The Art of Agile Development: Estimating

by James Shore

23 Apr, 2010

- *Next:* ***Chapter 9: Developing***
- *Previous:* ***Stories***
- *Up:* ***Chapter 8: Planning***

Full Text

Estimating

We provide reliable estimates.

Audience

Programmers

Programmers often consider estimating to be a black art—one of the most difficult things they must do. Many programmers find that they consistently estimate too low. To counter this problem, they pad their estimates (multiplying by three is a common approach) but sometimes even these rough guesses are too low.

Are good estimates possible? Of course! You just need to focus on your strengths.

What Works (and Doesn't) in Estimating

Part of the reason estimating is so difficult is that programmers can rarely predict how they will spend their time. A task that requires eight hours of uninterrupted concentration can take two or three days if the programmer must deal with constant interruptions. It can take even longer if the programmer works on another task at the same time.

Part of the secret to good estimates is to predict the effort, not the calendar time that a project will take. Make your estimates in terms of *ideal engineering days* (often called *story points*): the number of days a task would take if you focused entirely on it and experienced no interruptions.

> Estimate in ideal time.

Ideal time alone won't lead to accurate estimates. I've asked some of the teams I've worked with to measure exactly how long each task takes them. One team gave me 18 months of data, and even though we estimated in ideal time, the estimates were never accurate.

Still, they were *consistent*. For example, one team always estimated their stories at about 60% of the time they actually needed. This may not sound very promising. How useful can inaccurate estimates be, especially if they don't correlate to calendar time? Velocity holds the key.

Velocity

Although estimates are almost never accurate, they are *consistently* inaccurate. While the estimate accuracy of *individual* estimates is all over the map—one estimate might be half the actual time, another might be 20 percent more than the actual time—the estimates *are*consistent in aggregate. Additionally, although each iteration experiences a different set of interruptions, the amount of time required for the interruptions also tends to be consistent from iteration to iteration.

As a result, you can reliably convert estimates to calendar time if you aggregate all the stories in an iteration. A single scaling factor is all you need.

This is where velocity comes in. Your *velocity* is the number of story points you can complete in an iteration. It's a simple, yet surprisingly sophisticated tool. It uses a feedback loop: every iteration's velocity reflects what the team actually achieved in the previous iteration.

> To predict your next iteration's velocity, add the *estimates* of the stories that were "done done" in the previous iteration.

This feedback leads to a magical effect. When the team underestimates their workload, they are unable to finish all of their stories by the iteration deadline. This causes their velocity to go down, which in turn reduces the team's workload, allowing them to finish everything on

time the following week.

Similarly, if the team overestimates their workload, they find themselves able to finish more stories by the iteration deadline. This causes their velocity to go up, which increases the team's workload to match their capacity.

Velocity is an extremely effective way of balancing the team's workload. In a mature XP team, velocity is stable enough to predict schedules with a high degree of accuracy (see **Risk Management** earlier in this chapter).

Velocity and the Iteration Timebox

Velocity relies upon a strict iteration timebox. To make velocity work, *never* count stories that aren't "done done" at the end of the iteration.*Never* allow the iteration deadline to slip, not even by a few hours.

ALLY
Energized Work

You may be tempted to cheat a bit and work longer hours, or to slip the iteration deadline, in order to finish your stories and make your velocity a little bit higher. Don't do that! Artificially raising velocity sabotages the equilibrium of the feedback cycle. If continued, your velocity will gyrate out of control, which will likely reduce your capacity for energized work in the process. This will further damage your equilibrium and your ability to meet your commitments.

> One project manager wanted to add a few days to the beginning of an iteration so his team could "hit the ground running" and have a higher velocity to show to stakeholders. By doing so, he set the team up for failure: they couldn't keep that pace in the following iteration. Remember that velocity is for predicting schedules, not judging productivity. See **Reporting** in Chapter 6 for ideas about what to report to stakeholders.

