Chapter 15. Planning

“We're agile; we don't plan” and “We'll be done when we're done” were common statements in the early years following the publication of the Agile Manifesto. I suspect that many people on some of the early agile teams that took this stance knew that they were giving up something valuable when they threw planning out the window. But, theirs was a natural reaction to the prior cultures in which they'd worked. Too many developers hated planning because the plan had never been of any personal benefit to them. Instead, plans were often weapons used against the developers: “You said you'd be done by June; it's June. Make it happen.”

As inappropriate as it was for some organizations to use plans as weapons, it was equally inappropriate to throw planning out altogether. As a former vice president of engineering for a handful of companies where agile development had been central to our success, I also knew that Scrum teams could and should plan. In fact, not only can agile and Scrum teams plan, according to research by Kjetil Melaker-Ostvold and Magne Jørgensen, agile teams often plan more accurately than teams using a sequential process (2005).

Planning is a fundamental aspect of Scrum. Scrum teams commit to always working on the features with the highest value. To do this, the team and product owner must have an estimate of how much a feature will cost to develop; otherwise, they are prioritizing on desirability alone. Similarly, it is important to estimate how long a feature will take to develop—a feature that misses a critical market window will deliver much less value. Clearly, for a Scrum team to live up to its promise of working in priority order, planning must be an essential practice.

In this chapter, we look beyond the basics and consider some planning challenges I see many organizations still facing well into their adoption of Scrum. We start by looking at the need to progressively refine plans rather than starting with fully detailed plans. We next look at why overtime is not a solution to schedule problems. After that, I make the argument that organizations should learn to favor changing scope rather than the other critical project planning parameters of schedule, resources, or quality. Finally, the chapter concludes with advice on separating the estimates created by a team from the commitments the team makes.

Progressively Refine Plans

In Chapter 14, “The Product Backlog,” we learned that the product backlog should be progressively refined. Capabilities that will be added well into the future are initially put on the product backlog as epics and
later split into smaller user stories. Eventually these stories are so small that they do not need to be split further. But they are then refined one last time by adding conditions of satisfaction that describe high level tests, which will be used to determine whether the story has been completed.

A good Scrum team takes a similar approach to planning. Just as an epic describes the essence of a feature but leaves out specifics, an early plan captures the essence of what will be delivered but leaves the specifics for later. Subsequent plans add the necessary details but only when that detail can be supported by the knowledge gained through the project thus far. Leaving details out of an initial plan does not mean we cannot make commitments about what will be included when a project is finished. We can still make commitments, but those commitments must leave room for changes commensurate with the amount of uncertainty on the project.

For example, consider the case of a team developing a new web-based genealogy product. The team has a firm deadline in six months and needs to be able to convey exactly what will be delivered by then. The team can provide a lot of detail about the most important features (which will have been prioritized highest on the product backlog). If manually drawing family trees is a high priority, the initial plan will include a great deal of detail about that feature. The product backlog might mention showing a layout grid, snapping items to a grid, showing rulers, manually inserting page breaks, and so on.

A feature further down the product backlog will include less detail. We might write, “As a user, I want to upload photos so that I can attach them to a person in the family tree.” This gives the team and product owner the flexibility later to support only JPG and GIF files even though the initial hope had been to support seven or eight image formats.

**Objection**

“We do outsourced, contract development. Our customers want to know before they sign the contract which file formats they’ll get. I can’t leave details like that out.”

Although deferring implementation details allows the team more flexibility in finding the best solution, you don’t need to progressively refine all feature descriptions. If image file formats are critical to your customers, then you’re right. These should be detailed in the product backlog and plans that accompany the contract. Other, less critical features can be refined as they rise to the top of the product backlog.

There are many advantages to progressively refining a plan. Chief among them are the following:

- It minimizes the time investment. Planning is necessary, but it can be time consuming. The time spent estimating and planning is best viewed as an investment; we want to invest in planning only to the extent that our effort is rewarded. If we create a detailed project plan at the start of the project, that
plan will be based on many assumptions. As the project progresses, we'll find that some of our assumptions were wrong, which will invalidate plans based on them.

- **It allows decisions to be made at the optimal time.** Progressively refining the plan helps the team avoid falling into the trap of making too many decisions at the outset of the project. Project participants become more knowledgeable about their project day by day. If a decision does not need to be made today and can be safely deferred until tomorrow, we should defer making that decision until everyone is one day smarter.

- **It allows us to make course changes.** One thing we can always be certain of is that things change. Planning enough that we know the general direction but not all of the specifics leaves the team with the flexibility to alter course as more is learned. Notice that I've carefully avoided the common phrase course correction. There is no one "correct course" that is known in advance.

- **It helps us avoid falling into the trap of believing our plans.** No matter how well we understand that the unexpected can happen and that no plan is safe from change, a thorough, well-documented plan can fool us into believing everything has been thought of. Progressively refining a plan reinforces the idea that even the best plan is subject to change.

### Things to Try Now

- Review a current release plan. Identify parts of the plan that are prematurely precise or detailed.
- Make a list of the reasons why your organization creates plans that are prematurely precise. Are there specific individuals or groups that demand such plans? Could they be persuaded to start with less detail? If so, meet with them and present the reasons for doing so.

