2 Agile Software Engineering

Bringing a software product to the market quickly is critically important. This is true for all types of products—from simple mobile apps to large-scale enterprise products. If a product is released later than planned, a competitor may have already captured the market or you may have missed a market window, such as the beginning of the holiday season. Once users have committed to a product, they are usually reluctant to change, even to a technically superior product.

Agile software engineering focuses on delivering functionality quickly, responding to changing product specifications, and minimizing development overheads. An "overhead" is any activity that doesn't contribute directly to rapid product delivery. Rapid development and delivery and the flexibility to make changes quickly are fundamental requirements for product development.

A large number of "agile methods" have been developed. Each has its adherents, who are often evangelical about the method's benefits. In practice, companies and individual development teams pick and choose agile techniques that work for them and that are most appropriate for their size and the type of product they are developing. There is no best agile method or technique. It depends on who is using the technique, the development team, and the type of product being developed.

2.1 Agile methods

In the 1980s and early 1990s, there was a widespread view that the best way to create good software was to use controlled and rigorous software development processes. The processes included detailed project planning, requirements specification and analysis, the use of analysis and design methods supported by software tools, and formal quality assurance. This view came from the software engineering community that was responsible for developing large, long-lived software systems such as aerospace and government systems. These were "one-off" systems, based on the customer requirements.

This approach is sometimes called plan-driven development. It evolved to support software engineering where large teams developed complex, long-lifetime systems. Teams were often geographically dispersed and worked on the software for long periods of time. An example of this type of software is a control system for a modern aircraft. Developing an avionic system might take five to ten years from initial specification to on-board deployment.

Plan-driven development involves significant overhead in planning, designing, and documenting the system. This overhead is justifiable for critical systems where the work of several development teams must be coordinated and different people may maintain and update the software during its lifetime. Detailed documents describing the software requirements and design are important when informal team communications are impossible.

If plan-driven development is used for small and medium-sized software products, however, the overhead involved is so large that it dominates the software development process. Too much time is spent writing documents that may never be read rather than writing code. The system is specified in detail before implementation begins. Specification errors, omissions, and misunderstandings are often discovered only after a significant chunk of the system has been implemented.

To fix these problems, developers have to redo work that they thought was complete. As a consequence, it is practically impossible to deliver software quickly and to respond rapidly to requests for changes to the delivered software.

Dissatisfaction with plan-driven software development led to the creation of agile methods in the 1990s. These methods allowed the development team to focus on the software itself, rather than on its design and documentation. Agile methods deliver working software quickly to customers, who can then propose new or different requirements for inclusion in later versions of the system. They reduce process bureaucracy by avoiding work that has dubious long-term value and eliminating documentation that will probably never be used.

The philosophy behind agile methods is reflected in the agile manifesto[1] that was agreed on by the leading developers of these methods. Table 2.1 shows the key message in the agile manifesto.

[1]Retrieved from http://agilemanifesto.org/. Used with permission.

Table 2.1 The agile manifesto

We are uncovering better ways of developing software by doing it and helping others to do it. Through this work, we have come to value:

- individuals and interactions over processes and tools;
- working software over comprehensive documentation;
- customer collaboration over contract negotiation;
- responding to change over following a plan.

While there is value on the items on the right, we value the items on the left more.

All agile methods are based on incremental development and delivery. The best way to understand incremental development is to think of a software product as a set of features. Each feature does something for the software user. There might be a feature that allows data to be entered, a feature to search the entered data, and a feature to format and display the data. Each software increment should implement a small number of product features.

With incremental development, you delay decisions until you really need to make them. You start by prioritizing the features so that the most important features are implemented first. You don't worry about the details of all the features—you define only the details of the feature that you plan to include in an increment. That feature is then implemented and delivered. Users or surrogate users can try it out and provide feedback to the development team. You then go on to define and implement the next feature of the system.

I show this process in Figure 2.1, and I describe incremental development activities in Table 2.2.

