Robert C. Martin Series

PRENTICE HALL PTR

Agile Estimating and Planning

Forewords by Jim Highsmith, Agile Practice Director, Cutter Consortium, and Gabrielle ene ield , Director, Agile Product De elopment, ahoo

Mike Cohn

Contents

Foreword

Everywhere in the agile world I hear the same questions:

- How do I plan for large teams?
- What size iteration should we use?
- How should I report progress to management?
- How do I prioritize stories?
- How do I get the big picture of the project?

These questions, and many others, are skillfully addressed in this book. If you are a project manager, project lead, developer, or director, this book gives you the tools you need to estimate, plan, and manage agile projects of virtually any size.

I have known Mike Cohn for five years. I met him shortly after the Agile Manifesto was signed. Mike joined the Agile Alliance with a unique enthusiasm and energy. Any project he took on, he completed, and completed well. He was visible, and he was helpful. He very quickly became indispensable to the fledgling organization.

Now he has put the same level of competence, thoroughness, and energy into this book. And it shows. It shows big time.

It shows, because this book gives advice that is innately practical. This is not a book of theoretical abstractions. As a reader, you will not be spending your time in the clouds, looking at the problems at the 30,000 foot level. Instead, Mike

provides concrete practices, techniques, tools, charts, formulae, and—best of all—cogent advice. This book is a how-to manual for estimating and planning.

Laced throughout the book are anecdotes that expose Mike's experience in using the techniques and tools he is describing. He tells you when they have worked for him, and when they haven't. He tells you what can go wrong, and what can go right. He makes no promises, offers no silver bullet, provides no guarantee. Yet at the same time he leaves you with little doubt that he is offering a large measure of his own hard-won experience.

There have been many books that have touched upon the topics of agile estimation and planning. Indeed, there have been a few that have made it their primary topic. But none have matched the depth and utility of this book, which covers the topic so completely, and so usefully, that I think it is bound to be regarded as the definitive work.

OK, I know I'm gushing, but I'm excited. I'm excited that so many long-standing questions have finally been answered with sufficient competence. I'm excited that I now have a tool I can give to my clients when they ask the hard questions. I'm excited that this book is ready, and that you are about to read it.

I am pleased and honored to have this book in my series. I think it's a winner.

Robert C. Martin
Series Editor

Foreword

In reading a book or manuscript for the first time I always ask myself the same question, "What is the author adding to the state of the art on this subject?" In Mike's case the answer is twofold: His book adds to our knowledge of "how" to do estimating and planning, and it adds to our knowledge of "why" certain practices are important.

Agile planning is deceptive. At one level, it's pretty easy—create a few story cards, prioritize them, allocate them to release iterations, then add additional detail to get a next iteration plan. You can show a team the basics of planning in a couple of hours, and they can actually turn out a tolerable plan (for a small project) in a few more hours. Mike's book will greatly help teams move from producing tolerable plans to producing very good plans. I'm using my words carefully here—I didn't say a great plan, because as Mike points out in this book, the difference between a good (enough) plan and a great plan probably isn't worth the extra effort.

My early thoughts about Mike's book have to do with the concept of agile planning itself. I'm always amused, and sometimes saddened, by the lack of understanding about agile planning. We hear criticisms like "agile project teams don't plan," or "agile teams won't commit to dates and features." Even Barry Boehm and Richard Turner got it wrong in *Balancing Agility and Discipline: A Guide for the Perplexed* (Addison-Wesley, 2004) when they talk about plan-driven versus agile methods. Actually, Boehm and Turner got the idea right, but the terms wrong. By plan-driven they actually mean "greatly weighting the balance of anticipation versus adaptation toward anticipation," while in agile

methods the weighting is the opposite. The problem with the words "plan-driven" versus "agile" is that it sends entirely the wrong message—that agile teams don't plan. Nothing could be further from the state of the practice. Mike's book sends the right message—planning is an integral part of any agile project. The book contains a wealth of ideas about why planning is so important and how to plan effectively.

First, agile teams do a lot of planning, but it is spread out much more evenly over the entire project. Second, agile teams squarely face the critical factor that many non-agile teams ignore—uncertainty. Is planning important?—absolutely. Is adjusting the plan as knowledge is gained and uncertainty reduced important?—absolutely. I've gone into too many organizations where making outlandish early commitments, and then failing to meet those commitments, was acceptable, while those who tried to be realistic (and understanding uncertainty) were branded as "not getting with the program," or "not being team players." In these companies failure to deliver seems to be acceptable, whereas failure to commit (even to outlandish objectives) is unacceptable. The agile approach, as Mike so ably describes, is focused on actually delivering value and not on making outrageous and unachievable plans and commitments. Agile developers essentially say: We will give you a plan based on what we know today; we will adapt the plan to meet your most critical objective; we will adapt the project and our plans as we both move forward and learn new information; we expect you to understand what you are asking for—that flexibility to adapt to changing business conditions and absolute conformance to original plans are incompatible objectives. *Agile Estimating and Planning* addresses each of those statements.

Returning to the critical issue of managing uncertainty, Mike does a great job of looking at how an agile development process works to reduce ends uncertainty (what do we really want to build) and means uncertainty (how are we going to build it), concurrently. Many traditional planners don't understand a key concept—you can't "plan" away uncertainty. Plans are based on what we know at a given point in time. Uncertainty is another way of expressing what we don't know—about the ends or the means. For most uncertainties (lack of knowledge) the only way to reduce the uncertainty and gain knowledge is to execute—to do something, to build something, to simulate something—and then get feedback. Many project management approaches appear to be "plan, plan, plan-do." Agile approaches are "plan-do-adapt," "plan-do-adapt." The higher a project's uncertainties, the more critical an agile approach is to success.

I'd like to illustrate the "how's" and "why's" of Mike's book by looking at Chapters 4 and 5, which detail how to estimate story points or ideal days and provide an explanation of the pros and cons of each. While I have used both

approaches with clients, Mike's words crystallized my thinking about the benefits of story-point estimation, and I realized that story points are part of an evolution, an evolution toward simplicity. Software development organizations have long looked for an answer to the question, "How big is this piece of software." A home builder can do some reasonable estimating based on square footage. While estimates from builders may vary, the size is fixed (although finish work, material specifications, and more will also impact the estimates) and remains a constant. Software developers have long searched for such a measurement.

In software development we first utilized lines-of-code to size the product (this measure still has its uses today). For much day-to-day planning, lines-of-code proved to be of limited use for a variety of reasons, including the amount of up-front work required to estimate them. Next on the scene came function points (and several similar ideas). Function points eliminated a number of the problems with lines-of-code, but still required a significant amount of up-front work to calculate (you had to estimate inputs, outputs, files, and so on). But what dooms function points from widespread use is their complexity. My guess is that as the complexity of counting has gone up—a quick perusal of the International Function Point User Group (IFPUG) website indicates the degree of that complexity—the usage in the general population has gone down.

However, the need to estimate the "size" of a software project has not diminished. The problem with both historical measures is twofold—they are complex to calculate and they are based on a waterfall approach to development. We still need a size measure, we just need one that is simple to calculate and applicable without going through the entire requirements and design phases.

The two critical differences between story points and either lines-of-code or function points are that they are simpler to calculate and they can be calculated much earlier. Why are they simpler? Because they are based on relative size more than absolute size. Why can they be calculated earlier? Because they are based on relative size more than absolute size. As Mike points out, story-point estimating is about sitting around discussing stories (gaining shared knowledge) and guestimating the relative story size. Relative sizing, as opposed to absolute sizing, goes remarkably quickly. Furthermore, after a few iterations of sizing and delivering, the accuracy of a team's guestimates improves significantly. Mike's description of both the "how" and the "why" of story-point versus ideal days estimating provides keen insight into this critical topic.

Another example of Mike's thoroughness shows up in Chapters 9 to 11, on the prioritization of stories. Mike isn't content with telling us to do the highest value stories first, he actually delves into the key aspects of value: financial benefits, cost, innovation/knowledge, and risk. He carefully defines each of these

aspects of value (including a primer on Net Present Value, Internal Rate of Return, and other financial analysis tools), and then provides several schemes (with varying levels of simplicity) for weighting decisions using these different aspects of value.

Often, people new to agile development think that if you are doing the twelve or nineteen or eight practices of a particular methodology that you are therefore Agile, or Extreme, or Crystal Clear, or whatever. But in reality, you are Agile, Extreme, or otherwise when you know enough about the practices to adapt them to the reality of your own specific situation. Continuous learning and adaptation are core to agile development. What Mike does so well in this book is provide us with the ideas and experience that help take our agile estimating and planning practices to this next level of sophistication. Mike tells us "how" in depth—for example, the material on estimating in story points and ideal days. Mike tells us "why" in depth—for example the pros and cons of both story points and ideal days. While he usually gives us his personal recommendation (he prefers story points), he provides enough information so we feel confident in tailoring practices to specific situations ourselves.

So, this, in the end, identifies Mike's significant contribution to the state of the art—he helps us think through estimating and planning practices at a new depth of knowledge and experience (the how) and then helps us frame decisions about using this new knowledge in adapting these practices to new, unique, or merely specific situations (the why). Of the half-dozen books I regularly recommend to clients, Mike has written two of them. *Agile Estimating and Planning* goes on my "must read" list for those wanting to understand the state of the art in this aspect of agile project management.

Jim Highsmith
Agile Practice Director, Cutter Consortium
Flagstaff, Arizona
August 2005

Foreword

"Agile development won't work at Yahoo! except maybe on small tactical projects, because teams don't do any planning and team members can't estimate their own work. They need to be told what to do."

This is a real quote and something I've heard more than once since I began guiding the rollout of agile at Yahoo! People who don't understand the concepts of agile development think that it's simply a case of eliminating all documentation and planning, giving teams a license to hack. The reality could not be further from the truth.

"Teams don't do any planning." Those who say this forget that an agile team spends half a day every other week coming up with a list of tasks they'll perform in order to deliver some user-valued functionality at the end of the two weeks. That teams spread planning out across a project, rather than doing it all up front, is seen as a lack of planning. It's not, and agile teams at Yahoo! are creating products that please our product managers far more than traditional teams ever did.

"Team members can't estimate their own work and need to be told what to do." This is a classic misperception. Giving a product or project manager sage-like qualities to be able to foresee what others, who are experts in their own work, can really deliver is business suicide. Often this approach is really a way of saying yes to the business when asked to deliver on unrealistic goals. Team members are then forced to work around the clock and cut corners. And we wonder why people are burnt out and morale is so low in our industry.

Estimating and planning are among the topics I get the most questions about, especially from new teams. Having a simple approach to planning, not only for an iteration but for an entire project is invaluable. Product managers have to be concerned with meeting revenue goals and having a predictable release plan. Teams can still be flexible and change course as desired, but it's important to have a roadmap to follow. It's not enough to go fast if you are heading in the wrong direction. Learning how to estimate and plan are some of the most important ingredients for success if you hope to successfully implement agile in your organization.

Mike's estimating and planning class is the most popular of the agile classes we run at Yahoo! It gives teams the skills and tools they need to do just the right amount of planning to optimize results. If you follow Mike's advice, does it really work? Yes. The success of agile in Yahoo! has been incredible. I've had teams return from Mike's class and immediately put his advice into action. We are getting products to market faster and teams genuinely love the agile approach.

Why are agile estimating and planning methods more effective than traditional methods? They concentrate on delivering value and establishing trust between the business and the project teams. Keeping everything highly transparent, and letting the business know of any changes as they come up, means that the business can adapt quickly to make the best decisions. At my last company I saw us go from a state of permanent chaos, where we had an extremely ambitious roadmap but couldn't deliver products, to a predictable state where we could genuinely sign up for projects that we could deliver. The business said they might not always like the answers (they always want things tomorrow, after all), but at least they believed our answers and were not frustrated from feeling that they were being consistently lied to.