### Don’t Plan on Overtime to Salvage a Plan

Long ago, when I first started managing software developers, I thought it would be the easiest job in the world. In my experience, and as a programmer, it seemed that programmers routinely underestimated how long things would take. I thought that all I would need to do as a manager would be ask individuals to create their own estimates and then keep the heat on them to meet those estimates. Because the estimates would be low more often than not, I reasoned that we'd finish earlier than if I prepared a schedule for the team.

This worked quite well for the first few months. As a non-Scrum team back in the 1980s, many of the first tasks on the schedule had loosely defined deliverables. Analysis was done when we called it done. Design was done when the deadline for design being done arrived. The first few features to be programmed were finished on schedule. I made a few team members work overtime to meet the deadlines—after all, they were the ones who gave those estimates, not me. The overtime wasn't excessive; a few extra hours this week,
maybe half a day next Saturday. But after a few months of this I noticed we were working more overtime and it wasn't helping as much. Corners we had cut during earlier crunch periods were coming back to haunt us. We were also either finding or making more bugs than before.

My solution back then? More overtime.

No, it didn't work. It also didn't work on the next few projects where I repeated the cycle. But I did eventually learn that teams cannot be pushed infinitely hard and that beyond a certain point, working more hours in a week will move the team backward rather than forward.

In the early days of Extreme Programming this was known as the **fourty-hour workweek**, based on the eight-hour day common in the United States. Soon, though, the principle was renamed to **sustainable pace** to reflect that many countries have a standard different from 40 hours and that it is sometimes acceptable to work longer than 40 hours in a week. Watch any marathon, and each runner will seem to be running at a personally sustainable pace. After all, the runner will keep it up for 26.2 miles. Look more closely, however, and you'll notice that the pace is not entirely consistent from mile to mile. Each works a little harder going up the hill and maybe recovers slightly coming down it. At the finish line, most accelerate and sprint at a pace that is not sustainable beyond the finish line.

Sustainable pace should mean the same to a Scrum team: Most of the time the team runs at a nice, even pace, but every now and then team members need to kick it up a gear, such as when nearing a finish line or perhaps attacking a critical, user-reported defect. Working overtime occasionally does not violate the goal of working at a sustainable pace. Authors of *Extreme Programming Explained* Kent Beck and Cynthia Andres concur.

Overtime is a symptom of a serious problem on the project. The XP rule is simple—you can’t work a second week of overtime. For one week, fine, crank and put in some extra hours. If you come in on Monday and say “To meet our goals, we’ll have to work late again,” then you already have a problem that can’t be solved by working more hours. (2004, 60)

**Learning the Hard Way**

When Clinton Keith was the CTO of High Moon Studios, a developer of Triple-A video games, he learned the hard way to take seriously this admonishment against more than one week of overtime. The video game industry is one of the few that still has a dominant annual trade show. Theirs is called the Electronic Entertainment Expo, or E3. Important upcoming titles are shown at E3, and deals are struck between studios and publishers. It is natural for teams to put in overtime leading up to the biggest showcase of the year for their games. Not only do team members want their games to show well to the press and possible business partners, but they also want to impress their friends from other companies who will be at the
show.

For years, Keith had encouraged his teams to work overtime for the months leading up to E3. But, now that he and High Moon had embraced Scrum, teams had been working at a consistent and sustainable pace. Still, old habits die hard. With a few weeks left before the show, Keith asked his teams for some mandatory overtime. As Kent Beck might have predicted, velocity did go up in the first week, as shown in Figure 15.1. But in the second week, although velocity was still higher than without the overtime, it was below that of the first week. By the third week, velocity was insignificantly above the pre-overtime pace. In the fourth week of overtime, velocity was actually below what the team had been achieving at its sustainable pace (Keith 2006).

Figure 15.1 High Moon Studios found that subsequent weeks of overtime actually lowered velocity. Printed with permission of Clinton Keith, AgileGame Development.

Although ingrained habits might be difficult to break, after a manager experiences something like this and sees hard evidence from his teams that extended periods of overtime are counterproductive, the lesson finally sinks in.

Getting There

The argument in favor of working at a sustainable pace says that teams get more done that way than they do when cycling between an unsustainable pace and a recovery period. Graphically, this is shown in
Figure 15.2: The team working at a sustainable pace completes the same amount of work each period of time. The team working at an unsustainable pace exceeds that amount of work during some periods. But during other periods it is recovering from having worked unsustainably and complete less work. In Figure 15.2 the question is whether the area under the sustainable pace curve (representing the total work done by that team) is greater than the area under the unsustainable pace curve. Another way to think about this is if you have five kilometers to run, will you be faster running at a consistent pace or alternating periods of all-out sprinting and walking?

Figure 15.2: The amount of work completed is shown by the area under each line.