ALLY
Slack

Velocity tends to be unstable at the beginning of a project. Give it three or four iterations to stabilize. After that point, you should achieve the same velocity every iteration, unless there's a holiday during the iteration. Use your iteration slack to ensure that you consistently meet your commitments every iteration. I look for deeper problems if the team's velocity changes more than one or twice per quarter.

Tips For Accurate Estimates

You can have accurate estimates if you:

1. Estimate in terms of ideal engineering days (story points), not calendar time.
2. Use velocity to determine how many story points the team can finish in an iteration.
3. Use iteration slack to smooth over surprises and deliver on time every iteration.
4. Use risk management to adjust for risks to the overall release plan.

How to Make Consistent Estimates

ALLY
Risk Management

There's a secret to estimating. *Experts automatically make consistent estimates*[1]. All you have to do use a consistent estimating technique. When you estimate, pick a single, optimistic value. How long will the story take if you experience no interruptions, can pair with anyone else on the team, and everything goes well? There's no need to pad your estimates or provide a probabilistic range with this approach. Velocity automatically applies the appropriate amount of padding for short-term estimates and risk management adds padding for long-term estimates.

[1]*Unless you have a lot of technical debt. If you do, add more iteration slack (see **Slack** earlier in this chapter) to compensate for the inconsistency.*

There are two corollaries to this secret. First, if you're an expert but you don't trust your ability to make estimates, relax. You automatically make good estimates. Just imagine the work you're going to do and pick the first number that comes into your head. It won't be right, but it *will* be consistent with your other estimates. That's sufficient.

> To make a good estimate, go with your gut.

Second, if you're not an expert, the way to make good estimates is to become an expert. This isn't as hard as it sounds. An expert is just a beginner with lots of experience. To become an expert, make a lot of estimates with relatively short timescales and pay attention to the results. In other words, follow the XP practices.

All of the programmers should participate in estimating. At least one customer should be present to answer questions. Once the programmers have all the information they need to make an estimate, one programmer will suggest an estimate. Allow this to happen naturally. The person who is most comfortable will speak first. Typically this is the person who has the most expertise.

> I assume that programmers are the constraint in this book (see **XP Concepts** in Chapter 3), which means they should be the only ones estimating stories. The system constraint determines how quickly you can deliver your final product, so only the constraint's estimates are necessary. If programmers aren't the constraint on your team, talk with your mentor about how to adjust XP for your situation.

If the suggested estimate doesn't sound right, or if you don't understand where it came from, ask for details. Alternatively, if you're a programmer, provide your own estimate and explain your reasoning. The ensuing discussion will clarify the estimate. When all of the programmers confirm an estimate, write it on the card.

> If you don't know a part of the system well, practice making estimates for that part of the system in your head, then compare your estimate with those of the experts. When your estimate is different, ask why.

At first, different team members will have differing ideas of how long something should take. This will lead to inconsistent estimates. If the team makes estimates as a group, programmers will automatically synchronize their estimates within the first several iterations.

Comparing your estimates to the actual time required for each story or task may also give you the feedback you need to become more consistent. To do so, track your time as described in **Reporting** in Chapter 6.

How to Estimate Stories

Estimate stories in story points. When you start estimating stories, think of a story point as an ideal day.

> It's okay to estimate in half-points, but quarter-points shouldn't be necessary.

When you start estimating stories, imagine the engineering tasks you will need to implement it. Ask your on-site customers about their expectations, focusing on those things that would affect your estimate. If you encounter something that seems expensive, provide the customers with less costly alternatives.

As you gain experience with the project, you will begin to make estimates intuitively rather than mentally breaking them down into engineering tasks. Often, you'll think in terms of similar stories rather than ideal days. For example, you might think that a story is a typical report, and typical reports are one point. Or you might think that a story is a complicated report, but not twice as difficult as a regular report, and estimate it at one and a half points.

Sometimes you will need more information to make an accurate estimate. In this case, make a note on the card. If you need more information from your customers, write "??" (for "unanswered questions") in place of the estimate. If you need to research a piece of technology further, write "Spike" in place of the estimate and create a spike solution story (see **Stories** earlier in this chapter).

