

Estimating in Actual Time

Moses M. Hohman, PhD

Center for Functional Genomics, Northwestern University, Evanston, Illinois
Robert H. Lurie Comprehensive Cancer Center, Northwestern University, Chicago, Illinois
mmhohman@northwestern.edu

Abstract

In an effort to improve our understanding of the way we work, my team grouped together ideas from various sources to come up with an estimation scheme that avoided the confusing notion of "ideal" time, used an appropriate level of granularity for the size of the story, and explicitly acknowledged the uncertainty inherent in estimation. In this experience report, I describe the forces that influenced our choices, explain how we estimated, and then present what we learned about estimating this way, both the good and the bad.

1. Introduction

Estimation methods vary. The original XP book recommended estimating in "Ideal Engineering Time" [Beck, 2000]. Some people found the notion of ideal time problematic, so someone came up with story points [Beck and Fowler, 2001]. In the new edition of Extreme Programming Explained, Beck notes that he has come to "prefer to work with real time estimates". [Beck, 2005]

Each method reflects an underlying philosophy of and experience with estimation. In this paper we report on an estimation method based on actual (or real) time and three-sided estimates, derived from DeMarco and Lister's "triangular uncertainty diagrams" [DeMarco and Lister, 2003]. We examine what we've learned using this method, and compare and contrast with the other methods mentioned above.

2. Existing Methods

Existing agile estimation methods have a number of similarities. Developers accept responsibility for their share of the available work rather than being told what to do. As part of this accepted responsibility, developers have the right and the duty to estimate how long implementation will take, rather than being told how long it should take. Agile estimation methods explicitly account for non-programming time (meetings, conference calls, seminars, socializing, etc.), and attempt to measure the percent of time spent on programming to help plan realistic completion dates. Most methods benchmark estimates to previously completed tasks using "Yesterday's Weather", the idea that the time to complete a story today should be close to the time required to complete a similar story "yesterday" [Beck and Fowler, 2001]. Estimation by comparison helps developers estimate more realistically. Finally, project trackers

compare the actual effort spent on a story with the estimate, both to help plan releases and to provide feedback about estimation effectiveness to the team.

First we will define a number of terms used throughout the paper, then we will briefly survey the most popular existing methods used in agile software development.

2.1. Ideal time

In his book, "Extreme Programming Explained", Beck described the concept of "Ideal Engineering Time", the time required to complete a story "with no interruptions or meetings" [Beck, 2000, p. 90]. This idea emphasizes that software developers do other things during the day besides program. They go to meetings, they attend seminars, they socialize with coworkers, they clean their desks, etc. It is difficult for a developer to realistically incorporate all of these other daily tasks into an estimate. So, developers estimate in ideal time.

Project managers and trackers then track the amount of ideal time completed by the team in an iteration to determine the team's velocity. At the end of each week or two-week long iteration trackers simply add up the estimates of the stories completed. To project the team's performance into the future, trackers can take this sum of completed estimates, or "velocity", and multiply by the number of iterations remaining in the release. If "Yesterday's Weather" holds, the velocity for each iteration should remain relatively constant, and trackers should be able to predict how much total work can be completed in the remaining iterations.

Underlying this method is the philosophy that estimating in actual, calendar time is impossible, because accounting for distractions is too difficult. In order to plan in actual time, trackers compute the team's velocity, a ratio between ideal and actual time that roughly tells you how much the team can complete in future iterations and in the release as a whole.

The velocity is an approximate factor. The idea is that too much precision would be both a false representation of the team's ability to predict the future and in fact not really that useful, because inaccuracies usually average out in the long run. A passage from *Planning Extreme Programming* expresses this well:

Let's be clear, estimation is an art at best. You're not going to get accurate figures however hard you try. With a little bit of effort you can, however, get good enough numbers, and you can

get better numbers over time. [Beck and Fowler, 2001, p. 57]

It is better to be open and honest with customers about the level of confidence they should have with your predictions. Developers estimate and trackers track in coarse-grained units, whole days and iterations. Instead of worrying about a half day here and there, the team tries to get things about right, and then manages scope with the customer to meet release dates.