Intellectual arguments that working at a sustainable pace is most productive are unlikely to convince most doubters. After all, do you really believe a tortoise (at its sustainable pace) could beat a hare (with its sprint/sleep strategy)? I know Aesop's fables are supposed to reinforce great truths, but I'd have to see it to believe it. The same can be said of the truth of sustainable pace; most organizations need to collect their own data to be convinced that overtime is not a solution to long-term schedule problems. After you see the data for your teams (like Clinton Keith did in Figure 15.1), it's easy to see that prolonged overtime does not increase productivity.

Unfortunately, getting people to try working at a sustainable pace so that you can even collect that data is no easy feat. I've found the following arguments helpful in pleading that case:

- **Working at a sustainable pace leaves extra capacity for when you need it.** If a team is constantly running at an all-out pace, it will not have the extra reservoir of energy for the time when extra effort is truly necessary.

- **It leaves time for more creativity.** Real productivity comes not just from working more hours but from occasionally coming up with creative solutions that either dramatically shorten the schedule or vastly improve the product. Teams working at a sustainable pace will be more likely to have the mental
energy to come up with these ideas.

• **Stop arguing that our brains are exhausted after six hours a day.** Teams need to stop telling management that their brains are exhausted after six hours of hard thinking and that working beyond that is impossible. Many executives work 12 hours or more per day. Maybe that work is less brain intensive, but developers will get nowhere telling such executives that it is impossible to program for more than six hours. Besides, how many developers who make this argument during their day jobs go home and contribute at night to an open source project for fun? As passion increases, so does productivity.

• **It's worth a try.** If an experiment will result in useful, objective data, most decision makers will support the experiment, as long as it's done at the right time. Early in the project when there's less apparent time pressure, start collecting data about the team's velocity when working at a sustainable pace. Later, when overtime is mandated, don't argue against it. Instead, try to gain the agreement that you'll continue the overtime more than a few weeks only if the data shows that velocity has increased.

### If Not Overtime, What?

Extended overtime is a popular tool because it's cheap, easy, and occasionally effective. It takes nothing more than a manager saying, "I expect you here on Saturday," and has an immediate cost of no more than an occasional pizza. If we wish to establish a culture in which we eliminate such an attractive, seemingly free tool as overtime, we need to offer something in its place.

Tony Schwartz and Catherine McCarthy of the Energy Project believe they have the solution. They point out that time is a finite resource—we cannot add hours to our day. Energy is different, they say: We can add energy. We know this intuitively. There are days we come into the office energized and are hyperproductive. And then there are the days we do little more than watch the clock. If we can add energy to the team, it will have more of the former and fewer of the latter days (2007).

One of the best ways of adding energy is increasing passion. The more passionate people are about their projects, the more likely they are to fully engage on them each day. The product owner is the key here. Product owners need to convey a compelling vision around the product being developed so that team members are enthusiastic about working on it.

### See Also

Suggestions on how to convey this compelling vision were given in the section, "Energize the System," in Chapter 12, "Leading a Self-Organizing Team."

Another good technique advocated by Schwartz and McCarthy is to take brief but regular breaks (2007). A
20-minute walk outside or a quick chat with a coworker works to restore focus and energy to the main task. As a personal example, I used 30-minute sprints to write this book. At the start of each writing sprint I would turn off all distractions such as e-mail and my phone. I would then turn over a 30-minute sand timer. At the end of each half hour I could either turn the timer immediately back over if the writing was going well, or I could grant myself five or ten minutes to check e-mail, return a call, or just look outside.

Francesco Cirillo has long advocated a similar approach he calls “pomodoro,” which is Italian for tomato. In Cirillo’s pomodoro approach, team members work in 30-minute increments. At the start of each increment, a tomato-shaped kitchen timer is set for 25 minutes. The team works diligently during that timewth no distractions from e-mail, the phone, or so on. When the timer goes off, team members take a five-minute break. During these five minutes they can walk around, stretch, share stories, and so on but are discouraged from doing things like talking about work or checking e-mail. Every fourth pomodoro, Cirillo encourages a longer break of from 15 to 30 minutes (2007).

Cirillo’s advice of a longer mental break twice a day fits with human ultra-dian rhythms. These are 90- to 120-minute cycles during which the body moves between high- and low-energy states. Psychologist Ernest Rossi says that “the basic idea is that every hour and a half or so you need to take a rest break—if you don’t...you get tired and lose your mental focus, you tend to make mistakes, get irritable and have accidents” (2002).

Things to Try Now

- Commit to running the next two or three sprints without any overtime. After doing that, evaluate how much work was completed, the quality of that work, and the creativity and energy of team members during the sprint.
- Consider trying Cirillo’s pomodoro approach or a less rigid variation of it. Work for 30 or 60 minutes without interruption. Then take a five- or ten-minute break to look outside, walk around the building, or talk to coworkers.

Favor Scope Changes When Possible

The Project Management Institute (PMI) has long drawn the “iron triangle” shown in Figure 4-5. The iron triangle is meant to show the interdependent relationships between scope, cost (resources), and schedule. It is often drawn by a project manager and handed to the project’s customer accompanied by the words, “Pick any two.” By this the project manager means that as long as he has some flexibility in one of the three dimensions, he can meet the customer’s expectations on the other two.
**Figure 15.3** The iron triangle illustrates the relationship between scope, resources, and schedule.

