Extracted from:

Software Estimation Without Guessing

Effective Planning in an Imperfect World

The Pragmatic Bookshelf

Raleigh, North Carolina

The Pragmatic Programmers

Software Estimation Without Guessing

Effective Planning in an Imperfect World

George Dinwiddie
edited by Adaobi Obi Tulton

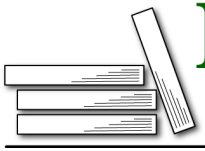
Software Estimation Without Guessing

Effective Planning in an Imperfect World

George Dinwiddie

The Pragmatic Bookshelf

Raleigh, North Carolina



Our Pragmatic books, screencasts, and audio books can help you and your team create better software and have more fun. Visit us at *https://pragprog.com*.

The team that produced this book includes:

Publisher: Andy Hunt
VP of Operations: Janet Furlow
Executive Editor: Dave Rankin
Development Editor: Adaobi Obi Tulton
Copy Editor: Sean Dennis
Indexing: Potomac Indexing, LLC
Layout: Gilson Graphics

For sales, volume licensing, and support, please contact *support@pragprog.com*.

For international rights, please contact *rights@pragprog.com*.

ISBN-13: 978-1-68050-698-3
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—December 2019

Are We Going Fast Enough?

How do we know how fast we *should* be able to go? After all, given the current circumstances, we're going at exactly the right speed. That's a tautology. But is there some small thing that's holding us back, something that's within our control?

When tracking progress over a significant period of time, it's naive to think that the rate of accomplishment will remain constant. The only way that's likely to happen is if someone is managing the data to make it look constant because they think that it will reflect better on them. (See Hiding Reality, on page ?.) Things change. People and teams of people cycle through better times and tougher times. On something as hard to measure as software development, even the measurement is unlikely to remain consistent over time. Expect uncertainty and learn to live with it.

In spite of the uncertainty, you can use this information to guide your exploration of what may be happening. Ask not only how fast have you been going, but how that rate has been changing. Look for trends and patterns. Don't jump to assumptions about the competence, motivation, or individual performance of people. The more likely causes are systemic effects.

What are the components that affect changes in velocity? These lists are surely not complete, but they are some of the common contributors that I've seen.

Slowing Down

When the data tells you that work is slowing down, the first thing you should look for is bottlenecks in the process. Are work items building up at some stage in the process? A Cumulative Flow Diagram, on page ? can make such a buildup obvious. Watching the flow of work items in a visual management tool can reveal bottlenecks just as well, if you look for them.

Also, check if the pressure to deliver is driving teams to leave technical debt in the code:

- The code diverges from clearly expressing the problem domain.
- Small messes get left behind for "someday" when there's time to clean them up.
- Code gets more and more complex, and therefore harder and harder to expand and enhance.
- Duplication grows when there are areas that people are afraid to touch, or can't easily know about as they're writing code.

- Code gets harder to read, and therefore it takes longer to find where to change it and how the change should happen.

One of the key telltales of this situation is a rising bug count. If you have an honest relationship with the programmers, though, they can likely tell you about this. Some programmers may not be aware of techniques of keeping code well-factored with lots of easily rearranged modules instead of a few rigid ones.

As the bug reports come in, at some point effort needs to be spent addressing them. This points out hidden undoneness of the functionality that's been shipped so far. It wasn't really completed as it didn't do everything expected of it. You might not have discovered these deficiencies at the time, so you congratulated yourself on completing the functionality. Now, going back to complete it is taking away from your capacity to implement new functionality. If you're not careful, this can lead to a runaway spiral of less and less completion plus more and more bugs.

Maybe the work environment has become less conducive to working together. There may be increased friction in communicating information and ideas between people. Or unrelated noise is distracting people. Perhaps an increase in the number of people working on the project has increased the number of relationships and communication paths beyond what can be handled. Maybe the communication has become more point-to-point rather than many-to-many among the group.

Could it be that changes in the oversight or evaluation of the group is causing them to do more overhead work at the expense of development? Are there more reports? Is there more fear of blame, resulting in more CYA documentation that eats away at the team capacity? Is micromanagement interrupting the flow of work?

Perhaps you've been doing the easier work to make a good show of progress and have deferred the harder or riskier work until now. Or maybe you're just counting your progress in larger units of work. Maybe you're estimating more optimistically? It could be that your unit of measurement has changed more than your rate of doing work.