These ideas are important, and our team found them to be very useful for our planning and tracking processes. Indeed, in the early days we tried to use this method to estimate and track our progress. We found, however, that there were a number of difficult details this method did not address.

For example, computing velocity on a per iteration basis and then multiplying by the number of iterations in the release only works if all iterations have the same length. A number of factors influence iteration length, the most important being developer vacation time and time spent on other projects. To account for this, we divided the amount of ideal time completed in an iteration by the number of calendar days and by the number of developers, yielding a velocity of ideal time completed per calendar day per developer.

Further complicating matters, however, most of our programmers worked on more than one project at a time. We tried to handle this by assuming that a given developer, let's call her Cathy, planned to work 50% of her time on project X (say) for the next ten weeks. This is where the method really started to break down for us. There was no way to know whether Cathy was really spending 50% of her time on project X or not using this method, especially because the other 50% of her time was not spent on programming tasks. We did not estimate and track those other tasks, so we could not tell whether the amount of project X work she did each week really represented 50% of her time. Even if we had tracked that other work, Cathy may well have performed differently compared to her estimates for those tasks, yielding a different velocity, meaning a unit of completed work different and incomparable with that of project X.

Also, we noticed that some stories went way over estimate (sometimes taking three or four times the estimate, even accounting for velocity). Part of the problem was that we had estimated these stories poorly. The biggest factors were failure to identify all of the tasks required for story completion and failure to communicate with users sufficiently during implementation, leading to significant rework of incorrect code. Even accounting for these failures, however, we felt that some story estimates simply had a greater level of uncertainty than we could represent with a single number, while other stories we felt confident we could estimate more accurately.

Although the underlying philosophy of this method cautions against trying to estimate too precisely, one can theoretically estimate in ideal minutes if one chooses. This can be a bit confusing. What is the smallest unit of time one should use? To fit with a coarse-grained estimation approach, we felt that units smaller than half a day were inappropriate. However, this led to another awkward problem, in that we often had stories (usually bugs) whose implementation did not take much time at all, so estimates of half a day would be far too generous. We tried to group these smaller stories into larger story bundles. This was sometimes not possible, however, and frankly we felt it an unnecessary burden to have to do this.

More confusing for our team, however, was the simple notion of ideal time itself. Trying to deal with two different quantities, ideal and actual time, both expressed in units of days, led to many misunderstandings and long-winded conversations. We did not even try to explain ideal and actual time to our customers—the research scientists and technicians who wanted us to write software—because we had so many problems with the distinctions even within our team.

The confusion really became apparent during tracking. During an iteration, it was difficult, if not impossible, to determine how far a developer was with a given story. For example, in a ten-calendar-day iteration, after three days, how far should Cathy be with a one-day story? A reasonable estimate would be to multiply the percent of Cathy's time spent on project X (50%) by the number of actual days passed (three calendar days), then multiplying by Cathy's velocity (say 0.5 ideal days per calendar day), yielding 0.75 days. So, Cathy should be about three-quarters of the way finished. But, on a given day, Cathy might not actually spend 50% of her time on project X. She might spend 100% of her time one day, and no time the next. In order to know how much time the story actually took, we needed to know, among all of the stories Cathy worked on that iteration, how much time she spent on the one-day story. Otherwise, at the end of an iteration, we only know that it took her ten calendar days to complete three stories (say), and no accurate idea of how much of that time she spent on each.

We tried asking the developers what percentage of their "ideal" programming time they spent each day on each issue. This did not go too well, because developers found percentages of ideal programming time a difficult quantity to report. Often developers would give answers like 6% on this issue, 24% on that issue, and 70% on that other issue, answers clearly manufactured to come out to an even 100%. We needed to find a better way.

2.2. Story points

To overcome some of these downsides, someone (I could not track down who) invented story points. Story points are an arbitrary, indivisible unit of story

“complexity”, intended to map linearly to the actual, calendar time necessary for completion. By “map linearly” I mean that two story points are supposed to take twice as much calendar time to complete as one story point. To emphasize story points’ abstract nature, people sometimes call them by other names (e.g. “gummi bears” [Jeffries et al, 2000]).

