From: Steven Sinofsky <[redacted]>
To: Jeffrey Epstein <[redacted]>
Subject: Fw: [New post] Learning by slipping
Date: Tue, 14 May 2013 02:08:22 +0000
Importance: Normal

Sent from Surface RT
Check out [redacted]

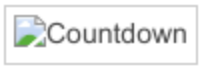
From: Learning by Shipping
Sent: Thursday, May 2, 2013 2:14 AM
To: [redacted]

Respond to this post by replying above this line

New post on **Learning by Shipping**

Learning by slipping

by Steven Sinofsky



"Slipping" or missing the intended completion or milestone date of software projects is as old as software itself. There's a rich history of our industry tracking intended v. actual ship dates and speculating as to the length of the slip and the cause. Even with all this history, slipping is a complex and nuanced topic worth a bit of discussion about slipping as an engineering concept.

Slipping

I've certainly had my fair share of experience slipping. Projects I've worked on have run the full spectrum from landing exactly on time to slipping 20-30% from the original date. There's never a nice or positive way to look at slipping since as an engineer you're only as good as your word. So you can bet the end of every project includes a healthy amount of introspection about the slip.

Big software projects are pretty unique. The biggest challenge is that large scale projects are rarely "repeated" so the ability to get better through iteration keeping some things constant is limited. This is different than building a bridge or a road where many of the steps and processes can be improvements from previous projects. In large scale software you rarely do the same thing with the same approach a second or third time.

While software is everywhere, software engineering is still a very young discipline that rapidly changes. The tools and techniques are wildly different today than they were just a few years ago. Whether you think about the languages, the operating systems, or the user experience so much of what is new software today is architected and implemented in totally new ways.

Whenever one talks about slipping, at some basic level there is a target date and a reality and slipping just means that the two are not the same (Note: I've yet to see a software project truly finish early). There's so much more to slipping than that.

What's a ship date

In order to slip you need to know the ship date. For many large scale projects the actual date is speculation and of course there are complexities such as the release date and the availability date to "confuse" people. This means that discussions about slipping might themselves be built on a foundation of speculation.

The first order of business is that a ship date is in fact a single date. When people talk about projects shipping "first quarter" that is about 90 different dates and so that leaves everyone (on the team and elsewhere) guessing what the ship date might be. A date is a date. All projects should have a date. While software itself is not launching to hit a Mars orbit, it is important that everyone agree on a single date. Whether that date is public or not is a different question.

In the world of continuously shipping, there's even more of a challenge in understanding a slip. Some argue that "shipping" itself is not really a concept as code flows to servers all the time. In reality, the developers on the team are working to a date—they know that one day they come to work and their code is live which is a decidedly different state than the day before. That is shipping.

Interestingly, the error rate in short-term, continuous projects can often (in my experience) be much higher. The view of continuously shipping can lead to a "project" lasting only a month or two. The brain doesn't think much of missing by a week or two, but that can be a 25 – 50% error rate. On a 12 month project that can mean it would stretch to 15-18 months, which does sound like a disaster.

There's nothing about having a ship date that says it needs to be far off. Everything about having a date and hitting it or slipping can apply to an 8 week sprint or a 3 year trek. Small errors are a bigger part of a short project but small errors can be amplified over a long schedule. Slipping is a potential reality regardless of the length of the schedule.

The key thing from the team's perspective about a ship date is that there is one and everyone agrees. The date is supported by the evidence of a plan, specifications, and the tools and resources to support the plan. As with almost all of engineering, errors early in the process get magnified as time goes by. So if the schedule is not believable or credible up front, things will only get worse.

On the other hand, a powerful tool for the team is everyone working towards this date. This is especially true for collaboration across multiple parts of the team or across different teams in a very large organization. When everyone has the same date in mind then everyone is doing the same sorts of work at the same time, making the same sorts of choices, using the same sorts of criteria. Agreeing on a ship date is one of the most potent cross-group collaboration tools I know.

