Chapter 20. Sprint Execution

Sprint execution is the work the Scrum team performs to meet the sprint goal. In this chapter I focus on the principles and techniques that guide how the Scrum team plans, manages, performs, and communicates during sprint execution.

Overview

Sprint execution is like a mini project unto itself—all of the work necessary to deliver a potentially shippable product increment is performed.

Timing

Sprint execution accounts for the majority of time during a sprint. It begins after sprint planning and ends when the sprint review starts (see Figure 20.1).

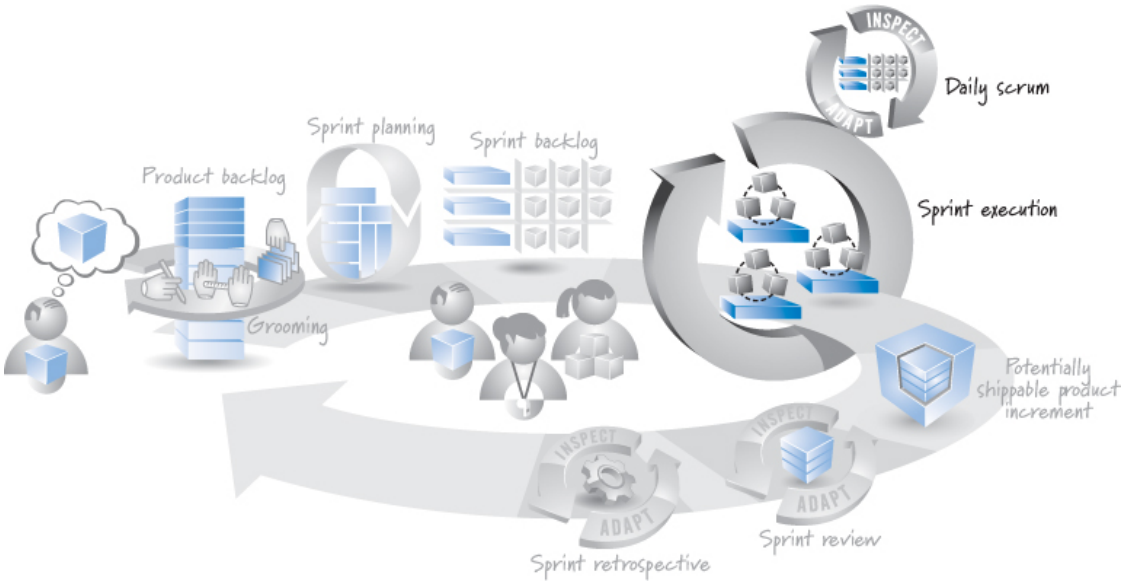

Figure 20.1. When sprint execution happens

On a two-week-long sprint, sprint execution might account for about eight out of the ten days.

Participants

During sprint execution the development team members self-organize and determine the best way to meet the goal established during sprint planning.

The ScrumMaster participates as the coach, facilitator, and impediment remover, doing whatever is possible to help the team be successful. The ScrumMaster doesn't assign work to the team or tell the team how to do the work. A self-organizing team figures these things out for itself.

The product owner must be available during sprint execution to answer clarifying questions, to review intermediate work and provide feedback to the team, to discuss adjustments to the sprint goal if conditions warrant, and to verify that the acceptance criteria of product backlog items have been met.

Process

Figure 20.2 illustrates the sprint execution activity.

