\(^{18}\) To that end, some scholars suggest cutting out the jargon, using short sentences, giving examples, giving directions, and even using mathematics as simple ways to disambiguate statutes.\(^{19}\) These common sense methods do provide the drafter with some practical guidelines. However, these methods are

---

\(^{14}\) See Benson, supra note 7, at 520-22.

\(^{15}\) Id. at 531.

\(^{16}\) Gopen, supra note 10, at 345.

\(^{17}\) Benson, supra note 7, at 544; Sean Flammer, Persuading Judges: An Empirical Analysis of Writing Style, Persuasion, and the Use of Plain English, 16 LEGAL WRITING 183, 189 (2010).

\(^{18}\) Gopen, supra note 10, at 337.

\(^{19}\) Conard, supra note 1, at 473-80.
inadequate because they fail to provide any “theoretical underpinning” to their craft.\(^{20}\) The problem of poor drafting runs deeper than the problem of legalese; the more central problem is the lack of overall theory of legislative method. Citing Judge Posner, Professor Maxeiner claims that poor statutory drafting results from the lack of an overall theory of legislation in the United States.\(^{21}\) An overall theory of legislation requires methods of drafting, which legislatures and other drafting bodies have long neglected. Recent scholarship has focused on political process and statutory interpretation, but has paid “little attention to how legislatures should write rules.”\(^{22}\)

b. Ambiguity

A statute of poor quality is also characterized by ambiguity.\(^{23}\) An ambiguous statute will have multiple seemingly valid interpretations that will lead to different results. In this paper, the term “ambiguity” is restricted to mean the uncertainty of what is written.\(^{24}\) Uncertainty is often either a semantic or a syntactic problem.\(^{25}\) A statute may also be incomplete and not cover everything that the drafter had intended which leads to unintended and undesired

\(^{20}\) Blackwell, \textit{supra} note 13, at 234.


\(^{22}\) \textit{Id.}

\(^{23}\) Allen & Engholm, \textit{supra} note 2, at 381.

\(^{24}\) \textit{Id.}

\(^{25}\) \textit{Id.}
consequences. Both incompleteness and uncertainty introduce imprecision in the statute. Refer to Figure 1 for the relationship of these terms as used in this note.

Figure 1

Ambiguity may be either semantic or syntactic

Semantic uncertainty refers to a word or a phrase having wide range of different meanings and shades of meanings. Semantic problems are sometimes unavoidable but often desirable in that a more general term can be used in a rule to “cover unforeseen circumstances.” Also, many statutes include a “definitions” section to limit terms’ semantic ambiguity.

Syntactic problems arise from ambiguities in the syntactic structure and the relationships between the components of the sentence. Professor Layman Allen and C. Rudy Engholm introduced the normalization technique which can “virtually [eliminate] the likelihood of

26 Glazer-Esh, supra note 4, at 665.

27 Allen & Engholm, supra note 2, at 381.

28 This figure is adapted from Allen & Engholm, supra note 2, at 381.

29 Id. at 381-82.

30 Id. at 384.

31 See, e.g., H.R. REP. NO. 1249-2, at 1 (2011). This is the Leahy-Smith America Invents Act which will be examined later in this paper.

32 Allen & Engholm, supra note 2, at 382.
ambiguity” that the syntactic structure of a sentence causes. This method “clarifies structural or syntactic relationships between and among propositions stating factual or legal conditions … and propositions stating legal consequences.” Normalization “may be described … as a consistent combination of a limited set of standard legal syntax terms with traditional typographic techniques such as capitalization, outline labeling with letters and numbers, and indentation of text.” The set of legal syntax terms include IF, IF AND ONLY IF, THEN, AND, and OR, with specific rules that limit the type of proposition that can come before and after each term. Additionally, each proposition is a complete grammatical sentence and states either a condition or a legal result. Different syntax terms connect conditions and results, and the entire normalized statute is presented in an outline format. The normalization technique is “a compromise between expression in ordinary prose and expression in the mathematical notation of symbolic logic.” Other features of symbolic logic may be useful for the normalization technique. For example, utilizing symbols to represent propositions may increase clarity in compound propositions. Some scholars have claimed that using the normalization method in statutory drafting also enhances readability; experiments have shown that subjects can read

33 Gray, supra note 12, at 455.

34 Id. at 434.

35 Id. at 436.

36 Id.

37 Id. at 437.


39 See id. at 834.
normalized statutes faster and more accurately than non-normalized statutes.\textsuperscript{40} The following is a simple normalized statute:

\begin{verbatim}
IF
  (1) the superintendent receives a request for discharge, AND
  (2) the superintendent does not admit the patient under § 33-6-103,
THEN
  (3) the superintendent shall release the patient within twelve (12) hours after receipt of
       the request or at the time stated in the request, whichever is later.\textsuperscript{41}
\end{verbatim}

There are, however, limitations to normalization. Normalization requires a “translation interface.”\textsuperscript{42} The normalization technique results in an outline of the statute in the form of propositions connected by syntax terms. This final outline is neither fully English nor symbolic logic, but a hybrid language.\textsuperscript{43} Even if drafters draft statutes unambiguously in this hybrid language, it must be translated back to English because most people, including lawyers, are unfamiliar with symbolic logic. This translation process from the hybrid language to English will introduce new syntactic ambiguities that may have been absent in the outline form.\textsuperscript{44} So although normalization may eradicate many of the syntactic ambiguities, the translation process may introduce syntactic ambiguities in the final, translated, English version of the statute.\textsuperscript{45}

Furthermore, normalization cannot systematically detect the imprecision arising out of incompleteness. While drafters may use normalization to eradicate both semantic and syntactic ambiguities, especially in highly complex situations, normalization does not detect

\begin{flushright}
\textsuperscript{40}Gray, \textit{supra} note 12, at 435, 447.
\textsuperscript{41}Gray, \textit{supra} note 12, at 435.
\textsuperscript{42}Blackwell, \textit{supra} note 13, at 239.
\textsuperscript{43}See Allen \textit{supra} note 38, 833.
\textsuperscript{44}Blackwell, \textit{supra} note 13, at 240.
\textsuperscript{45}\textit{Id.} at 239.
\end{flushright}
incompleteness and oversight.\textsuperscript{46} For example, the normalization process may result in the following statement in the hybrid language:

\begin{center}
IF P THEN Q.
\end{center}

The drafter can readily verify that this statement is syntactically error free because it obeys every rule of the hybrid-language. The drafter can also verify that this statement is semantically error free by examining each of the constituent propositions, P and Q. If, however, the drafter has overlooked another possible condition or legal result, then the final statement will be incomplete. For example, the final statement could have required an additional legal result:

\begin{center}
IF P THEN Q AND R.
\end{center}

The drafter cannot readily detect such imprecision from incompleteness just by using normalization and symbolic logic.

Lastly, normalization does not address poor conceptual organization,\textsuperscript{47} a problem that will be discussed in the next section. Due to these limitations, drafters may not exclusively rely on normalization to improve statutory drafting.

\medskip

c. Conceptual Organization

In addition to legalese and ambiguity, poor conceptual organization is another characteristic of poor statutory drafting. Poor conceptual organization results in “redundancy, inconsistency, ambiguity, and overall lack of coordination.”\textsuperscript{48} While semantic and syntactic ambiguities are located at the propositional and statement level,\textsuperscript{49} poor conceptual organization is

\textsuperscript{46}Id. at 239-40.

\textsuperscript{47}See id. at 233-34. See generally Allen & Engholm, supra note 2.

\textsuperscript{48}Blackwell, supra note 13, at 289.

\textsuperscript{49}See generally Allen, supra note 38, at 833.
found at a higher structural level.\textsuperscript{50} A common result of multiple conceptual linkage and complex structural organization is the abundance of cross references, which are means of coping with conceptually related topics physically scattered throughout a statute.\textsuperscript{51}

Cross references in general are useful.\textsuperscript{52} Without cross references, a reader interpreting just one section would need to be familiar with the rest of the statute because there may be some “exception lurking elsewhere….”\textsuperscript{53} However, some cross references may increase the ambiguity because of the “varying degrees of inexplicitness that add to the difficulty of knowing how the [statute] is going to behave.”\textsuperscript{54} Even highly explicit cross references, in large numbers, may “add distracting clutter.”\textsuperscript{55}

Professor Blackwell has suggested using hypertext to address the ambiguities that an abundance of cross references causes. Hypertext is “[a] system for writing and displaying text that can be linked in multiple ways to related documents….”\textsuperscript{56} It is “a nonlinear system of information browsing and retrieval based on associative links between documents,” that is used on the internet “to link pages and multimedia files.”\textsuperscript{57} One scholar has even

\textsuperscript{50}Blackwell, \textit{supra} note 13, at 234.