Shown in the center of the iron triangle of Figure 15.3 is quality. This is because quality—like the federal agents who arrested Al Capone—is considered untouchable. Unfortunately, this is rarely the case; as such, quality is often used as a fourth side of the iron “triangle.”

As part of transitioning to Scrum, key project stakeholders, developers, and product owners will need to learn to make changing scope their first choice. It is far easier to lock down the schedule, resources, and quality of a project. This does not mean that we don’t sometimes instead fix the scope of the project and allow the schedule or resources to vary. However, our bias should be toward adjusting scope to fit available resources and schedule.

**Considering the Alternatives**

To see why we should favor changing scope over the other options, suppose we are part of a team that has just finished the ninth month of what we expected to be a 12-month project. At this point everyone realizes that the full scope of the project cannot be delivered on schedule with the current team. What are our choices?

**Cut Quality?**

Perhaps we should cut some quality corners—skip a little testing, leave a few bugs unfixed. Reducing quality is rarely explicitly considered, yet it is often the go-to option when the project is running behind.
we reduce quality, perhaps by altering our definition of what bugs must be fixed before we ship and perhaps by skipping the stress testing, in our scenario we may succeed in finishing the project in three months. The problem is that reducing quality is short sighted. These decisions, if made, will come back to haunt the team on the next release. The team will probably be under equivalent deadline pressure then but will also have to find a way to pay off the technical debt they racked up to meet the last deadline.

Scrum teams have learned that the best way to go fast is by keeping the quality of the system high throughout its development. In 1979, Philip Crosby wrote that “quality is free” and that “what costs money are the unquality things—all the actions that involve not doing jobs right the first time” (1). So if we try to meet our deadline in three months by cutting quality between now and then, there is a good chance that we will only succeed in slowing ourselves down because of the rework and instability in the system, even over the short term.

A further problem with cutting quality is deciding how much and what to cut. It is hard to predict the impact of shortcutting quality. And if Crosby is right, attempts to shortcut quality could result in a longer schedule. Imagine for a moment that you’ve been asked to move a deadline up from six months to five and that for some reason everyone (including you) agrees that cutting quality is the way to do this. How much quality would you need to cut to shorten the schedule by a month? Specifically, which items would you choose to test less and how much less? Which validation steps would you skip? The difficulty of these questions illustrates how unpredictable cutting quality is as an attempt to shorten a schedule.

**Add Resources?**

What about meeting the planned schedule by adding resources? Tossing a few (or a few dozen) more developers onto the team ought to bring the schedule in, many will think. Unfortunately, it’s not that simple. In *The Mythical Man-Month*, Fred Brooks wrote that “adding manpower to a late software project makes it later” (1995, 25). With 3 months left in our 12-month project, it is quite possible that the time spent training the new team members, the additional communication overhead, and so on will negate the benefit the additional developers bring over such a short period. If we were 1 month into our 12-month project when we realized we wouldn’t be able to deliver everything on schedule, adding people may have made more sense, because they would have had longer to contribute.

Even if we were to debate the relative merits of adding people and when it is too late to do so, what we cannot debate is that the impact of adding people is unpredictable. Because we cannot be positive of the impact, adding resources is risky.

**Extend the Schedule?**

So if we can’t ensure success on our hypothetical project by either reducing quality or increasing resources,
that leaves us with changing the scope or the schedule. Let’s first consider changing the schedule. From the development team’s perspective, changing the schedule is a wonderful option. If our project cannot be delivered in the originally planned three remaining months, all we need to do is estimate how much additional time is needed and then announce that as the new schedule. Apart from the difficulty of successfully re-estimating the completion date, there is little risk to the developers in adjusting the schedule.

Unfortunately, adjusting the date can be very hard for the business. Commitments have often been made to customers or investors. Advertising plans, including the big Super Bowl commercial, have been timed to coincide with the release date. New personnel may have been hired to handle increased calls to sales or support. Training sessions may have been scheduled. And so on. Although changing the deadline is a wonderful, easy-to-implement option for the development team, it is not always feasible. When it is an option, though, it should be considered.

**Adjust the Scope?**

Finally, what about changing the scope of the release? Yes, yes, “changing scope” is a polite way of saying “dropping things.” However, is dropping something always so bad? When our project was first planned we drew the line after a set of features and said, “That’s what you’ll get.” Suppose we drew this line after the 100th feature. I’m positive that the product owner was disappointed that the team would not commit to delivering item 101 on the product backlog. That feature was dropped even before the project began.

And so the product owner is entitled now to be disappointed that we have discovered that we can finish only the first 99, not 100, items on the product backlog. However, this is not the end of the world. At least it’s not if the team has been working in priority order. If the five items being dropped from the release are the five items of lowest priority, and if we assume (as we generally should) that the team did the best work it was capable of under the circumstances, then the product owner is receiving the best possible product in the amount of time provided and with the team available.

Is dropping features disappointing? Absolutely. Would it have been better if we could have better predicted how much would be completed by the target date? Definitely. Is it realistic to expect perfection in these predictions? Sadly, no.

