In Agile, What Should We Estimate?

A version of this article was originally published on AgileConnection.

Sometimes the highest priority is not enough.

"Ladies and gentlemen, we have reached our cruising altitude. You have experienced the new release of our avionics software. "Takeoff" was the highest priority feature, and we loaded it on the plane as soon as it was done. As you just saw, it worked flawlessly. The next priority for the development team is "Landing," and the team will radio that software up to the plane as soon as it's potentially shippable. Don't worry, we have plenty of fuel, so sit back and relax, and thank you for flying Agile Airlines."

This is a joke of course. But while no professional software organization would let passengers take off if they couldn't land, it illustrates that the answer to the question, "When will software be ready to deliver?" isn't as simple as "Just choose the highest priority features first, then you can deliver working software at the end of any sprint."

What should we estimate?

There's been a lot of confusing discussion in the agile community surrounding the question, "Should we estimate?" Often times, however, confusion can stem from lack of clarity. So, it can be helpful to expand the question to make it, and thus the response to it, more useful and straightforward. Perhaps the question that should be asked is: "What should we estimate when we are using agile methods, and what choices will we make based on an estimate?"

The QSM Agile Round Table was formed to discuss the role of estimation in agile environments. QSM customers shared their questions, challenges, and experiences on the relevance and benefits of scope-based estimation in an agile environment. "What should we estimate?" was the first question we addressed. We agreed early on that there were two distinct meanings to the term "estimate" in the agile community, and that leads to confusion. On the one hand, the term "estimate" is used for sprint planning. This is trying to answer the question, "What stories should we develop over the next two weeks (or however long the time-boxed sprint will be)?" That's a fundamentally different question from the larger-scale question, "What duration and cost should I plan for, and how likely is it that the plan will succeed?" We agreed that this larger-scale question was still important in an agile environment and we wanted to understand how agile methods impacted the way we go about such estimates.

The first thing we needed to understand was, "To what do we apply this larger scale question?" We don't ask this about each sprint! "How long will a two-week sprint take" isn't a very interesting question. Traditionally, this question is asked about a project. In many agile environments, however, we no longer organize work as projects in the traditional sense, so what is it we are planning for? What plays the role of "project" that we should estimate?

To start answering that, we need to look at some techniques commonly used by agile teams.

What is potentially shippable software?

Agile methods emphasize value and quality. To maximize value, backlogs are prioritized, and to the extent possible, stories are developed in priority order. Additionally, the team only "counts" a story if it's complete, and to be complete, it must be well tested. At the end of each sprint, the highest priority functionality that the team has had time to develop is working—the software is "potentially shippable." But to complete a story within a single sprint, broader epics and stories must be broken down into developer-sized bites--parts of the story that can be completed in a short period of time.

These techniques allow each iteration review to accomplish its primary purpose: get feedback on software completed so far, so we can learn what needs to change going forward. This prepares us to develop another chunk in the next sprint, making progress on software that will deliver value to the eventual users.

What is consumable value?

An inevitable consequence of breaking stories into developer-sized bites is that the smaller pieces developed in a particular sprint may not be the full features that users will need. That's ok: the remaining pieces will be developed in the coming sprints, and the team learns from feedback on the earlier bites. But until enough of the higher level epics and stories are developed to make the software useful for its intended purpose, users cannot consume the software.

Partway through our work on a release of our products at QSM, we sometimes propose an enhancement to meet a specific customer request. To avoid having to delay the release, we may propose "just enough" to meet that specific request. Our lead often criticizes such proposals as "half a feature." When customers use the enhancement, they will also expect related functionality. The enhancement has value, but cannot be consumed unless there are other enhancements as well.

It's easy to come up with blatant examples:

- ATM software that lets you put in your card and enter your PIN, but you can't get money from the machine.
- ATM software that lets you get money from the machine after validating your card, but you can't enter your PIN.
- Payroll software that can compute the paychecks, but can't print checks or make direct deposits.
- An airplane that can take off but not land.

Consumable value comes in all sizes, and a large scale consumable epic can often be broken down into consumable stories, some of which may be further broken down into smaller consumable stories. Some of those may turn out to be developer-sized bites that can be developed in a single iteration. But often, when a consumable story is broken into developer-sized bites to fit into a single sprint, the individual bites, while useful for review and feedback, will not be consumable by themselves. Over several iterations, the bites combine into a consumable meal. Consumable value is a collection of stories that provide enough value to satisfy users. This cannot arbitrarily be forced into a time box.