Figure 2.1 Incremental development

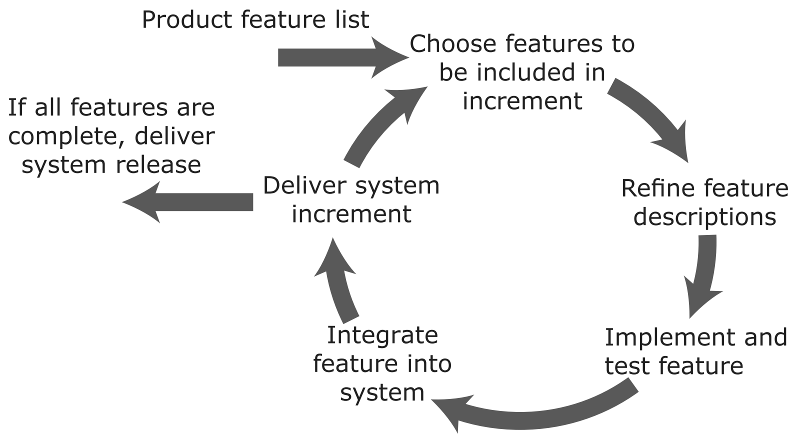

Product feature list
Choose features to be included in increment
Refine feature descriptions
Implement and test feature
Integrate feature into system
Deliver system increment
If all features are complete, deliver system release

Table 2.2 Incremental development activities

Activity	**Description**
Choose features to be included in an increment	Using the list of features in the planned product, select those features that can be implemented in the next product increment.
Refine feature descriptions	Add detail to the feature descriptions so that the team members have a common understanding of each feature and there is sufficient detail to begin implementation.
Implement and test	Implement the feature and develop automated tests for that feature that show that its behavior is consistent with its description. I explain automated testing in Chapter 9.
Integrate feature and test	Integrate the developed feature with the existing system and test it to check that it works in conjunction with other features.
Deliver system increment	Deliver the system increment to the customer or product manager for checking and comments. If enough features have been implemented, release a version of the system for customer use.

Of course, reality doesn't always match this simple model of feature development. Sometimes an increment has to be devoted to developing an infrastructure service, such as a database service, that is used by several features; sometimes you need to plan the user interface so that you get a consistent interface across features; and sometimes an increment has to sort out problems, such as performance issues, that were discovered during system testing.

All agile methods share a set of principles based on the agile manifesto, so they have much in common. I summarize these agile principles in Table 2.3.

Almost all software products are now developed with an agile approach. Agile methods work for product engineering because software products are usually stand-alone systems rather than systems composed of independent subsystems. They are developed by co-located teams who can communicate informally. The product manager can easily interact with the development team. Consequently, there is no need for formal documents, meetings, and cross-team communication.

Table 2.3 Agile development principles

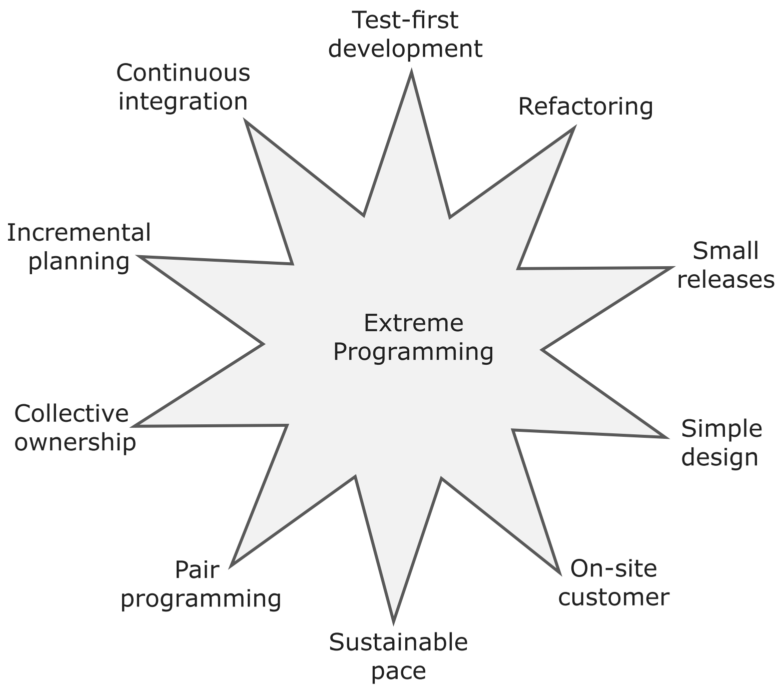
Principle	Description
Involve the customer	Involve customers closely with the software development team. Their role is to provide and prioritize new system requirements and to evaluate each increment of the system.
Embrace change	Expect the features of the product and the details of these features to change as the development team and the product manager learn more about the product. Adapt the software to cope with changes as they are made.
Develop and deliver incrementally	Always develop software products in increments. Test and evaluate each increment as it is developed and feed back required changes to the development team.
Maintain simplicity	Focus on simplicity in both the software being developed and the development process. Wherever possible, do what you can to eliminate complexity from the system.
Focus on people, not the development process	Trust the development team and do not expect everyone to always do things in the same way. Team members should be left to develop their own ways of working without being limited by prescriptive software processes.