\textsuperscript{51}Blackwell, \textit{supra} note 13, at 234, 293 n120.

\textsuperscript{52}See Gray, \textit{supra} note 12, at 437 (“[I]t is desirable to use explicit cross-references whenever possible to avoid redundancy.”).

\textsuperscript{53}Allen & Engholm, \textit{supra} note 2, at 391.

\textsuperscript{54}Id.


suggested that “[l]aw is inherently hypertextual.” 58 This solution, however, does not address the underlying problem of bad conceptual organization. 59 Because hypertext is merely “a technological refinement of … cross-reference,” it does not reduce the ambiguities that arise from a poorly organized statute that unnecessarily increases the required number of cross-references. 60 A better organization of a statute may reduce the ambiguity by minimizing the number of conceptual links across physically scattered provisions of a statute.

Other legal scholars suggest legal dialectic as a way to improve conceptual organization in statutory drafting, particularly by drafting with consistency, sound arrangement, and normal usage.61 In order to “add rigor” to this art of organizing the statutes into conceptual categories, Professor Blackwell suggests adopting “principles developed in the unrelated field of computer software design.” 62 Specifically, he recommends using object-oriented analysis and design methods as a means of better arranging statutes into conceptually coherent structures.63

In summary, poor statutory drafting has the following characteristics: legalese, ambiguity, and poor conceptual organization. These characteristics are analogous to those found in software development. In addition, some of the previous solutions mentioned above are also analogous to different software development methods. There are also many parallels between the two disciplines of statutory drafting and software development. All these parallels are examined in the following section.

58 Samuelson, supra note 57, at 2048 (1996).

59 Blackwell, supra note 13, at 258.

60 Blackwell, supra note 13, at 258.


62 Blackwell, supra note 13, at 237.

63 Blackwell, supra note 13, at 267-88. See infra Part V.
III. The Parallels Between Statutory Drafting and Software Development

There are obvious differences between statutes and computer programs. Drafters write statutes in natural, human language and computer programmers write programs in programming language. People execute statutes, and computers execute computer programs. The biggest distinction is that statutes are ambiguous, but texts written in programming languages are not.64 Despite these differences, there are “some common laws [that] hold for both natural language texts and computer programs,” which “enables one to draw parallels between these two different types of human writings.”65 Due to these parallels, some software development tools and methods can be useful for statutory drafting. There is one major potential benefit of using software development tools and methods. Because texts in programming languages are unambiguous, using some of the software development tools and methods may reduce ambiguity of the resulting statute.

One obvious similarity between statutes and programs is that drafters of both write in a language. Both contain strings of symbols, a set of vocabularies, and are written according to grammatical rules.66 Furthermore, statutes have more in common with programs in that they are both commands.67 The law instructs a person how to behave, and the program instructs the computer processor which command to execute.68 The primary goal of statues is to be accurate

64 Kokol et al., Computer and Natural Language Texts – A Comparison Based on Long-Range Correlations, 50 JOURNAL OF AMERICAN SOCIETY FOR INFORMATION SCIENCE 1295, 1297 (1999).
65 Id.
66 Id. at 1300.
67 See HOBBES, LEVIATHAN 162 (1651) where Hobbes defines civil law under section XXVI.
68 JOHN AUSTIN, THE PROVINCE OF JURISPRUDENCE DETERMINED 13 (1832) (“By Every command, the party to whom it is directed is obliged to do or to forbear.”).
and to “give legal effect to Government policy.”

Analogously, the primary goal of software development is to create software that works, that accurately fulfills the customer’s intent. Another important goal of statutes is to be simple, clear, and easy to understand to the “persons who administer the law and citizens who must obey it.…” As with statutes, programs should be simple, clear, and direct, because this will make debugging and modification much easier later on, ensuring that the program will function properly. In programming, clarity is even more important than efficiency, because “the harder it is for people to grasp the intent of any given section, the longer it will be before the program becomes operational.” Writing clear, “readable code” is the only way to ensure creating “efficient programs that are also easy to maintain and modify.” In order to meet the intent of the customers, programmers need to create programs with high reusability and well-known concepts and designs.

---

69 Jack Stark, Should the Main Goal of Statutory Drafting Be Accuracy or Clarity?, 15 STATUTE L. REV. 207, 209-10 (1994).

70 Id. at 208. See BROOKS, FRED P. BROOKS, THE MYTHICAL MAN-MONTH 8 (Anniversary ed. 1995). (“[O]ther people set one’s objectives.…’

71 Reed Dickerson, Legislative Drafting: A Challenge to the Legal Profession, AMERICAN BAR ASSOCIATION JOURNAL, July 1954, at 635 (“Sound government depends upon legislation that says the right thing in the right way, in language that is as clear, simple, and accessible as possible.”).


“Debugging” is a process in which “bugs,” the malfunctions, of a piece of software is found and fixed. See infra note 86 and accompanying text.

73 See KERNIGHAN & PLAUGER, supra note 72, at 11, 123 (where “efficiency” is defined as “reduction of overall cost” in both “machine time over the life of the program,” the “time spent by the programmer,” and “by the users of the program.”). See also id. at 127 (“Make it clear before you make it fast.”); id. at 128 (“Don’t sacrifice clarity for small gains in ‘efficiency.’”)

74 Id. at 123.

75 See ERICH GAMMA, RICHARD HELM, RICHARD JOHNSON & JOHN VLISSIDES, DESIGN PATTERNS: ELEMENTS OF REUSABLE OBJECT-ORIENTED SOFTWARE 1 (1995). The authors are commonly referred to as “The Gang of Four.”
known patterns and reusable code increases clarity, because it allows others who must maintain
and modify the code to quickly understand it.\textsuperscript{76}

There are more parallels between statutes and software in the way their writers create
them.\textsuperscript{77} The programmer “works only slightly removed from pure” ideas; he shapes and forms
these ideas to represent real world objects into code.\textsuperscript{78} Similarly, the drafter works only slightly
removed from ideas of justice and public policy.\textsuperscript{79} Like the “program construct,” a statute is
“real in the sense that it moves and works.”\textsuperscript{80} While a program can “produc[e] visible outputs
separate from the construct itself,” a statute has the output of affecting the way people live.\textsuperscript{81}
The ways in which a statute is written, executed and interpreted, are also analogous to the ways
in which a piece of code is written, executed, and debugged. The U.S. government has three
branches: legislative, executive, and judicial.\textsuperscript{82} The legislative branch makes the law.\textsuperscript{83} The
executive branch executes the law, and the judicial branch interprets the law.\textsuperscript{84} Analogously,

\textsuperscript{76} See KERNIGHAN & PLAUGER, supra note 72, at 10, where clarity is enhanced by the use of library functions. See
also id. at 61 where clarity is enhance by modularization.

\textsuperscript{77} For a comparison of writing a novel and programming in view of these three stages of creative activity, see
BROOKS, supra note 70, at 15, which draws from the Dorothy Sayers’ classic book on creativity, The Mind of the
Maker.

\textsuperscript{78} Compare id. at 7 which contrasts the way in which the programmer and the poet both work with a tractable
medium. See Part V.c

\textsuperscript{79} Maxeiner, supra note 21, at 518.

\textsuperscript{80} BROOKS, supra note 70, at 7.

\textsuperscript{81} Id. at 7-8.

\textsuperscript{82} U.S. CONST. art. I-III.

\textsuperscript{83} Id. at art. I § 8.