Keeping the "viable" in your Minimal Viable Product.

Making sure that you deliver enough value to users doesn't mean you can't incrementally release features over time. A large epic may be broken into consumable pieces, with some of those pieces released first, and enhanced over time. Eric Ries popularized the term "minimal viable product," and the related concept, "releasing in small batches," is looked highly upon by many agile organizations. But "the smaller the better" works only up to a point, which is why Reis didn't call it "Minimal Product." The product must be viable, in other words, provide consumable value. And Reis is very clear that what is viable depends on your customers, your competition, and your goals for the product.[1]

#YesEstimate to get to consumable value

Since we expect it will take multiple sprints to deliver consumable value, it's worthwhile to estimate what it will take to deliver that value. That is, it makes sense to answer the compound question, "What consumable value do we expect to achieve (described at a high level, not in detail), what duration and cost should we plan for, and how likely is it that the plan will succeed?" Based on that, we can propose a feasible plan that is both attainable and meets the company's business requirements.

This is very different from iteration or sprint planning, where the primary question is, "given where we are now, and given the backlog we have in front of us, what developer-sized bites should we include in the next time-boxed sprint?" While that may involve estimates of how big the individual stories are (Planning Poker, anyone?), it's fundamentally different from an estimate to achieve consumable value. The #NoEstimates movement has grown up around simplifying sprint planning through proper backlog grooming, including the breakdown of bigger stories into smaller bites and careful prioritization.

The purpose of estimates based on consumable value is to help make overall decisions, often decisions that must be made before development of specific stories even starts. Some of these decisions are:

- Where should we invest our limited resources? Most companies have many good ideas for new software, enhancements to existing software, new platforms and architectures, and other types of value. Usually there are more good ideas than resources to carry them out. Choosing the collection of ideas to invest in requires balancing the benefits to be delivered with the resources and time it will take to produce those benefits, then choosing among those ideas based on the combination of benefit and what it takes to deliver.
- What team or teams should I allocate to a particular idea and for how long? There's usually a complex tradeoff among the time it takes to realize benefit, the likely cost of the development to get the benefit within that time, and the risk of meeting that plan.
- What are the tradeoffs between benefit and the time and effort it takes to deliver and the cost of delay? Agile teams have learned to appreciate ideas such as "minimal viable product," "develop in small batches," and "weighted shortest job first;" delaying delivery has costs of its own. However, reducing scope to speed up delivery impacts the consumable value of the delivered product. These tradeoffs cannot be made arbitrarily, so considering multiple estimates for several different scenarios allows us to make informed plans, yielding results that are both minimal and viable.
- How can we coordinate software delivery with other organizations involved? Software development is rarely an island. Plans for software delivery must be coordinated with other initiatives and the organizations involved in them. Other organizations have to plan based on when they can expect value from the delivered software.

Delivery of consumable value comes in many forms.

It used to be a lot easier, in theory, to answer the question "What should we estimate?" The answer was "turn a proposal into a project, and include an estimate of the duration and effort it will take to complete that project." Of course, it was never actually that simple in practice. This seemingly clear answer was based on the classic notion of a project, a temporary endeavor undertaken to create a unique product or service with a definite beginning and end, and software development is hardly ever that delineated. Many agile methods are designed to mitigate the problems we encounter trying to stick to that definition, and many agile organizations no longer organize their work around the classic notion of project. How your company organizes software development work depends on many factors. Here are a few ways you may recognize, among the many variations we see in practice. Most of these and more were represented in the QSM Agile Round Table.

- We are developing internal systems for our company. Thus, there is a unique deployed instance of the system in production (or perhaps a few, segmented by organization), though there may be multiple other instances for development and testing. We promote changes through the varying levels and eventually those changes are consumed by the users of the production system. We may have an agile team working independently on each system, or we may be using an "agile at scale" methodology to coordinate the work of many teams across many systems.
- We are a product company; we develop software to sell to our customers. We periodically release new versions, perhaps fix packs, or perhaps major releases with innovative new features. We may bundle up several innovative features into an annual release. Our customers decide whether or not each new release provides enough value to them to justify consuming it.
- We are a system integrator or contract development shop. Our customers desire new or enhanced software, and we bid for that business.
- We contract out software development. We need to plan the software budget to meet our business goals, and we will evaluate bids for the development.
- We push out updates of our software to our subscribers when they become available (continuous deployment, like those updates you usually blindly accept on your cell phone).