2.2 Extreme Programming

The ideas underlying agile methods were developed by a number of different people in the 1990s. However, the most influential work that has changed the culture of software development was the development of Extreme Programming (XP). The name was coined by Kent Beck in 1998 because the approach pushed recognized good practice, such as iterative development, to "extreme" levels. For example, regular integration, in which the work of all programmers in a team is integrated and tested, is good software engineering practice. XP advocates that changed software should be integrated several times per day, as soon as the changes have been tested.

XP focused on new development techniques that were geared to rapid, incremental software development, change, and delivery. Figure 2.2 shows 10 fundamental practices, proposed by the developers of Extreme Programming, that characterize XP.

Figure 2.2 Extreme Programming practices



The developers of XP claim that it is a holistic approach. All of these practices are essential. In reality, however, development teams pick and choose the techniques that they find useful given their organizational culture and the type of software they are writing. Table 2.4 describes XP practices that have become part of mainstream software engineering, particularly for software product development. The other XP practices shown in Figure 2.2 have been less widely adopted but are used in some companies.

I cover these widely-used XP practices, in later chapters of the book. Incremental planning and user stories are covered in Chapter 3, refactoring in Chapter 8, test-driven development in Chapter 9, and continuous integration and small releases in Chapter 10.

You may be surprised that "Simple design" is not on the list of popular XP practices. The developers of XP suggested that the "YAGNI" (You Ain't Gonna Need It) principle should apply when designing software. You should include only functionality that is requested, and you should not add extra code to cope with situations anticipated by the developers. This sounds like a great idea.

Unfortunately, it ignores the fact that customers rarely understand system-wide issues such as security and reliability. You need to design and implement software to take these issues into account. This usually means including code to cope with situations that customers are unlikely to foresee and describe in user stories.

Table 2.4 Widely adopted XP practices

Practice	Description
Incremental planning/ user stories	There is no "grand plan" for the system. Instead, what needs to be implemented (the requirements) in each increment are established in discussions with a customer representative. The requirements are written as user stories. The stories to be included in a release are determined by the time available and their relative priority.
Small releases	The minimal useful set of functionality that provides business value is developed first. Releases of the system are frequent and incrementally add functionality to the previous release.
Test-driven development	Instead of writing code and then tests for that code, developers write the tests first. This helps clarify what the code should actually do and that there is always a "tested" version of the code available. An automated unit test framework is used to run the tests after every change. New code should not "break" code that has already been implemented.
Continuous integration	As soon as the work on a task is complete, it is integrated into the whole system and a new version of the system is created. All unit tests from all developers are run automatically and must be successful before the new version of the system is accepted.
Refactoring	Refactoring means improving the structure, readability, efficiency, and security of a program. All developers are expected to refactor the code as soon as potential code improvements are found. This keeps the code simple and maintainable.

Practices such as having an on-site customer and collective ownership of code are good ideas. An on-site customer works with the team, proposes stories and tests, and learns about the product. However, the reality is that customers and surrogate customers such as product managers have many other things to do. It is difficult for them to find the time to be fully embedded in a development team.

Collective ownership discourages the individual ownership of code, but it has proved to be impractical in many companies. Specialists are needed for some types of code. Some people may work part-time on a project and so cannot participate in its "ownership." Some team members may be psychologically unsuited to this way of working and have no wish to "own" someone else's code.

In pair programming two developers create each code unit. It was proposed by the inventors of XP because they believed the pair could learn from each other and catch each other's mistakes. They suggested that two people working together were more productive than two people working as individuals. However, there is no hard evidence that pair programming is more productive than individual work. Many managers consider pair programming to be unproductive because two people seem to be doing one job.