Developers estimate each story by assigning a number of story points, attempting to ensure that stories of similar complexity receive the same number of points. Usually the range of points developers can assign are restricted to be whole numbers only (to avoid false precision) and to a certain range, say one to five points. The latter restriction reminds developers to break large and ungainly stories down into smaller chunks the team can more reliably estimate and track. Trackers measure velocity in number of points completed per iteration.

Story points address a number of problems with ideal time. Because story points are not units of time, there can be no confusion between ideal and actual. It is easier to explain to customers that their budget for the next iteration is twelve points (instead of twelve ideal days, and by the way that iteration is going to take a calendar month). This benefit was probably the primary motivator for the invention of story points, and many teams that started out with ideal time migrated (at least temporarily) over to story points for this reason. Some customers had a hard time understanding how a story with an estimate of three (ideal) days could take longer than three days to complete. For both ideal time estimation and story point estimation, estimates are really indications of complexity, not of time until completion. Using an abstract estimation unit makes this distinction clear to both customers and the development team.

Dividing points into fractions of points is less natural than dividing a day into hours and minutes. This allays confusion about the level of precision one should use when estimating. The story points approach reemphasizes the approximate nature of estimation and progress tracking.

However, we found that story points did not solve the most important problems we had with ideal time. There was still no attempt to address the varying uncertainty in estimates from story to story. With a one-number estimate, whether in points or ideal days, a developer cannot express that while two issues will probably take the same amount of time, the first issue is more likely to go way over estimate than the second. We wanted developers to be able to express this, and to try to figure this information into our iteration and release planning.

More importantly, story points did not address the problems that arose because our team worked on more than one project at a time. Both story points and ideal time are estimates of complexity, and tracking the completion of units of complexity is difficult at best [WikiWikiWeb, 2005]. When a developer worked on

multiple issues and multiple projects within a single iteration, we found it impossible to determine how long he or she had actually spent on each issue, and thus very difficult to measure velocity.

3. Background

Before I explain our estimation method, I will review some background information on our team and our work, and mention several important characteristics that influenced our approach.

Our team has always been small, four to five people at any given time. Instead of working on one project, we have always spread time between two major projects, along with other occasional, smaller, short-term projects.

One major project, Neuromice, is a public web application (www.neuromice.org) that offers mutant mice produced by a consortium of three research centers (Northwestern, the Tennessee Mouse Genome Consortium, and the Jackson Laboratory) to the neuroscience and genetics communities. Neuromice has both internal users, at each of the three centers, and public users worldwide, potential purchasers of the mice. Neuromice is written in Java.

The other major project, MouseDB, is an intranet web application for the research center at Northwestern. MouseDB helps the scientists manage mutant mouse production by tracking mice pedigrees and their associated data. MouseDB has roughly 30 users at Northwestern—including research technicians, research scientists and animal care personnel—and a few users at affiliated labs in the US. MouseDB inherited a code base written in ColdFusion. We continued with a hybrid of ColdFusion and Java, more recently moving toward porting the application completely to Java.

Our team’s prior experience with estimation was mixed. Overall manager of the team, I had been exposed to agile methods and estimation as a software developer at ThoughtWorks. Other team members had varying levels of programming and software experience, from no professional experience to many years’ worth. This experience was mostly with “code and fix” approaches and/or single-person development.

Team members were primarily “creative types” better at zoning in on a problem and worse at multi-tasking. None of us had deeply ingrained organization skills. We supported each other in this regard, and were able to keep things running relatively smoothly. However, this did not come naturally.

When most people joined the development team, the MouseDB project was somewhat distressed (Neuromice had not yet commenced). Not only was the team small for the job and spread between two projects, the customer was also upset with progress. Talk was in the air of getting rid of internal development and outsourcing the work. While this stress lessened over the life of the project, it was important for the development team to be

able to show progress and to account for any schedule or scope slips.

There were a number of ancillary tasks (e.g., maintaining two Center websites) that had no time budgeted for them, and no schedule set out for their completion. These tasks sometimes interrupted our flow, when customers suddenly decided they really needed the websites overhauled in the midst of a MouseDB release.