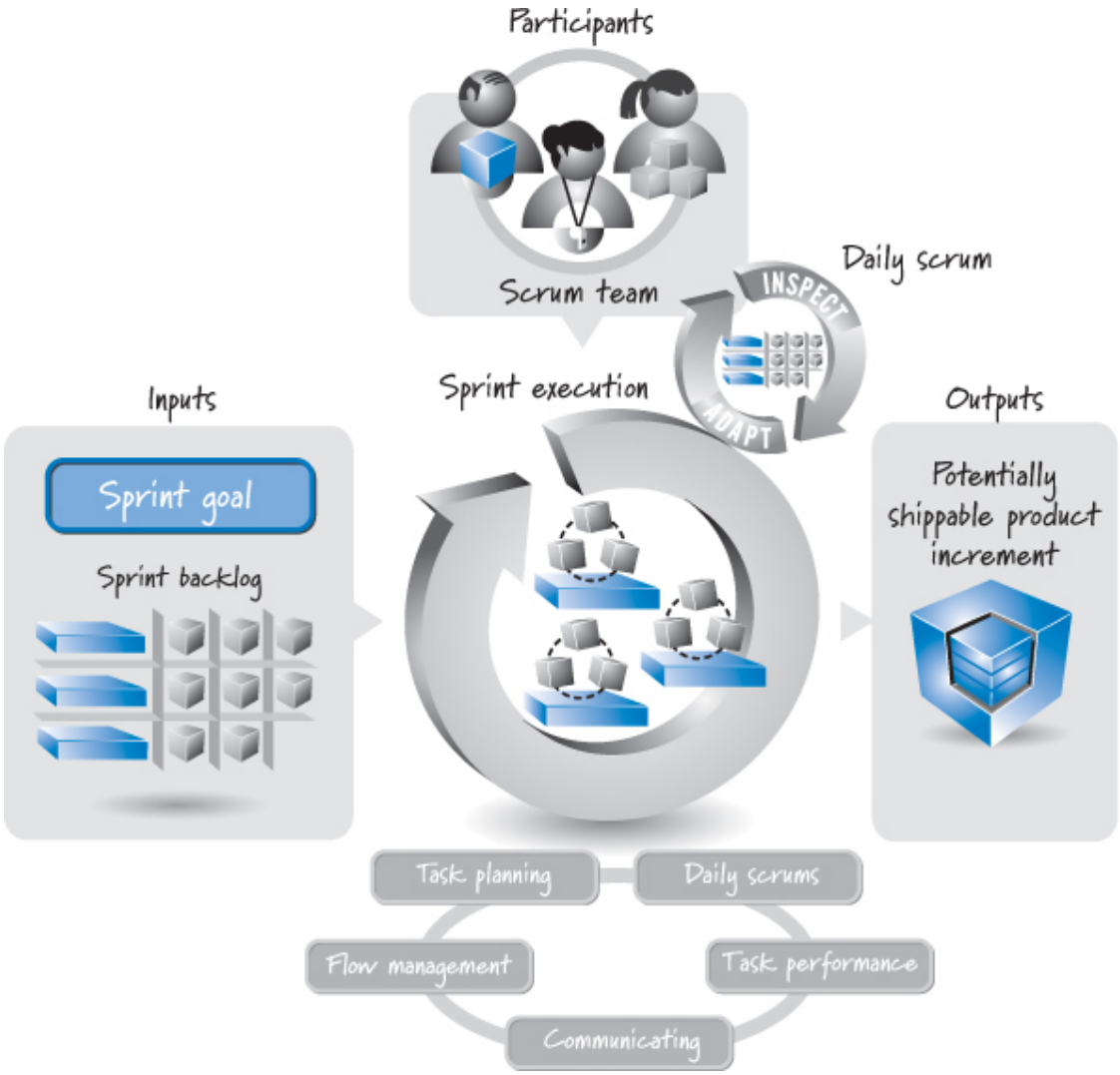

Figure 20.2. Sprint execution activity

The inputs to sprint execution are the sprint goal and the sprint backlog that were generated during sprint planning. The output from sprint execution is a potentially shippable product increment, which is a set of product backlog items completed to a high degree of confidence—where each item meets the Scrum team's agreed-upon definition of done (see Chapter 4). Sprint execution involves planning, managing, performing, and communicating the work necessary to create these working, tested features.

Sprint Execution Planning

During sprint planning the team produces a *plan* for how to achieve the sprint goal. Most teams create a sprint backlog, which typically lists product backlog items and their associated tasks and estimated effort-hours

(see Figure 19.6). Although the team probably could create a full task-level sprint execution plan (the equivalent of a project plan for the sprint, perhaps in the format of a Gantt chart), the economics of doing so are hard to justify.

First, it's not clear that a team of five to nine people needs a Gantt chart to dictate who should do the work and when for the next short-duration sprint. Second, even if the team wanted to create a Gantt chart, it would be inaccurate soon after the team begins working. Sprint execution is where the rubber meets the road. A massive influx of learning comes from actually building and testing something. This learning will disrupt even the best-conceived early plan. As a result, the team wastes valuable time putting a plan together, only to waste even more time changing it to reflect the reality of sprint execution.

Of course, some up-front planning is helpful for exposing important task-level dependencies. For example, if we know that a feature we're creating during the sprint must be subjected to a special two-day-long stress test, it would be wise for the team to sequence the work so that this test starts at least two days before the end of sprint execution.

A good principle for sprint execution is to approach task-level planning opportunistically rather than trying to lay out up front a complete plan of how to do the work (Goldberg and Rubin 1995). Allow task planning to occur continuously during sprint execution as the team adapts to the evolving circumstances of the sprint.

Flow Management

It's the team's responsibility to manage the flow of work during sprint execution to meet the sprint goal. It must make decisions such as how much work the team should do in parallel, when work should begin on a specific item, how the task-level work should be organized, what work needs to be done, and who should do the work.

When answering these questions, teams should discard old behaviors, such as trying to keep everyone 100% busy (the consequences of which are described in Chapter 2), believing that work must be done sequentially, and having each person focus on just her part of the solution.

Parallel Work and Swarming

An important part of managing flow is determining how many product backlog items the team should work on in parallel to maximize the value delivered by the end of the sprint. Working on too many items at once contributes to team member multitasking, which in turn increases the time required to complete individual items and likely reduces their quality.

Figure 20.3 shows a simple example that I use in my training classes to illustrate the cost of multitasking.