Speeding Up

If things seem to be speeding up, perhaps you're getting better at what you're doing. It's relatively rare that you get better at programming in a time short enough to be a noticeable productivity boost, but it's possible. Maybe you've all learned some clever techniques from each other and that's boosted your

output. Or you've become better at working together as a group, gaining synergy from the best skills of each person combining into a group effort.

Sometimes teams start off with a lot of speculative framework development. If they guessed right about their needs, then maybe it's paying off now. Conversely, perhaps they've started postponing the hard work until later, creating a false sense of progress. Could they be taking shortcuts that will later show up as technical debt, unfinished functionality, and bugs?

Or maybe it's an illusion of measurement. It could be the team has gotten better at splitting User Stories, and are counting smaller units of work. Or maybe, having been burned before, they're now estimating more pessimistically. This is especially likely to be true if there is pressure to increase velocity.

Oscillating

If rate of accomplishment seems to be alternately speeding up and slowing down, then it could be that the development team is correcting based on feedback that is delayed. Systems engineering shows that delays in a feedback loop result in a late start to correction and subsequent overcorrection, causing oscillations. The delay in feedback can be external, or it can be created within the team's work system. For example, if the work items take longer than the measurement interval, the feedback on accomplishments gets delayed until the next measurement interval. Reducing the size of the work items will help the data reflect reality more clearly.

It could also be that the division of work is inconsistent in sizing. Or that estimation is haphazard. Either would affect your data measurement. It might also be that there is noise in the collection of the data. Or that frequent perturbations in the team makeup or environment make a steady pace impossible.

None of these patterns are necessarily indicators of a problem, but it's definitely good to notice the variability of velocity if you're projecting it into the future.

Pushing Our Limits

If development speed *is* the critical bottleneck, how can you determine how fast you can go unless you push as hard as you can? How can you be sure the development team is not loafing if you don't keep pushing for more speed?

I was discussing this topic with a colleague, and she described how a runner learns to go further, faster, by using a technique known as interval running. As a beginner to long-distance running, she could improve her time and endurance by running for two minutes and walking for one. As she got in

better shape, she could reduce the length of the walking interval, or replace it with jogging. She emphasized that even the best conditioned long-distance runners do not try to maintain the same speed all the time. Pushing the limit for short periods of time, though, can condition you for faster performance in the future.

In project work, the same principles hold true. People burn out faster when they're pushed to go at top speed all the time. If they're trying to maintain a constant speed, they also don't hit the highest speeds of which they're capable.

Certainly, people can decide to push themselves harder from time to time, usually in response to some triggering event. But they will need recovery time from short pushes.

Short pushes of increased effort can increase your capability. It helps you stretch your abilities. It can't do that, though, if you're in a panic. Learning happens best when it's safe to fail, and when you're thoughtful about what you're doing. You can look at the need for a speedup, talk about ways you can do that, and try it out. If you have some historical data for comparison, you can check whether your attempts to speed up are working or not. This gives you new tools for times that require more speed, and other tools that might help you speed up, on average.

When moving at a steady pace, it's difficult to judge whether or not you are proceeding at a pace that's fast enough, but not too fast. Our ability to judge differences is much better than our ability to judge absolute conditions. We can compare our progress during a slower interval with our progress with a faster one. Which gets more done? Which has fewer problems?

You should be vigilant and sensitive to the potential for burnout or other byproducts of schedule pressure. These can quickly eliminate any advantage of going faster. By the time you notice them, it's usually too late to avoid a significantly long recovery period. Short experiments, starting with *very* short, are the best way to avoid overstressing.

These short experiments shouldn't be restricted to trying to go faster doing the same things. Try some different techniques, too. We mentioned in Balancing Speed and Risk, on page ? that trying shortcuts can increase variability and decrease predictability. If we're trying experiments, then we are seeking decreased predictability intentionally, in the hope that we find other things of value. Some of that value might be increased speed, but you should also be on the lookout for other ways to increase effectiveness.

By observing performance under a variety of conditions, you can tune the pace to maximize overall performance. Maximizing overall performance, however, will never be reached by trying to maintain peak performance. If trying to improve on performance, never focus solely on speed.