This books keeps it real. It doesn't tell you how to become 100% accurate with your estimates. That would be a waste of effort and impossible to achieve. Mike's book doesn't attempt to give you pretty templates to be filled out, instead he makes you think and learn how to approach and solve problems. No project, product, or organization is the same, so learning the thinking and principles are far more important. Mike brings his vast real-world experiences and personality to life in this book. It's real and it's honest. This book definitely belongs at the top of your reading list.

Gabrielle Benefield
Director, Agile Product Development
Yahoo!

Chapter 6

Techniques for Estimating

"Prediction is very difficult,
especially about the future."
—Niels Bohr, Danish physicist

The more effort we put into something, the better the result. Right? Perhaps, but we often need to expend just a fraction of that effort to get *adequate* results. For example, my car is dirty, and I need to wash it. If I wash it myself, I'll spend about an hour on it, which will be enough to wash the exterior, vacuum the interior, and clean the windows. For a one-hour investment, I'll have a fairly clean car.

On the other hand, I could call a car-detailing service and have them wash my car. They'll spend four hours on it. They do everything I do but much more thoroughly. They'll also wax the car, shine the dashboard, and so on. I watched one time, and they used tiny cotton swabs to clean out the little places too small to reach with a rag. That's a lot of effort for slightly better results. For me, the law of diminishing returns kicks in well before I'll use a cotton swab on my car.

We want to remain aware, too, of the diminishing return on time spent estimating. We can often spend a little time thinking about an estimate and come up with a number that is nearly as good as if we had spent a lot of time thinking about it. The relationship between estimate accuracy and effort is shown in Figure 6.1. The curve in this graph is placed according to my experience, corroborated in discussions with others. It is not based on empirical measurement.

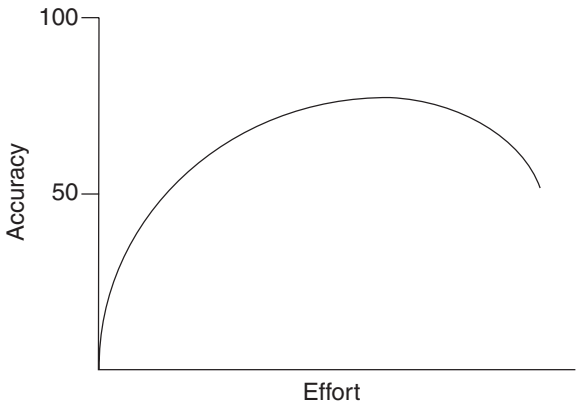

Figure 6.1 Additional estimation effort yields very little value beyond a certain point.

To understand this relationship better, suppose you decide to estimate how many cookies I've eaten in the past year. You could put no effort into the estimate and just take a random guess. Mapping this onto Figure 6.1, you'd be completely to the left on the effort axis, and your estimate would be unlikely to be accurate. You could move to the right on the effort axis by spending a half-hour or so researching national averages for cookie consumption. This would improve your accuracy over the pure guess. If you felt the need to be more accurate, you could do some research—call my friends and family, subpoena my past cookie orders from the Girl Scouts, and so on. You could even follow me around for a day—or, better yet, a month—and then extrapolate your observations into how many cookies you think I eat in a year.

Vary the effort you put into estimating according to purpose of the estimate. If you are trying to decide whether or not to send me a box of cookies as a gift, you do not need a very accurate estimate. If the estimate will be used to make a software build versus buy decision, it is likely enough to determine that the project will take six to twelve months. It may be unnecessary to refine that to the point where you can say it will take seven or eight months.

Look carefully at Figure 6.1, and notice a couple of things. First, no matter how much effort is invested, the estimate is never at the top of the accuracy axis. No matter how much effort you put into an estimate, an estimate is still an estimate. No amount of additional effort will make an estimate perfect. Next, notice how little effort is required to move the accuracy up dramatically from the baseline. As drawn in Figure 6.1, about 10% of the effort gets 50% of the potential accuracy. Finally, notice that eventually, the accuracy of the estimate declines. It is

possible to put too much effort into estimating, with the result being a less accurate estimate.

When starting to plan a project, it is useful to think about where on the curve of Figure 6.1 we wish to be. Many projects try to be very high up the accuracy axis, forcing teams far out on the effort axis even though the benefits diminish rapidly. Often, this is the result of the simplistic view that we can lock down budgets, schedules, and scope and that project success equates to on-time, on-budget delivery of an up-front, precisely planned set of features. This type of thinking leads to a desire for extensive signed requirements documents, lots of up-front analysis work, and detailed project plans that show every task a team can think of. Then, even after all this additional up-front work, the estimates still aren't perfect.

Agile teams, however, choose to be closer to the left in a figure like Figure 6.1. They acknowledge that we cannot eliminate uncertainty from estimates, but they embrace the idea that small efforts are rewarded with big gains. Even though they are less far up the accuracy/effort scale, agile teams can produce more reliable plans because they frequently deliver small increments of fully working, tested, integrated code.

Estimates Are Shared

Estimates are not created by a single individual on the team. Agile teams do not rely on a single expert to estimate. Despite well-known evidence that estimates prepared by those who will do the work are better than estimates prepared by anyone else (Lederer and Prasad 1992), estimates are best derived collaboratively by the team, which includes those who will do the work. There are two reasons for this.

First, on an agile project we tend not to know specifically who will perform a given task. Yes, we may all suspect that the team's database guru will be the one to do the complex stored procedure task that has been identified. However, there's no guarantee that this will be the case. She may be busy when the time comes, and someone else will work on it. So because anyone may work on anything, it is important that everyone have input into the estimate.

Second, even though we may expect the database guru to do the work, others may have something to say about her estimate. Suppose that the team's database guru, Kristy, estimates a particular user story as three ideal days. Someone else on the project may not know enough to program the feature himself, but he may know enough to say, "Kristy, you're nuts; the last time you worked on a feature like that, it took a lot longer. I think you're forgetting how hard it was last

time." At that point Kristy may offer a good explanation of why it's different this time. However, more often than not in my experience, she will acknowledge that she was indeed underestimating the feature.

The Estimation Scale

Studies have shown that we are best at estimating things that fall within one order of magnitude (Miranda 2001; Saaty 1996). Within your town, you should be able to estimate reasonably well the relative distances to things like the nearest grocery store, the nearest restaurant, and the nearest library. The library may be twice as far as the restaurant, for example. Your estimates will be far less accurate if you are asked also to estimate the relative distance to the moon or a neighboring country's capital. Because we are best within a single order of magnitude, we would like to have most of our estimates in such a range.

Two estimation scales I've had good success with are

- 1, 2, 3, 5, and 8
- 1, 2, 4, and 8

There's a logic behind each of these sequences. The first is the Fibonacci sequence.[1] I've found this to be a very useful estimation sequence because the gaps in the sequence become appropriately larger as the numbers increase. A one-point gap from 1 to 2 and from 2 to 3 seems appropriate, just as the gaps from 3 to 5 and from 5 to 8 do. The second sequence is spaced such that each number is twice the number that precedes it. These nonlinear sequences work well because they reflect the greater uncertainty associated with estimates for larger units of work. Either sequence works well, although my slight personal preference is for the first.

Each of these numbers should be thought of as a bucket into which items of the appropriate size are poured. Rather than thinking of work as water being poured into the buckets, think of the work as sand. If you are estimating using 1, 2, 3, 5, and 8, and have a story that you think is just the slightest bit bigger than the other five-point stories you've estimated, it would be OK to put it into the five-point bucket. A story you think is a 7, however, clearly would not fit in the five-point bucket.

1. A number in the Fibonacci sequence is generated by taking the sum of the previous two numbers.

You may want to consider including 0 as a valid number within your estimation range. Although it's unlikely that a team will encounter many user stories or features that truly take no work, including 0 is often useful. There are two reasons for this. First, if we want to keep all features within a 10x range, assigning nonzero values to tiny features will limit the size of largest features. Second, if the work truly is closer to 0 than 1, the team may not want the completion of the feature to contribute to its velocity calculations. If the team earns one point in this iteration for something truly trivial, in the next iteration their velocity will either drop by one or they'll have to earn that point by doing work that may not be as trivial.

If the team does elect to include 0 in their estimation scale, everyone involved in the project (especially the product owner) needs to understand that $13 \times 0 \neq 0$. I've never had the slightest problem explaining this to product owners, who realize that a 0-point story is the equivalent of a free lunch. However, they also realize there's a limit to the number of free lunches they can get in a single iteration. An alternative to using 0 is to group very small stories and estimate them as a single unit.

Some teams prefer to work with larger numbers, such as 10, 20, 30, 50, and 100. This is fine, because these are also within a single order of magnitude. However, if you go with larger numbers, such as 10 to 100, I still recommend that you pre-identify the numbers you will use within that range. Do not, for example, allow one story to be estimated at 66 story points or ideal days and another story to be estimated at 67. That is a false level of precision, and we cannot discern a 1.5% difference in size. It's acceptable to have one-point differences between values such as 1, 2, and 3. As percentages, those differences are much larger than between 66 and 67.

User Stories, Epics, and Themes

Although in general, we want to estimate user stories whose sizes are within one order of magnitude, this cannot always be the case. If we are to estimate everything within one order of magnitude, it would mean writing all stories at a fairly fine-grained level. For features that we're not sure we want (a preliminary cost estimate is desired before too much investment is put into them) or for features that may not happen in the near future, it is often desirable to write one much larger user story. A large user story is sometimes called an *epic*.

Additionally, a set of related user stories may be combined (usually by a paper clip if working with note cards) and treated as a single entity for either

estimating or release planning. Such a set of user stories is referred to as a *theme*. An epic, by its very size alone, is often a theme on its own.

By aggregating some stories into themes and writing some stories as epics, a team is able to reduce the effort they'll spend on estimating. However, it's important that they realize that estimates of themes and epics will be more uncertain than estimates of the more specific, smaller user stories.

User stories that will be worked on in the near future (the next few iterations) need to be small enough that they can be completed in a single iteration. These items should be estimated within one order of magnitude. I use the sequence 1, 2, 3, 5, and 8 for this.

User stories or other items that are likely to be more distant than a few iterations can be left as epics or themes. These items can be estimated in units beyond the 1 to 8 range I recommend. To accommodate estimating these larger items I add 13, 20, 40, and 100 to my preferred sequence of 1, 2, 3, 5, and 8.

Deriving an Estimate

The three most common techniques for estimating are

- Expert opinion
- Analogy
- Disaggregation

Each of these techniques may be used on its own, but the techniques should be combined for best results.

Expert Opinion

If you want to know how long something is likely to take, ask an expert. At least, that's one approach. In an expert opinion-based approach to estimating, an expert is asked how long something will take or how big it will be. The expert relies on her intuition or gut feel and provides an estimate.

This approach is less useful on agile projects than on traditional projects. On an agile project, estimates are assigned to user stories or other user-valued functionality. Developing this functionality is likely to require a variety of skills normally performed by more than one person. This makes it difficult to find suitable experts who can assess the effort across all disciplines. On a traditional project

for which estimates are associated with tasks, this is not as significant of a problem, because each task is likely performed by one person.

A nice benefit of estimating by expert opinion is that it usually doesn't take very long. Typically, a developer reads a user story, perhaps asks a clarifying question or two, and then provides an estimate based on her intuition. There is even evidence that says this type of estimating is more accurate than other, more analytical approaches (Johnson et al. 2000).

