April 20, 2000

Software Engineering for Startups

Todd Sedano, Carnegie Mellon University

Available at: https://works.bepress.com/todd_sedano/3/
After programming in the large, what can we apply to smaller projects?
The Department of Defense began Software Engineering as a discipline to curb the number of software project failures and learn how to build software better. By the nature of the Department of Defense’s call to arms, research has focused on large software projects where the correction of bad practices has the greatest yield. Small software projects receive less attention. Practices that don’t scale to large projects still get the job done for small projects. The effects of bad development style can be corrected by heroic efforts. When the company grows and the team sizes increases, so does the need for more heroes.

My interests include finding Effective Processes and learning which processes are best suited for startups. The nature of small business, the agility to change direction quickly, allows it to handle minor bad software practices better. However, failure for a startup may crater the company where as a large company may recover from the fumble. Due to the fierce competition for a startup with limited resources, we should consider tasks that reduce the time to market.

A startup strives to create equity
The purpose of a startup is not to realize an idea, but to realize an opportunity. There are plenty of good ideas, but only a few opportunities exist. Inventing a better mousetrap is not the key. It’s hard to find an opportunity that has the correct combination of idea, market window, lack of competitors, an experienced executive team, barriers to entry for competitors, and an understanding of the customers’ need. A startup must create a useful product within the shortest timeframe possible in order to be the first or second company in a well-defined market niche.

Although most startups fail because of poor business decisions, we can still look at their software development to understand how we can make it better. While on a startup project, the schedules are so tight; it’s hard to find the time to do reflective thinking on how to optimize the development. If developers can make a better product in the same timeframe or make the same product sooner, then we have helped create equity.

Code Reviews – an overlooked lifesaver
The argument in favor of code reviews is pretty simple. During a code review, when a bug is found, it’s location and cause is known at discovery. When it’s removed early in the development, integration and testing will be easier. During testing, when a bug is found, it’s location and cause is not known. Especially in multi-threaded environments, a bug can be very tedious to locate.

One might think that promoting code reviews of critical components, the core engine or kernel, would be an easy sell. However, some developers and managers are so focused on producing code and the schedule they downplay the benefit for an activity they may
never have tried. Software engineering research has been clear, the benefit of code reviews out ways the overhead.

Acceptance of code reviews very. On some projects, developers review code for the core engine. On other projects, developers never review code.

*Review all critical code, peer review all other code*

**Running aground while traveling without a project plan**

One mode of operation involves using a greedy algorithm to schedule tasks. What needs to be done next? Developers follow their gut instinct on the ordering of task and identifying what is on the critical path.

Unfortunately, using this method is only as good as the instincts. The critical path can easily be misidentified. There’s little point in preparing for a website launch in six weeks if it takes two months to order and configure a production stack. (A website for a San Francisco company prepared to launch on its staging stack since its production stack wouldn’t be ready.)

It’s easy to miss key factors that are necessary later on. It’s easy to miss beneficial tasks such as looking for reuse or designing for reuse. Business plays an important role in determining the plan and schedule. After creating a plan, it’s necessary to periodically update the plan. We’ve all seen companies that couldn’t deliver what they promise. The sooner a company can identify that, the easier it is to reset the customer’s expectations. Finding the middle ground between not having a plan and creating a plan that is too detailed is key.

*Aim to be like Hannibal, “I love it when a plan comes together.”*

**Task estimation – Looking beyond function points.**

Research into estimation focused on creating better estimates for the project plan and schedule. Due to a lack of historical data and because each project tends to be a one-off project, this can be difficult for a startup. Several startups that I’ve seen have built a system with many similar components. By spending the overhead to track time for the first component, they then predicted the end date for all components. The usefulness of that knowledge must be compared with the overhead. Executive teams find this a reasonable cost.

*Determine what you need estimates for.*

*Focus is on business needs not software process analysis.*

---

Incremental development – minimizing the risk of software integration

The market pushes companies toward incremental releases to get the first version product out early. If we can apply incremental releases to a product, we can also apply incremental builds to each release. The purpose is to get the core engine functional and bolt on additional components with each build. Once the core engine works, we can test a thread through the entire system. At that point, we know we can actually deliver a product that’s not vaporware; it simply is a matter of time before all the features are in place. This reduces integration risk between subsystems in a form of incremental integration. This allows us to do performance testing on the core engine in the middle of a project when there is still time to fix major issues.

First develop a working system, and then make it useful, and then make it faster.

Do we put Problem Frames on milk cartons?

I haven’t seen anyone using problem frames. If you have, please let me know. I’ll predict that it’ll be useful for developers involved in multiple domains. This aids the developer to understand each new domain better and to learn what questions to ask when approaching a new domain. I see this more applicable to consultants than for the average developer.