Some of these are like classic projects, others much less so. The relationship between delivering consumable value with software and "a temporary endeavor with a definite beginning and end" is much more tenuous than project and portfolio planners used to think it was. Some may argue it was always so, but we're now explicitly incorporating it in our methods.

Set goals for delivering consumable value and estimate what it takes.

So if we don't have projects to estimate, and since delivery can take so many forms, how do we make the decisions about what to invest in and the other related decisions? The specifics and the terminology depend both on your business and the methodologies you use, but in all cases you can set a goal for a collection of features that will provide consumable value, and estimate what it takes to reach the goal. You will no doubt have multiple possible ideas for such goals, including variations on what features should be included (different possibilities of what is simultaneously minimal and viable), and you can have multiple estimates for each, trading off time and cost. This gives you information to make decisions about which goals to invest in, where to apply resources, how to plan related initiatives and the other decisions you need to make to move forward with development.

Your goals should be specific enough to estimate and make decisions about. "Improve the user's experience" is potentially the vision you have for your software investments, but is likely too vague to determine concrete steps to reach the goal. On the other hand, don't fall into the "big upfront requirements" trap. You do not need to define all the details of features in order to make decisions about your goals, and using agile methods to allow

those details to emerge over the course of development will help you “build the right product.”

The way you deliver software can affect how you map goals to delivery. If you push new features to production, you may push the features when the goal is reached. You may even push some features to production that can stand alone before the entire goal is reached (this may involve the development technique of “feature branches” to ensure consistent code). If instead you have annual releases, you may bundle up multiple goals into a single release, estimating whether it’s plausible to reach that combined goal in the time allotted. If you’re using an agile at scale methodology, you may plan for multiple goals to be developed concurrently, using the estimates to vary the resource allocation over time; the actual releases of the software to production may be planned based on your delivery capabilities and methods.

Note, however, that when we say a goal is a collection of features, we are using the word “feature” in a generic sense. Your development methodology may have a specific meaning for “feature”, in which case you may want to use a different term. It’s important to realize that features come in all sizes. They may be very specific (send e-mail confirmations whenever the shipping status of a purchase changes, fix the bug in a specific calculation), or they may be higher level (revamp the reporting capabilities), or anything in between (allow a car rental to be added to hotel reservations). Consumable value, after all, is in the eye of the eventual consumer as well as your company’s business needs.

Using the estimate with agile methods: is an estimate a commitment?

This is an old question, of course. Software teams have always been reluctant to give an estimate, which inherently has a degree of uncertainty, because management often takes the cost (which primarily affects staffing levels) and duration as firm commitments. This can lead, as we all know, to nightmare “death marches” and cancelled projects when the reality of development doesn’t match the commitment. Part of the reason for this is poor estimation techniques that underestimate. But even with sophisticated estimation techniques based on history, estimates inherently have a degree of uncertainty.

In the QSM Agile Round Table, we discussed how organizations handle the question of uncertainty in an estimate. Bonnie Brown, Hewlett Packard Enterprise, said some teams handle estimation risk by putting the burden on the product manager to manage scope within the committed time. They estimate cost and duration based on the understanding of the planned scope (the “goal” in the terms of this article), and commit resources for the duration of that estimate, which determined the number of time-boxed sprints. Then, as Bonnie said, the product manager would decide on the priorities at each sprint and could get “whatever scope they want within that duration.” This fits with what some agile practitioners say, “In agile, we time-box duration and vary scope.” However, other members of the roundtable said that didn’t work for their customers. The customer expected the result to meet the original goal. As the work progressed in an agile fashion, the customer certainly refined and modified the original scope, but the customer did not accept the notion that “all the scope that fits the original estimate” was sufficient if it did not meet the goal. Consumable value is in the eye of the beholder.

Agile methods increase the uncertainty.

Perhaps the biggest impact of agile methods on early, goal-level estimation is that we don't even pretend to have a detailed description of the scope when we need to estimate. The details of the scope emerge throughout the work, as large epics are refined into development-sized bites. There is no upfront signed-off requirements document on which to base an estimate. Of course, part of the rationale for agile methods is that the signed-off requirements document never really stuck anyway, or if it did, we had a result that nobody liked. Emergent requirements based on feedback at each sprint is the best way to "build the right product."

But how, then, can you estimate an expected duration and cost from a scope only defined at a high level? In estimation terms, how can you determine the size of your goal? The QSM Agile Round Table addressed these questions, and that will be the topic of the next two articles in this series.

---

1. Reis, Eric, *The Lean Startup*, Crown Publishing Group, 2011