> *The team has gathered to estimate stories. Mary, one of the on-site customers, starts the discussion. Amy, Joe, and Kim are all programmers.*
>
> ***Mary:*** Here's our next story. *(She reads the story card aloud, then puts it on the table.)* "Report on parts inventory in warehouse."
>
> ***Amy:*** We've done so many reports by now that a new one shouldn't be too much trouble. They're typically one point each. We already track parts inventory, so there's no new data for us to manage. Is there anything unusual about this report?
>
> ***Mary:*** I don't think so. We put together a mock-up. *(She pulls out a printout and hands it to Amy.)*
>
> ***Amy:*** This looks pretty straightforward. *(She puts the paper on the table. The other programmers take a look.)*
>
> ***Joe:*** Mary, what's this `age` column you have here?

Mary: That's the number of business days since the part entered the warehouse.

Joe: You need business days, not calendar days?

Mary: That's right.

Joe: What about holidays?

Mary: We only want to count days that we're actually in operation. No weekends, holidays, or scheduled shutdowns.

Kim: Joe, I see what you're getting at. Mary, that's going to increase our estimate because we don't currently track scheduled shutdowns. We would need a new UI, or a data feed, in order to know that information. It could add to the complexity of the admin screens, and you and Brian have said that ease of admin is important to you. Do you really need age to be that accurate?

Mary: Hmm. Well, the exact number isn't that important, but if we're going to provide a piece of information, I would prefer it to be accurate. What about holidays—can you do that?

Kim: Can we assume that the holidays will be the same every year?

Mary: Not necessarily, but they won't change very often.

Kim: Okay, then we can put them in the config file for now rather than creating a UI for them. That would make this story cheaper.

Mary: You know, I'm going to need to look into this some more. This field isn't that important and I don't think it's going to be worth the added administration burden. Rather than business days, let's just make it the number of calendar weeks. I'll make a separate story for dealing with business days. *(She takes a card and writes, "Report part inventory age in business days, not calendar weeks. (no holidays, weekends, or scheduled shutdowns)".)*

Kim: Sounds good. That should be pretty easy then, because we already track when the part went into the warehouse. What about the UI?

Mary: All we need to do is add it to the list of reports on the reporting screen.

Kim: I think I'm ready to estimate. *(Looks at other programmers.)* This looks like a pretty standard report to me. It's another specialization for our reporting layer with a few minor logic changes. I'd say one point.

Joe: That's what I was thinking, too. Anybody else?

(The other programmers nod.)

Joe: One point. *(He writes "1" on the story card.)* Mary, I don't think we can estimate the business day story you just created until you know what kind of UI you want for it.

Mary: That's fair. *(Writes "??" on the business day card.)* Our next story...

How to Estimate Iteration Tasks

During iteration planning, programmers will create engineering tasks that allow them to deliver the stories planned for the iteration. Each engineering task is a concrete, technical task such as "update build script" or "implement domain logic".

ALLY
Iteration Planning

Estimate engineering tasks in ideal hours rather than ideal days. When you think about the estimate, be sure to include everything necessary for the task to be "done done"—testing, customer review, and so forth.
See **Iteration Planning** earlier in this chapter for more information.

ALLY
"Done Done"

When Estimating is Difficult

If the programmers understand the requirements and are experts in the required technology, they should be able to estimate a story in less than a minute. If the programmers need to discuss the technology, or if they need to ask questions of the customers, then estimating may take longer. I look for ways to bring discussions to a close if an estimate takes longer than five minutes, and I look for deeper problems if every story involves detailed discussion.

One common cause of slow estimates is inadequate customer preparation. To make their estimates, programmers often ask questions the customers haven't considered. In some cases, customers will disagree on the answer and need to work it out.

A *customer huddle*—in which the customers briefly discuss the issue, come to a decision, and return—is one way to handle this. Another way is to write "??" in place of the estimate and move on to the next story. The customers then work out the details at their own pace, and the programmers estimate the story later.

Expect the customers to be unprepared for programmer questions during the first several iterations. Over time, they will learn to anticipate most of the questions programmers will ask.