Analogy

An alternative to expert opinion comes in the form of estimating by analogy, which is what we're doing when we say, "This story is a little bigger than that story." When estimating by analogy, the estimator compares the story being estimated with one or more other stories. If the story is twice the size, it is given an estimate twice as large. There is evidence that we are better at estimating relative size than we are at estimating absolute size (Lederer and Prasad 1998; Vicinanza et al. 1991).

When estimating this way, you do not compare all stories against a single baseline or universal reference. Instead, you want to estimate each new story against an assortment of those that have already been estimated. This is referred to as triangulation. To triangulate, compare the story being estimated against a couple of other stories. To decide if a story should be estimated at five story points, see if it seems a little bigger than a story you estimated at three and a little smaller than a story you estimated at eight.

Disaggregation

Disaggregation refers to splitting a story or feature into smaller, easier-to-estimate pieces. If most of the user stories to be included in a project are in the range of two to five days to develop, it will be very difficult to estimate a single story that may be 100 days. Not only are large things notoriously more difficult to estimate, but also in this case there will be very few similar stories to compare. Asking "Is this story fifty times as hard as that story" is a very different question from "Is this story about one-and-a-half times that one?"

The solution to this, of course, is to break the large story or feature into multiple smaller items and estimate those. However, you need to be careful not to go too far with this approach. The easiest way to illustrate the problem is with a nonsoftware example. Let's use disaggregation to estimate my golf score this weekend. Assume the course I am playing has eighteen holes each with a par of four. (If you're unfamiliar with golf scoring, the par score is the number of shots

it should take a decent player to shoot his ball into the cup at the end of the hole.)

To estimate by disaggregation, we need to estimate my score for each hole. There's the first hole, and that's pretty easy, so let's give me a three on that. But then I usually hit into the lake on the next hole, so that's a seven. Then there's the hole with the sandtraps; let's say a five. And so on. However, if I'm mentally re-creating an entire golf course it is very likely I'll forget one of the holes. Of course, in this case I have an easy check for that, as I know there must be eighteen individual estimates. But when disaggregating a story, there is no such safety check.

Not only does the likelihood of forgetting a task increase if we disaggregate too far, but summing estimates of lots of small tasks also leads to problems. For example, for each of the 18 holes, I may estimate my score for that hole to be in the range 3 to 8. Multiplying those by 18 gives me a full round range of 54 to 144. There's no chance that I'll do that well or that poorly. If asked for an estimate of my overall score for a full round, I'm likely to say anywhere from 80 to 120, which is a much smaller range and a much more useful estimate.

Specific advice on splitting user stories is provided in Chapter 12, "Splitting User Stories."

Planning Poker

The best way I've found for agile teams to estimate is by playing planning poker (Grenning 2002). Planning poker combines expert opinion, analogy, and disaggregation into an enjoyable approach to estimating that results in quick but reliable estimates.

Participants in planning poker include all of the developers on the team. Remember that *developers* refers to all programmers, testers, database engineers, analysts, user interaction designers, and so on. On an agile project, this will typically not exceed ten people. If it does, it is usually best to split into two teams. Each team can then estimate independently, which will keep the size down. The product owner participates in planning poker but does not estimate.

At the start of planning poker, each estimator is given a deck of cards. Each card has written on it one of the valid estimates. Each estimator may, for example, be given a deck of cards that reads 0, 1, 2, 3, 5, 8, 13, 20, 40, and 100. The cards should be prepared prior to the planning poker meeting, and the numbers should be large enough to see across a table. Cards can be saved and used for the next planning poker session.

For each user story or theme to be estimated, a moderator reads the description. The moderator is usually the product owner or an analyst. However, the moderator can be anyone, as there is no special privilege associated with the role. The product owner answers any questions that the estimators have. However, everyone is asked to remain aware of the effort/accuracy curve (Figure 6.1). The goal in planning poker is not to derive an estimate that will withstand all future scrutiny. Rather, the goal is to be somewhere well on the left of the effort line, where a valuable estimate can be arrived at cheaply.

After all questions are answered, each estimator privately selects a card representing his or her estimate. Cards are not shown until each estimator has made a selection. At that time, all cards are simultaneously turned over and shown so that all participants can see each estimate.

It is very likely at this point that the estimates will differ significantly. This is actually good news. If estimates differ, the high and low estimators explain their estimates. It's important that this does not come across as attacking those estimators. Instead, you want to learn what they were thinking about.

As an example, the high estimator may say, "Well, to test this story, we need to create a mock database object. That might take us a day. Also, I'm not sure if our standard compression algorithm will work, and we may need to write one that is more memory efficient." The low estimator may respond, "I was thinking we'd store that information in an XML file—that would be easier than a database for us. Also, I didn't think about having more data—maybe that will be a problem."

The group can discuss the story and their estimates for a few more minutes. The moderator can take any notes she thinks will be helpful when this story is being programmed and tested. After the discussion, each estimator re-estimates by selecting a card. Cards are once again kept private until everyone has estimated, at which point they are turned over at the same time.

In many cases, the estimates will already converge by the second round. But if they have not, continue to repeat the process. The goal is for the estimators to converge on a single estimate that can be used for the story. It rarely takes more than three rounds, but continue the process as long as estimates are moving closer together. It isn't necessary that everyone in the room turns over a card with exactly the same estimate written down. If I'm moderating an estimation meeting, and on the second round four estimators tell me 5, 5, 5, and 3, I will ask the low estimator if she is OK with an estimate of 5. Again, the point is not absolute precision but reasonableness.

The Right Amount of Discussion

Some amount of preliminary design discussion is necessary and appropriate when estimating. However, spending too much time on design discussions sends a team too far up the effort/accuracy curve of Figure 6.1. Here's an effective way to encourage some amount of discussion but make sure that it doesn't go on too long.

Buy a two-minute sand timer, and place it in the middle of the table where planning poker is being played. Anyone in the meeting can turn the timer over at any time. When the sand runs out (in two minutes), the next round of cards is played. If agreement isn't reached, the discussion can continue. But someone can immediately turn the timer over, again limiting the discussion to two minutes. The timer rarely needs to be turned over more than twice. Over time this helps teams learn to estimate more rapidly.

Smaller Sessions

It is possible to play planning poker with a subset of the team, rather than involving everyone. This isn't ideal but may be a reasonable option, especially if there are many, many items to be estimated, as can happen at the start of a new project.

The best way to do this is to split the larger team into two or three smaller teams, each of which must have at least three estimators. It is important that each of the teams estimates consistently. What your team calls three story points or ideal days had better be consistent with what my team calls the same. To achieve this, start all teams together in a joint planning poker session for an hour or so. Have them estimate ten to twenty stories. Then make sure each team has a copy of these stories and their estimates and that they use them as baselines for estimating the stories they are given to estimate.

When to Play Planning Poker

Teams will need to play planning poker at two different times. First, there will usually be an effort to estimate a large number of items before the project officially begins or during its first iterations. Estimating an initial set of user stories may take a team two or three meetings of from one to three hours each. Naturally, this will depend on how many items there are to estimate, the size of the team, and the product owner's ability to clarify the requirements succinctly.

Second, teams will need to put forth some ongoing effort to estimate any new stories that are identified during an iteration. One way to do this is to plan

to hold a very short estimation meeting near the end of each iteration. Normally, this is quite sufficient for estimating any work that came in during the iteration, and it allows new work to be considered in the prioritization of the coming iteration.

Alternatively, Kent Beck suggests hanging an envelope on the wall with all new stories placed in the envelope. As individuals have a few spare minutes, they will grab a story or two from the envelope and estimate them. Teams will establish a rule for themselves, typically that all stories must be estimated by the end of the day or by the end of the iteration. I like the idea of hanging an envelope on the wall to contain unestimated stories. However, I'd prefer that when someone has a few spare minutes to devote to estimating, he find at least one other person and that they estimate jointly.

Why Planning Poker Works

Now that I've described planning poker, it's worth spending a moment on some of the reasons why it works so well.

First, planning poker brings together multiple expert opinions to do the estimating. Because these experts form a cross-functional team from all disciplines on a software project, they are better suited to the estimation task than anyone else. After completing a thorough review of the literature on software estimation, Jørgensen (2004) concluded that "the people most competent in solving the task should estimate it."

Second, a lively dialogue ensues during planning poker, and estimators are called upon by their peers to justify their estimates. This has been found to improve the accuracy of the estimate, especially on items with large amounts of uncertainty (Hagafors and Brehmer 1983). Being asked to justify estimates has also been shown to result in estimates that better compensate for missing information (Brenner et al. 1996). This is important on an agile project because the user stories being estimated are often intentionally vague.

Third, studies have shown that averaging individual estimates leads to better results (Hoest and Wohlin 1998) as do group discussions of estimates (Jørgensen and Moløkken 2002). Group discussion is the basis of planning poker, and those discussions lead to an averaging of sorts of the individual estimates.

Finally, planning poker works because it's fun.

Summary

Expending more time and effort to arrive at an estimate does not necessarily increase the accuracy of the estimate. The amount of effort put into an estimate should be determined by the purpose of that estimate. Although it is well known that the best estimates are given by those who will do the work, on an agile team we do not know in advance who will do the work. Therefore, estimating should be a collaborative activity for the team.

Estimates should be on a predefined scale. Features that will be worked on in the near future and that need fairly reliable estimates should be made small enough that they can be estimated on a nonlinear scale from 1 to 10 such as 1, 2, 3, 5, and 8 or 1, 2, 4, and 8. Larger features that will most likely not be implemented in the next few iterations can be left larger and estimated in units such as 13, 20, 40, and 100. Some teams choose to include 0 in their estimation scale.

To arrive at an estimate, we rely on expert opinion, analogy, and disaggregation. A fun and effective way of combining these is planning poker. In planning poker, each estimator is given a deck of cards with a valid estimate shown on each card. A feature is discussed, and each estimator selects the card that represents his or her estimate. All cards are shown at the same time. The estimates are discussed and the process repeated until agreement on the estimate is reached.

Discussion Questions

1. How good are your estimates today? Which techniques do you primarily rely on: expert opinion, analogy, or disaggregation?
2. Which estimation scale do you prefer? Why?
3. Who should participate in planning poker on your project?

Chapter 7

Re-Estimating

"There's no sense in being precise
when you don't even know what you're talking about."
—John von Neumann

One of the most common questions about estimating with story points or ideal days is "When do we re-estimate?" To arrive at an answer it is critical to remember that story points and ideal days are estimates of the overall size and complexity of the feature being implemented. Story points in particular are not an estimate of the amount of time it takes to implement a feature, even though we often fall into the trap of thinking of them as such. The amount of time that implementing a feature will take is a function of its size (estimated in either ideal days or story points) *and* the team's rate of progress (as reflected by its velocity).

If we keep in mind that story points and ideal time estimate size, it's easier to see that we should re-estimate only when we believe a story's relative size has changed. When working with story points or ideal time, we do not re-estimate solely because a story took longer to implement than we thought. The best way to see this is through some examples.

Introducing the SwimStats Website

Throughout the rest of this chapter and some of the upcoming chapters, we will be working on SwimStats, a hypothetical website for swimmers and swim coaches. SwimStats will be sold as a service to competitive age-group, school,

and college swim teams. Coaches will use it to keep track of their roster of swimmers, organize workouts, and prepare for meets; swimmers will use the site to see meet results, check their personal records, and track improvements over time. Officials at swim meets will enter results into the system. A sample screen from SwimStats is shown in Figure 7.1.

Boulder Valley Summer Swim League | Search | League | Dual Meets | Community | News

Savannah Cohn — Lafayette Seals — Gender: F

Leagues (Email us.)