\textsuperscript{84} Id. at art. II § 1; id. at art. III § 1.
software is written by the programmer, executed by the processor, and debugged using a debugger.\textsuperscript{85}

Due to these parallels between statutes and programs, both statutory drafting and software development share common problems and solutions. For example, the problems analogous to legalese occur in programming. While texts written in programming language may be unambiguous to the compiler, it may nevertheless be ambiguous to the programmers who write or maintain it, in a way that’s too “clever” for other programmers to easily understand.\textsuperscript{86} The parallels extend to the common sense solutions to reduce legalese, which include cutting out the jargon, using short sentences, giving examples, giving directions, and using mathematics to simplify statutes.\textsuperscript{87} Likewise, programmers are advised to write simply and directly, to test their inputs, and to comment their code.\textsuperscript{88}

The problems of incompleteness and uncertainty are commonplace in programming. Programs are often “incompletely delivered (no source code or test cases) and poorly

\textsuperscript{85}To “write” a piece of software means to actually write the human readable code, that is then translated into machine readable form by a compiler. A “compiler” is a program that is created specifically for the purpose of translation. Programs are written in programming languages, such as C++ and Java, or in hardware architecture specific assembly language (which are less readable and have closer correspondence to the machine readable form).

A program is “executed” after it has been converted to machine readable form, when the processor steps through each of the instructions and executes the command.

“Debugging” is a process in which “bugs,” the malfunctions, of a piece of software is found and fixed. One important method is to start the program with a test value and going through the code, line by line. This “debugging” process is analogous to the judicial branch, in that both performs the function of applying what is written to one specific instance or set of input data.

\textit{See} TOBEN \AE GIDIUS MOGENSEN, BASICS OF COMPILER DESIGN 1-3 (Anniversary ed., 2010).

\textsuperscript{86} KERNIGHAN & PLUGER, supra note 72, at 1-2.

\textsuperscript{87} Conard, supra note 1.

\textsuperscript{88} KERNIGHAN & PLUGER, supra note 72, at 84, 142-43. To “comment” a code is to write in natural language explaining what the code does.
In statutes, incompleteness may stem from simple oversight; this is true in programming as well. Incompleteness in software development also stems from the inability to test thoroughly. For nearly all programs, it would be impossible to test every line of code with every possible input due to limited time and resource, because each variable can store one of many different possible values.

As in statutes, programs contain semantic and syntactic ambiguities. Syntax errors refer to an error in which a piece of code does not conform to the strict grammatical rules of the programming language. Because programming languages have a smaller set of vocabularies and are “much more restricted and formal than natural languages,” the compiler can immediately detect syntax errors the programmer makes. However, even if a piece of code does not contain any syntax errors, a programmer can still write in a way that is ambiguous to the programmer.

---

89 See BROOKS, supra note 70, at 8. Professor Brooks even recommends trimming the features when the development team falls behind schedule, which of course, is another cause of incompleteness – the program not doing everything it was originally designed to do. Id. at 24.

90 See KERNIGHAN & PLauger, supra note 32, at 104 (“Simple oversight is the most common way to botch initialization.”).

91 See BROOKS, supra note 70, at 183 (“Digital computers have … have very large number of states. This makes … testing them hard. Software systems have orders of more magnitude more states than computers do.”). Not that any programmer would want to test and debug their program so thoroughly, even with infinite processing power and memory. “[D]esigning grand concepts is fun; finding nitty little bugs is just work.” BROOKS, supra note 70, at 8.

92 See KERNIGHAN & PLauger, supra note 72, at 28 (where an example illustrates both a semantic and syntactic mistake.).

93 Kokol, supra note 64.

94 BROOKS, supra note 70, at 225. See Kokol, supra note 64 where natural languages are described as ambiguous in contrast to programming languages which are unambiguous. For a description of “compiler,” see note 85.

95See KERNIGHAN & PLauger, supra note 72, at 1-2 for examples of two functionally equivalent lines of code that stores an identity matrix. The first is written in a way that is ambiguous to the programmer, while the second is much less ambiguous.
Furthermore, while the compiler can detect some basic semantic errors, most semantic errors are harder to detect.\textsuperscript{96}

Consistent use of programming conventions can reduce some instances of semantic ambiguity. Just as “definitions” sections can limit semantic ambiguities in statutes, predefined sections of the code can clearly define other parts of the code.\textsuperscript{97} Also, using pseudo code to either plan out the program before hand or to understand an existing code is similar to the use of symbolic logic in statutes.\textsuperscript{98} Pseudo code is a way of writing out the structure of the code in a less formal way, with less restraint on the syntax, to quickly plan out how the code will be structured. The programmer must translate pseudo code into a programming language,\textsuperscript{99} just as the drafter translates the symbolic logic notation into natural language.

Programs also suffer from poor conceptual unity and design.\textsuperscript{100} Just as how hypertext can link conceptually similar sections of a statute, programming editors and integrated development environments, or IDEs, can provide links to where a function or a class is defined.\textsuperscript{101} An IDE is a “set of tools that aids application development” that allows the

\textsuperscript{96} Initialization of variables, for example, can be detected by java compilers. \textit{Compile Time Error Messages: Java Glossary, CANADIAN MIND PRODUCTS}, http://mindprod.com/jgloss/compileerrormessages.html. \textit{See BROOKS, supra} note 1, at 196 (“Language-specific smart editors” can at “most … promise … freedom from syntactic errors and simple semantic errors.”).

\textsuperscript{97} \textit{See MISFELDT, BUMGARDNER & GRAY, supra} note 72, at 112 (“Initialize all variables.”); \textit{id.} at 113 (“Initialize Member Variables in the Initializer List.”). These principles illustrate the need to clearly define variables and members (and by extension, class instances) in a consistent location.

\textsuperscript{98} \textit{See supra} notes 29-39 for a description of symbolic logic.

\textsuperscript{99} Definition of pseudo code, \textsc{dictionary.com}, http://dictionary.reference.com (search for “pseudo code”).

\textsuperscript{100} \textit{See BROOKS, supra} note 70, at 42-44 (“[M]ost programming systems reflect conceptual disunity far worse than that of cathedrals.”); \textit{id.} at xi (“The flaws in design … pervade especially the control program.”); GAMMA, HELM, JOHNSON & VLISSIDES, \textit{supra} note 28, at 1 (“[N]ew designers are overwhelmed by the options available…..”).

\textsuperscript{101} Most Integrated Development Environments (“IDE”), such as Microsoft Visual Studio, allows the programmer to open up the section of the code that is being used. A “function” is a section of code that, given a set of inputs, provides a set of outputs, and can be used repeatedly. A function is “called” whenever it is used. A “class” is a collection of variables, functions, and classes that can be used to represent an object. \textit{See Peter Müller,}
programmer to “write and edit source code”, find errors, and see highlighted code syntax, among other features. 102 While IDEs can boost productivity by providing links to function and class definitions, such features do not eradicate the problem of bad designs. 103 On the other hand, many software developers have recognized object-oriented analysis and design “as the basis for a revolution in the creation of computer software,”104 which may also aid statutory drafting. 105

In summary, statutes and software share many similarities. They both share a common goal of clarity, and because of that, many software development methods correspond to statutory drafting methods. Due to these similarities, there are other methods and aspects of software development that may be useful in statutory drafting.

IV. The Requirements for Selecting Tools and Methods

In order to evaluate the software development tools that may be useful for statutory drafting, each of these tools must meet some requirements. Although programming and drafting are similar, they are different in many aspects. This note presents these requirements to select the programming tools that will most likely be useful in drafting. In programming, using the right tool or method cannot reduce any of the complexity that is essential to the task, but it may

---


103 See BROOKS, supra note 70, at 196 where Professor Brooks has accurately predicted that the “biggest gains” to be made in programming environments are ways to “keep track of myriads of details that must be recalled accurately by the individual programmer….”

104 Blackwell, supra note 13, at 268. See also id., at 189 where Professor Brooks accurately predicts that object-oriented programming will have a great impact on software development.

105 See supra text accompanying note 24.
reduce the accidental complexity that results from using the wrong tool or method.\textsuperscript{106} Likewise, using the right tool or method in statutory drafting can reduce some of the accidental complexity that results from using a wrong particular tool or method.\textsuperscript{107}

The new 35 U.S.C. § 102 from the America Invents Act can illustrate the distinction between essential and accidental complexity.\textsuperscript{108} This section is about novelty, on whether the public or someone other than the inventor already knows about the invention that is the subject of the patent application. The new § 102(a) presents two ways in which a prior art is specified. Section 102(a)(1) precludes novelty if the invention was available to the public before the patent application filing date. Section 102(a)(2) precludes novelty if the invention was the subject of a patent or application by another inventor. Section 102(b) presents two exceptions corresponding to two subsections of 102(a): Exceptions to 102(a)(1) are placed under 102(b)(1), and exceptions to 102(a)(2) is placed under 102(b)(2).