4. Our method

We tried using ideal programming time estimates, but we never used story points. We ran into several problems with ideal time, due to a combination of the approach and our team's situation.

4.1. Problems we experienced

The estimates for some of our stories were very short, and sometimes it was unnatural to group many of these together into a single story. We needed a nonlinear range of values for our estimates, with finer granularity for small stories and coarser granularity for large stories.

Also, we had problems with some stories going way over estimate. Estimates were off for many reasons. Developers sometimes did not account for all tasks involved, or they were simply overly optimistic. However, sometimes the developers were fully aware that there was a chance that an issue could go way over or under estimate, but they could not say ahead of time for sure whether this would happen. In these cases, there was greater uncertainty in the estimate than the developers could represent with a single number. We wanted to try to capture this uncertainty and plan based on it.

Members of the team found ideal time to be too confusing. The main problem was with tracking effort expended on each story. Tracking this quantity would allow us to measure velocity and to give developers feedback on their estimates so they could improve their estimation skills.

An example is instructive. Say that on a given workday, a developer, Mike, works on the following tasks:

- Project X, Story 11 for 2 hours
- Project Y, Story 21 for 1 hour
- Project Y, Story 22 for 2 hours
- General meeting, 1 hour
- Project Y, meeting for 1 hour
- Debugging network outage for 0.5 hour
- Phone calls, emails and small talk, 0.5 hour

We will make the simplifying assumption that Mike has a velocity of 0.5 ideal days per calendar day for both projects. Story 11 has an estimate of 0.5 ideal days. Story 21 has an estimate of 1 ideal day, and Story 22 has an estimate of 2 ideal days. How much progress has Mike

made? Is he splitting his time as expected between Project X and Project Y?

One reasonable response to these questions is "I don't care on a day-to-day basis. I will measure each week at the close of the iteration." In fact this is the standard response one would expect from the XP and agile literature. However, a problem arises in trying to allocate that week's worth of effort among the various issues completed. If Mike does not complete the expected amount of work, were the stories or the nonprogramming activities to blame? If it was the stories that went over, we want to know which ones, and by how much, so we can target help to those problem areas. In our experience, asking a developer what percentage of time they spent on a story over the last five days leads to wild guesses. These guesses were especially fuzzy because our developers split their days among more than one project. Daily measurement not only increases accuracy, we also found it was much more likely to get done. Although, as creative types, we sometimes had problems even with this regular recording habit, we found we remembered to track effort expended more often when we made it part of our daily routine.

In our initial attempt to track progress for ideal time estimates, we would have asked Mike what percentage of his ideal day he spent on Story 11. Because ideal estimates ignore "distractions" like meetings and debugging network outages, Mike should consider his ideal day just the five hours he spent on stories. Mike spent 40% of his programming day on Story 11. We expected Mike to complete 0.5 ideal days of work (velocity times one actual day) total, spread across the two projects. How to spread the expected work is tricky, however, because it is unclear how to determine what percentage of the day Mike spent on each project. How do we account for time spent in non-project-specific tasks, like the general meeting and network outage?

4.2. Our approach

We went through ridiculous mental gymnastics like these for several months. Each day we would ask developers what percentage of their programming day they spent on each story, and at least once a week we had to reiterate what we meant by this question. Finally, in the midst of a discussion with John, our tracker, about how to use these numbers to measure our velocity across projects, it hit John and me that the important piece of information was that Mike had worked two hours on Story 11. This was two hours of undistracted programming time, with no intervening meetings, no phone calls, and no network outages. Since Mike's estimates were in undistracted hours, we could easily track his progress by comparing two hours with an estimate of 0.5 days, or four hours. Mike should be about halfway done. In this reformulation, ideal time is real programming time, and we track this time directly.

We realized that tracking progress in terms of the actual number of programming hours spent per day would be much more straightforward. Velocity became the number of real programming hours each programmer spent on a project per eight-hour workday. Thus, for the workday described above, Mike's velocity was 2 hours/day on Project X, and 3 hours/day on Project Y. This concrete calculation was simple to record and simple to understand, and would have been simple to explain to our customers, had they asked.