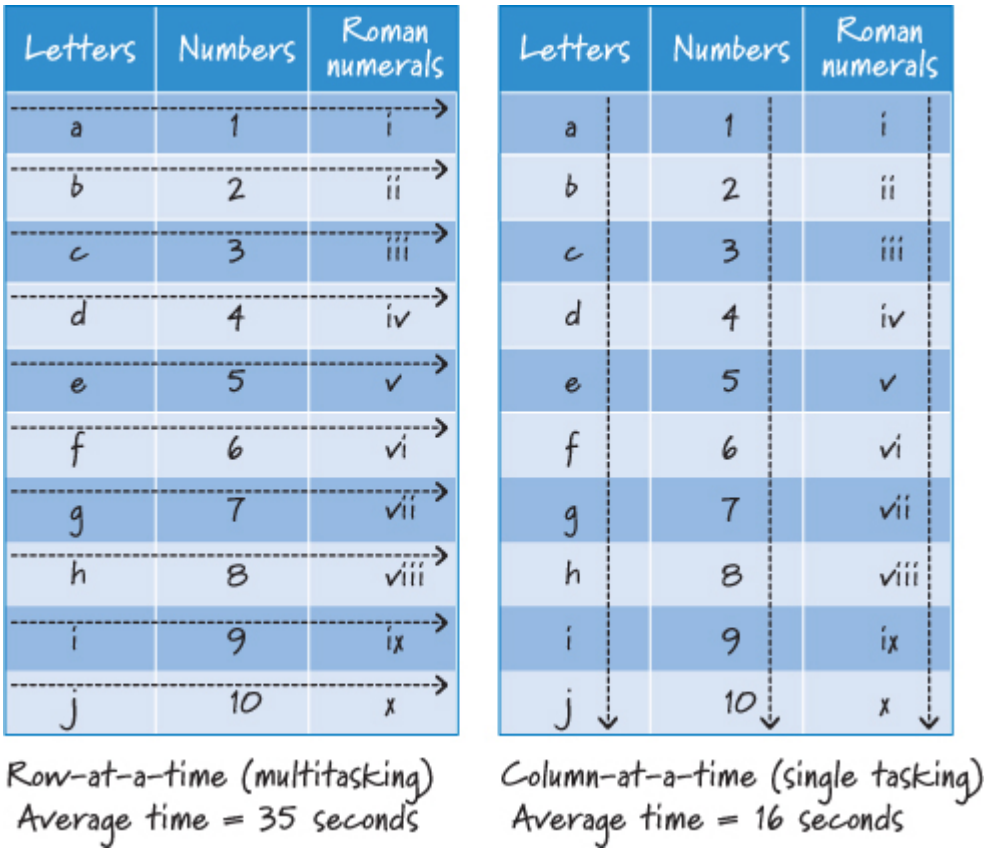

Figure 20.3. Cost of multitasking

The goal is to complete two identical tables by writing the letters *a* to *j*, the numbers 1 to 10, and Roman numerals i to x. One table is completed a row at a time, and the other is completed a column at a time. The row-at-time table represents multitasking (do the letter task, then do the number task, then do the Roman numeral task, and then repeat the sequence for the next letter, number, and Roman numeral). The column-at-a-time table represents single tasking.

The typical results shown in Figure 20.3 are that most people complete the column-at-a-time table in about half the time of the row-at-a-time table. Give it a try and time yourself, and you'll see! Also, if people make any errors, they make them when completing the row-at-a-time table. So, even for simple multitasking the overhead can be quite high. Imagine the waste involved when multitasking on complex project work.

Just as working on too many items at the same time is wasteful, working on too few items at a time is also wasteful. It leads to underutilization of team member skills and capacity, resulting in less work being completed and less value being delivered.

To find the proper balance, I recommend that teams work on the number of items that leverages, but does not overburden, the T-shaped skills (see Chapter 11) and available capacity (see Chapter 19) of the team members. The goal is to reduce the time required to complete individual items while maximizing the total value delivered during the sprint (typically the number of items completed during the sprint).

A term frequently used to describe this approach is **swarming**, where team members with available capacity gather to work on an item to finish what has already been started before moving ahead to start work on new items. Teams with a Musketeer attitude and some degree of T-shaped skills swarm. Teams that still think in terms of individual roles wind up with some members far ahead and others who are mired in unfinished work. A classic individual-role-focused thought is "The testers might still have 'their' work to finish up, but I'm finished coding this feature, so I'm

off to start coding the next one." In a team that swarms, people would understand that it is typically better to stay focused and help get the testing done instead of running ahead to start working on new features.

Some people mistakenly believe that swarming is a strategy to ensure that team members are 100% busy. This is not the goal of swarming. If we wanted to ensure that people were 100% busy, we would just start working on all product backlog items at the same time! Why don't we do that? Because the extensive multitasking required to make that happen would ultimately slow the flow of completed items. Swarming, on the other hand, helps the team remain goal focused instead of task focused, which means it gets more things done, faster.

While swarming favors working on fewer items concurrently, it doesn't necessarily mean working on only one product backlog item at a time. One item at a time might be correct in a given context, but just saying that all team members should collectively focus on a single item at a time is potentially dangerous. A different number of items might be appropriate when we consider the actual work that needs to be done, the skills of the team members, and other conditions that exist at the time a decision to start or not start working on another item needs to be made.

Another dangerous approach would be to apply waterfall thinking at the sprint level and treat sprint execution like a mini waterfall project. Using this approach, we would start working on all product backlog items at the same time. First we would analyze all of the items to be worked on this sprint, then design them all, then code them all, and then, finally, test them all (see Figure 20.4).