Part of the essential complexity derives from the number of conceptual links across different sections and subsections.\textsuperscript{109} Bad statutory organization may increase the accidental complexity if it increases the number and the length of physical separation of these links unnecessarily. The essential complexity in § 102 lies in the fact there are four conceptual links in 102(a) and 102(b): between 102(a)(1) and 102(a)(2), between 102(a)(1) and 102(b)(1), between 102(a)(2) and 102(b)(2), and between 102(a)(2) and 102(b)(2). The structure of 102(a) and (b) increases the accidental complexity by separating two conceptual links that a restructuring can eliminate. Thankfully, in this specific instance, the relative shortness 102(a)

\textsuperscript{106} See BROOKS, supra note 70, at 182.

\textsuperscript{107} See infra Part V.a for examples of essential and accidental complexity.

\textsuperscript{108} See supra note 22.

\textsuperscript{109} See supra Part II.c and accompanying text.
means that physical separation between the related subsections are minimal, thereby only slightly increasing accidental complexity. However, this added accidental complexity need not exist at all with a better organization of the statute; specifically, 102(b)(1) can be incorporated under 102(a)(1), and 102(b)(2) can be incorporated under 102(a)(2). This reorganization would eliminate two conceptual links that are present in the current arrangement.

This note proposes four requirements for every tool and method that Parts V and VI introduce. These requirements are as follows: (1) there should be a decrease in overall complexity and ambiguity, (2) the tool or method should not require the drafter or the reader to possess intimate knowledge of software development, (3) the resulting statute should be clear and accessible to the people it affects, (4) the tool or method should not require either the drafter or the reader to perform machine-like repetitive tasks or memorization. These requirements will ensure that the tools and methods are best suited for statutory drafting.

a. Decrease in Overall Complexity and Ambiguity

The goal of introducing these software development tools and methods is to reduce overall ambiguity for those who read the statute. These tools and methods should also reduce the accidental complexity of the task of drafting. While using a particular tool or a method can reduce one source of accidental complexity or ambiguity, it may also introduce other sources of complexity and ambiguity. For example, the reorganization of § 102 above may indeed reduce one source of accidental complexity by reducing the number of conceptual links. However, it also increases the physical length of one of the remaining conceptual links, between 102(a) and 102(b). This increase in physical length then causes a slight increase in the accidental complexity of the organization and may be a source of ambiguity to the reader.
Software development tools or methods may also present accidental complexity to the drafter, which may or may not also increase ambiguity for the reader. For example, the use of nominalization and symbolic logic in statutory drafting would require a translation, which would introduce syntactic ambiguities.\textsuperscript{110} Using such methods require the drafters to learn a hybrid language, which is another source of accidental complexity.\textsuperscript{111} However, using normalization methods may decrease the overall accidental complexity of the task as compared to using natural human language, and if so, learning a hybrid language may very well be justified.\textsuperscript{112} Similarly, any other method that requires a translation will have a similar source of accidental complexity in the translation process. The first requirement, then, is that the overall complexity and ambiguity should decrease with the utilization of the tool or the method.

b. Knowledge of software development

Another potential source of accidental complexity is the ordinary drafter’s unfamiliarity with programming. Most drafters are not programmers, and are generally technology averse. The legal field has “an innate aversion to change.”\textsuperscript{113} If the tool or the method requires the drafter to possess a significant set of technological or programming knowledge, then it would have a lower probability of being adopted. On the other hand, if the tool or the method is widely known or shares similar concepts and ideas with technologies the general population or the

\textsuperscript{110} See supra note 85 for a description of what translation means.

\textsuperscript{111} See supra note 42-46 and accompanying text.

\textsuperscript{112} See Blackwell, supra note 13, at 240.

\textsuperscript{113} Richard L. Marcus, The Impact of Computers on the Legal Profession: Evolution or Revolution?, 102 NW. U. L. REV. 1827, 1830 -32 (2008). For example, while today’s law schools use laptops, emails, and websites, “[c]onsiderably more radical law school reliance on computers can be imagined,” where schools can “bundle services … to produce a dispersed law school experience at much less cost to the students.” Due to law schools’ slow adaption to new technologies, “the current law school experience resembles the precomputer experience far more than it differs.” See also Blackwell, supra note 13, at260 (“[T]he law … has generally been conservative in adapting to change and adopting new technologies.”).
drafters already employ, then it would significantly reduce the potential amount of accidental complexity in using the tool or method causes. For example, hypertext is used extensively on the internet, and thus it is a familiar concept to most people.\textsuperscript{114} Many lawyers work and think in a way similar to the way hypertext works, even when using traditional print media.\textsuperscript{115} Therefore, lawyers may in the future “learn to use digital technologies to facilitate the intertextual nature of legal analysis.”\textsuperscript{116} Thus, the drafter should not have to know intimate details about programming in order to utilize these tools and methods; any tool or method employed must be similar to what the drafter is already familiar with.\textsuperscript{117}

c. Clarity and Accessibility

Third requirement is that the resulting statute should be clear and accessible to the people it affects and to the general public. For this requirement, clarity refers to any additional ambiguity that might be present in the final statute that requires the reader to be familiar with software development tools or methods. Not everyone in the general public is familiar with or has access to various technologies.\textsuperscript{118} One of the limitations of the hypertext solution, for example, is that “those persons who do not have electronic access … are unable to gain access to

\textsuperscript{114} See Samuelson, \textit{supra} note 57, at 2050.

\textsuperscript{115} Id. (“When lawyers do legal research in a traditional print-based law library, they engage in a lot of nonlinear activities. Tools, such as the West key number system, enable legal researchers to skip past most parts of judicial opinions to locate their golden nuggets.”).

\textsuperscript{116} Id. at 2051.

\textsuperscript{117} See Allen, \textit{supra} note 38, at 833 (“This approach is a compromise between expression in ordinary prose and expression in the mathematical notation of symbolic logic-enough like ordinary prose to be understood easily by any careful reader, enough like symbolic logic to achieve some of its important advantages.”).

\textsuperscript{118} See \textit{supra} note 114-15.
the official form of a hypertext statute.”  Inaccessible statute raises the “possibility of violations of due process, [and] equal protection.”

One requirement of due process is that people must have “notice of what the law requires of them so that they may obey it and avoid its sanctions.” People without electronic access to the hypertext statute, for example, would only have the printed version of the statute that is an “approximation or a paraphrase of the official version, and the more complex the statute, the more approximate the paraphrase.”

Equal protection concerns are present as well. There are a high percentage of racial minority households that do not have access to internet at home. Educational attainment has a huge correlation to internet availability at home as well. Due to this difference in access to technology, if a statute requires technology for proper access, the statute could be “subject to charges of denial of equal protection rights.”

Due to these concerns, the final statute should not be drafted in a way that it is only accessible by the more technologically savvy or comprehensible to only those who are familiar

---

119 Blackwell, supra note 13, at 261.
120 Blackwell, supra note 13, at 261-62.
121 Id.
122 Id.
123 U.S. Census Bureau, Table 1. Reported Internet Usage for Households, by Selected Householder Characteristics: 2009, http://www.census.gov/hhes/computer/publications/2009.html (2009). For racially White households, 29.5% did not have internet at home. For Asian households, 19.5%. For Black households, 45.5%. For Hispanic households, 47.2%.
124 Id. For households with less than high school graduate, 67.8% did not have internet access at home. For high school graduate, 42.5%. For some college or associate’s degree, 25.3%. For bachelor’s degree or higher degree, 11.5%.
125 Blackwell, supra note 13, at 266-67.
with programming concepts. Also, drafters should remove from the final statute any
programming ideas or concepts that can potentially confuse the general public.

d. No repetitive tasks or memorization.

Fourthly, any software development tool or method should not require the drafter or the
reader to perform machine-like repetitive task or memorization. There are basic differences in
the way humans and machines work. Computers can perform repetitive tasks much quicker than
a human being, and can access huge amounts of memory almost instantaneously. Many software
development tools and methods were designed to take advantage of this. 126 Therefore, any tool
or method that requires inordinate amounts of repetitive tasks or memory is not viable in the
statutory drafting context. For example, Professor Gray’s utilization of the normalization
method excludes the use of symbols to represent propositions in compound propositions. 127
Using symbols to make substitutions are a quick and easy task for computers, but a laborious
task for humans; this is the same reason that the abundance of cross-references can be a
significant source of ambiguity and clutter. 128

In summary, this note proposes four requirements that provide a baseline for comparison.
Using these four requirements, the next sections will examine some of the software development
tools and methods.