Programmer inexperience can also cause slow estimating. If the programmers don't understand the problem domain well, they will need to ask a lot of questions before they can make an estimate. As with inadequate customer preparation, this problem will go away in time. If the programmers don't understand the *technology*, however, immediately create a spike story and move on.

ALLY
Spike Solutions

Some programmers try to figure out all the details of the requirements before making an estimate. However, only those issues which would change the estimate by a half-point or more are relevant. It takes practice to figure out which details are important and which you can safely ignore.

This sort of over-attention to detail sometimes occurs when a programmer is reluctant to make estimates. A programmer who worries that someone will use her estimate against her in the future will spend too much time trying to make her estimate perfect rather than settling on her first impression.

Programmer reluctance may be a sign of organizational difficulties or excessive schedule pressure, or it may also stem from past experiences that have nothing to do with the current project. In the latter case, the programmers will usually come to trust the team over time.

ALLY
Trust

To help address these estimation issues, ask leading questions. For example:

- Do we need a customer huddle on this issue?
- Should we mark this "??" and come back to it later?
- Should we make a spike for this story?
- Do we have enough information to estimate this story?
- Will that information change your estimate?
- We've spent five minutes on this story. Should we come back to it later?

Explaining Estimates

It's almost a law of physics: customers and stakeholders are invariably disappointed with the amount of features their teams can provide. Sometimes they express that disappointment out loud. The best way to deal with this is to ignore the tone and treat the customers' questions as honest requests for information.

In fact, a certain amount of back-and-forth is healthy: it helps the team focus on the high-value, low-cost elements of the customers' ideas. (See **The Planning Game** earlier in this chapter.)

One common challenge is to say, "Why does that cost so much?" Resist the immediate urge to defend yourself and your sacred honor, pause a moment to collect your thoughts, then list the issues you considered when coming up with the estimate. Suggest options for reducing the cost of the story by reducing scope.

Your explanation will usually satisfy your customers. In some cases, they'll ask for more information. Again, treat these questions as simple requests. If there's something you don't know, admit it, and explain why you made your estimate anyway. If a question reveals something that you haven't considered, change your estimate.

Be careful, though: the questions may cause you to doubt your estimate. Your initial, gut-feel estimate is most likely correct. Only change your estimate if you learn something genuinely new. Don't change it just because you feel pressured. As the programmers who will be implementing the stories, you are the most qualified to make the estimates. Be polite, but firm:

> Politely but firmly refuse to change your estimates when pressured.

> I'm sorry you don't like these estimates. We believe our estimates are correct, but if they're too pessimistic, our velocity will automatically increase to compensate. We have a professional obligation to you and this organization to give you the best estimates we know how, even if they are disappointing, and that's what we're doing.

If a customer reacts with disbelief or browbeats you, he may not realize how disrespectful he's being. Sometimes making him aware of his behavior can help.

> From your body language and the snort you just made, I'm getting the impression that you don't respect or trust our professionalism. Is that what you intended?

Customers and stakeholders may also be confused by the idea of story points. I start by providing a simplified explanation:

> A story point is an estimation technique that automatically compensates for overhead and estimate error. Our velocity is *v*, which means we can accomplish *v* points per week. (Substitute your velocity for *v*.)

If the questioner pushes for more information, I explain all of the details of ideal days and velocity. That often inspires concern. "If our velocity is ten ideal days and we have six programmers, shouldn't our velocity be 30?"

You can try to explain that ideal days aren't the same as developer days, but that has never worked for me. Now I just offer to provide detailed information.

> Generally speaking, our velocity is ten because of estimate error and overhead. If you like, we'll perform a detailed audit of our work next week and tell you exactly where the time's going. Would that be useful?

(**Reporting** in Chapter 6 discusses time usage reports in detail.)

These questions often dissipate as customers and stakeholders gain trust in the team's ability to deliver. If they don't, or if the problems are particularly bad, enlist the help of your project manager to defuse the situation. **Trust** in chapter 6 has further suggestions.