Reasons to slip

Even with a great plan, a team on the same page, and a well-known date, stuff can happen. When stuff happens, the schedule pressure grows. What are some of the reasons for slipping?

- **Too much work, aka "we picked too much stuff".** The most common reason for slipping is that the team signed up to do more work than could be done. The most obvious solution is to do less stuff. In practice it is almost impossible to do less once you start (have you ever tried to cut the budget on a kitchen remodel once it starts? You cut and cut and end up saving no money but costing a lot of time.) The challenge is the inter-connected nature of work. You might want to cut a feature, but more often than not it connected to another feature either upstream or downstream.
- **Some stuff isn't working, aka "we picked the wrong architecture".** This causal factor comes from realizing that the approach that is halfway done just won't work, but to redo things will take more time than is available. Most architecturally oriented developers in this position point to a lack of time up front thinking about the best approach. More agile minded developers assume this is a normal part of "throw away the first version" for implementing new areas. In all cases, there's not much you can do but stick with what you have or spend the time you don't have (i.e. slipping).
- **Didn't know what you know now, aka "we picked the wrong stuff".** No matter how long or short a project, you're learning along the way. You're learning about how good your ideas were or what your competitors are doing, for example. Sometimes that learning tells you that what you're doing just won't fly. The implications for this can run from minimal (if the area is not key) to fairly significant (if the area is a core part of the value). The result in the latter case can be a big impact on the date.
- **Change management, aka "we changed too much stuff".** As the project moves forward, things are changing from the initial plans. Features are being added or removed or reworked, for example. This is all normal and expected. But at some point you can get into a position where there's simply been too much change and the time to get to a known or pre-determined is more than the available time.

The specifics of any slip can also be a combination of these and it should be clear how these are all interrelated. In practice, once the project is not on a schedule all of these reasons for slipping begin to surface. Pretty soon it just looks like there's too much stuff, too much is changing, and too many things aren't "right".

That is the nature of slipping. It is no one single thing or one part of a project. The interrelationships across people, code, and goals mean that a slip is almost always a systemic problem. Recognizing the nature of slipping leads to a better understanding of project realities.

Reality

In reality, slips are what they are and you just have to deal with them. In software, as in most other forms of engineering, once you get in the position of missing your date things get pretty deterministic pretty quickly.

In the collective memories of most large projects that slipped are the heroes or heroic work that saved a project. That could very well happen and does, but from a reliable or repeatable engineering perspective these events are circumstantial and hard to reproduce project over project. Thus the reality of slipping is that you just have to deal with it.

The most famous description of project scheduling comes from Frederic P. Brooks who authored "The Mythical Man-Month" in 1975. While his domain was the mainframe, the ideas and even the metrics are just as relevant almost 40 years later. His most famous aphorism about trying to solve a late project by adding resources is:

> *When a task cannot be partitioned because of sequential constraints, the application of more effort has no effect on schedule. The bearing of a child takes nine months, no matter how many women are assigned.*

Software projects are generally poorly partitioned engineering – much like doing a remodel in a tiny place you just can't have all the different contractors in a tiny place.

There are phases and parts of a project in large scale software that are very amenable to scale with more resources, particularly in testing and code coverage work, for example. Adding resources to make code changes runs right up against the classic man-month reality. Most experienced folks refer to this as "physics" implying that these are relatively immutable laws. Of course as with everything we do, context matters (unlike physics) and so there are ways to make things work and that's where experience in management and most importantly experience as a team working together on the code matters.

The triad of software projects can be thought of as features, quality, and schedule. At any given point you're just trading off against each of those. But if it were only that easy.

Usually it is easy to add features at the start, unaware of precisely how much the schedule or quality will be impacted. Conversely, changing features at other times becomes increasingly difficult and obviously so. From a product management/program management perspective, this is why feature selection, feature set understanding, and so on is so critical and why this part of the team must be so crisp at the start of a project. In reality, the features of a product are far less adaptable than one might suspect. Products where features planned are not delivered can sometimes feel incomplete or somehow less coherent.