V. The Tools

126 See KERNIGHAN & PLAUGER, supra note 72, at 12 (“Let the machine do the dirty work.”).

127 See supra note 24-30 and accompanying text.

128 See supra note 45 and accompanying text.
The following tools in software development are easy to understand and programmers/drafters can use them in a variety of ways to reduce ambiguity and overall accidental complexity. They are variables, data structures, and integrated development environments.129

a. Variables

In programming, variables hold large range of values.130 There is an analogue to variables in statutes – a “term of art,” which is a “word or a phrase that has special meaning in a particular context.”131 Similar to variables used in programming, a “term of art” can mean one thing in a statute, and something else in a different statute. For example, in Johnson v. United States, the Supreme Court found the phrase “physical force” to have numerous meanings, and that the common law meaning applies unless the meaning does not fit.132

Scope is an important characteristic of variables. The scope of a variable in programming refers to where the programmer can access the variable from.133 A variable’s scope may be within a function, or a class, depending on where and how the programmer

---

129 See supra note 68 for a definition of integrated development environment.


133 JAMES GOSLING, BILL JOY, GUY STEELE, GILAD BRACHA, ALEX BUCKLEY, THE JAVA LANGUAGE SPECIFICATION 130 (Java SE 7 ed., 2012).
declared the variable. Simp ily put, the variable’s scope can range from just a few lines to across multiple files and systems.

If “term of art” is analogous to variables, what is the scope of a “term of art” and how are they declared? Some of the “term of art” is developed over time by common law; some are defined in the “definitions” section of an act. These “definitions” sections are usually at the beginning of the entire act, and contain definitions of term of art for the entire act. So for example, the definitions section in the Copyright Act, 17 U.S.C. § 101, contains all the definitions for the entire title. In programming, such an arrangement would be like declaring all the variables that the programmer will use in the program at the beginning or at the highest scope. Such practice would not work for a larger number of variables as it would be harder for the programmer to discern the purpose, of where, when, why, and how the programmer uses each variable. Thus, such arrangement would reduce clarity and readability for the programmer. In statutes, if the reader is not aware of every term that is defined in the definitions section, then understanding the statute can be a time intensive and repetitive task of seeing if any of the terms have a specific meaning in the definitions section. By contrast, programmers do not arrange variable declarations for the entire program all at the beginning except for very small programs.

---

134 See supra note 102 for a definition of a “function.” See infra note 187 for a definition of “class.”

135 See supra note 14, 125.


137 See Oracle, Code Conventions for the Java Programming Language, http://www.oracle.com/technetwork/java/javase/documentation/codeconventions-141270.html#2991. In the example under Placement, the variable int2 is placed in the inner ‘if’ block, and not in the outer myMethod function. Also, placing all variable declarations in the outer most block will potentially cause more local declarations that hide declarations at higher levels. Id.

138 MISFELDT, BUMGARDNER & GRAY, supra note 72, at 5 (For clarity, “[e]nsure that each … variable has a clear purpose. Explain where, when, why, and how to use each.”).
Programming editors and IDEs also provide a partial solution to unwieldy number of variables with vast scopes, by providing a link to the variable declaration. An analogous technological solution such as the use of hypertext would work well for statutes for definition sections. Using hypertext, every defined term would provide a link to its definition in the definition section. This task does not even need to be done by the drafter; after a statute is drafted, a simple program can add the hypertext information automatically by replacing every instance of a defined term in the statute with a link to its definition. Hypertext fulfills the second requirement that the drafter should not need to know much about programming. Also, regarding the fourth requirement, use of hypertext would reduce the repetitive task of looking up every term, and the reader will not need to have an archival knowledge of the definitions section. Using hypertext would also fulfill the first requirement since the drafter can add hypertext post-draft and would not have any new source of accidental complexity if the statute viewing software hides hypertext information from the user.

However, the disadvantage is that “a statute that officially includes hypertext as a structural feature cannot be precisely duplicated in print media.” This means that a statute would not be reasonably obtainable to those without access to software that can properly parse, view, and link the hypertext in the statutes. Even if such software was readily available, and even though navigating through hypertext material would only require a minimum amount of knowledge that is very similar to browsing the internet, such requirements would prevent some members of the general public from accessing the statutes. Thus this solution of using the hypertext specifically fails to satisfy the third requirement of being accessible to the public. If

---

139 See supra note 102 and accompanying note.

140 See supra note 56 for a description of hypertext.

141 Blackwell, supra note 13, at 259.
the final statutes contain hypertext material in the print version, the extra semantics and symbols would cause confusion and add to the overall accidental complexity for the public without the viewer software.

However, the fact that these definitions are very similar to the way variable scope work in programming, suggests a different solution. In programming, if a programmer will use a variable only within a function, he should declare the variable inside the function and not outside of it.\textsuperscript{142} Similarly, if a drafter will only use a word or a phrase within one section or a chapter, he should only define it within that section or chapter. Some of the more industry-specific chapters of the Copyright Act already have this feature,\textsuperscript{143} and this can be extended to other sections. A simple program can perform this process of limiting the definitions to a chapter post-drafting. The program would look for words or a phrase in across multiple chapters, and if found, place the definition in the definition section of the entire title. Otherwise, the program can place the definition section of the chapter where the term is used.

Such post-draft organization of definitions may reduce the size of the title definition section and decrease the physical separation between some of the terms and their use and thereby decrease some accidental complexity. Thus the first requirement is satisfied; the overall accidental complexity and ambiguity will never increase from using this method.\textsuperscript{144} The physical separation of the definitions of the terms and the use of the terms will also be shorter. The second requirement is also satisfied in that this task can be automated post drafting by a computer program; the drafter need not possess any programming skills. The third requirement,

\begin{footnotesize}
\begin{itemize}
  \item \textsuperscript{142} See Oracle, Code Conventions for the Java Programming Language, http://www.oracle.com/technetwork/java/javase/documentation/codeconventions-141270.html#2991.
  \item \textsuperscript{144} See supra Part III.a
\end{itemize}
\end{footnotesize}
that the final statute be clear and accessible to the reader, is also satisfied because to the ordinary person the resulting statute will not be any different except that the definitions will be reorganized in some instances. Unlike the hypertext solution to the definitions, the final statute will not require a software reader. The fourth requirement, however, is not satisfied; some repetitive task or archival knowledge may yet be required because the reader of the statute must now look at the definition section of the chapter first before looking at the definition section of the title. Thus, this solution would require one additional definition lookup by the reader for every term of art, which is similar to computer’s repetitive task of looking up memory.\footnote{Merriam-Webster, \textit{lookup}, http://www.merriam-webster.com/dictionary/lookup.}

However, this repetitive task may actually benefit the reader, in that the reader will realize that when a term is defined higher up in the chapter or in the title, the reader will know that term is being used elsewhere and may have a broader usage than the specific section that he was looking at.

A final solution is analogous to the way in which IDEs handle variables. In most IDEs, variable names are highlighted differently from keywords or operators.\footnote{See supra note 103. A “keyword” is a word in the programming language that is reserved for a specific use. For example, in C or Java, “for” is a keyword that denotes a start of a loop. See Kolkol, supra note 26 for a more detailed explanation.} When the programmer types in the name of a variable, he can inspect information about the variable instantly.\footnote{Quick Info, MSDN (Jan. 15, 2010), http://msdn.microsoft.com/en-us/library/137ey1yz.aspx. Note that Quick Info can also be used in other identifiers.} Similarly, any term of art that is defined in a different section in a statute can have a different typography. This is also done in hypertext, but in that solution the visual representation is only useful using the viewer software and the extra hypertext tags can cause confusion without the
Here, the difference in visualization of defined terms of art should not cause confusion when viewed from print media but can also be enhanced using a software viewer that can parse the extra commands in the text. A special typography and/or metadata convention can denote in the print media which words and phrases are defined in a separate section. For example, one possible metadata convention is a parenthetical information right after the term of art wherever it is used outside of a definition section. So in the Copyright Act, whenever the term “anonymous work” is found outside of a definition section, it is immediately followed by parenthetical information that states where it can be found. This may look something like: “(‘anonymous work’ is defined in § 101).” Of course, other typography can just as easily denote the metadata indicating that a term is defined in another section.