How to Improve Your Velocity

Your velocity can suffer for many reasons. The following options might allow you to improve your velocity:

Pay Down Technical Debt

The most common technical problem I see is excessive technical debt. This has a bigger impact on team productivity than any other factor does. Make code quality a priority and your velocity will improve dramatically. However, this isn't a quick fix. Teams with excessive technical debt often have months—or even years—of cleanup ahead of them. Rather than stopping work to pay down technical debt, fix it incrementally. Iteration slack is the best way to do so, although you may not see a noticeable improvement for several months.

ALLY
Slack

Improve Customer Involvement

If your customers aren't available to answer questions when programmers need them, programmers either have to wait or make guesses about the answers. Both of these hurt velocity. To improve your velocity, make sure that a customer is always available to answer programmer questions.

ALLY
Sit Together

Support Energized Work

Tired, burned-out programmers make costly mistakes and don't put forth their full effort. If your organization has been putting pressure on the team, or if programmers have worked a lot of extra hours, shield the programmers from organizational pressure and consider instituting a no-overtime policy.

ALLY
Energized Work

Offload Programmer Duties

If programmers are the constraint for your team—as this book assumes—then hand any work that other people can do *to* other people. Find ways to excuse programmers from unnecessary meetings, shield them from interruptions, and have somebody else to take care of organizational bureaucracy such as time sheets and expense reports. You could even hire an administrative assistant for the team to handle all non-project-related matters.

Provide Needed Resources

Most programming teams have all of the resources they need. However, if your programmers complain about slow computers, insufficient RAM, or inavailability of key materials, get it for them. It's always a surprise when a company nickle-and-dimes its software teams. Does it make sense to save $5,000 in equipment costs if it costs your team half an hour per programmer every day? A team of six programmers will recoup that cost within a month. And what about the opportunity costs of delivering fewer features?

Add Programmers (Carefully)

Velocity is related to the number of programmers on your team, but unless your project is woefully understaffed and experienced personnel are readily available, adding people won't make an immediate difference. As [Brooks] famously said, "adding people to a late project only makes it later." Expect the new employees to take a month or two to be productive. Pair programming, collective code ownership, and a shared workspace will help reduce that time, though adding junior programmers to the team can actually *decrease* productivity.

Likewise, adding to large teams can cause communication challenges that decrease productivity. Six programmers is my preferred size for an XP team and I readily add good programmers to reach that number. Past six, I am very cautious about adding programmers, and I avoid team sizes greater than ten programmers.

Questions

How do we modify our velocity if we add or remove programmers?

If you add or remove only one person, try leaving your velocity unchanged and see what happens. Another option is to adjust your velocity proportionally to the change. Either way, your velocity will adjust to the correct number after another iteration.

How can we have a stable velocity? Team members take vacation, get sick, and so forth.

Your iteration slack should handle minor variations in people's availability. If a large percentage of the team is away, as during a holiday, your velocity may go down for an iteration. This is normal. Your velocity should recover in the next iteration.

If you have a small number of programmers—four or fewer—you may find that even one day of absence is enough to affect your velocity. In this case, you may wish to use two-week iterations. See **Iteration Planning** earlier in this chapter for a discussion of the trade-offs.

What should we use as our velocity at the beginning of the project?

For your first iteration, just make your best guess. Set your velocity for the next iteration based on what you actually complete. Expect your velocity to take three or four iterations to stabilize.

Your organization may want you to make release commitments before your velocity has stabilized. The best approach is to say, "I don't have a schedule estimate yet, but we're working on it. I can promise you an estimate in three or four weeks. We need the time to calibrate the developers' estimates to what they actually produce."

Isn't it a waste of time for all of the programmers to estimate stories together?

It does take a lot of programmer-hours for all of the programmers to estimate together, but this isn't wasted time. Estimating sessions are not just for estimation—they're also a crucial first step in communicating and clarifying requirements. Programmers ask questions and clarify details, which often leads to ideas that the customers haven't considered. Sometimes this collaboration reduces the overall cost of the project. (See **The Planning Game** earlier in this chapter.)