**See Also**

Specific advice on estimating how much functionality can be delivered and by when will be provided in the next section, “Separate Estimating from Committing.”
So, then, back to our example of realizing we will not finish everything in the three months remaining on our hypothetical project. Is dropping scope a valid response in this situation? From the development team's perspective, absolutely. If it isn't going to finish all of the desired work, simply figure out what it is likely to finish and don't do the work past there. If the team is behaving agilely—in particular, driving the system to a potentially shippable state by the end of each sprint—the team will have no challenges in dropping some scope.

**See Also**

For more on the importance of driving to a potentially shippable state at the end of each sprint, see Chapter 14, “Sprints.”

From the business's perspective, dropping scope is always a bad thing. But what alternatives are there? We've established that reducing quality to meet the deadline is not a good thing. We've also established that the effect of adding people is unpredictable. That leaves the business with extending the deadline or dropping scope. Because of the likely issues with changing the deadline, reducing scope is often the preferred option, again assuming that features have been worked on in priority order.

**Project Context Is Key**

Making the appropriate trade-offs between the items on the iron triangle is all about making the appropriate decisions within the context of your project. I'm not advocating that scope always be the first thing to go. I'm certainly not advocating that reducing scope can be taken lightly. What I do want organizations to learn is that changing scope is often more feasible than we may have realized in the past, and that it is often the best side of the iron triangle to adjust.

**Objection**

"This product is like a car; a car is no good if it has an engine but no brakes. I need it all."

True. There are a certain number of mandatory features on a car. I'll even grant that on all cars made since Fred Flintstone was driving, both an engine and brakes have been mandatory. However, there are many other features, even on a car, that are optional—sun roof, air conditioning, traction sensors, and so on.
Again, remember that the team should be working in priority order. This means that the software equivalents of the engine and the brakes are done first. Then when we realize we can’t deliver all desired scope, the features on the chopping block are the ones that are undeniably nice to have but are not truly fundamental. If a project is in a position where the deadline can only be met by dropping truly mandatory features (engine, brakes, and so on), then it really is time to consider other alternatives, possibly including canceling the project.

“If the product includes less than what we’ve planned, no one will buy it.”

This is really the same situation as the car. The real problem here is that the plan was created without a sufficient margin of safety. I most commonly hear this objection when the project planning process consisted of creating the product backlog, determining the earliest possible date that all that work might be completed, and committing that date to customers or users. If the product truly needs features that cannot be delivered by the committed date, then this is one of the occasions when extending the date is necessary. The problem in this case is more that the project was improperly planned than anything else.

**Separate Estimating from Committing**

A fundamental and common problem in many organizations is that estimates and commitments are considered equivalent. A development team (agile or not) estimates that delivering a desired set of capabilities will take seven months with the available resources. Team members provide this estimate to their manager who passes the estimate along to a vice president who informs the client. And in some cases the estimate is cut along the way to provide the team with a “stretch goal.”

The problem here is not that the team’s estimate of seven months is right or wrong. The problem is that the estimate was turned into a commitment. “We estimate this will take seven months” was translated into “We commit to finishing in seven months.” Estimating and committing are both important, but they should be viewed as separate activities.

I need to pick up my daughter from swim practice tonight. I asked her what time she’d be done (which we defined as finished swimming, showered, and ready to go home). She said, “I should be ready by 5:15.” That was her estimate. If I had asked for a firm commitment—be outside the facility by the stated time or I’ll drive away without you—she might have committed to 5:25 to allow herself time to recover from any problems, such as a slightly longer practice, the coach’s watch being off by five minutes, a line at the showers, and so on. To determine a time she could commit to, my daughter would still have formed an estimate. But rather than telling me her estimate directly, she would have converted it to a deadline she could commit to.
The Right Data to Do This

A good organization learns to separate estimating from committing. We estimate first and then, based on how confident we are of the estimate, we convert it into a commitment. But without a good estimate to start with, a team’s commitment will be meaningless. To come up with a good estimate, the product owner and team must be equipped with the right data. Most important, they need to know two critical things:

- The size of the work to be performed
- The team’s expected rate of progress through that work

To size the user stories on the product backlog, most teams use either story points or ideal days, as described in Agile Estimating and Planning (Cohn 2007). The rate at which product backlog items are completed is known as velocity. Velocity is simply the sum of the story-point or ideal-day estimates of the product backlog items completed in each sprint; with most teams using a rule of no partial credit. Working with these values, the product owner is able to see how much functionality can be delivered by various dates. Let’s see how the product owner can use this information to make informed scope/schedule trade-off decisions.

An Example

Consider Table 15.1, which shows the actual velocities of a team I worked with. The first thing you should notice is that velocity is volatile—it can bounce around quite a bit from sprint to sprint. This is because it is impossible to perfectly assign estimates to user stories on the product backlog; some will turn out bigger than estimated and some smaller. Similarly, teams might encounter more interruptions in some sprints or have greater focus in others. I equate a team’s velocity to the number of points scored by a sports team in a sequence of games. My favorite sports team is the Los Angeles Lakers. In their last nine basketball games they scored 101, 94, 102, 102, 107, 93, 114, 117, and 97 points. Compare the volatility of those scores with the volatility shown in Table 15.1.