In terms of the requirements, the first requirement states that the tool should not increase the overall accidental complexity or ambiguity; the IDE-inspired solution satisfies this requirement. As for the second requirement, the drafter does not need to have any programming skill or knowledge as a programmer can do this post drafting. For the third requirement, the reader does not need to have any programming knowledge either because the final statute will only contain extra parenthetical (or other typographical) information that points to the source of definition of the term. Furthermore, no extra repetitive lookups or archival knowledge is required by the reader of the statute. If the average reader did not know about the term of art, he would have had to look up the definitions anyway or would have not realized that the term was
defined in another section in the first place. If he did know the definition already, then it only takes him minimal amount of time to skip over the extra parenthetical (or other typographical) information. This IDE-inspired solution, then keeps the reader from having to possess archival knowledge of the statute.

The idea of variables in programming suggested the solutions above in hypertext of definitions, reorganizing the definitions according to scope, and using different typography and metadata such as parenthetical to denote where a term is defined. Moreover, the drafter can combine all three solutions. Similar to the use of hypertext, a software viewer can easily parse the parenthetical information and analyze it grammatically to determine whether the information is in the format of “(‘term’ is defined in § x.)”. If the term is defined in another section, then the software can provide a link or a “quick info” to that part of the definition section, much like the hypertext solution. The distinct advantage in using both the parenthetical solution and the hypertext solution, over just using the hypertext solution, is that the statute is much more human readable without the hypertext tags.

Likewise, the parenthetical solution is compatible with the definition scope solution. A program can first insert and reorganize the definitions, then insert parenthetical information everywhere the statute uses the definitions. Again, combining these two solutions is superior to using just the scope solution. The only problem with the scope solution was that it required an extra lookup; without the parenthetical information, the reader would have had to look for the term of art in the definitions subsection, then the definitions in the chapter, then in the title.

---


153 See *supra* note 149.

154 See *supra* note 133-136 and accompanying text.
However, if the lookup information is provided as parenthetical information, such concerns will be alleviated and the benefits of reorganizing the definitions will be fully realized because the reader of the statute would only need to look at the right definition section. So the combination of these two solutions meets the fourth requirement of no repetitive task or memorization for the reader.

b. Data Structures

One of first tools that a programmer learns about is the data structures, which are ways to arrange and group data. Some examples of data structures are arrays and trees.\textsuperscript{155} A data structure is a grouping of data.\textsuperscript{156} An array is a sequential grouping of data that are identical in type.\textsuperscript{157} A tree is a hierarchical data structure that consists of a set of nodes that start off from the root node and branch out into child nodes. A node can contain a data type.\textsuperscript{158} A programmer can choose which data structure to use based on the task at hand, and one program may utilize any number of different data structures.\textsuperscript{159} Statutes, on the other hand, use one or two types of data structures. Drafters arrange most of the statutes sequentially, sometimes into blocks of sections in chapters.\textsuperscript{160} In programming, this would be like using arrays for every task at hand,


\textsuperscript{156}Id.

\textsuperscript{157}Id.

\textsuperscript{158}Clifford A. Shaffer, \textit{Data Structures and Algorithm Analysis} 145, 195 (2012). An example of trees can be found in Patent Act: any independent claim would be a root of a tree, and any dependent claim would be a child of the claim that it is dependent on. However, any multiple-dependent claims would prevent it from being a tree.

\textsuperscript{159}Gray, \textit{supra} note 155 (under Structures or "structs").

\textsuperscript{160}See, e.g., 17 U.S.C. §§ 100-1332 (2006). The Copyright Act is divided into 13 chapters.
which would be slow, inefficient, and add a lot of accidental complexity in most situations.\footnote{See Clifford A. Shaffer, \textit{Data Structures and Algorithm Analysis} 145 (2012) (“The list representation … have a fundamental limitation: Either search or insert can be made efficient, but not both at the same time.”).} Similarly, with most statutes, much accidental complexity is introduced due to the linear arrangement. If, on the other hand, the drafter arranged the statutes analogous to another data structure that conceptually fits better with the statute, then this arrangement will reduce some of the accidental complexities that results from the structure of the statute.

Using the “tree” data structure in particular can be an effective way to organize sections of statute. A “tree” is a data structure that hierarchical data structure that consists of a set of nodes.\footnote{Id. at 145, 195.} A node is a data or a collection of data that can point or link to another node.\footnote{Id.} The root node, which is the top level of the tree, is a single node that branches out to a different number of child nodes. Each of these child nodes can itself be a parent node and branch out further. Two nodes that share the same parent node are referred to as sibling nodes.\footnote{Id. at 196.} There is no circularity in trees, which means that nodes cannot form a loop. The “tree” structure is used to manipulate hierarchical data as well as making information easy to search.\footnote{An Extensive Examination of Data Structures Using C# 2.0, MSDN (Jan. 15, 2012), http://msdn.microsoft.com/en-us/library/ms379572(v=vs.80).aspx.} The “tree” structure should be familiar to most people who have used a file system of a computer that is arranged in a hierarchy that arrange files using folders and subfolders.\footnote{Subdirectory, WEBOPEDIA, http://www.webopedia.com/TERM/S/subdirectory.html.}
Drafters can readily reorganize statutes into a tree structure on the sub-section level because of the way they are formatted.\textsuperscript{167} Sometimes the different sections on the same level have a different relationship to the parent node; nonetheless, all these different relationships are represented in the same way. For example, 17 U.S.C. § 104 is about the national origin of the copyrighted act, and it has four subsections, (a) through (d). The first two subsections (a) and (b) discuss the two different types of works (published and unpublished), while subsections (c) and (d) discuss the effect of international treaties (the Berne Convention and the Phonogram Treaties). The relationship between (a) and (b) to the parent, § 104, is different from the relationship between (c) and (d) to the parent. In other words, the difference in relationship is not apparent from the structure. The following reorganization demonstrates how a restructuring can make the difference in relationship obvious.

<table>
<thead>
<tr>
<th>§ 104 Subject Matter of Copyright: National Origin</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a) Types of works</td>
</tr>
<tr>
<td>(1) Published</td>
</tr>
<tr>
<td>(2) Unpublished</td>
</tr>
<tr>
<td>(b) International Treaties</td>
</tr>
<tr>
<td>(1) Berne Convention</td>
</tr>
<tr>
<td>(2) Phonogram Treaties</td>
</tr>
</tbody>
</table>

In a small section such as this one, failure to use tree structure may not be a source of much ambiguity. However, for bigger sections, such as 17 U.S.C. § 108, such restructuring can add more clarity. Grouping sub-sections that are related as siblings will allow the reader to quickly understand the purpose of all the subsections by reading just one subsection.\textsuperscript{168} Also, searching through the subsections can become a lot easier; each subsection can contain a short

\textsuperscript{167} See, \textit{e.g.}, 17 U.S.C. § 108 (2006). The section, like many others, is organized as a hierarchical tree structure.

\textsuperscript{168} See \textit{supra} note 152 for a description of sibling nodes.
typographically denoted descriptive phrase. For example, the 17 U.S.C. § 108 may look like this when the subsection headings are bolded:

§ 104 Subject Matter of Copyright: National Origin
(c) Types of works
   (3) Published
   (4) Unpublished
(d) International Treaties
   (3) Berne Convention
   (4) Phonogram Treaties

This way of reorganizing the subsections meets all the requirements set out in Part IV. First, this solution does not add to the overall accidental complexity or ambiguity. The gains in clarity from restructuring offset any resulting complexity or ambiguity. Admittedly, this restructuring may in some cases introduce new conceptual links, as was the case in the new 35 U.S.C. 102.\textsuperscript{169} However, drafters may exercise discretion in utilizing the tree structure as to ensure that no new conceptual links are introduced in the process. For the second requirement, subsection restructuring does not require the drafter to have much programming knowledge or skill because it is easy to understand and analogous to hierarchical file systems that are commonly used in today’s computers. Thirdly, the final statute does not require the reader to have any programming. The reader does not need to know anything about the tree structure to read the statutes. Also, unlike the hypertext solution, no viewer software is required to view the changes. Finally, this solution does not require additional repetitive tasks or archival knowledge; it arguably lowers the amount of knowledge required because it facilitates a quicker grasp of sibling nodes just by reading one of the child nodes.