Some of our stories were going way over estimate (e.g., twenty calendar days for a four-day estimate). By tracking real effort expended we could tell when a story was nearing the estimate, and begin to direct help to the developer or pair at that point. Also, if a story started to go way over estimate, we could triage earlier. We also discussed this problem with our developers and learned that, for some of these issues, the developer had no real idea how long implementation would take, and had just given us an optimistic guess.

To deal with the latter problem, I borrowed ideas from a book I had been reading, Tom DeMarco and Tim Lister's "Waltzing with Bears" [DeMarco and Lister, 2003]. Rather than collect single-number estimates from developers, we decided to ask for three estimates: early, likely and late. The likely estimate was the familiar single-number estimate, the amount of undistracted programming time the story would most likely take. The early/late estimates were measures of the least/greatest number of undistracted days the story could take. Thus, a developer might feel he could complete a story most likely in two days. But there was a chance that, because he might have to rework part of the database schema in order for the object-relational mapping tool to behave properly, the story could take four days. However, he was confident that his work would take no fewer than 1.5 days. With three-sided estimates, as we call them, the developers can express this kind of information.

We also hoped that using three-sided estimates would have another benefit, outlined in DeMarco and Lister's book. By explicitly acknowledging that a range of outcomes were possible, we thought developers would do a better job estimating. DeMarco and Lister point out that "the earliest articulated date often becomes the deadline" in software [DeMarco and Lister, 2003]. They call this the "nano-percent" date, because it is the first date when the probability of completion becomes slightly greater than zero. On this date, the developer has a negligibly small, but positive chance of completing the story. Both comparing with similar tasks already completed and using "Yesterday's Weather" help mitigate overly optimistic estimation. However, we also thought that by asking developers to provide an early estimate, the likely estimate would no longer be the "nano-percent" estimate.

To use these real-time, three-sided estimates in iteration and release planning, we adopted the following

approach. We used the likely estimates to plan iterations and releases. We determined the average velocity for each developer from previous work, marked out planned vacations and holidays, accounted for the fact that some of us only planned to work 50% of our time on the project at hand, and came up with a budget of programming hours before the release date. We allocated stories according to this budget to the release and to iterations based on the likely estimates.

As the release progressed, developers entered the time spent each day on stories into a web application, and we monitored how well the time spent matched the likely estimate. We made no special considerations for pair programming in our estimation process. Our developer team paired only about ten to twenty percent of the time. We assumed that the increased efficiency of two people working together smarter and more intensely on a story would counterbalance the decreased efficiency of doubling up personnel. Each developer in the pair recorded time they spent each day, and we compared that total time to the story's estimate.

When a story neared its likely estimate, we checked in with the developer or pair to see how things were going, if they needed additional help, if we needed to triage, etc. If the story went over the likely estimate, we tried to give the developer daily attention, to make sure the person or pair had what they needed to complete the task as quickly as possible. We also compared the overrun to the overall release progress, to determine whether previous savings might absorb the story's overrun, or whether the release date or scope might be at risk. If the story went over the late estimate, we considered this a serious problem. A discussion of the reasons for the overrun became part of the iteration or release post-mortem. The ability to do this kind of close monitoring was one of the real benefits of this method for our team. Because we were a small team with more than enough work to go around, and because we had experienced overruns and performance problems in the past, we wanted early warning of problems so we could take appropriate action.

If it looked like our release scope or date was at risk, we would give our customers the option either to extend the release date or to cut scope. If we found that we had extra time, we again would give our customers the option of either an earlier release or more scope with the same release date.

Following standard agile principles, we were putting choice in the hands of our customers, and adding visibility into our process. Also, we emphasized fixed release dates with managed scope, which we found made scheduling the rollout of a new release easier. Third, by using likely dates, we would hit our scope targets a significant part of the time, and sometimes we would even be early.