Working at a sustainable pace, with no overtime, is attractive in principle. Team members should be more productive if they are not tired and stressed. However, it is difficult to convince managers that this sustainable working will help meet tight delivery deadlines.

Extreme programming considers management to be a collective team activity; normally, there is no designated project manager responsible for communicating with management and planning the work of the team. In fact, software development is a business activity and so has to fit with broader business concerns of financing, costs, schedules, hiring and managing staff, and maintaining good customer relationships. This means that management issues cannot simply be left to the development team. There needs to be explicit management where a manager can take account of business needs and priorities as well as technical issues.

2.3 Scrum

In any software business, managers need to know what is going on and whether or not a software development project is likely to deliver the software on time and within its budget. Traditionally, this involves drawing up a project plan that shows a set of milestones (what will be achieved), deliverables (what will be delivered by the team), and deadlines (when a milestone will be reached). The "grand plan" for the project shows everything from start to finish. Progress is assessed by comparing that plan with what has been achieved.

The problem with up-front project planning is that it involves making detailed decisions about the software long before implementation begins. Inevitably things change. New requirements emerge, team members come and go, business priorities evolve, and so on. Almost from the day they are formulated, project plans have to change. Sometimes this means that "finished" work has to be redone. This is inefficient and often delays the final delivery of the software.

On this basis, the developers of agile methods argued that plan-based management is wasteful and unnecessary. It is better to plan incrementally so that the plan can change in response to changing circumstances. At the start of each development cycle, decisions are made on what features should be prioritized, how these should be developed and what each team member should do. Planning should be informal with minimal documentation and with no designated project manager.

Unfortunately, this informal approach to management does not meet the broader business need of progress tracking and assessment. Senior managers do not have the time to become involved in detailed discussions with team members. Managers want someone who can report on progress and take their concerns and priorities back to the development team. They need to know whether the software will be ready by the planned completion date, and they need information to update their business plan for the product.

This requirement for a more proactive approach to agile project management led to the development of Scrum. Unlike XP, Scrum is not based on a set of technical practices. Rather, it is designed to provide a framework for agile project organization with designated individuals (the ScrumMaster and the Product Owner) who act as the interface between the development team and the organization.

The developers of Scrum wanted to emphasize that these individuals were not "traditional" project managers who have the authority to direct the team. So they invented new Scrum terminology for both individuals and team activities (Table 2.5). You need to know this Scrum jargon to understand the Scrum method.

Two key roles in Scrum are not part of other methods:

1. The *Product Owner* is responsible for ensuring that the development team always focuses on the product they are building rather than diverted to technically interesting but less relevant work. In product development, the product manager should normally take on the Product Owner role.

2. The *ScrumMaster* is a Scrum expert whose job is to guide the team in the effective use of the Scrum method. The developers of Scrum emphasize that the ScrumMaster is not a conventional project manager but is a coach for the team. The ScrumMaster has the authority within the team on how Scrum is used. However, in many companies that use Scrum, the ScrumMaster also has some project management responsibilities.