League	Team(s)	Season start date	Events Swam	First	Second	Third
2005 BVSSL	Lafayette Seals	05/15/2005	18	12	4	2

Click on Team Name, Swimmer's Name or Meet Date to view more information. Meets are sorted with most recent meet on top.

Indicates Winner

07/9/2005 Lafayette Seals at Meadow Hills

Event Number	Age Group	Event	Time	Place	Points
14	Girls 9-10	100 Free	2:09.48	1	6
34	Girls 9-10	50 Back	1:06.41	1	6
64	Girls 9-10	50 Breast	1:13.97	1	6

07/16/2003 Lafayette Seals host Huntington Beach Blue Dolphins

Event Number	Age Group	Event	Time	Place	Points
14	Girls 9-10	100 Free	2:03.48	1	6
34	Girls 9-10	50 Back	1:04.41	1	6
64	Girls 9-10	50 Breast	1:13.43	1	6

Figure 7.1 One screen from the SwimStats website.

When Not to Re-Estimate

Using SwimStats as an example, let's first look briefly at a case when we should not re-estimate. Suppose we have the stories shown in Table 7.1. At the conclusion of the first iteration, the first two stories are complete. The team doesn't feel good about this because they thought they would complete twice as many points (twelve rather than six) per iteration. They decide that each of those stories was twice as big or complex as initially thought, which is why they took twice as long as expected to complete. The team decides to double the number of story points associated with each. This means that their velocity was twelve (two six-point stories), which they feel better about.

However, before the project started, the team considered all four stories of Table 7.1 to be of equivalent size and complexity, so each was estimated at three

story points. Because they still believe these stories are equivalent, the estimates for Stories 3 and 4 must now also be doubled. The team has given themselves more points for the stories they completed, which doubled their velocity. But, because they also doubled the amount of work remaining in the project, their situation is the same as if they'd left all the estimates at three and velocity at six.

Table 7.1 Some Stories and Estimates for the SwimStats Website

Story ID	Story	Estimate
1	As a coach, I can enter the names and demographic information for all swimmers on my team.	3
2	As a coach, I can define practice sessions.	3
3	As a swimmer, I can see all of my times for a specific event.	3
4	As a swimmer, I can update my demographics information.	3

Velocity Is the Great Equalizer

What's happened here is that velocity is the great equalizer. Because the estimate for each feature is made relative to the estimates for other features, it does not matter if our estimates are correct, a little incorrect, or a lot incorrect. What matters is that they are consistent. We cannot simply roll a die and assign that number as the estimate to a feature. However, as long as we are consistent with our estimates, measuring velocity over the first few iterations will allow us to hone in on a reliable schedule.

Let's look at another example. Suppose a project consists of fifty user stories, each of which is estimated as one story point. For simplicity, suppose that I am the only person working on this project and that I expect I can complete one story point per work day. So on a two-week iteration, I expect to finish ten stories and have a velocity of ten. Further, I expect to finish the project in five iterations (ten weeks). However, after the first iteration, rather than having completed ten stories, I've completed only five. If I let velocity take care of correcting my misperceptions, I will realize that the project will take ten iterations, because my velocity is half of what I'd planned.

What happens, though, if I re-estimate? Suppose I re-estimate the five completed stories, assigning each an estimate of two. My velocity is now ten (five completed stories, each re-estimated at two), and forty-five points of work remain. With a velocity of ten and with forty-five points remaining, I expect to finish the project in 4.5 iterations. The problem with this is that I am mixing

revised and original estimates. Using hindsight, I have re-estimated the completed stories at two points each. Unfortunately, when still looking forward at the remaining forty-five stories, I cannot predict which of those one-point stories I will want to say were worth two points in hindsight.

When to Re-Estimate

Let's continue working on the SwimStats website, this time with the user stories and estimates shown in Table 7.2.

Table 7.2 Initial Estimates for Some SwimStats Stories

Story ID	Story	Estimate
1	As a swimmer, I can see a line chart of my times for a particular event.	3
2	As a coach, I can see a line chart showing the progress over the season of all of my swimmers in a particular event.	5
3	As a swimmer, I can see a pie chart showing how many first, second, third, and lower places I've finished in.	3
4	As a coach, I can see a text report showing each swimmer's best time in each event.	3
5	As a coach, I can upload meet results from a file exported from the timing system used at the meet.	3
6	As a coach, I can have the system recommend who should swim in each event subject to restrictions about how many events a swimmer can participate in.	5

The first three of these stories each has to do with displaying a chart for the user. Suppose the team has planned the first iteration to include Stories 1, 2, and 6 from Table 7.2. Their planned velocity is thirteen. However, at the end of the iteration they have finished only Stories 1 and 6. They say they got less done than expected because Story 1 was much harder than expected and that it should have been "at least a six." Suppose that rather than one difficult story, the team has completely underestimated the general difficulty of displaying charts. In that case, if Story 1 turned out to be twice as big as expected, we can expect the same of Stories 2 and 3.

Let's see how this plays out across three scenarios.

Scenario 1: No Re-Estimating

In this scenario, we will leave all estimates alone. The team achieved a velocity of eight points in the last iteration. That leads us to the expectation that they'll average eight points in the upcoming iterations. However, the team knows they cannot complete Stories 2 and 3 in a single iteration, even though they represent only eight points. Because each of those stories involves charting, and because the team expects each charting story to be twice as big as its current estimate (just like Story 1 was), the team concludes that they cannot do Stories 2 and 3 in one iteration. It's eight points, but it's too much.

Scenario 2: Re-Estimating the Finished Story

In this scenario, let's see if adjusting only the estimate of Story 1 fixes this problem. After finishing the iteration, the team felt that Story 1 was twice as big as had been expected. So they decide to re-estimate it at six instead of three. That means that velocity for the prior iteration was eleven—six points for Story 1 and five points for Story 6.

Because no other stories are re-estimated, the team plans its next iteration to comprise Stories 2, 3, and 4. These stories are worth eleven points, the same amount of work completed in the prior iteration. However, they run into the same problem as in the first scenario: Stories 2 and 3 will probably take twice as long as expected, and the team will not be able to average eleven points per iteration, as expected.

Scenario 3: Re-Estimating When Relative Size Changes

In this scenario, the team re-estimates each of the charting stories. The estimates for Stories 1, 2, and 3 are double what is shown in Table 7.2. As in the second scenario, velocity for the first iteration is eleven—six points for Story 1 and five points for Story 6. Because velocity was eleven in the first iteration, the team expects approximately that velocity in the next iteration. However, when they plan their next iteration, only Story 2 will be selected. This story, initially estimated as five, was doubled to ten and is so big there is no room for an additional story.

Re-estimating was helpful only in this third scenario. This means that you should re-estimate a story only when its relative size has changed.

Re-Estimating Partially Completed Stories

You may also wish to re-estimate when the team finishes only a portion of a story during an iteration. Suppose the team has been working on a story that says, "As a coach, I can have the system recommend who should swim in each event." This story is initially estimated as five points, but it is deceptively complex.

Teams in a swim meet receive points based on the finishing places of the swimmers. However, planning for a swim meet is not as easy as putting the team's fastest swimmer for each event into that event. Each swimmer is limited in the number of events he or she can swim. This means we may not elect to have Savannah swim the 100-meter backstroke because we need her more in the 100-meter breaststroke. So suppose the team reaches the end of the iteration, and the system can optimize the assignment of swimmers to individual events. However, the team has not begun to think about how to assign swimmers to relay events. How many points should the team count toward the velocity of the current iteration? How many points should they assign to the remaining work?

First, let me point out that I'm generally in favor of an all-or-nothing stance toward counting velocity: if a story is done (coded, tested, and accepted by the product owner), the team earns all the points, but if anything on the story isn't done, they earn nothing. At the end of an iteration, this is the easiest case to assess: If everything is done, they get all the points; if anything is missing, they get no points. If the team is likely to take on the remaining portion of the story in the next iteration, this works well. Their velocity in the first iteration is a bit lower than expected because they got no credit for partially completing a story. In the second iteration, however, their velocity will be higher than expected because they'll get all of the points, even though some work had been completed prior to the start of the iteration. This works well as long as everyone remembers that we're mostly interested in the team's average velocity over time, not in whether velocity jumped up or down in a given iteration.

However, in some cases the unfinished portion of a story may not be done in the next iteration. In these cases it can be appropriate to allow the team to take partial credit for the completed portion of the story. The remaining story (which is a subset of the initial story) is re-estimated based on the team's current knowledge. In this case, the original story was estimated at five points. If the team feels that the subset they completed (scheduling individual events) is equivalent to three points or ideal days, they will give themselves that much credit. The unfinished portion of the original story in this case could be rewritten to be "As a coach, I can have the system recommend who should swim in each relay." The team could then estimate that smaller story relative to all other stories. The combined estimates would not need to equal the original estimate of five.

However, the two best solutions to allocating points for incomplete stories are not to have any incomplete stories and to use sufficiently small stories that partial credit isn't an issue.

The Purpose of Re-Estimating

Do not become overly concerned with the need to re-estimate. Whenever the team feels one or more stories are misestimated relative to other stories, re-estimate as few stories as possible to bring the relative estimates back in line. Use re-estimating as a learning experience for estimating future user stories. As Tom Poppendieck has taught me, "Failure to learn is the only true failure." Learn from each re-estimated story, and turn the experience into a success.

Summary

Remembering that story points and ideal days are estimates of the size of a feature helps you know when to re-estimate. You should re-estimate only when your opinion of the relative size of one or more stories has changed. Do not re-estimate solely because progress is not coming as rapidly as you'd expected. Let velocity, the great equalizer, take care of most estimation inaccuracies.

At the end of an iteration, I do not recommend giving partial credit for partially finished user stories. My preference is for a team to count the entire estimate toward their velocity (if they completely finished and the feature has been accepted by the product owner) or for them to count nothing toward their story otherwise. However, the team may choose to re-estimate partially complete user stories. Typically, this will mean estimating a user story representing the work that was completed during the iteration and one or more user stories that describe the remaining work. The sum of these estimates does not need to equal the initial estimate.

Discussion Questions

1. How does velocity correct bad estimates?
2. Why should you re-estimate only when the relative size has changed? Identify some examples on a current or recent project when the relative size of one or more features changed.

Chapter 9

Prioritizing Themes

"The indispensable first step to getting what you want is this: Decide what you want."
—Ben Stein

There is rarely, if ever, enough time to do everything. So we prioritize. The responsibility for prioritizing is shared among the whole team, but the effort is led by the product owner. Unfortunately, it is generally difficult to estimate the value of small units of functionality, such as a single user story. To get around this, individual user stories or features are aggregated into *themes*. Stories and themes are then prioritized relative to one another for the purpose of creating a release plan. Themes should be selected such that each defines a discrete set of user- or customer-valued functionality. For example, in developing the SwimStats website, we would have themes such as these:

- Keep track of all personal records and let swimmers view them.
- Allow coaches to assign swimmers to events optimally and predict the team score of a meet.
- Allow coaches to enter practice activities and track practice distances swum.
- Integrate with popular handheld computers for use at the pool.
- Import and export data.
- Allow officials to track event results and score a meet.

Each of these themes has tangible value to the users of the software. And it would be possible to put a monetary value on each. With some research we could determine that support for handheld computers is likely to result in ¤150,000 of new sales. We could compare that with the expected ¤200,000 in new sales if the next version can be used for scoring a swim meet. We could then prioritize those themes. There is more to prioritizing, however, than simply considering the monetary return from each new set of features.