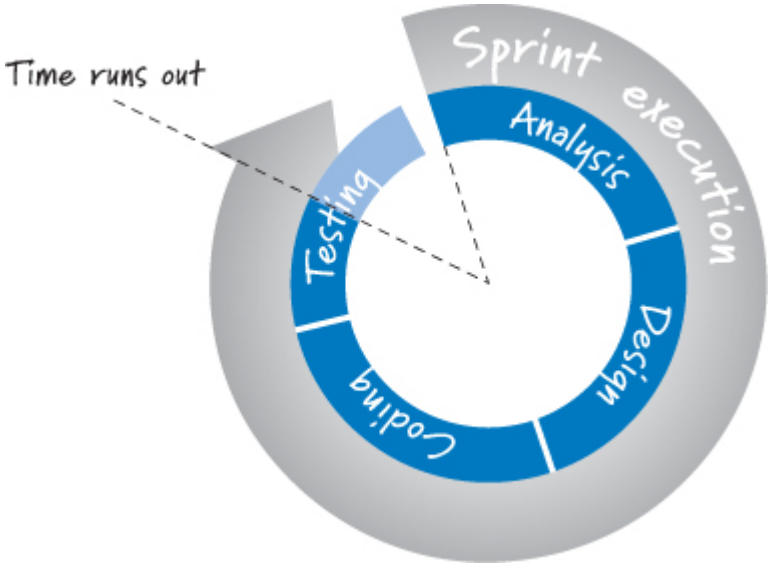

Figure 20.4. Mini waterfall during sprint execution—a bad idea

Although this approach may seem logical, it is very risky. What if the team runs out of time and doesn't finish all of the testing? Do we have a potentially shippable product increment? No; a reasonable definition of done would never allow untested features to be called done. By using a mini waterfall strategy, we could end up with 90% of each feature complete, but no feature 100% done. The product owner gets no economic value from partially done work.

Which Work to Start

Assuming that not all product backlog items are started simultaneously, at some point the team needs to determine which product backlog item to work on next.

The simplest way to select the next product backlog item is to choose the next-highest-priority item as specified by the product owner (via the item's position in the product backlog). This approach has the obvious advantage of ensuring that any items not completed during the sprint must be of lower priority than the ones that are completed.

Unfortunately, the simplest approach won't always work because technical dependencies or skills capacity constraints might dictate that items be

selected in a different order. The development team needs the ability to opportunistically make this selection as it sees fit.

How to Organize Task Work

Once the development team decides to start working on a product backlog item, it must determine how to perform the task-level work on that item. If we apply waterfall thinking at the level of a single product backlog item, we would analyze the item, design it, code it, and then test it.

Believing there is a single, predetermined, logical ordering to the work (for example, you have to build it before you can do any testing) blinds the team to the opportunity to do things in a different and perhaps more efficient way. For example, I frequently hear new teams say something like "What will our testers be doing early in the sprint while they are waiting for features to be ready for testing?" Typically I respond by saying that for teams doing **test-first development**, where tests are written before the development is performed, the "testers" are the *first* to work on a feature (Crispin and Gregory 2009)!

Traditional role-based thinking plagues many teams. What we need instead is value-delivery-focused thinking, where the team members opportunistically organize the tasks and who will work on them. In doing so, they minimize the amount of time that work sits idle and reduce the size and frequency at which team members must "hand off" work to one another. This might mean, for example, that two people pair up on the first day of sprint execution and work in a highly interleaved fashion, with rapid cycles of test creation, code creation, test execution, and test and code refinement, and then repeat this cycle. This approach keeps work flowing (no blocked work), supports very fast feedback so that issues are identified and resolved quickly, and enables team members with T-shaped skills to swarm on an item to get it done.

What Work Needs to Be Done?

What task-level work does the team perform to complete a product backlog item? Ultimately the team decides. Product owners and managers must trust that the team members are responsible professionals who have a vested interest in doing great work. As such, they need to empower these individuals to do the necessary work to create innovative solutions in an economically sensible way.

Of course, product owners and managers do have influential input to what task-level work gets done. First, the product owner ensures that the scope of a feature and its acceptance criteria are defined (part of the definition of ready described in Chapter 6), both of which provide boundaries for the task-level work.

Product owners and managers also provide business-facing requirements for the definition of done. For example, if the business requires that the features developed in each sprint be released to the end customer at the conclusion of the sprint, that decision influences the task-level work the team will perform (there is more work involved with getting features live on production servers than there is in getting them built and tested).

Overall, the product owner must work with the team to ensure that technical decisions with important business consequences are made in an economically sensible way. Some of these decisions are embedded in the more technically oriented aspects of the definition of done. For example, the Scrum team may collectively decide that having automated regression tests (which have an economic cost and benefit) is important and in this way influence task-level work (to create and run automated tests).

Other decisions are feature specific. There is often a degree of flexibility regarding how much effort a team should exert on a feature. For example, enhancing or polishing a feature might be technically appealing but simply not worth the extra cost to the product owner at this time or ever. Conversely, cutting corners on a design or shortchanging where, how, or when we do testing also has economic consequences that must be considered (see the discussion of technical debt in Chapter 8). The team is ex-

pected to work with the product owner to discuss these trade-offs and make economically sensible choices.

Who Does the Work?

Who should work on each task? An obvious answer is the person best able to quickly and correctly get it done. What if that person is unavailable? Perhaps she is already working on another, more important task, or maybe she is out sick and the task needs to get done immediately.

There are a number of factors that can and should influence who will work on a task; it's the collective responsibility of the team members to consider those factors and make a good choice.

When team members have T-shaped skills, several people on the team have the ability to work on each task. When some skills overlap among team members, the team can swarm people to the tasks that are inhibiting the flow of a product backlog item through sprint execution, making the team more efficient.

Daily Scrum

The daily scrum is a critical, daily inspect-and-adapt activity to help the team achieve faster, more flexible flow toward the solution. As I discussed in Chapter 2, the daily scrum is a 15-minute, timeboxed activity that takes place once every 24 hours. The daily scrum serves as an inspection, synchronization, and daily adaptive planning activity that helps a self-organizing team do its job better.