Table 2.5 Scrum terminology

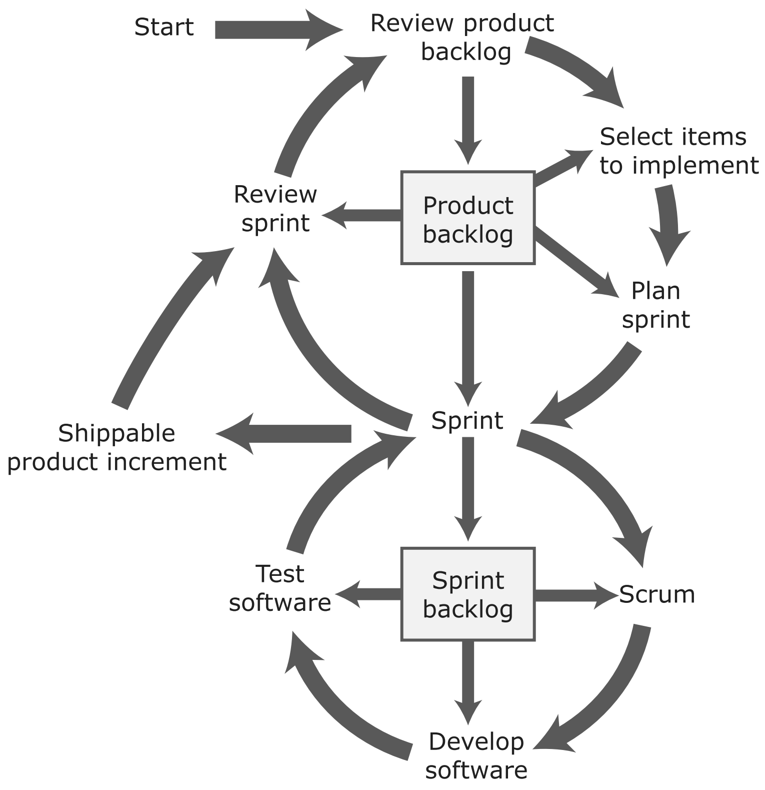
Scrum term	Explanation
Product	The software product that is being developed by the Scrum team.
Product Owner	A team member who is responsible for identifying product features and attributes. The Product Owner reviews work done and helps to test the product.
Product backlog	A to-do list of items such as bugs, features, and product improvements that the Scrum team has not yet completed.
Development team	A small self-organizing team of five to eight people who are responsible for developing the product.
Sprint	A short period, typically two to four weeks, when a product increment is developed.
Scrum	A daily team meeting where progress is reviewed and work to be done that day is discussed and agreed.
ScrumMaster	A team coach who guides the team in the effective use of Scrum.
Potentially shippable product increment	The output of a sprint that is of high enough quality to be deployed for customer use.
Velocity	An estimate of how much work a team can do in a single sprint.

The other Scrum term that may need explanation is “potentially shippable product increment.” This means that the outcome of each sprint should be product-quality code. It should be completely tested, documented, and, if necessary, reviewed. Tests should be delivered with the code. There should always be a high-quality system available that can be demonstrated to management or potential customers.

The Scrum process or sprint cycle is shown in Figure 2.3. The fundamental idea underlying the Scrum process is that software should be developed in a series of “sprints.” A sprint is a fixed-length (timeboxed) activity, with each sprint normally lasting two to four weeks. During a sprint, the team has daily meetings (Scrums) to review the work done so far and to agree on that day’s activities. The “sprint backlog” is used to keep track of work that is to be done during that sprint.

Sprint planning is based on the product backlog, which is a list of all the activities that have to be completed to finish the product being developed. Before a new sprint starts, the product backlog is reviewed. The highest-priority

Figure 2.3 Scrum cycle



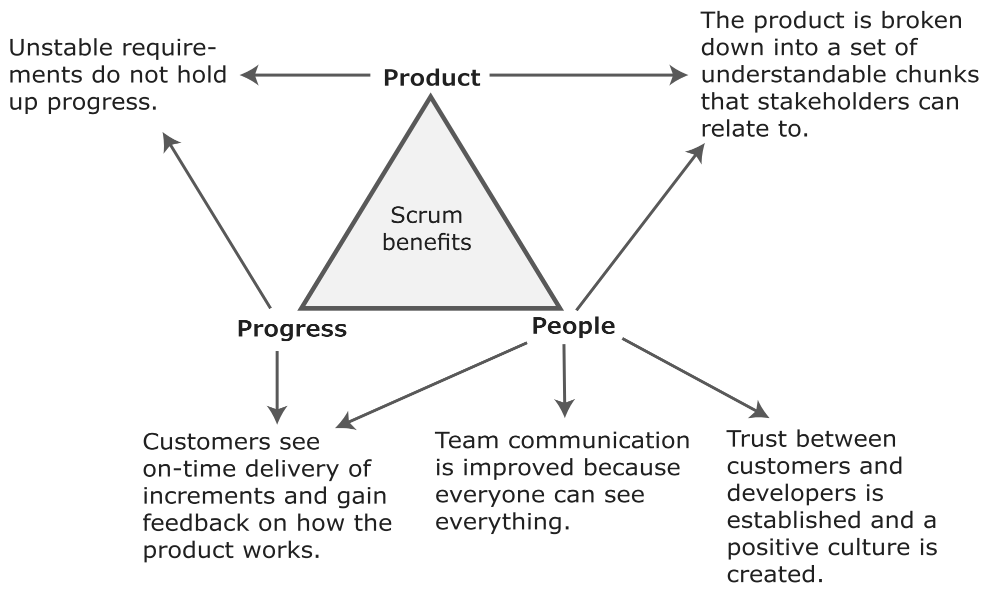
items are selected for implementation in the next sprint. Team members work together to plan the sprint by analyzing the selected items to create the sprint backlog. This is a list of activities to be completed during the sprint.