Factors in Prioritization

Determining the value of a theme is difficult, and product owners on agile projects are often given the vague and mostly useless advice of "prioritize on business value." This may be great advice at face value, but what is *business value?* To provide a more practical set of guidelines for prioritizing, in this chapter we will look at four factors that must be considered when prioritizing the development of new capabilities.

1. The financial value of having the features.
2. The cost of developing (and perhaps supporting) the new features.
3. The amount and significance of learning and new knowledge created by developing the features.
4. The amount of risk removed by developing the features.

Because most projects are undertaken either to save or to make money, the first two factors often dominate prioritization discussions. However, proper consideration of the influence of learning and risk on the project is critical if we are to prioritize optimally.

Value

The first factor in prioritizing work is the financial value of the theme. How much money will the organization make or save by having the new features included in the theme? This alone is often what is meant when product owners are given the advice to "prioritize on business value."

Often, an ideal way to determine the value of a theme is to estimate its financial impact over a period of time—usually the next few months, quarters, or possibly years. This can be done if the product will be sold commercially, as, for example, a new word processor or a calculator with embedded software would

be. It can also be done for applications that will be used within the organization developing them. Chapter 10, "Financial Prioritization," describes various approaches to estimating the financial value of themes.

It can be difficult to estimate the financial return on a theme. Doing so usually involves estimating the number of new sales, the average value of a sale (including follow-on sales and maintenance agreements), the timing of sales increases, and so on. Because of the complexity in doing this, it is often useful to have an alternate method for estimating value. Because the value of a theme is related to the desirability of that theme to new and existing users, it is possible to use nonfinancial measures of desirability to represent value. This will be the subject of Chapter 11, "Prioritizing Desirability."

Cost

Naturally, the cost of a feature is a huge determinant in the overall priority of a feature. Many features seem wonderful until we learn their cost. An important, yet often overlooked, aspect of cost is that the cost can change over time. Adding support for internationalization today may take four weeks of effort; adding it in six months may take six weeks. So we should add it now, right? Maybe. Suppose we spend four weeks and do it now. Over the next six months, we may spend an additional three weeks changing the original implementation based on knowledge gained during that six months. In that case, we would have been better off waiting. Or what if we spend four weeks now and later discover that a simpler and faster implementation would have been adequate? The best way to reduce the cost of change is to implement a feature as late as possible—effectively when there is no more time for change.

Themes often seem worthwhile when viewed only in terms of the time they will take. As trite as it sounds, it is important to remember that time costs money. Often, the best way to do this while prioritizing is to do a rough conversion of story points or ideal days into money. Suppose you add up the salaries for everyone involved in a project over the past twelve weeks and come up with ¤150,000. This includes the product owner and the project manager, as well as all of the programmers, testers, database engineers, analysts, user interface designers, and so on. During those twelve weeks, the team completed 120 story points. We can tell that at a total cost of ¤150,000, 120 story points cost ¤1,250 each. Suppose a product owner is trying to decide whether thirty points of functionality should be included in the next release. One way for her to decide is to ask herself whether the new functionality is worth an investment of ¤37,500 ($30 \times 1{,}250 = 37{,}500$).

Chapter 10, "Financial Prioritization," will have much more to say about cost and about prioritizing based on financial reward relative to cost.

New Knowledge

On many projects, much of the overall effort is spent in the pursuit of new knowledge. It is important that this effort be acknowledged and considered fundamental to the project. Acquiring new knowledge is important because at the start of a project, we never know everything that we'll need to know by the end of the project. The knowledge that a team develops can be classified into two areas:

- Knowledge about the product
- Knowledge about the project

Product knowledge is knowledge about *what* will be developed. It is knowledge about the features that will be included and about those that will not be included. The more product knowledge a team has, the better able they will be to make decisions about the nature and features of the product.

Project knowledge, by contrast, is knowledge about *how* the product will be created. Examples include knowledge about the technologies that will be used, about the skills of the developers, about how well the team functions together, and so on.

The flip side of acquiring knowledge is reducing uncertainty. At the start of a project there is some amount of uncertainty about what features the new product should contain. There is also uncertainty about how we'll build the product. Laufer (1996) refers to these types of uncertainty as *end uncertainty* and *means uncertainty*. End uncertainty is reduced by acquiring more knowledge about the product; means uncertainty is reduced through acquiring more knowledge about the project.

A project following a waterfall process tries to eliminate all uncertainty about what is being built before tackling the uncertainty of how it will be built. This is the origin of the common advice that analysis is about what will be built, and design is about how it will be built. Figure 9.1 shows both the waterfall and agile views of removing uncertainty.

On the waterfall side of Figure 9.1, the downward arrow shows a traditional team's attempt to eliminate all end uncertainty at the start of the project. This means that before they begin developing, there will be no remaining uncertainty about the end that is being pursued. The product is fully defined. The rightward arrow on the waterfall side of Figure 9.1 shows that means uncertainty (about

how the product will be built) is reduced over time as the project progresses. Of course, the complete up-front elimination of all end uncertainty is unachievable. Customers and users are uncertain about exactly what they need until they begin to see parts of it. They can then successively elaborate their needs.

Contrast this view with the agile approach to reducing uncertainty, which is shown on the right side of Figure 9.1. Agile teams acknowledge that it is impossible at the start of a project to eliminate all uncertainty about what the product is to be. Parts of the product need to be developed and shown to customers, feedback needs to collected, opinions refined, and plans adjusted. This takes time. While this is occurring the team will also be learning more about how they will develop the system. This leads to simultaneously reducing both end and means uncertainty, as shown in the agile view in Figure 9.1.

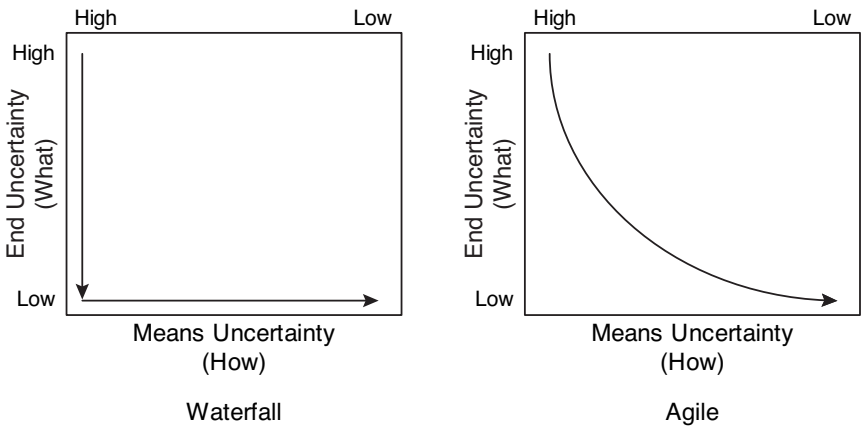

Figure 9.1 Traditional and agile views of reducing uncertainty. Adapted from Laufer (1996).

I've drawn the curve in the agile side of Figure 9.1 to show a preference toward early reduction of end uncertainty. Why didn't I draw a straight line or one that favors early reduction of means uncertainty? I drew the line as I did to reflect the importance of reducing uncertainty about what a product should be as early as possible. End uncertainty does not need to be eliminated at the outset (as hoped for in the traditional view), and it cannot be. However, one of the greatest risks to most projects is the risk of building the wrong product. This risk can be dramatically reduced by developing early those features that will best allow us to get working software in front of or in the hands of actual users.

Risk

Closely aligned with the concept of new knowledge is the final factor in prioritization: risk. Almost all projects contain tremendous amounts of risk. For our purposes, a risk is anything that has not yet happened but might and that would jeopardize or limit the success of the project. There are many different types of risk on projects, including:

- Schedule risk ("We might not be done by October")
- Cost risk ("We might not be able to buy hardware for the right price")
- Functionality risk ("We might not be able to get that to work")

Additionally, risks can be classified as either technological or business risks.

A classic struggle exists between the high-risk and the high-value features of a project. Should a project team start by focusing on high-risk features that could derail the entire project? Or should a project team focus on what Tom Gilb (1988) called the "juicy bits," the high-value features that will deliver the most immediate bang for the customer's buck?

To choose among them, let's consider the drawbacks of each approach. The risk-driven team accepts the chance that work they perform will turn out to be unneeded or of low value. They may develop infrastructural support for features that turn out unnecessary as the product owner refines her vision of the project based on what she learns from users as the project progresses. On the other hand, a team that focuses on value to the exclusion of risk may develop a significant amount of an application before hitting a risk that jeopardizes the delivery of the product.

The solution, of course, is to give neither risk nor value total supremacy when prioritizing. To prioritize work optimally, it is important to consider both risk and value. Consider Figure 9.2, which maps the relationship between the risk and value of a feature into four quadrants. At the top right are high-risk, high-value features. These features are highly desirable to the customer but possess significant development risk. Perhaps features in this quadrant rely on unproven technologies, integration with unproven subcontractors, technical innovation (such as the development of a new algorithm), or any of a number of similar risks. At the bottom right are features that are equally desirable but that are less risky. Whereas features in the right half of Figure 9.2 are highly desirable, features falling in the left half are of lower value.

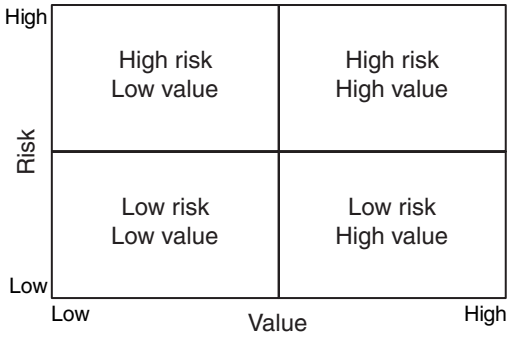

Figure 9.2 The four quadrants of the risk–value relationship.

The appropriate development sequence for the features is shown in Figure 9.3. The high-value, high-risk features should be developed first. These features deliver the most value, and working on them eliminates significant risks. Next are the high-value, low-risk features. These features offer as much value as the first set, but they are less risky. Therefore, they can be done later in the schedule. Because of this, use the guideline to work first on high-value features, but use risk as a tie-breaker.

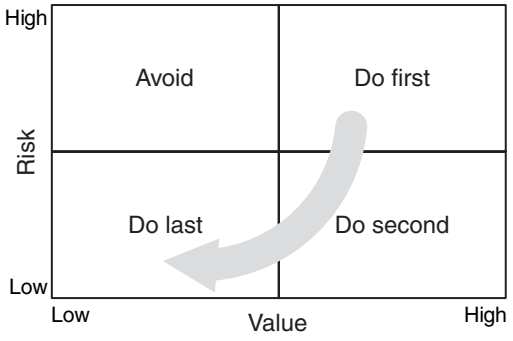

Figure 9.3 Combining risk and value in prioritizing features.

Next are the low-value, low-risk features. These are sequenced third because they will have less impact on the total value of the product if they are dropped, and because they are low risk.

Finally, features that deliver low value, but are high risk, are best avoided. Defer work on all low-value features, especially those that are also high risk. Try

to defer low-value, high-risk items right out of the project. There is no reason to take on a high degree of risk for a feature of limited value. Be aware that a feature's risk and value profile changes over time. A low-value, low-risk feature in the Avoid quadrant of Figure 9.3 today could be in the Do first quadrant six months from now if all other features have been finished.

Combining the Four Factors

To combine the four prioritization factors, think first about the value of the feature relative to what it would cost to develop today. This gives you an initial priority order for the themes. Those themes with a high value-to-cost ratio are those that should be done first.

Next, think of the other prioritization factors as moving themes forward or backward. Suppose that based on its value and cost, a theme is of medium priority. Therefore, the team would tend to work on this theme midway through the current release. However, the technology needed to develop this story is very risky. This would move the theme forward in priority and on the schedule.