The goal of the daily scrum is for people who are focused on meeting the sprint goal to get together and share the big picture of what is happening so that they can collectively understand how much to work on, which items to start working on, and how to best organize the work among the team members. The daily scrum also helps avoid waiting. If there is an issue that is blocking flow, the team would never have to wait more than a day to discuss it. Imagine if the team members got together only once a

week—they would deny themselves the benefits of fast feedback (see Chapter 3). Overall the daily scrum is essential for flow management.

Task Performance—Technical Practices

Development team members are expected to be technically good at what they do. I'm not saying that you need a team of superstars to use Scrum. However, working in short, timeboxed iterations where there is an expectation of delivering potentially shippable product increments does exert pressure on teams to get the job done with good control over technical debt. If team members lack appropriate technical skills, they will likely fail to achieve the level of agility needed to deliver long-term, sustainable business value.

If you are using Scrum to develop software, team members need to be skilled in good technical practices for developing software. I'm not referring to esoteric skills but instead to skills that have been in use for decades and are essential to being successful with Scrum or arguably any software development approach—for example, continuous integration, automated testing, refactoring, test-driven development, and so on. Today the agile community refers to many of these technical practices as Extreme Programming (Beck and Andres 2004), but most are practices that predate that label (see Figure 20.5 for a subset of the Extreme Programming technical practices).

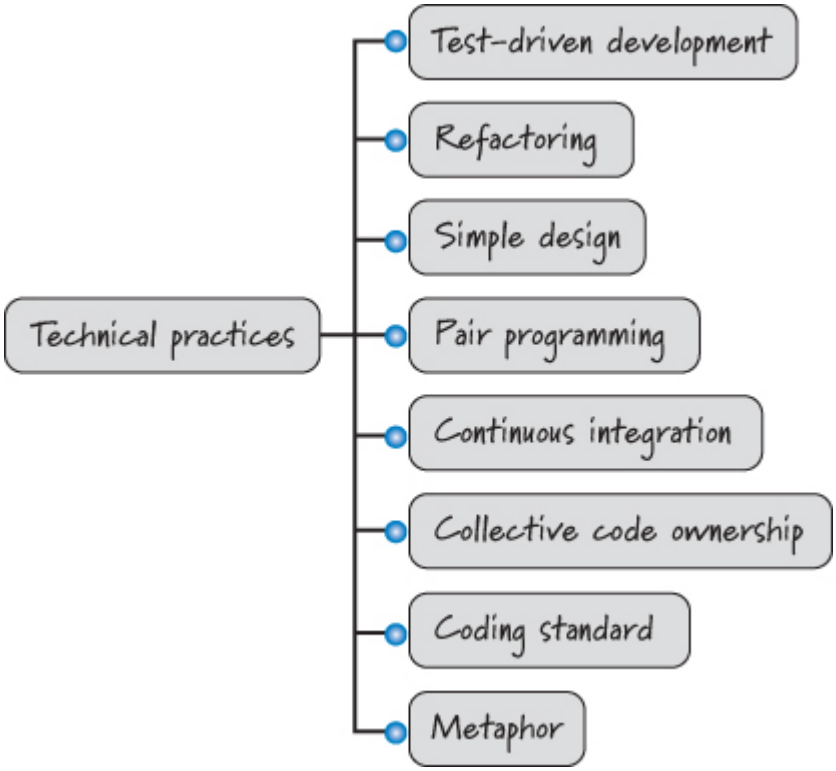

Figure 20.5. Subset of Extreme Programming technical practices

As an example, consider automated testing, which is necessary to support several of the practices in Figure 20.5. Development teams that don't focus on automating their tests will quickly start to slow down and take ever-increasing risks. At some point, it could take all of the sprint execution time just to manually rerun the regression tests for previously developed features. In such cases, the team might choose not to rerun all of the manual tests each sprint, which could allow defects to propagate forward, adding to the system's technical debt (increased risk). You won't be agile for very long if you don't start automating your tests.

Similar arguments can be made for other core technical practices. Most teams achieve the long-term benefits of Scrum only if they also embrace strong technical practices when performing task-level work.

Communicating

One of the benefits of working in short timeboxes with small teams is that you don't need complex charts and reports to communicate progress! Although any highly visible way of communicating progress can be used,

most teams use a combination of a task board and a burndown and/or burnup chart as their principal **information radiator**.

Task Board

The **task board** is a simple but powerful way to communicate sprint progress at a glance. Formally, the task board shows the evolving state of the sprint backlog over time (see Figure 20.6).