Table 15.1 The velocity of a team will vary from sprint to sprint, as this team’s data shows.
<table>
<thead>
<tr>
<th>Sprint Number</th>
<th>Velocity</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>34</td>
</tr>
<tr>
<td>2</td>
<td>41</td>
</tr>
<tr>
<td>3</td>
<td>27</td>
</tr>
<tr>
<td>4</td>
<td>45</td>
</tr>
<tr>
<td>5</td>
<td>35</td>
</tr>
<tr>
<td>6</td>
<td>38</td>
</tr>
<tr>
<td>7</td>
<td>40</td>
</tr>
<tr>
<td>8</td>
<td>39</td>
</tr>
<tr>
<td>9</td>
<td>40</td>
</tr>
</tbody>
</table>

Because a team's velocity will vary (perhaps somewhat dramatically) from sprint to sprint, I do not rely heavily on a single value. What I'm interested in is the likely range of future velocities, or what a statistician would call a confidence interval. As an example of a confidence interval, global warming between 2000 and 2050 is estimated to be between 0.1°C and 0.3°C per decade. In 2030, we'll be able to look back and calculate a precise value—say 0.21°C—but looking forward we use a range. Scientists have studied this and are 90% confident that the actual value (when we can calculate it in 2050) will fall between 0.1°C and 0.3°C per decade.

I'd like to know the same thing about a team's velocity. I'd like, for example, to say that my team is 90% likely to experience a velocity between 18 and 26 over the remaining 5 sprints of a project. Fortunately,
putting a confidence interval around a team’s velocity is not hard to do.

Start by gathering velocity data for as many past sprints as you can. You will need at least 5 sprints to calculate a 90% confidence interval. Throw out data from sprints that you do not think accurately represent the team as it will be going forward. For example, if new people were added to the team 8 sprints ago, I would look back at only the last 8 sprints. But, if team size has fluctuated between 5 and 7 people for the last 13 sprints and I expect it to continue to fluctuate like that, then I would include all 13 sprints. Use your judgment, but try to avoid bias in throwing out values that help you get the predicted range you want.

Once you have past velocity values, sort them from lowest to highest. Sorting the values from Table 15.1 produces the following list:

27, 34, 35, 38, 39, 40, 41, 45

Next, we want to use these sorted velocities to find a range that we are 90% confident contains the velocity the team will experience going forward. To do this, we will use Table 15.2, which shows which 2 data points in our sorted set of velocities to use to determine the 90% confidence interval. For example, in Table 15.1 we had 9 observed velocities. Looking at the first column of Table 15.2, we see two choices near to 9: 8 and 11. We round down and choose 8. Looking across from 8, we find the number 9 in the second column. This means we create a confidence interval using the second observation from the bottom and the second from the top in our sorted list of velocities. These values are 34 and 41. Therefore, we are 90% confident that this team’s average velocity will fall between 34 and 41.

Table 15.2: The nth lowest and nth highest observation in a sorted list of velocities can be used to find a 90% confidence interval.
<table>
<thead>
<tr>
<th>Number of Velocity Observations</th>
<th>$n^{th}$ Velocity Observation</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>2</td>
</tr>
<tr>
<td>11</td>
<td>3</td>
</tr>
<tr>
<td>13</td>
<td>4</td>
</tr>
<tr>
<td>16</td>
<td>5</td>
</tr>
<tr>
<td>18</td>
<td>6</td>
</tr>
<tr>
<td>21</td>
<td>7</td>
</tr>
<tr>
<td>23</td>
<td>8</td>
</tr>
<tr>
<td>26</td>
<td>9</td>
</tr>
</tbody>
</table>

We can now use this confidence interval to predict how much functionality can be provided by a given date. We can then use that knowledge to decide what scope and schedule to commit to. Suppose that the team in Table 15.4 has 3 sprints remaining before a release. To see how much the team is likely to complete in that time, we can multiply the number of sprints, 3, by the values of the confidence interval (34 and 41). We then count down that many story points into the product backlog and point to the range of functionality the team is likely to deliver. This can be seen in Figure 15.4, which also shows an arrow pointing at the team’s median value (39).

Figure 15.4 We can predict the amount of work that will be completed in five sprints for the team whose velocity was shown in Table 15.4.
Going from Estimate to Commitment

The three arrows in Figure 15.4 still show only estimates. For many projects, it will be necessary to translate those estimates into commitments. I would ideally like to convert these estimates into a commitment by saying something like, "In the next 5 sprints we believe we can commit to delivering between 170 [5 x 34] and 205 [5 x 41] story points, which means we'll deliver between here [top arrow] and there [bottom arrow] on the product backlog." This is the most realistic and accurate commitment. However, in many cases, product owners and their teams are asked to give a point estimate—"We commit to finishing exactly here." This is often the case for an organization doing outsourced, contract development work that needs to commit to a specific amount of functionality for a fixed-date contract.