All of the programmers need to be present to ensure that they all understand what they will be building. Having the programmers together also increases estimate accuracy.

What if we don't ask the right questions of the customer and miss something?

Sometimes you will miss something important. If Joe hadn't asked Mary about the age column, the team would have missed a major problem and their estimate would have been wrong. These mistakes happen. Information obvious to customers isn't always obvious to programmers.

Although you cannot *prevent* these mistakes, you can reduce them. If all of the programmers estimate together, they're more likely to ask the right questions. Mistakes will also decrease as the team becomes more experienced in making estimates. Customers will learn what details to provide and programmers will learn which questions to ask.

In the meantime, don't worry about it unless you encounter these surprises frequently. Address unexpected details when they come up. (See **Iteration Planning** earlier in this chapter.)

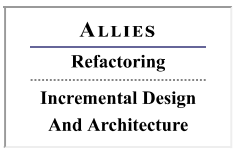
If unexpected details frequently surprise you, and the problem doesn't improve with experience, ask your mentor for help.

When should we re-estimate our stories?

Story estimates don't need to be accurate, just self-consistent. As a result, you only need to re-estimate stories when your understanding of the story changes in a way that affects its difficulty or scope.

To make our estimates, we made some assumptions about the design. What if the design changes?

XP uses incremental design and architecture, so the whole design gradually improves over time. As a result, your estimates will usually remain consistent with each other.



How do we deal with technical dependencies in our stories?

With proper incremental design and architecture, technical dependencies should be rare, although they can happen. I typically make a note in the estimate field: "six (four if foo story done first)."

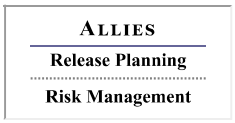
If you find yourself making more than a few of these notes, something is wrong with your approach to incremental design. Ask your mentor for help.

What if we want to get a rough estimate of project length, without doing release planning, before the project begins?

[DeMarco 2002] has argued that organizations set project deadlines based on the value of the project. In other words, the project is only worth doing if you can complete it before the deadline. (Some organizations play games with deadlines in order to compensate for expected overruns, but assume the deadline is accurate in this discussion.)

If this is true—and it coincides with my experience—then the estimate for the project isn't as important as whether or not you can finish it before the deadline.

To judge whether a project is worth pursuing, gather the project visionary, a seasoned project manager, and a senior programmer or two (preferably ones that would be on the project). Ask the visionary to describe the project goals and its deadline, then ask the project manager and programmers if they think it's possible. If it is, then you should gather the team and perform some real estimating, release planning, and risk management by conducting the first three or four iterations of the project.



This approach takes about four weeks and yields a release date that you can commit to. [McConnell 2005] provides additional options that are faster but less reliable.

Results

When you estimate well, your velocity is consistent and predictable with each iteration. You make commitments and meet them reliably. Estimation is fast and easy, and you can estimate most stories in a minute or two.

Contraindications

This approach to estimating assumes that programmers are the constraint (see **XP Concepts** in Chapter 3 for more about the Theory of Constraints). It also depends on fixed-length iterations, small stories, and small tasks. If these conditions aren't present, you need to use a different approach to estimating.

This approach also requires trust: developers need to believe that they can give accurate estimates without being attacked, and customers and stakeholders needto believe the developers are providing honest estimates. That trust may not be present at first, but if it doesn't develop, you'll run into trouble.

Regardless of your approach to estimating, never use missed estimates to attack developers. This is a quick and easy way to destroy trust.

Alternatives

There are many approaches to estimating. This one has the benefit of being both accurate and simple. However, its dependency on release planning for long-term estimates makes it labor intensive for initial project estimates. See [McConnell 2005] for other options.

Further Reading

Agile Estimating and Planning [Cohn] describes a variety of approaches to agile estimation.

Software Estimation: Demystifying the Black Art [McConnell] provides a comprehensive look at traditional approaches to estimation.

- *Next:* ***Chapter 9: Developing***
- *Previous:* ***Stories***
- *Up:* ***Chapter 8: Planning***