It's not necessary that this initial ranking followed by shifting forward and back be a formal activity. It can (and often does) take place entirely in the head of the product owner. The product owner will then typically present her priorities to the team, who may entreat the product owner to alter priorities slightly based on its assessment of the themes.

Some Examples

To make sure that these four prioritization factors are practical and useful, let's see how they can be applied to two typical prioritization challenges: infrastructure and user interface design. In the following sections, I'll consider a theme and show how these prioritization factors can be applied.

Infrastructure

One common prioritization challenge comes in the form of developing the infrastructure or architectural elements of an application. As an example, consider a security framework that will be used throughout an application. Considered solely on the merits of the value it delivers to customers, a security framework is unlikely to be prioritized into the early iterations of a project. After all, even though security is critical to many applications, most applications are not

generally purchased solely because of how secure they are. The application must do something before security is relevant.

The next prioritization factor is cost. Adding a security framework to our website today will probably cost less than adding the same security framework later. This is true for many infrastructure elements, and is the basis for many arguments in favor of developing them early. However, if a feature is developed early, there is a chance that it will change by the end of the project. The cost of these changes needs to be considered in any now/later decision. Additionally, introducing the security framework early may add complexity that will be a hidden cost on all future work. This cost, too, would need to be considered.

Our next factor says that we should accelerate the development of features that generate new product or project knowledge. Depending upon the product being built, a security framework is unlikely to generate relevant new knowledge about the product. However, developing the security framework may generate new knowledge about the project. For example, I worked on a project a handful of years ago that needed to authenticate users through an LDAP server. None of the developers had done this before, so there was a lot of uncertainty about how much effort it would take. To eliminate that uncertainty, the stories about LDAP authentication were moved up to around the middle of a project rather than being left near the end.

The final prioritization factor is risk. Is there a risk to the project's success that could be reduced or eliminated by implementing security earlier rather than later? Perhaps not in this example. However, the failure of a framework, key component, or other infrastructure is often a significant risk to a project. This may be enough to warrant moving development sooner than would be justified solely on the basis of value.

User Interface Design

Common agile advice is that user interface design should be done entirely within the iteration in which the underlying feature is developed. However, this sometimes runs counter to arguments that the usability of a system is improved when designers are allowed to think about the overall user interface up front. What can we learn from applying our prioritization factors?

First, will the development of the user interface generate significant, useful new knowledge? If so, we should move some of the work forward in the schedule. Yes, in many cases developing some of the main user interface components or the navigational model will generate significant, useful new knowledge about the product. The early development of parts of the user interface allows for the

system to be shown to real or likely users in an early form. Feedback from these users will result in new knowledge about the product, and this knowledge can be used to make sure the team is developing the most valuable product possible.

Second, will developing the user interface reduce risk? It probably doesn't eliminate technical risk (unless this is the team's first endeavor with this type of user interface). However, early development of features that show off the user interface often reduces the most serious risk facing most projects: the risk of developing the wrong product. A high prioritization of features that will show off significant user-facing functionality will allow for more early feedback from users. This is the best way of avoiding the risk of building the wrong product.

Finally, if the cost of developing the user interface will be significantly lower if done early, that would be another point in favor of scheduling such features early. In most cases, this is not the case.

So because of the additional learning and risk reduction that can be had, it seems reasonable to move earlier in the schedule those themes that will allow users to provide the most feedback on the usability and functionality of the system. This does not mean that we would work on the user interface in isolation or separate from the functionality that exists beneath the user interface. Rather, this means that it might be appropriate to move forward in the schedule those features with significant user interface components that would allow us to get the most useful feedback from customers and users.

Summary

Because there is rarely enough time to do everything, we need to prioritize what is worked on first. There are four primary factors to be considered when prioritizing.

1. The financial value of having the features.
2. The cost of developing (and perhaps supporting) the new features.
3. The amount and significance of learning and new knowledge created by developing the features.
4. The amount of risk removed by developing the features.

These factors are combined by thinking first of the value and cost of the theme. Doing so sorts the themes into an initial order. Themes can then be moved forward or back in this order based on the other factors.

Discussion Questions

1. A feature on the project to which you have been assigned is of fairly low priority. Right now it looks like it will make it into the current release, but it could be dropped if time runs out. The feature needs to be developed in a language no one on the team has any familiarity with. What do you do?
2. What types of product and project knowledge have your current team acquired since beginning the project? Is there additional knowledge that needs to be acquired quickly and that should be accounted for in the project's priorities?

Chapter 16

Estimating Velocity

"It is better to be roughly right
than precisely wrong."
—John Maynard Keynes

One of the challenges of planning a release is estimating the velocity of the team. You have the following three options:

- Use historical values.
- Run an iteration.
- Make a forecast.

There are occasions when each of these approaches is appropriate. However, regardless of which approach you are using, if you need to estimate velocity you should consider expressing the estimate as a range. Suppose you estimate that velocity for a given team on a given project will be 20 ideal days per iteration. You have a very limited chance of being correct. Velocity may be 21, or 19, or maybe even 20.0001. So instead of saying velocity will be 20, give your estimate as a range, saying perhaps instead that you estimate velocity will be between 15 and 24.

In the following sections, I'll describe each of the three general approaches—using historicals, running an iteration, and making a forecast—and for each, I'll also offer advice on selecting an appropriate range.

Use Historical Values

Historical values are great—if you have them. The problem with historical values is that they're of the greatest value when very little has changed between the old project and team and the new project and team. Any personnel or significant technology changes will reduce the usefulness of historical measures of velocity. Before using them, ask yourself questions like these:

- Is the technology the same?
- Is the domain the same?
- Is the team the same?
- Is the product owner the same?
- Are the tools the same?
- Is the working environment the same?
- Were the estimates made by the same people?

The answer to each question is often yes when the team is moving onto a new release of a product they just worked on. In that case, using the team's historical values is entirely appropriate. Even though velocity in a situation like this is relatively stable, you should still consider expressing it as a range. You could create a range by simply adding and subtracting a few points to the average or by looking at the team's best and worst iterations over the past two or three months.

However, if the answer to any of the preceding questions is no, you may want to think twice about using historical velocities. Or you may want to use historical velocities but put a larger range around them to reflect the inherent uncertainty in the estimate. To do this, start by calculating the team's average velocity over the course of the preceding release. If they completed 150 story points of work during 10 iterations, their average (mean) velocity was 15 points.

Before showing how to convert this to a range, take a look at Figure 16.1. This figure shows the cone of uncertainty that was introduced back in Chapter 1, "The Purpose of Planning." The cone of uncertainty says that the actual duration of a project will be between 60% and 160% of what we think it is. So to turn our single-point, average velocity into a range, I multiply it by 60% and 160%.[1] So if

1. Technically, I should divide it by 0.60 and 1.6. However, because 0.60 and 1.60 are meant to be reciprocals ($0.6 \times 1.6 = 0.96 \cong 1$), you get approximately the same values by multiplying.

our average historical velocity is 15, I would estimate velocity to be in the range of 9 to 24.

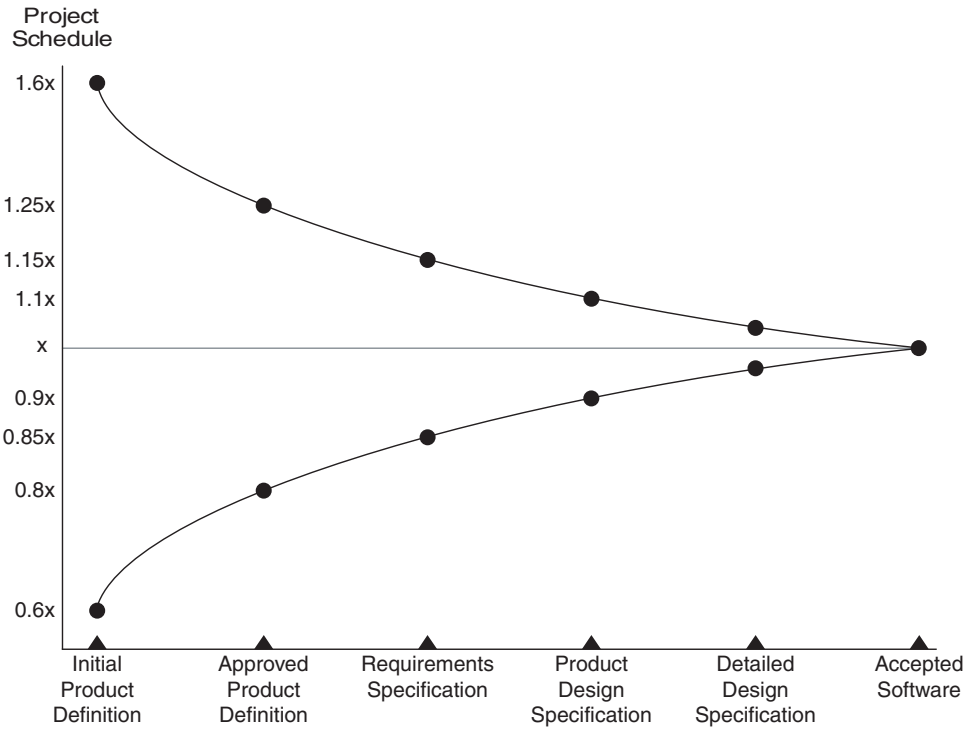

Figure 16.1 The cone of uncertainty around schedule estimates.

This range may feel large, but given the uncertainty at this point, it is probably appropriate. Constructing a range in this way helps the project team heed the advice offered in the quote at the start of this chapter that it is better to be roughly right than precisely wrong. A large range around the expected velocity will allow the team to be roughly right about it.

Run an Iteration

An ideal way to forecast velocity is to run an iteration (or two or three) and then estimate velocity from the *observed velocity* during the one to three iterations. Because the best way to predict velocity is to observe velocity, this should always be your default approach. Many traditional projects get under way with the

developers working on the "obvious" requirements or infrastructure, the analysts "finalizing" the requirements, and the project manager putting together a comprehensive list of tasks that becomes the project plan. All of this takes time—often, as long as a few iterations on an agile project.

I was with a development director recently who said that deadlines are not set on traditional projects in his company until about two months into a year-long project. It takes them that long to get the requirements "locked down" and a plan created. He told me that even after that much effort, their project plans were always off at least 50% and often more. We agreed that instead, he would use this up-front time to turn the team loose on the project, observe their velocity over two or three iterations, and then use that to plan a release date.

For similar reasons, as was the case with this development director, most project managers can hold off giving an estimate for at least one iteration. If that's your case, use the time to run an iteration and measure the velocity. Then create a range around that one data point, using the cone of uncertainty. So if you ran one iteration and had a velocity of 15, turn it into a range by multiplying by 0.60 and 1.6, giving a range of 9 to 24.

If a team can run three or more iterations before being giving an estimate of velocity, they have a couple of additional options for determining a range. First and easiest, they can simply use the range of observed values. Suppose the team has completed three iterations and had velocities of 12, 15, and 16. They could express velocity as likely to be within the range 12 to 16.

Alternatively, they could again use the cone of uncertainty. Although there's no solid empirical basis for the approach I'm about to describe, it does work, and it makes sense. Here's the approach: Calculate the average velocity for the iterations you've run. Then, for each iteration completed, move one step to the right on the cone of uncertainty. So for a team that has run one iteration, use the range for the "initial product definition" milestone. If the team has run two iterations, use the range for the "approved product definition" milestone (80% to 120%), and so on. For convenience, these numbers are shown in Table 16.1.