As mentioned above, we felt we had experienced significant derailments from sudden, drop-everything customer requests for projects not directly related to our

work. Because we wanted to track the amount of time spent on these tasks, as well as on other time drags, we asked developers to record not just the time spent on stories, but to record what they did during their entire workday. If they spent two hours dealing with a faulty network connection, they would record this. If developers spent three days upgrading our customers' website, we would also record this. Also, we categorized stories within the issue tracking web application as bugs, scope, maintenance, technical issues (e.g., database upgrades), external projects (the customer websites), and scope creep. We hoped that capturing this information would give us a better view into what the most significant time drags were.

Finally, we felt that because we could tailor our uncertainty, we could specify a narrower range of time for small stories, and a larger range of time for big stories. We provide estimates in fractional days, but we use finer granularity (0.1 or 0.2 days) for smaller estimates and coarser granularity (half or whole days) for larger estimates. This allowed us to use appropriate precision when estimating. I should note that for no good reason, we have always estimated in days (down to one-tenth of a day), but recorded effort in hours. This is a little strange, but has not created problems for us, because 10% of an eight-hour workday is approximately an hour. We should move to estimating in hours to keep the units consistent.

We have used this estimation scheme for over a year (since April 2004). We track stories, record estimates and track progress using in-house, custom software. We plan to open source the code base on SourceForge sometime this year after porting it to Ruby/Rails from ColdFusion (see <http://rhythm.sourceforge.net>).

5. Findings

Releases became smoother. We have noticed that our performance tends to fall somewhere between the likely and the late estimate, with the exception of a recent release in March, which hit the late estimate almost exactly. This indicates that we are still falling prey to the tendency to be too aggressive with our likely estimates. Because we have past data on poor estimation and other risks that we cannot include in velocity, we incorporate these risks into our release plans via a fudge factor, so we still tend to deliver the planned amount of scope by the release date. Because we know which stories really went over estimate, we can review these stories during the release post-mortem to try to improve our estimation and tune our risk accounting.

Having the three numbers for each estimate gave us a better feel for the progress of a story. Exceeding the likely estimate was a better marker for the starting point of management intervention and triage. This allowed us to leave developers to their work until this clear worry sign occurred. Of course, developers were still free to ask for help, either informally or during a standup meeting.

This method has allowed us to record the time we spend fixing bugs versus other work. As a result, we know that as we introduced more unit testing (and improved in other ways), the percentage of a release we spent fixing bugs went from 21% of the time to 11%.

5.1. Creative Types

As mentioned above, none of us has deeply ingrained organizational skills. We all are people who tend to forget the time, get engrossed in our work, and ignore the daily routine. As a result, we have found it hard sometimes to keep up with entering effort daily. In fact, in the last fifteen months, out of eight releases, we only have (nearly) complete effort data on four. If we forget to enter data for more than a couple days, instead of trying to come up with fake numbers for time spent that we cannot really remember, we just accept the lost data and begin anew with the next iteration.

This hampers our ability to measure velocity for a release. Also, we lose information about our average estimation accuracy. Any deeper data mining we might want to do (for example, measuring percentage of time spent on bugs) becomes impossible if the holes in our data get too large for a given release. We have considered automated reminders, for example emailing the tracker every day if any past days are missing effort records.

However, for day-to-day release and iteration tracking, losing data like this is not such a big problem. Even if we do not know how badly we went over estimate for the previous two weeks (say), we always know what remains to be done, what the total estimate for that work is, and what our velocity has been for iterations with complete data. Even if we forget to track occasionally, we still find that there is plenty of value in our estimation and tracking approach for day-to-day project management.

5.2. Discord

Perhaps the biggest problem we had with our estimation approach revolved around an ineffective mix of quantitative and qualitative management. At one point in the project, it became clear that one of our developers was having problems going way over estimate for stories of significant complexity. This developer was not writing unit tests either, so quality was also suffering. I wanted to gather evidence for my concerns, so I could present this evidence to the developer as part of a plan for improvement. Because I was spread thinly among many responsibilities, I tried to rely on our issue tracking tool, with its estimation and effort tracking features, to help me gather this evidence.