You determine the usefulness of Use Cases

You might think that use cases are easy, right? But I’ve seen this done with varying degrees of success (or failure.) On one project, we wrote use cases for requirements that did document the user interaction with the system well. The use case document was so successful, that we built a checklist for future documents from it. However, in some places, we left off exception handling details! On another project, the use case became so detailed in scope that it approached a design document classification. For example, a login use case was decomposed into five more use cases.

When doing requirements with Use Cases, focus the work to the right scope.

Design for release 1.0 when building release 0.5

One typical strategy to improve time to market is to divide the product into multiple releases with increasing feature set. Unfortunately, under tight deadlines, developers can act near-sighted by making decisions biased towards writing the least amount of code, which later won’t support the next feature set.

One large startup making internet applications in Foster City was transitioning from version 1.0 to version 2.0 and had to rewrite one of their components from scratch – the web server. This had become necessary because the 1.0 version did not support the functionality need by the 2.0 system and was poorly documented. This wasted an expensive developer’s time. This reset the maturity clock for that component. The server in 1.0 had already gone through QA, was debugged, and was working in a large fielded system.
Determine the right depth or context. Designing at a too high level produces ambiguity and leads to creativity in implementation phase. Designing at a low level tends to produce over design and create inconsistencies in the project.

*Be aware of the next release during design;*
*Encapsulate properties, algorithms, and data that is likely to change*

**Treating UML as a toolset, not as a hammer**
UML provides a collection of useful notations and a methodology. The usefulness of each notation depends upon the problem and what you need to communicate. Developers can go overboard and attempt to use every notation. Don’t overdue UML. Internet companies experimenting with UML suffer from this ailment more than startups.

*Determine which UML diagrams are useful.*
*Focus on what you are trying to communicate*

**Data Collection – Calculating wind velocity of a tornado**
PSP and TSP place an emphasis on data collection and analysis to determine the utility of changes in your personal process or your team’s process. Building such a discipline in conducive environments is tough, now try the startup world!

Data collection after a build is commonplace in calculating the number of bugs and the severity of bugs.

*Determine which metrics bring you value and emphasize those.*

**Risk Mitigation, Contingency planning – Lack of risk analysis is a risk**
Have you ever been on a perfect project? I haven’t. Because projects come with problems, the prudent team would try to understand them and maybe spend the time to reduce them. Some startups don’t do risk mitigation because the team is too busy. Further research is required into the overhead of executing risk mitigation compared to it benefit.

**Developers operating like sheep in the field of corporate culture**
After visiting enough cubicles and meeting with enough developers, we’ll still find reason for surprise. See if you can recognize these three observations. 1) In many software cultures, developers don’t like to say, “I don’t know.” 2) In some software cultures, developers don’t ask questions. They will make no headway on a project even though the person across from them knows the answer. They will even implement an architecture even if they don’t understand it. 3) In some software cultures, developers will hide problems. Admitting that there exists a problem implies ownership. In this environment, leads will need to routinely elicit the problems, so that they can be fixed.

Fortunately, there is a remedy for this. The executive team exudes a company’s culture.
Conclusion
For each aspect of software development covered in this paper, I’ve recorded its current use from a random, non-scientific sampling of startups. Predictions about future use assume short-sighted application. Further research should quantify the current use of each aspect of software development.

| Current practice and predicted use of software engineering in startups |
|---------------------------------|-----------------|-----------------|
| **Observed in practice**        | **Future use**   |                  |
| Code reviews                    | Frequently used | Very practical  |
| Project plans                   | Sometimes used  | Very practical  |
| Task duration estimation        | Sometimes used  | Good with similar components |
| Incremental development         | Sometimes used  | Very practical  |
| Problem frames                  | Not used        | Depends on company |
| Use cases                       | Sometimes used  | Unknown          |
| UML                             | Sometimes used  | Very practical  |
| Data collection                 | Not used, except for bug tracking | Not used |
| Risk mitigation                 | Sometimes used  | Very practical  |

Software engineering examines the problems of programming in the large in order to find mechanisms to avoid project failures and reduce development costs. Developers tend to see many of the mechanisms as unnecessary for small projects. The fun is to distinguish the context where each mechanism is appropriate and effective.

Knowledge gained in large software organizations may not be communicated to startups. Although people do move from large corporations to small startups, within the Silicon Valley, startups tend to have a younger workforce working on smaller sized projects. College programs focus on computer science fundamentals, not on how to complete a project that takes weeks to build. We need to continually identify which software practices’ benefits outweigh the overhead for small projects.

For any software practice to be successful, it needs the support of the executive team. If the company’s focus is solely to put a product out onto the market quickly, developers soon drop better practices in favor of shipping a product. The quality of the first release may be sufficient, but in the near term (six to twelve months) development effort will be wasted fixing problems that would have been easier to fix while still in design. The company reflects the attitude of the executive team.

Ironically, for most startups, it will be the decisions of the executive team that determines whether or not a startup succeeds, not the efforts of the developers.