Table 16.1 Multipliers for Estimating Velocity Based on Number of Iterations Completed

Iterations Completed	Low Multiplier	High Multiplier
1	0.6	1.60
2	0.8	1.25
3	0.85	1.15
4 or more	0.90	1.10

Suppose that a team has run three iterations with an average velocity of twenty during that period. For three iterations the appropriate range is 85% to 115%. This means that if the team's average velocity is twenty after three iterations, their actual true velocity by the end of the project will probably be in the range seventeen to twenty-three.

I normally don't extend this analysis past three or four iterations. I don't use the cone of uncertainty, for example, to pretend that after six iterations, the team precisely knows their velocity, and it won't waver through the end of the project.

Some organizations will resist starting a project without having a more specific idea how long it will take. In such cases, stress that the need to run a few iterations first stems not from a desire to avoid making an estimate, but to avoid giving an estimate without adequate foundation. You'll want to stress that the purpose of these initial iterations is to assess the dark corners of the system, better understand the technologies involved, refine the understanding of the requirements, and measure how quickly the team can make progress.

Make a Forecast

There are times when we don't have historicals, and it is just not feasible to run a few iterations to observe velocity. Suppose the estimate is for a project that won't start for twelve months. Or suppose the project may start soon, but only once a client signs a contract for the work. There are two key differences in cases like this. First, you want to minimize the expenditure on the project so you won't actually start running iterations on a project that may not happen or that is too far in the future. Second, any estimate of velocity on these projects must reflect a high degree of uncertainty.

In cases like these, we need to forecast velocity. Forecasting velocity is rarely your first option, but it's an important option and one you should have in your bag of tricks. The best way to forecast velocity involves expanding user stories into their constituent tasks, estimating those tasks (as we do when planning an iteration), seeing how much work fits into an iteration, and then calculating the velocity that would be achieved if that work were finished in an iteration. This involves the following steps:

1. Estimate the number of hours that each person will be available to work on the project each day.
2. Determine the total number of hours that will be spent on the project during the iteration.

3. Arbitrarily and somewhat randomly select stories, and expand them into their constituent tasks. Repeat until you have identified enough tasks to fill the number of hours in the iteration.
4. Convert the velocity determined in the preceding step into a range.

Let's see how this works through an example.

Estimate the Hours Available

Almost everyone has some responsibilities outside of the specific project that is their primary responsibility. There are emails to be answered, phone calls to be returned, company meetings to attend, and so on. The amount of time this takes differs from person to person and organization to organization. What it amounts to, though, is that project participants generally do not spend 100% of their time working on the project.

From observation and discussion with colleagues, my opinion is that most individuals who are assigned full time to a project spend between four and six hours per day on that project. This fits with reports that individuals spend 55% (Ganssle 2004) to 70% (Boehm 1981) of their time on project activities. At the high end, Kennedy (2003) reports that engineers in Toyota—with its highly efficient, lean process—are able to spend 80% of their time on their designated projects.

Use these numbers as parameters in estimating the amount of time individuals on your project team will be able to dedicate each day to the project. If you are part of a large bureaucracy, you will most likely be at the low end of the scale. If you are part of a three-person start-up in a garage, you'll probably be at the high end. For the purposes of this example, let's assume that the SwimStats team estimates they will each be able to dedicate six hours per day to the project.

Estimate the Time Available in an Iteration

This step is simple: Multiply the number of hours available each day by the number of people on the team and the number of days in each iteration. Suppose the SwimStats team includes one analyst, one programmer, one database engineer, and one tester. Four people each working six hours per day is twenty-four hours each day. In a ten-day iteration they put about 240 hours toward the project.

When I introduce this approach to some teams, they want to factor in additional adjustments for vacations, sick time, and other such interruptions. Don't bother; it's not worth the extra effort, and it's unlikely to be more accurate

anyway. These events are part of the reason why we don't plan on a team's being 100% available in the first place.

Getting More Time on Your Project

Regardless of how many hours team members are able to put toward a project each day, you'd probably like to increase that number. The best technique I've found for doing so was invented by Francesco Cirillo of XPLabs. Cirillo coaches teams to work in highly focused thirty-minute increments (Cirillo 2005). Each thirty-minute increment consists of two parts: twenty-five minutes of intense work followed by a five-minute break. These thirty-minute increments are called "pomodori," Italian for tomatoes and deriving from the use of tomato-shaped timers that ring when the twenty-five-minute period is complete.

Cirillo introduced this technique to Piergiuliano Bossi, who has documented its success with multiple teams (Bossi 2003; Bossi and Cirillo 2001). These teams would plan on completing ten pomodori (five hours) per day. If you find yourself with less productive time per day than you'd like, you may want to consider this approach.

Expand Stories and See What Fits

The next step is to expand stories into tasks, estimate the tasks, and keep going until we've filled the estimated number of available hours (240, in this case). It is not necessary that stories be expanded in priority order. What you really want is a fairly random assortment of stories. Do not, for example, expand all the one- and two-point stories and none of the three- and five-point stories. Similarly, do not expand only stories that involve mostly the user interface or the database. Try to find a representative set of stories.

Continue selecting stories and breaking them into tasks as long as the tasks selected do not exceed the capacity of the individuals on the team. For the SwimStats team, for example, we need to be careful that we don't assume the programmer and analyst are also fully proficient database engineers. Select stories until one skill set on the team can't handle any more work. Add up the story points or ideal days for the work selected, and that is the team's possible velocity.

Suppose we get the planned team together (or a proxy for them if the project will not start for a year), and we expand some stories as shown in Table 16.2. If we felt that the four-person SwimStats team could commit to this but probably no more, we'd stop here. This 221 hours of work seems like a reasonable fit within their 240 hours of available time. Our point estimate of velocity is then twenty-five.

Table 16.2 Hours and Points for Some SwimStats Stories

Story	Story Points	Hours for Tasks
As a coach, I can enter the names and demographic information of all swimmers on my team.	3	24
As a coach, I can define practice sessions.	5	45
As a swimmer, I can see all of my times for a specific event.	2	18
As a swimmer, I can update my demographics information.	1	14
As a swimmer, I can see a line chart of my times for a particular event.	2	14
As a coach, I can see a line chart showing the progress over the season of all of my swimmers in a particular event.	3	30
As a swimmer, I can see a pie chart showing how many first, second, third, and lower places I've finished in.	2	12
As a coach, I can see a text report showing each swimmer's best time in each event.	2	14
As a coach, I can upload meet results from a file exported from the timing system used at the meet.	5	50
Total	**25**	**221**

Put a Range Around It

Use whatever technique you'd like to turn the point estimate of velocity into a range. As before, I like to multiply by 60% and 160%. For the SwimStats team, this means our estimate of twenty-five story points per iteration becomes an estimate of fifteen to forty.

A Variation for Some Teams

Some teams—especially those with a significant number or part-time members—should not plan with a single number of hours that everyone is available. These teams may have members who are allocated for dramatically smaller portions of their time. In these cases, it can be useful to create a table like the one shown in Table 16.3.

For the SwimStats team, as shown in Table 16.3, Yury and Sasha are dedicated full time to the project. SwimStats is Sergey's only project, but he has

some other managerial and corporate responsibilities that take up some of his time. Carina is split between SwimStats and another project. She has very few responsibilities beyond the two projects, and so she could put close to six hours per day on them. However, she needs to move back and forth between the two projects many times each day, and this multitasking affects her productivity, so she is shown as having only two productive hours on SwimStats per day.

Table 16.3 Calculating Availability on a Team with Part-Time Members

Person	Available Hours per Day	Available Hours per Iteration
Sergey	4	40
Yury	6	60
Carina	2	20
Sasha	6	60
Total		**180**

Remember Why We're Doing This

Keep in mind that the reason we're forecasting velocity in this way is that it is either impossible or impractical for the team to run an iteration, and they do not yet have any historical observations. This may be the case because the team doesn't yet exist, and you are tasked with planning a project that starts a few months from now.

If, for example, you are in an environment where you are doing strategic planning and budgeting well in advance of initiating a project, forecasting velocity in this way can be your best approach.

Which Approach Should I Use?

Determining which approach to use is often simpler than this variety of choices may make it appear. Circumstances often guide you and constrain your options. In descending order of desirability, follow these guidelines to estimate velocity:

- If you can run one or more iterations before giving an estimate of velocity, always do so. There's no estimate like an actual, and seeing the team's actual velocity is always your best choice.
- Use the actual velocity from a related project by this team.

- Estimate velocity by seeing what fits.

Regardless of which approach you use, switch to using actual, observed values for velocity as soon as possible. Suppose that you choose to estimate velocity by seeing what fits in an iteration because the project is not set to begin for six months and the organization needs only a rough guess of how long the project will take. Once the project begins and you are able to measure actual velocity, begin using those actuals when discussing the project and its likely range of completion dates.

Summary

There are three ways of estimating velocity. First, you can use historical averages if you have them. However, before using historical averages, you should consider whether there have been significant changes in the team, the nature of the project, the technology, and so on. Second, you can defer estimating velocity until you've run a few iterations. This is usually the best option. Third, you can forecast velocity by breaking a few stories into tasks and seeing how much will fit into an iteration. This process is very similar to iteration planning.

Regardless of which approach you use, estimates of velocity should be given in a range that reflects the uncertainty inherent in the estimate. The cone of uncertainty offers advice about the size of the range to use.

Discussion Questions

1. In Table 16.2, stories that were estimated to have the same number of story points did not have the same number of task hours. Why? (If you need a refresher, see the section "Relating Task Estimates to Story Points" in Chapter 14, "Iteration Planning."
2. Complete a table like Table 16.3 for your current project. What might you try to increase the amount of time each person is able to devote to the project?

Chapter 22

Why Agile Planning Works

"If you want a guarantee,
buy a toaster."
—Clint Eastwood in *The Rookie*

Having arrived at this point, we are well equipped to look at why agile estimating and planning are successful. First, remember the purpose of estimating and planning. Planning is an attempt to find an optimal solution to the overall product development question: features, resources, and schedule. Changing any one of these causes changes in the others. While planning, we are exploring the entire spectrum of possible solutions of how to mix these three parameters such that we create the best product possible. The estimates and plans that we create must be sufficient for serving as the basis for decisions that will be made by the organization. In this chapter, we consider the agile estimating and planning process described so far by this book to see how it achieves these goals.

Replanning Occurs Frequently

One of the ways in which agile estimating and planning support the efficient exploration of the new product development solution space is through frequent replanning. At the start of each new iteration, a plan for that iteration is created. The release plan is updated either after each iteration or, at worst, after every few iterations. Acknowledging the impossibility of creating a perfect plan goes a long way toward reducing the anxiety that accompanies such a goal (Githens 1998).

Knowing that a plan can be revised at the start of the next iteration shifts a team's focus from creating a perfect plan (an impossible goal) to creating a plan that is useful right now.

For a plan to be useful, it must be accurate, but we accept that early plans will be imprecise. One of the reasons we replan is to remove that imprecision progressively. That is, an early plan may say that a project will deliver 300 to 400 story points in the third quarter. That plan may later turn out to be accurate (305 story points are delivered in August), but this plan is not particularly precise. During the early stages of a project, this plan may very likely be a sufficient basis for making decisions. Later, though, to remain useful, the plan will gain precision, and we may say that the project will deliver 380 to 400 story points in September.

An agile estimating and planning process recognizes that our knowledge is always incomplete and requires that plans be revised as we learn. As a team learns more about the product they are building, new features may be added to the release plan. As the team learns more about the technologies they are using or about how well they are working together, expectations about their rate of progress are adjusted. For a plan to remain useful, this new knowledge needs to be incorporated into the plan.