When asked to turn a range like 170–205 story points into a point estimate that will be used as a commitment, it is tempting to react by saying, "Well, if you want a guarantee of what we can commit to, then it's the 170." If this is all you are willing to commit to, you will probably be able to achieve it, but you are taking on the risk right now of angering the person you are committing to. This type of decision can be viewed as trading long-term risk (there isn't much risk of delivering less than that amount of functionality) for short-term risk (your client, customer, or boss could be angry at you now if this is all you can commit to).

An alternative—committing to deliver the amount indicated by the high end of the confidence
interval—makes the opposite trade-off. There is a lot of long-term risk (you might not be able to deliver that much) but very little short-term risk (everyone will think you're a superstar today for being able to commit to that much).

So, the three arrows of Figure 15.4 do not tell us what commitment to make. They tell us the likely range for our commitment. As an example, a contract development company with a lot of people on the bench might want to commit at or near the bottom arrow; the company's goal is likely to get people off the bench and back onto client engagements, even if the company takes on some risk to make that happen. Alternatively, a contract development company that has all of its developers fully engaged may commit near the top arrow for the next project.

**Note**

For a fixed-scope project, similar analysis can be done to determine the likely range of how many sprints the project will take. To do this, sum the estimates of the required scope and divide that by both of the values that make up your confidence interval.

**Historical Velocity Forms the Basis for Committing**

When I present information such as this to my clients, a common complaint is that they would love to do this type of analysis, but they cannot because they don't have the data. There's a very simple solution to this: get the data. Doing so is easier than you think. Let's see how you can do this in two common problematic situations: when you have a completely new team that has never worked together and when the team size is changing or will change during the project.

**The Team Has Never Worked Together**

If a team has never worked together, it has no historical velocity data. The best solution in this case is to turn the team members loose on the project and let them run at least one sprint before making commitments. Running two or three sprints would be even better, of course. I understand that running one sprint before committing is not always possible, so here's an alternative approach.

Starting with a product backlog that has been estimated in story points or ideal days, pull the team together. Have members conduct a sprint planning meeting. Have them select one user story at a time from the product backlog, identify the tasks necessary to complete it, estimate each task in hours, and then decide whether they can commit to completing that user story in a sprint. They can select user stories in
any order; we're not planning a real sprint but are instead trying to see how much work they can likely do in a sprint. In fact, this works best if the team grabs user stories more or less randomly from the product backlog. Encourage the team to make whatever assumptions it would like about the state of the system if they grab a user story that would be technically challenging if done in an early sprint. When team members have committed to a set of user stories and say they cannot commit to any more, add up the story-point or ideal-day estimates that were assigned earlier to the selected user stories. This becomes one estimate of the team's velocity. If you can get the team to do it, have it plan a second sprint this way and average the results. This decreases the impact of a bad estimate or two in the first sprint.

Objection

"The team doesn't even exist yet. We'll hire people if it looks like we can do the project in the desired time frame."

Find others in the organization who have similar skills and experience to those you'd hire for the team. If you think you'll eventually hire two good programmers, one great programmer, two solid testers, a top-notch user experience designer, and one database engineer with five years of experience, then find a somewhat similar group of individuals. Invite them to a meeting and ask them to imagine they are the team assigned to this new project. Encourage them to take the estimating seriously—after all, maybe some of them will end up on the team.

When the meeting is over, you may want to adjust the derived velocity estimate up or down based on how you think the eventual team will compare to the estimators and based on how seriously you think the estimators performed the job.

Calculating an initial velocity this way is a first step, but we need to turn it into a range, as we did when we had historical data and created a confidence interval. One way to put a range around the estimated velocity is to use your intuition, perhaps raising and lowering the estimate by 25%, or more if you think the team took the exercise less seriously than it should have.

Another way to put a range around the estimated velocity is to adjust it based on the relative standard deviation calculated from velocities of other teams. Relative standard deviation is simply standard deviation expressed in percentage terms. Looking back at Table 15.1, we can calculate the standard deviation of those velocity values as 5.1. The average velocity of that data was 37.6. Dividing 5.1 by 37.6 and rounding the result gives 1.4%, which is the relative standard deviation. If you have data, like that shown in Table 15.1, for a number of teams, you can calculate the relative standard deviation for each team and then take the average of those values and apply it to the estimated velocity. This will give you a reasonable expectation of the new team's velocity as a range. Although this approach does work reasonably well, I
want to reiterate that it would be better to have historical data for that team or to run a sprint or two before making a commitment. I am presenting this alternative approach for the times when neither of those is possible.

**Team Size Changes Frequently**

A different type of problem occurs when team size changes or is expected to change frequently. As in the previous case, my first answer is an easy one: Stop changing the team. Teams benefit tremendously from having a stable membership. Of course, team composition will change over the long term, but try not to exacerbate the problem by moving people back and forth between teams as is common in many organizations.

My second answer is again to collect data so that you are prepared and can anticipate the impact of team size changes. To do this, have someone in the organization keep track of the percentage change in velocity for the first few sprints following any change in team size. You want to track the change for a couple of sprints following a change of team size because velocity almost always drops in the first sprint, even when team size goes up. This is the result of increased communication, productive team members taking time to get a new member productive, and so on. In my experience, the long-term impact of the change is apparent by the third sprint after the change.