Another use of the “tree” data structure is to restructure the section numbers themselves. Because the “tree” structure is useful for hierarchical representation, it can readily represent the patent application sections of the Patent Act, 35 U.S.C. chapter 11. The first section, § 111

\textsuperscript{169} See supra note 156.
describes the required elements of a patent application, while some of the following sections describe the individual requirements in general. Drafters can then renumber these following sections so that this inherent hierarchical structure can be evident from the section numbers themselves. For example, a drafter can separate the elements of an application from rest of the requirements, and order them alphabetically. Then, the drafter can renumber the later statutes that correspond to those elements to better represent the tree structure. For example, 35 U.S.C. § 112 can be renamed to § 111A.

Drafters can perform similar section level restructuring in the Copyright Act. For example, 17 U.S.C. § 106 lists the exclusive rights in copyrighted works and § 107 through § 122 lists the limitation on those rights. Some of the limitation sections refer to one specific subsection or to multiple subsections of § 106; this means there is an inherent hierarchy among these sections that is not apparent from the section number. Drafters can readily renumber these limitation sections restructure them to make the hierarchy clearer. For example, § 109 is a limitation on § 106 (3). This structure would be evident by relabeling the subsections of § 106 into alphabets and renaming §109 to §106C.\footnote{Of course then §106A would be inconsistently numbered.}

Such section level restructuring would work well in limited situations where there is a clear intra level tree hierarchy and each node has one parent node. Unfortunately, neither example with the Patent Act and the Copyright Act above strictly follow the tree structure. In the Copyright Act, for example, some of the limitation sections limit more than one of the subsections of 17 U.S.C. §106. For example, §110 is a limitation of rights to performances and displays of copyrighted works which are covered under subsections §106(4)-(6). In the Patent Act, 35 U.S.C. §112 covers specifications that are used in both the regular patent application and
in the provisional application. Since a node in a tree cannot have more than one parent, a drafter cannot use this data structure in this instance. Having a different section numbering convention would then lower the accidental complexity and ambiguity in chapter 1 of the Copyright Act or in chapter 11 of the Patent Act.

Drafters can use other programming data structures to solve this problem, such as a directed acyclic graph (“DAG”). Graphs are a set of nodes and edges. An edge connects two nodes. A direct edge denotes that one node points to the other node. An acyclic graph means that there are no cycles: nodes do not form a loop. A directed acyclic graph, then, is a graph where there are no loops and all the edges are directed. Unlike the tree structure, DAG does not store the hierarchical information. Also, such a structure would be unfamiliar to both the drafter and the reader, and may require too much programming knowledge to be useful. Using an overly complex data structure that not a lot of people are familiar with would increase the accidental complexity and ambiguity, and thus not meet the first three requirements from Part IV. So while drafters can utilize the “tree” data structure to restructure and rename sections and some subsections in order to better convey the relationship between different sections and subsections, more complex structures have more limited uses due to their complexity.

c. Integrated Development Environments

172 See supra note ___ for a description of trees.
173 Shaffer, supra note 161, at 373.
174 Id.
175 See id.
176 See supra Part III.a-c
Most Integrated Development Environments (IDEs) have many features that can aid the program to write code quickly and clearly.\textsuperscript{177} One feature of IDEs that this note already examined was the typography of variables and keywords.\textsuperscript{178} IDEs also provide other features or conveniences that can aide statutory drafting.

One feature that some of the IDEs have is indicating incorrect syntax of a given code.\textsuperscript{179} Programming languages are simpler than natural languages.\textsuperscript{180} It is easier to determine whether a code fragment is syntactically correct than it is to determine whether a natural language is grammatically correct.\textsuperscript{181} Similarly, symbolic logic has very strict syntactic rules; a reader can easily detect any syntactic errors. Because of this parallel between symbolic logic and programming languages, it would be possible to build an IDE for symbolic logic that can detect syntactic errors such as checking for grammatically complete propositions.\textsuperscript{182}

Another feature that drafters can adapt from IDEs is the use of the debugger and test cases to check that a piece of statute covers all different possibilities.\textsuperscript{183} A drafter can eliminate some uncertainties if something similar to a debugger allows the drafter to step through the symbolic logic and verify each outcome of a test case. For example, the new America Invents

---

\textsuperscript{177} See supra note 68 for a description of IDEs.

\textsuperscript{178} See supra Part V.a

\textsuperscript{179} Technically, the compiler actually determines whether there are any syntactic errors. However, in most IDEs, the compiler and the IDE works hand in hand to find the errors and to indicate where it is in the code.

\textsuperscript{180} See supra note 64.

\textsuperscript{181} See Kokol et al., supra note 64, at 1297.

\textsuperscript{182} See Gray supra note 12, at 441.

\textsuperscript{183} A “debugger” works with the compiler and the IDE to run the program with a test value given by the programmer and allows him to step through the code line by line while examining the value of variables to ensure that each line is doing what is expected.

A “test case” is a set of input values to the program that will result in a predictable outcome. The programmer can use a set of test cases to check that his program is working correctly for wide range of values.
Act, 35 U.S.C. § 102, changes the way that the Patent and Trademark Office handles prior art and exceptions. Some commentators in the patent field have commented that this expanded exception adds a source of huge uncertainty to the Patent Act. The drafter can specify a variety of test cases so that he tests all the different combinations of filing dates and disclosure dates. He can then check the outcome and see if it conforms to the intent with which the statute was drafted.

One final aid that is used in some IDEs is the syntactic tree visualizer. Symbolic logic entails using a set of symbols a set of syntactic terms (such as AND, and OR) to create a syntax tree. For example: P OR Q would be equivalent to a tree with OR as the parent node, and P and Q as its child nodes. This syntax tree can be visualized like any other tree. Such visualization can aid the drafters analyze the logic of the normalized statute. Some IDEs can generate higher level flowcharts that visually represent the flow of the program. Symbolic logic itself is a representation of the flow of the statute, and so adding a visual element to it would aid the drafter in working through each step of the logic.

All of these IDE improvements have some common pitfalls, in that they would require the drafter to possess a higher amount of familiarity with programming skills and practices. However, since each of these IDE tools are built on the expectation that the drafter is using the symbolic logic, it may be reasonable to assume that such a drafter would be more familiar with

188 See Allen & Engholm supra note 2, at 3816 (the diagrams are parts of a flowchart.).
programming. Also, since most drafters are legislative staffers, perhaps programmers could be recruited for the more programming intensive tasks. Furthermore, the resulting statute would not require the general public to possess any programming skills. However, this again highlights the weakness of using symbolic logic in that it would require a translation.\footnote{See supra note 42.} No matter how many syntactic ambiguities the use of symbolic logic and IDE tools can eliminate, there is a potential that the translation process may introduce other syntactic ambiguities. Finally, as for the requirement regarding repetitive task or archival knowledge, the drafter who utilizes the debugger will need to test the same section of code repeatedly. While such a task may seem repetitive, debugging can lead to less semantic ambiguity. Overall, ambiguities for the reader can be reduced at the cost of accidental complexity paid by the drafter. This is a good tradeoff since the statute is read more often than it is written and since the use of these solutions assumes that the drafter has some knowledge in symbolic logic and programming.

In conclusion, drafters can use some of the tools and concepts that programmers use in programming, such as variables, data structures, and IDEs, to reduce accidental complexity and ambiguity in statutory drafting. However, there were some limitations to each of them. As always, the right tools must be used for the right problems.

VI. The Methods

Another set of solutions to poor statutory drafting are analogous to programming language paradigms: procedural programming and object oriented programming. The procedural paradigm contemplates programs as processes that organize sections of code in “functions.”\footnote{Gray, supra note 155 (under “High level languages”). See note 68 for a description on functions.}
Alternatively object oriented programming paradigm contemplates programs as a collection of objects and organizes sections of code under that object.\textsuperscript{191} “Objects” and “classes” are a collection of data types such as variables, other objects or functions. They can be used to represent a real world object or more abstract ones.\textsuperscript{192} Each paradigm modularizes the code in different ways; the first in “functions” and the second in “objects.” While previous works have suggested that “some model statutes are modular to some extent because they set forth alternate provisions, building blocks from among which enacting jurisdictions may choose,”\textsuperscript{193} drafters have underutilized the modularity concept in many other statutes. First, this note will more closely examine the concept of modularity the following section. Then the section after that will examine the two programming paradigms in order to see the different ways of modularization.