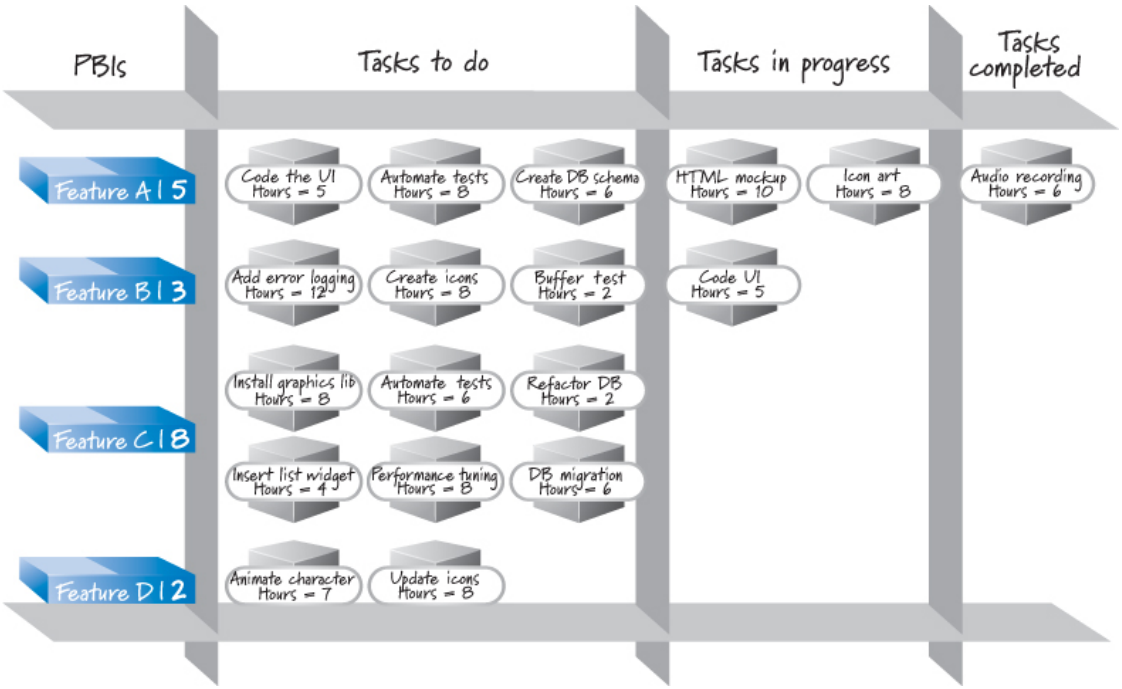

Figure 20.6. Example task board

On this task board each product backlog item planned to be worked on during the sprint is shown with the set of tasks necessary to get the item done. All tasks initially start off in the “to do” column. Once the team determines that it is appropriate to work on an item, team members start selecting tasks in the “to do” column for the item and move them into the “in progress” column to indicate that work on those tasks has begun. When a task is completed, it is moved to the “completed” column.

Of course, Figure 20.6 is just an example of how a task board might be structured. A team may choose to put other columns on its task board if it thinks that visualizing the flow of work through other states is helpful. In fact, an alternative agile approach called Kanban (Anderson 2010) uses

just such a detailed board to visualize the flow of work through its various stages.

Sprint Burndown Chart

Each day during sprint execution team members update the estimate of how much effort remains for each uncompleted task. We could create a table to visualize this data. Table 20.1 shows an example of a 15-day sprint that initially has 30 tasks (not all of the days and tasks are shown in the table).

Table 20.1. Sprint Backlog with Estimated Effort Remaining Each Day

Tasks	D1	D2	D3	D4	D5	D6	D7	D8	D9	. . .	D15
Task 1	8	4	4	2							
Task 2	12	8	16	14	9	6	2				
Task 3	5	5	3	3	1						
Task 4	7	7	7	5	10	6	3	1			
Task 5	3	3	3	3	3	3	3				
Task 6	14	14	14	14	14	14	14	8	4		
Task 7						8	6	4	2		
Tasks 8–30	151	139	143	134	118	99	89	101	84		0
Total	200	180	190	175	155	130	115	113	90		0

In Table 20.1 the number of hours remaining for each task follows the general trend of being smaller each day during the sprint—because tasks are being worked on and completed. If a task hasn't yet been started (it is still in the task board "to do" column), the size of the task might appear the same from day to day until the task is started. Of course, a task might turn out to be larger than expected, and if so, its size may actually increase day over day (see Table 20.1, task 4, days 4 and 5) or remain the same size even after the team has started working it (see Table 20.1 task 1, days 2 and 3)—either because no work took place on the task the previous day, or work did take place the previous day but the estimated effort remaining is the same.

New tasks related to the committed product backlog items can also be added to the sprint backlog at any time. For example, on day 6 in Table 20.1 the team discovered that task 7 was missing, so it added it. There is no reason to avoid adding a task to the sprint backlog. It represents real work that the team must do to complete a product backlog item that the team agreed to get done. Permitting unforeseen tasks to be added to the sprint backlog is not a loophole for introducing new work into the sprint. It simply acknowledges that during sprint planning we may not be able to fully define the complete set of tasks needed to design, build, integrate, and test the committed product backlog items. As our understanding of the work improves by doing it, we can and should adjust the sprint backlog.

If we plot the row labeled "Total" in Table 20.1, which is the sum of the remaining effort-hours across all uncompleted tasks on a given day, on a graph, we get another of the Scrum artifacts for communicating progress —the sprint burndown chart (see Figure 20.7).