This was a mistake. Using effort tracking to manage performance problems leads to mixed messages about the purpose of estimation and tracking progress. Rather than a tool to help developers, it becomes a tool to monitor them. Especially with our emphasis on daily recording of effort in terms of hours worked on a story, tracking can feel like

punching a clock. Depending on team culture and personality, this may or may not be a problem. In truth, however, all sorts of things were going on with that developer, the team and the project, none of which showed up in the developer's daily effort reports. The issue tracking tool could tell me that this developer was going way over estimate, but this in and of itself was not important. What was important was why s/he missed estimates. This experience reminded me that there is no substitute for one-on-one, personal interaction for these sorts of situations. Managers must use quantitative and qualitative approaches appropriately. Because recording effort daily is mildly annoying, to be effective managers must exert care not bring in any additional negative overtone. Generally, getting to the bottom of performance problems requires trust between manager and employee, and quantitative monitoring undermines trust. My brief experience has taught me that the issues involved are so subjective and subtle that gut feel is generally a better guide than measurement. This is a lesson we hear often, but as a developer recently turned manager, it is one lesson I admit I cannot hear often enough.

For these reasons, we decided that effort recording was particularly unsuited to handling these sorts of problems. To make this explicit, we adopted a policy that we would only use effort reports to help us plan iterations and releases, not as evidence of developer performance. Developers only had to record the effort they spent on stories during their daily work. They did not have to report time spent in meetings, or completing other tasks. The one downside of this was that we could no longer tell how much time we were wasting on meetings and so forth. However, we felt satisfied that we could deal with this on a qualitative basis.

5.3. Faking it

Sometimes, if we are not careful, we can end up faking our estimates. We pick an estimate, which we call the likely estimate, and then pick early and late estimates around it at what seems like cosmetically reasonable intervals. When we catch ourselves doing this, we find that our likely estimate is more of an early estimate, and the late and early estimates do not carry much meaning. This has taught us that it is important to always think carefully about and justify each of the three estimates—early, likely and late. Otherwise you will fall prey to over-optimism and DeMarco and Lister's "nano-percent" date.

5.4. Others

Our estimation scheme may be too complicated for teams. We tried to strike an appropriate balance between complexity and power, by keeping the practice as simple as it could possibly be while still affording us the various benefits described above. However, this choice will not work for all people; some may prefer less complicated approaches like story points.

6. Conclusion

Over the course of a year, my team tried various options for estimation and effort tracking, mostly variations on ideal time. These experiences, and some outside reading, led us to an estimation approach that tracks effort and estimates in the same units—real, undistracted programming hours. Instead of picking one estimate, our developers give us three—early, likely and late. Developers record effort spent on stories daily, so we can track progress and outcome clearly and cleanly for each story in a release, even if developers work on multiple projects at the same time.

This method has provided a number of benefits for us. On a day-to-day planning level, we have found we are more able to tell when stories run over estimate, so we can help developers earlier when they run into problems. We have better information about effort expended. Because of three-sided estimates, our developers estimate better, and we have noticed that in general we release on time with expected scope more often with fewer headaches. And, because we have better information about effort expended, we know that we still have room for improvement.

When we do a meticulous job recording effort for an entire release, we are able to mine this detailed data set for other interesting metrics, like percentage of time spent fixing bugs. These numbers help us understand our problem areas, and measure our progress in tackling them.

This estimation scheme will not work for every team, however. Our team has a number of characteristics—we are small, spread across more than one project, under pressure to manage risk and explain any slipups, and troubled by historical performance problems we did not fully understand—that make this estimation scheme appropriate.

References

- Demarco, Tom and Timothy Lister. *Waltzing with Bears: Managing Risk on Software Projects*. 2003.
- Beck, Kent. *Extreme Programming Explained: Embrace Change*. 1st ed. 2000.
- Beck, Kent and Cynthia Andres. *Extreme Programming Explained: Embrace Change*. 2nd ed. 2005.
- Beck, Kent and Martin Fowler. *Planning Extreme Programming*. 2001.
- WikiWikiWeb, <http://c2.com/cgi/wiki?GummiBearsConsideredHarmful>, visited June 22, 2005.
- Jeffries et al, *Extreme Programming Installed*. 2000.