It is almost impossible to ever shorten a schedule. And once you start missing dates there is almost no way to "make up for time". If you have an intermediate step you miss by two weeks, there's a good chance the impact will be more than two weeks by the end of a project. The developers/software engineers of a project are where managing this work is so critical. Their estimates of how long things will take and dependencies across the system can make or break the understanding of reality.

Quality is the most difficult to manage and why the test leadership is such a critical part of the management structure of any project. Quality is not something you think about at the end of the project nor is it particularly malleable. While a great test manager knows quality is not binary at a global level, he/she knows that much like error bars in physics a little bit of sub-par quality across many parts of the project compounds and leads to a highly problematic, or buggy, product. Quality is not just bugs but also includes scale, performance, reliability, security, and more.

Quality is difficult to manage because it is often where people want to cut corners. A product might work for most cases but the boundary conditions or edge cases show much different results. As we all know, you only get one chance to make a first impression.

On a project of any size there are many moving parts. This leads to the reality that when a project is slipping, it is never one thing—one team, one feature, one discipline. A project that is slipping is a product of all aspects of a project. Views of what is "critical path" will need to be reconciled with reality across the whole project, taking into account many factors. Views from other parts of the organization, the rumor mill, or just opinions of what is holding up the project are highly suspect and often as disruptive to the project as the slip itself. That's why when faced with a slipping project, the role of management and managing through the slip is so critical.

What to do

When faced with a slip, assuming you don't try to toss some features off the side, throw some more resources at the code, or just settle for lower quality there are a few things to work on.

First and foremost, it is important to make sure the team is not spending energy finger pointing. As obvious as that sounds, there's a natural human tendency to avoid having the spotlight at moments like this. One way to accomplish that, improperly, is to shine the light on another part of the project. So the first rule of slipping is "we're all slipping". What to do about that might be localized, but it is a team effort.

What else can be done?

- **Don't move the goalposts (quality, features, architecture).** The first thing to do is to avoid taking drastic actions with hard to measure consequences. Saying you're going to settle for "lower quality" is impossible to measure. Ripping out code that might not work but you understand has a very different risk profile than the "rewrite". For the most part, in the face of slipping the best thing to do is keep the goals the same and move the date to accomplish what you set out to do.
- **Think globally, act locally.** Teams will often take actions that are very local at times of slipping. They look to cut or modify features that don't seem critical to them but have important upstream or downstream impact, sometimes not well understood on a large project. Or feature changes that might seem small can have a big impact on planned positioning, pricing, partnerships, etc. The approach of making sure everyone is checking/double checking on changes is a way to avoid these "surprises".
- **Everyone focuses on being first, not avoiding being last.** When a project has more than a couple of teams contributing and is faced with a tight schedule, there's a tendency for a team to look around to just make sure they are not the team that is worse off. A great leader I once worked with used to take these moments to remind every part of the project to focus on being first rather than focusing on being "not last". That's always good advice, especially when followed in a constructive manner.
- **Be calm, carry on.** Most of all, slipping is painful and even though it is all too common in software, the most important thing to do during crunch time is to remain calm and carry on. No one does good work in a panic and for the most part the quality of decisions and choices degrades if folks are operating under too many constraints that can't get met. It is always bad for business, customers, and the team to slip. But if you are slipping you have to work with what you've got since most of the choices are usually even less desirable.

Managing a software project is one of the more complex engineering endeavors because of the infinite nature of change, complexity of interactions, and even the "art" that still

permeates this relatively new form of engineering. Scheduling is not yet something we have all mastered and slipping is still a part of most large projects. The more that *Software Eats the World* *($)*, the more the challenges of software project management will be part of all product and service development.

Given that, this post tried to outline some of the causes, realities, and actions one could take in the face of learning by slipping.

--Steven Sinofsky

Steven Sinofsky | May 1, 2013 at 10:00 pm | Tags: competition, management, tension | Categories: posts | URL: http://wp.me/p31nkB-7V

Comment See all comments