Estimates of Size and Duration Are Separated

A common planning flaw (on traditional as well as many agile teams) is confusing estimates of size and duration. Clearly, the size of an effort and the time needed to complete that effort are related, but many additional factors affect duration. A project of a given size will take programmers of different skill and experience levels different amounts of time. Similarly, duration is affected by the size of the team working on the project. A four-person team may take six months (twenty-four person-months). An eight-person team may reduce that to four calendar months but thirty-two person-months, even though the project is the same size in both cases.

To see the difference between estimates of size and duration, suppose I show you a book and ask you how long it will take you to read it. You can tell from its title that it's a novel, but I won't let you look inside the book to see how many pages are in it, how wide the margins are, or how small the type is. To answer my question about how long it will take you to read this book, you first estimate the number of pages. Let's assume you say 600. Then you estimate your rate of progress at one page per minute. You tell me it will take you 600 minutes, or 10

hours. In arriving at this estimate of duration (10 hours), you first estimated the size of the job (600 pages).

Agile estimating and planning succeed because estimates of size and duration are separated. We start by estimating the size of a project's user stories in story points. Because story points are abstract and notional, they are pure estimates of size. We then estimate a rate of progress, which we call velocity. Our estimates of size and velocity are then combined to arrive at an estimate of duration. Our estimating and planning process benefits from this clear and total separation of estimates of size and duration.

Plans Are Made at Different Levels

Because agile plans cover three different levels—the release, the iteration, and the current day—each plan can be made with a different level of precision. Having plans with different time horizons and different levels of precision has two main benefits. First, it conveys the reality that the different plans are created for different reasons. The daily plan, as committed to by each participant in a team's daily meeting, is fairly precise: Individuals express commitments to complete, or at least make progress on, specific tasks during the day. The iteration plan is less precise, listing the user stories that will be developed during an iteration and the tasks thought necessary to do so. Because each user story is imperfectly specified, there is also some vagueness around what it means to say that the story will be developed in the iteration. Finally, the release plan is the least precise of all, containing only a prioritized list of desired user stories and one or more estimates of how much of the desired functionality is likely to be delivered by the desired release date.

The second benefit of planning at different levels is that it helps the team view the project from different perspectives. An iteration plan is necessary for completing the highly coordinated work of a cross-functional team within a short iteration. A release plan provides a broader perspective on the project and ensures that the team does not lose the forest of the release for the trees of an iteration. A team that works iteration to iteration without awareness of a more distant goal runs the risk of continually pursuing short-term goals while missing out on targeting a truly lucrative longer-term goal. Short-term goals may be inconsistent with the desired long-term result.

Plans Are Based on Features, Not Tasks

A traditional plan in the form of a Gantt chart, PERT chart, or work breakdown structure focuses on the tasks needed to create a product. An agile plan focuses instead on the features that will be needed in the product. This is a key distinction, because it forces the team to think about the product at the right level—the features. When features are developed iteratively, there is less need for up-front thinking about the specific tasks necessary. The work of an iteration emerges, and the team will think of or discover all of the tasks as they are needed. What's more important is that the team think about the features that are being developed. Colleague Jim Highsmith (2004b) has stated that "agile planning is 'better' planning because it utilizes features (stories, etc.) rather than tasks. It is easy to plan an entire project using standard tasks without really understanding the product being built. When planning by feature, the team has a much better understanding of the product." At a task level, many project plans look the same; every agile plan is specific to the product being developed.

Small Stories Keep Work Flowing

From queuing theory (Poppendieck and Poppendieck 2003; Reinertsen 1997), we learn the importance of focusing on *cycle time*, the amount of time something takes to go from the start of a process to the end of that process. On a software project, cycle time is the time from when the team begins work on a feature until that feature delivers value to users. The shorter the cycle time, the better.

A key influence on cycle time is the variability in the time it takes to develop a new feature. One of the best ways to reduce variability is to work with reasonably small and similar size units of work. The estimating and planning process outlined in this book supports this by advising that teams estimate their short-term work within approximately one order of magnitude. Larger user stories can exist further down a project's prioritized requirements list. However, as those features near the top of the list (when they will be scheduled into an iteration that is beginning), they are disaggregated into smaller pieces.

Work in Process Is Eliminated Every Iteration

High amounts of work in process slow a team. On a software project, work in process exists on any feature that the team has started developing but has not yet finished. The more work in process there is, the longer any new feature will take to develop, as the new feature must follow the already-started work. (Or the new

feature needs to be expedited, jumping ahead of the work in process, but that just compounds the problem for the next feature that cannot be expedited.) So work in process leads to increased cycle times. And in the previous section, we learned the importance of maintaining a short cycle time.

One of the reasons why agile planning succeeds is that all work in process is eliminated at the end of each iteration. Because work is not automatically rolled forward from one iteration to the next, each iteration is planned afresh. This means that work on a feature not fully implemented in one iteration will not necessarily be continued in the next. It often will be, but there's no guarantee. This has the effect of eliminating all work in process at the start of each iteration.

Because work in process is eliminated at the start of each iteration, teams are more easily able to work efficiently in short iterations. This means a shorter feedback loop from users to the project team, which leads to faster learning as well as more timely risk mitigation and control.

Tracking Is at the Team Level

Traditional approaches to estimating and planning measure and reward progress at the individual team member level. This leads to a variety of problems. For example, if finishing a task early results in a programmer's being accused of giving a padded estimate for the task, the programmer will learn not to finish early. Rather than finish early, she won't report a task as complete until its deadline.

Agile estimating and planning successfully avert this type of problem by tracking progress only at the team level. This is one of the reasons that Chapter 14, "Iteration Planning," included the advice that individuals should refrain from signing up for specific tasks during iteration planning. Similarly, there are no individual burndown charts prepared—only a teamwide burndown chart.

Uncertainty Is Acknowledged and Planned For

Many traditional, prescriptive plans make the mistake of believing that features can be locked down at the beginning of a project. Plans are then made that do not allow changes or force changes through an onerous change control process. This leads us to delivering projects with features that users don't want. When we create a plan early in a project and do not update the plan as we acquire new knowledge, we lose the opportunity to synchronize the plan with reality.

With an agile approach to estimating and planning, teams acknowledge both end and means uncertainty. End uncertainty (about the product ultimately being built) is reduced as product increments are shown to potential users and other stakeholders at the end of each iteration. Their feedback and responses are used to fine-tune future plans. Means uncertainty (about how the product will be built) is reduced as the team learns more about the technologies in use and themselves. Early discovery that a particular third-party component cannot meet performance requirements, for example, may lead the team to find an alternative. The time to find and switch to the new component will need to be factored into new plans once the need is identified.

A Dozen Guidelines for Agile Estimating and Planning

With all of these reasons in mind, the following is a list of a dozen guidelines for successful agile estimating and planning.

1. **Involve the whole team.** Primary responsibility for certain activities may fall to one person or group, as prioritizing requirements is primarily the responsibility of the product owner. However, the whole team needs to be involved and committed to the pursuit of the highest-value project possible. We see this, for example, in the advice that estimating is best done by the whole team, even though it may be apparent that only one or two specific team members will work on the story or task being estimated. The more responsibilities are shared by the team, the more success the team will have to share.
2. **Plan at different levels.** Do not make the mistake of thinking that a release plan makes an iteration plan unnecessary, or the other way around. The release, iteration, and daily plans each cover a different time horizon with a different level of precision, and each serves a unique purpose.
3. **Keep estimates of size and duration separate by using different units.** The best way to maintain a clear distinction between an estimate of size and one of duration is to use separate units that cannot be confused. Estimating size in story points and translating size into duration using velocity is an excellent way of doing this.
4. **Express uncertainty in either the functionality or the date.** No plan is certain. Be sure to include an expression of uncertainty in any release plan you produce. If the amount of new functionality is fixed, state your uncertainty as a date range ("We'll finish in the third quarter" or "We'll finish in between seven and ten iterations"). If the date is fixed instead, express uncertainty

about the exact functionality to be delivered ("We'll be done on December 31, and the product will include at least these new features, but probably no more than those other new features"). Use bigger units (iterations, months, and then quarters, for example) when the amount of uncertainty is greater.

5. **Replan often.** Take advantage of the start of each new iteration to assess the relevancy of the current release plan. If the release plan is based on outdated information or on assumptions that are now false, update it. Use replanning opportunities to ensure that the project is always targeted at delivering the greatest value to the organization.
6. **Track and communicate progress.** Many of a project's stakeholders will have a very strong interest in the progress of the project. Keep them informed by regularly publishing simple, very understandable indicators of the team's progress. Burndown charts and other at-a-glance indicators of project progress are best.
7. **Acknowledge the importance of learning.** Because a project is as much about generating new knowledge as it is about adding new capabilities to a product, plans must be updated to include this new knowledge. As we learn more about our customers' needs, new features are added to the project. As we learn more about the technologies we are using or about how well we are working as a team, we adjust expectations about our rate of progress and our desired approach.
8. **Plan features of the right size.** Functionality that will be added in the near future (within the next few iterations) should be decomposed into relatively small user stories—typically, items that will take one or two days up to no more than ten days. We are best at estimating work that is all within one order of magnitude in size. Working with user stories that fall within these ranges will provide the best combination of effort and accuracy. It will also provide stories that are small enough to be completed during one iteration for most teams. Of course, working with small user stories can become quite an effort on longer projects. To balance this, if you are creating a release plan that will look more than two or three months into the future, either write some larger stories (called *epics*) or estimate the more distant work at the theme level to avoid decomposing large stories into small ones too far in advance.
9. **Prioritize features.** Work on features in the order that optimizes the total value of the project. In addition to the value and cost of features when prioritizing, consider the learning that will occur and the risk that will be reduced by developing a feature. Early elimination of a significant risk can often justify developing a feature early. Similarly, if developing a particular

feature early will allow the team to gain significant knowledge about the product or their effort to develop it, they should consider developing that feature early.

10. **Base estimates and plans on facts.** Whenever possible, ground your estimates and plans in reality. Yes, there may be times in some organizations when it will be necessary to estimate things like velocity with very little basis in fact, and Chapter 16, "Estimating Velocity," presented some valid techniques for doing so. However, whenever possible, estimates and plans should be based on real, observed values. This goes, too, for an estimate of how much of a feature is complete. It's easy to tell when a feature is 0% done (we haven't started it), and it's relatively easy to tell when we're 100% done (all tests pass for all of the product owner's conditions of satisfaction). It's hard to measure anywhere in between—is this task 50% done or 60% done? Because that question is so hard, stick with what you can know: 0% and 100%.
11. **Leave some slack.** Especially when planning an iteration, do not plan on using 100% of every team member's time. Just as a highway experiences gridlock when filled to 100% capacity, so will a development team slow down when every person's time is planned to full capacity.
12. **Coordinate teams through lookahead planning.** On a project involving multiple teams, coordinate their work through rolling lookahead planning. By looking ahead and allocating specific features to specific upcoming iterations, interteam dependencies can be planned and accommodated.

Summary

The purpose of agile planning is to discover iteratively an optimal solution to the overall product development questions of which features with which resources in what timeline. An agile approach to estimating and planning succeeds in finding such a solution because plans are made at different levels and replanning occurs frequently; because plans are based on features rather than tasks; because size is estimated first and then duration is derived from the size estimate; because small stories keep work flowing, and work in process is eliminated at the end of every iteration; because progress is measured at the team, rather than the individual, level; and because uncertainty is acknowledged and planned for.

Discussion Questions

1. Are there other reasons you can think of why agile estimating and planning succeed?
2. Which of the dozen guidelines apply to your current estimation and planning process? Would that process benefit by following the others?