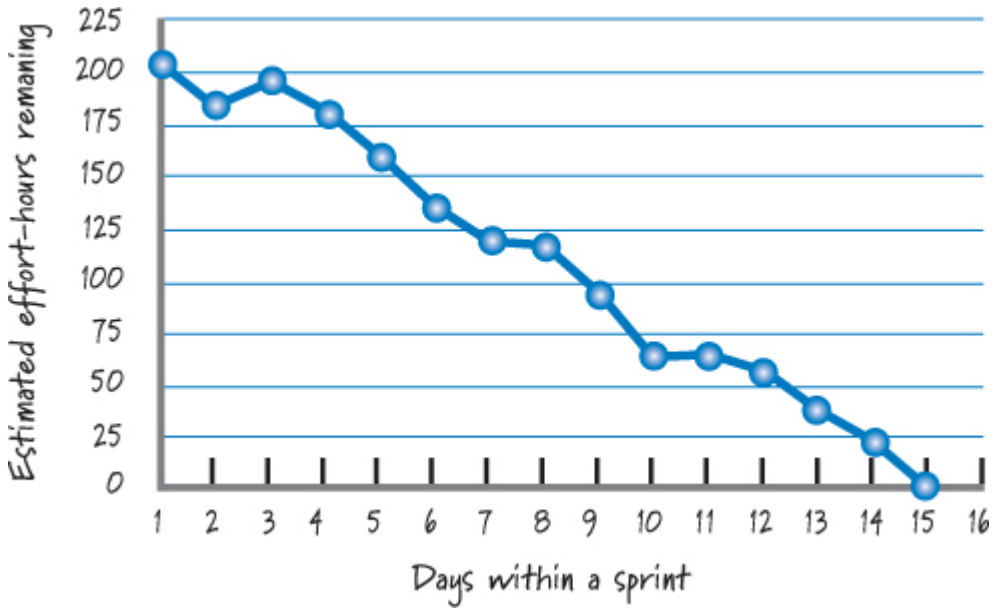

Figure 20.7. Sprint burndown chart

In Chapter 18 I discussed release burndown charts, where the vertical axis numbers are either in story points or ideal days and the horizontal axis numbers are in sprints (see Figure 18.11). In sprint burndown charts the vertical axis numbers are the estimated effort-hours remaining, and

the horizontal axis numbers are days within a sprint. Figure 20.7 shows that we have 200 estimated effort-hours remaining on the first day of the sprint and zero effort-hours remaining on day 15 (the last day of a three-week-long sprint). Each day we update this chart to show the total estimated effort remaining across all of the uncompleted tasks.

Like release burndown charts, sprint burndown charts are useful for tracking progress and can also be used as a leading indicator to predict when work will be completed. At any point in time we could compute a trend line based on historical data and use that trend line to see when we are likely to finish if the current pace and scope remain constant (see Figure 20.8).

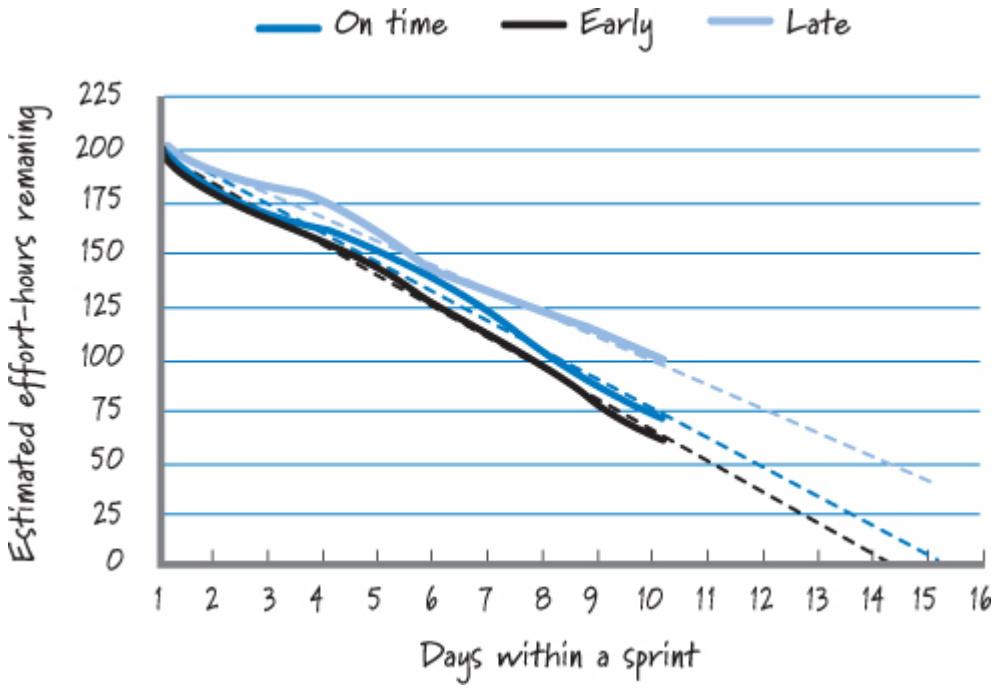

Figure 20.8. Sprint burndown chart with trend lines

In this figure, three different burndown lines are superimposed to illustrate distinct situations. When the trend line intersects the horizontal axis close to the end of the sprint duration, we can infer that we're in reasonable shape ("On time"). When it lands significantly to the left, we should probably take a look to see if we can safely take on additional work ("Early"). But when it lands significantly to the right ("Late"), that raises a flag that we're not proceeding at the expected pace or that we've taken on too much work (or both!). When that happens, we should dig deeper to

see what's behind the data and what, if anything, needs to be done. By projecting the trend lines, we have another important set of data that adds to our knowledge of how we are managing flow within our sprint.