**See Also**

This type of organization-wide metric is a good thing for the Project Management Office (PMO) to collect. The PMO is discussed in Chapter 20, “Human Resources, Facilities, and the PMO.”

My recommendation is to calculate the change in velocity not against the last sprint prior to the change but against the average value from the preceding five sprints. You could look back further, but often that isn’t possible. Remember, we’re trying to solve a problem in an environment where team sizes are changing frequently. So if we try to look back eight sprints, we might find that the team size changed over that interval.

What this leads to is something like Table 15.3. This table shows that the team in the first row went from six members to seven and experienced a 20% drop in velocity in the first sprint after the change, a 4% drop in the next sprint, followed by a 12% increase in velocity in the third sprint. The team in the last row has increased from seven to eight members but has only completed one sprint since doing so. The remaining columns of the last row will be filled in at the end of the next two sprints.

**Table 15.3** Collecting data on the effect of changing team size.
<table>
<thead>
<tr>
<th>Initial Team Size</th>
<th>New Team Size</th>
<th>Sprint +1</th>
<th>Sprint +2</th>
<th>Sprint +3</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>7</td>
<td>-20%</td>
<td>-4%</td>
<td>+12%</td>
</tr>
<tr>
<td>6</td>
<td>7</td>
<td>0%</td>
<td>-6%</td>
<td>+15%</td>
</tr>
<tr>
<td>7</td>
<td>5</td>
<td>-12%</td>
<td>-8%</td>
<td>-8%</td>
</tr>
<tr>
<td>8</td>
<td>6</td>
<td>-20%</td>
<td>-20%</td>
<td>-16%</td>
</tr>
<tr>
<td>7</td>
<td>8</td>
<td>-15%</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

A spreadsheet like this can be kept up to date in no more than a few minutes per sprint, even when tracking dozens of teams. It is intentionally kept simple—I do not, for example, track whether it was a programmer or tester or other who moved onto or off the project. It is entirely likely that you already have a great deal of the raw data needed to create a spreadsheet like Table 15.3. If each team has been tracking its velocity and knows who was on the team during each sprint, you could re-create quite a bit of historical data.

**See Also**

You can download a spreadsheet for tracking this type of data from [www.SucceedingWithAgile.com](http://www.SucceedingWithAgile.com).

You can use data like this to answer a wide variety of questions such as these:
- What will this team’s velocity be if we add two people?
- How soon could we get this project done if we added a person to each team?
- If I want this set of projects done by the end of the year, how many people would we need to add?
- What would be the impact of not approving the new employees in the budget?
- What would be the impact of a 15% layoff?

As a simple example using only the data in Table 15.3, suppose you have been asked to estimate how much more work could be completed in the next seven sprints if the team were to grow from six to seven people. By averaging the first two rows in Table 15.3 you estimate that velocity will go down about 10% in the first sprint, be down about 5% in the second sprint, but then be up about 15% from then on as the new member is
fully assimilated. These values are shown in Table 15.4.

For simplicity, Table 15.3 has only two rows for teams increasing from six to seven people. For a real project, I would want more data than this before starting to base decisions on it.

**Table 15.4 Calculating the impact of going from six to seven team members.**

<table>
<thead>
<tr>
<th>Sprint</th>
<th>Velocity Change</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>−10%</td>
</tr>
<tr>
<td>2</td>
<td>−5%</td>
</tr>
<tr>
<td>3–7</td>
<td>+13%</td>
</tr>
</tbody>
</table>

By averaging the velocity changes, we calculate that the velocity is expected to change by just over 7% over seven sprints. You can now answer your boss by saying that going from six to seven people (a 17% increase in head count and possibly budget) will allow you to deliver approximately 7% more functionality over the planned seven sprints. The product owner should be able to use this information to determine whether the increased cost is worthwhile.

**Things to Try Now**

- Start collecting data relevant to your situation. Minimally, start a spreadsheet to capture the velocity of each team each sprint. Consider including data about team size changes or other factors if such changes are relevant in your organization.
- After you’ve gathered enough data on historical velocity, start producing a chart like Figures 15.1 and 15.2 so that it can be used by each project’s product owner to make scope versus schedule trade-off decisions.

**Summary**

Becoming proficient at planning is a critical skill for any Scrum team. In this chapter we looked at ways
for a team to move beyond the basics of sprint and release planning and achieve greater benefits by
  • Progressively refining plans
  • Working at a sustainable pace
  • Looking first at changing scope when it’s impossible to complete everything in the desired time frame
  • Treating estimates as separate from commitments

Additional Reading

Cohn, Mike. 2005. *Agile estimating and planning*. Addison-Wesley Professional.
The most thorough book on both estimating and planning on agile projects. It covers the advantages and disadvantages of both story points and ideal days, the two most common units for estimating by Scrum teams. It introduces the popular Planning Poker technique for estimating. Also covered in detail are prioritizing and planning in a variety of circumstances.

This paper was written by two respected researchers from the Simula Research Lab. It describes an in-depth survey of software development projects and concludes that agile projects have smaller effort overruns than projects using a sequential development process. A number of possible reasons are cited, including better requirements specifications (a product backlog) and improved customer communication.