During implementation, the team implements as many of the sprint backlog items as they can in the fixed time period allowed for the sprint. Incomplete items are returned to the product backlog. Sprints are never extended to finish an incomplete item.

A sprint produces either a shippable product increment that can be delivered to customers or an internal deliverable. Internal deliverables, such as a product prototype or an architectural design, provide information for future sprints. If the sprint output is part of the final product, it should be complete. Unless the team has to change the software functionality, it should not have to do any more work on that software increment in future sprints.

On completion of a sprint, a review meeting is held involving all team members. The team discusses what went well during the sprint, what problems arose, and how the problems were tackled. Team members also reflect on the effectiveness of the tools and methods used. The aim of this meeting is for the team to learn from each other to avoid problems and to improve productivity in later sprints.

Figure 2.4 The top five benefits of using Scrum



The key benefits that come from using Scrum relate to the product being developed, the progress of the project, and the people involved (Figure 2.4).

Scrum has been very influential in the development of agile software engineering. It provides a framework for "doing" software engineering without prescribing the engineering techniques that should be used. However, Scrum is prescriptive in defining roles and the Scrum process. In *The Scrum Guide*[2], the "keepers" of the Scrum method state:

> *Scrum's roles, artefacts, events, and rules are immutable and although implementing only parts of Scrum is possible, the result is not Scrum. Scrum exists only in its entirety and functions well as a container for other techniques, methodologies, and practices.*

That is, they believe you should not pick and choose a subset of Scrum practices. Rather, you should take the whole of the method on board. It seems to me that this inflexibility contradicts the fundamental agile principle that individuals and interactions should be preferred over processes and tools. This principle suggests that individuals should be able to adapt and modify Scrum to suit their circumstances.

In some circumstances, I think it makes sense to use some of the ideas from Scrum without strictly following the method or defining the roles as exactly envisaged in Scrum. In general, "pure Scrum" with its various roles can't

[2]*The Scrum Guide* This definitive guide to the Scrum method defines all the Scrum roles and activities. (K. Schwaber and J. Sutherland, 2013).

Table 2.6 Examples of product backlog items

1. As a teacher, I want to be able to configure the group of tools that are available to individual classes. (feature)
2. As a parent, I want to be able to view my children's work and the assessments made by their teachers. (feature)
3. As a teacher of young children, I want a pictorial interface for children with limited reading ability. (user request)
4. Establish criteria for the assessment of open source software that might be used as a basis for parts of this system. (development activity)
5. Refactor user interface code to improve understandability and performance. (engineering improvement)
6. Implement encryption for all personal user data. (engineering improvement)

be used by teams with fewer than five people. So, if you are working with a smaller development team, you have to modify the method.

Small software development teams are the norm in startups, where the whole company may be the development team. They are also common in educational and research settings, where teams develop software as part of their learning, and in larger manufacturing companies, where software development is part of a broader product development process.

I think that motivated teams should make their own decisions about how to use Scrum or a Scrum-like process. However, I recommend that three important features of Scrum should be part of any product development process: product backlogs, timeboxed sprints, and self-organizing teams.

2.3.1 Product backlogs

The product backlog is a list of what needs to be done to complete the development of the product. The items on this list are called product backlog items (PBIs). The product backlog may include a variety of different items such as product features to be implemented, user requests, essential development activities, and desirable engineering improvements. The product backlog should always be prioritized so that the items that will be implemented first are at the top of the list.

Product backlog items are initially described in broad terms without much detail. For example, the items shown in Table 2.6 might be included in the product backlog for a version of the iLearn system, which I introduced in Chapter 1. In Chapter 3 I explain how system features can be identified from a product vision. These then become PBIs. I also explain how user stories can be used to identify PBIs.