The sprint backlog and the sprint burndown charts always use estimated effort *remaining*. They do not capture actual effort expended. In Scrum there is no specific need to capture the *actuals*; however, your organization might choose to do so for non-Scrum reasons such as cost accounting or tax purposes.

Sprint Burnup Chart

Analogous to how a release burnup chart is an alternative way of visualizing progress through a release, a sprint burnup chart is an alternative way to visualize progress through a sprint. Both represent the amount of work completed toward achieving a goal, the release goal in one case and the sprint goal in the other.

Figure 20.9 shows an example sprint burnup chart.

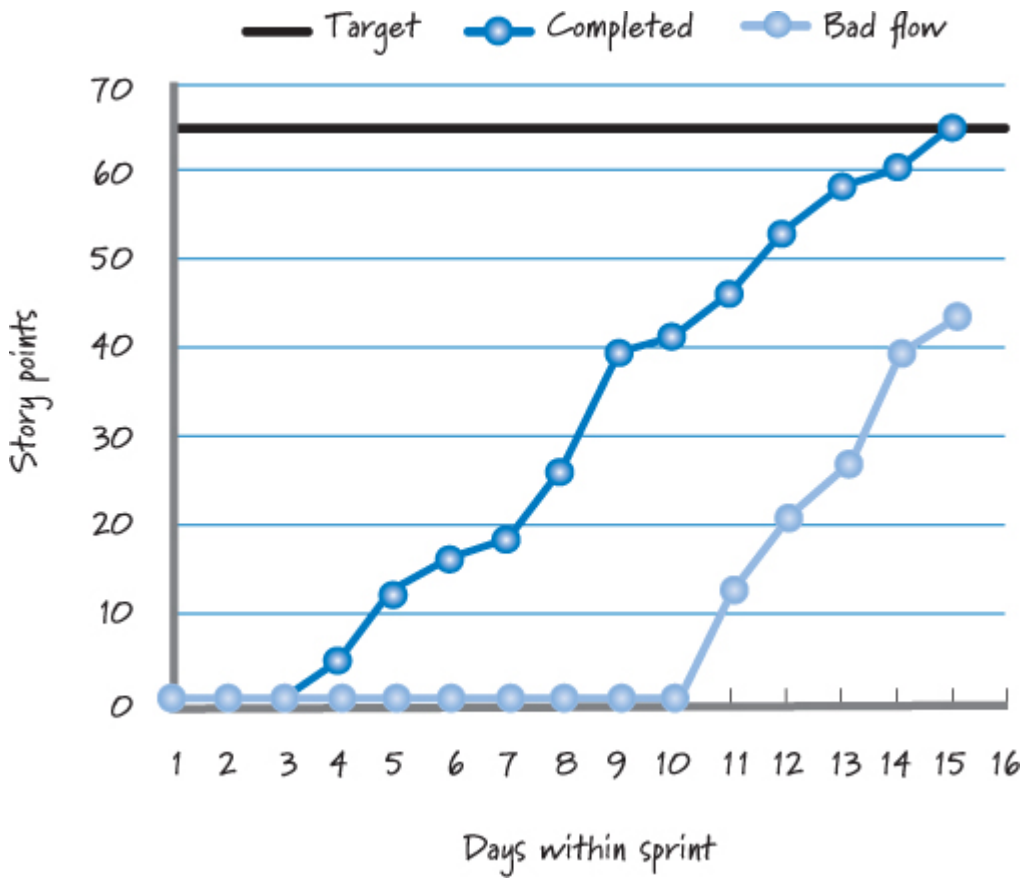

Figure 20.9. Sprint burnup chart

In sprint burnup charts the work can be represented in either effort-hours (as in the sprint burndown chart) or in story points (as shown in Figure 20.9). Many people prefer to use story points in their burnup charts, because at the end of the sprint the only thing that really matters to the Scrum team is business-valuable work that was completed during the sprint, and that is measured in story points (or ideal days), not task hours.

Also, if we measure story points of completed product backlog items, at a glance we can get a good feel for how the work is flowing and how the team is completing product backlog items through the sprint. To illustrate this point a third line (labeled “Bad flow”) is included on the sprint burnup chart in Figure 20.9 (normally this line would not be on the chart; it is added in this example for comparison purposes). The “Bad flow” line illustrates what the burnup chart might look like if the team starts too many product backlog items at the same time, delays completion of items until later in the sprint, fails to meet the sprint target because of the reduced velocity of doing too much work in parallel, works on product backlog items that are large and therefore take a long time to finish, or takes other actions that result in bad flow.

Closing

In this chapter I discussed sprint execution, which accounts for the majority of the time during a sprint. I emphasized that sprint execution is not guided by a complete up-front plan that specifies what work will be done, when it will be done, and who will do it. Rather, sprint execution is performed opportunistically, leveraging the skills of the team, feedback from work already completed, and the evolving, unpredictable circumstances of the sprint. This doesn’t mean that sprint execution is chaotic, but rather that it is guided by the application of good flow management principles, which determine how much work to do in parallel, which work to start, how to organize that work, who will do the work, and how

much effort to invest in the work. In this context I discussed the value of the daily scrum meeting as an important activity in flow management. I also mentioned the importance of good technical practices in achieving high levels of agility. I concluded by discussing the various ways that the Scrum team can visually communicate sprint progress through the task board, sprint burndown chart, and sprint burnup chart. In the next chapter I will discuss the sprint review activity that naturally follows sprint execution.