\textbf{a. Modularity}

A module is broadly defined as “a … functional assembly … for use with other such assemblies.”\textsuperscript{194} Legal modularity is defined as “the practice of creating a legal document by selecting and cobbling together terms from a source compendium or different sources.”\textsuperscript{195} This definition is generally accurate in describing the actual process. Modularity in computer

\begin{footnotesize}
\begin{enumerate}
\item \cite{191} For a brief comparison of these two paradigms, see Object-Oriented Programming, MSDN (Jan 15, 2010), http://msdn.microsoft.com/en-us/library/11ekc77d(v=VS.80).aspx.
\item \cite{192} Gray, supra note 155 (under “classes”).
\item \cite{194} Module, Merriam-Webster Dictionary (Jan. 15, 2012), http://www.merriam-webster.com/dictionary/module.
\item \cite{195} Radin, supra note 193.
\end{enumerate}
\end{footnotesize}
programming entails hiding of information from one part of the code to another to minimize the connection between sections of code.\textsuperscript{196}

There are many benefits to modularization. Because of the hiding of information, modularization techniques can organize sections of code into coherent sections. Cross references, which are huge sources of ambiguity, are minimized because information in one section is hidden from another section and cannot be accessed. In programming, systems of modules are easier to visualize and design.\textsuperscript{197} Systems are easier to change because programmers can swap different modules in and out.\textsuperscript{198} “The degree of … modularity determines the adaptability and changeability of the program.”\textsuperscript{199} Perhaps the biggest advantage of modularity is that it promotes reusability; programmers can reuse different sections of code in different places.

Some questions remain as to how the modules are assembled. What are some of the guiding principles in selection and cobbling together the modules? Which type of information is hidden? Other questions remain as to how modularization can work in the context of statutory drafting. How is the hiding of information enforced in a printed media, or even during the drafting process? The complexities of statutes and other legal documents stem from the fact that all parts of the statute are available to be referenced from anywhere else on the statute. While the availability of “more information is usually considered better” in law, such a huge amount of flexibility can potentially result in a greater amount of complexity.\textsuperscript{200}

\begin{flushleft}

\textsuperscript{197} See generally, MARTIN FOWLER, UML DISTILLED 1 (3rd. ed 2004) (explaining the use of UML as a way to describe and design software systems.)

\textsuperscript{198} BROOKS, supra note 70, at 118.

\textsuperscript{199} Id. at 143.

\textsuperscript{200} Smith, supra note 120, at 1178.
\end{flushleft}
The next two subsections will consider two different ways of modularity that drafters can utilize in statutory drafting. As with the tools in Part V, each of these paradigms will be held to the same requirements of reducing overall complexity and ambiguity, not requiring too much knowledge of programming skill from the drafters or from the readers, and not requiring any repetitive tasks or archival knowledge by the drafters or the readers.

b. Procedural Programming

In the procedural programming paradigm, modularity is centered on functions which are also called procedures. Functions have predefined inputs and outputs and perform one specialized task. Functions can be nested, in that they can call other functions, even themselves. The benefits of using functions is that they can be reused, therefore lowering the cost of maintenance and debugging, the time and resources required to fix or to update a program. In statutory drafting, some of the statutes are written in a way that is analogous to functions. For example, 35 U.S.C. § 151 is similar to a function in that the Patent and Trademark office “give[s] or mail[s]” the notice to the applicant.

Modularization hides information. In procedural programming, each function hides from its caller the details of how that function performs its task. For example, many sections of the Patent Act can reference 35 U.S.C. § 151, which is about issuing of patent. The sections that reference it do not need to contain any information about the details on how the patent is granted to the applicant, but only that the applicant somehow receives the notice of allowance. All the details of how this notice is sent is hidden within § 151. For example, there are details about the issuance regarding payment of fees that do not necessarily need to be known by other sections.

\footnote{Gray, \textit{supra} note 155 (under “Simple Programs”).}
The “interface,” in general, is ways in which objects communicate with each other. The interface between sections, then, should be very limited so that a thorough knowledge of the other section is not required to learn the new one. For example, if section A refers to section B, then the reader of section A should only need to know in general what section B does without having to know specific details of how section B accomplishes its task.

Drafters can use some of the tools from Part V to enforce modularity in statutes. Specifically, Part V.a presented variables as a way to restrict the scope of the variables to subsections, sections and chapters. Drafters can use similar tools to hide information within a section by exposing only the part of the interface that the caller section is required to know.

c. Object Orient Programming

Another major programming paradigm is Object Oriented Programming (OOP). Some basic understanding is OOP is required to discuss the features of the paradigm that may aid statutory drafting. Instead of focusing on what needs to be done (verbs), OOP focuses on objects (nouns) that represent real-world counterparts. For example, suppose you were to write a program about commuting from your house to your workplace. In procedural programming, a programmer might write a function called “Commute.” However, with OOP, a programmer might write a class called “Car” or “Vehicle.”

One of the concepts of OOP is encapsulation, which entails hiding the components of the class from other classes. From any scope outside the class, a programmer cannot use components designated as “private”, while the programmer can use components designated as “public.” This restriction reduces interdependence between classes and components, which in turn decreases overall complexity. This hiding of components increases modularity, because it

---

prevents the programmer from writing the “spaghetti” code which results from his being able to
call any function and use any data from anywhere in the code. Poorly written “spaghetti” code
and a statute cluttered with cross references are conceptually similar. They are both caused by
the fact that any section can refer to any other section. By enforcing the idea of encapsulation on
statutory drafting, drafters can avoid much of this “spaghetti” statutory drafting. Because the
current statutory drafting has no hiding of information, there is no exploitation of modularity (as
through encapsulation in OOP), which results in a mess.

Another important concept in OOP is inheritance. This is when a statute defines a class
(“child” or “subclass”) based upon another class (“parent” or “superclass”). In Java, a
commonly used programming language, the “protected” keyword indicates that those
components can be used in the subclasses. A key feature of inheritance is overloading methods.
The “hub-and-spoke” model used in the Texas Commercial Code is similar to this concept of
inheritance.

The benefit of using inheritance is that it promotes modularity and code reuse. For
example, a base class contains functions that can only be used from itself or its descendents.
This is another way of limiting or hiding information, which again is crucial to effective use of
modularity.

Finally, in Object Oriented Programming, there is the concept of polymorphism, where
one type can be treated as another class. In Java, a class can be defined to implement multiple
“interfaces”. This allows polymorphism in that an instance of such a class can be considered as a

---

203 See, for example, 17 U.S.C. § 118 (2006) which contains numerous cross references, which in turn cross reference many other sections, etc.


205 Blackwell, supra note 13, 250.

type of that interface. Again, the purpose of looking at this particular concept of OOP is because it is another way of hiding information, which is crucial to effective use of modularity.

In programming, the compilers enforces all these information hiding. A compiler is a tool that takes the human readable source code and converts it into another code, such as the object code which the computer can read and execute. The compiler checks whether each reference to a variable, function, or class is accessible from that particular level, and warns the programmer if it finds any problems. When symbolic logic or some hybrid language is used instead of a human language, such compilers can be built as well. However, if a hybrid language is not used, all the information hiding will have to be enforced by the drafter, which would represent a significant source of accidental complexity to the drafter, because checking all the different constraints is a repetitive task that is ideally suited for a computer. All of the burden will be placed on the drafter to learn the concepts of programming and to apply them in statutory drafting. However, the reader will not need to know any of the programming concepts because the final statute will be better organization, less use of cross-references, and less need for knowledge of other sections of the statute.

VII. Conclusion

This note has examined the problem of poor statutory drafting, why the problem should be solved and some of the common characteristics such as legalese, ambiguity, and poor conceptual organization. Then this note examined the parallels between statutory drafting and software development, between the problems and the solutions, as a motivation to use other software development methods. Another key motivation for using software development methods in statutory drafting was that programming languages are unambiguous. However,
because there are major differences between natural languages and programming languages, some software development methods will work better than others for statutory drafting. In order to measure which methods are effective, this note then laid out some of the requirements for software development methods. Using these requirements, this note examined some of the software development tools and methods, and suggested the ways in which statutory drafting can be improved.