Table 2.7 Product backlog item states

Heading	Description
Ready for consideration	These are high-level ideas and feature descriptions that will be considered for inclusion in the product. They are tentative so may radically change or may not be included in the final product.
Ready for refinement	The team has agreed that this is an important item that should be implemented as part of the current development. There is a reasonably clear definition of what is required. However, work is needed to understand and refine the item.
Ready for implementation	The PBI has enough detail for the team to estimate the effort involved and to implement the item. Dependencies on other items have been identified.

Table 2.6 shows different types of product backlog items. The first three items are user stories that are related to features of the product that have to be implemented. The fourth item is a team activity. The team must spend time deciding how to select open-source software that may be used in later increments. This type of activity should be specifically accounted for as a PBI rather than taken as an implicit activity that takes up team members' time. The last two items are concerned with engineering improvements to the software. These don't lead to new software functionality.

PBIs may be specified at a high level and the team decides how to implement these items. For example, the development team is best placed to decide how to refactor code for efficiency and understandability. It does not make sense to refine this item in more detail at the start of a sprint. However, high-level feature definitions usually need refinement so that team members have a clear idea of what is required and can estimate the work involved.

Items in the product backlog are considered to be in one of three states, as shown in Table 2.7. The product backlog is continually changed and extended during the project as new items are added and items in one state are analyzed and moved to a more refined state.

A critical part of the Scrum agile process is the product backlog review, which should be the first item in the sprint planning process. In this review, the product backlog is analyzed and backlog items are prioritized and refined. Backlog reviews may also take place during sprints as the team learns more about the system. Team members may modify or refine existing backlog items or add new items to be implemented in a later sprint. During a product backlog review, items may be moved from one state to another.

Figure 2.5 Product backlog activities

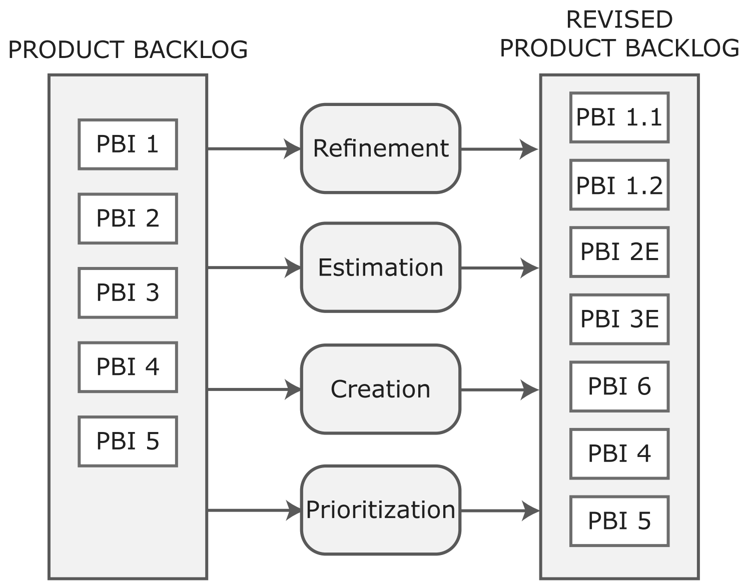


Figure 2.5 shows the four operations that may modify the product backlog. In this example, backlog item 1 has been split into two items, items 2 and 3 have been estimated, items 4 and 5 have been re-prioritized, and item 6 has been added. Notice that the new item 6 has a higher priority than existing items 4 and 5.

The Scrum community sometimes uses the term "backlog grooming" to cover these four activities:

1. *Refinement* Existing PBIs are analyzed and refined to create more detailed PBIs. This may also lead to the creation of new backlog items.
2. *Estimation* The team estimates the amount of work required to implement a PBI and adds this assessment to each analyzed PBI.
3. *Creation* New items are added to the backlog. These may be new features suggested by the product manager, required feature changes, engineering improvements, or process activities such as the assessment of development tools that might be used.
4. *Prioritization* The PBIs are reordered to take new information and changed circumstances into account.

Backlog prioritization is a whole-team activity in which decisions are made on which items to work on during a sprint. Input from product managers is essential because they should know about customer needs and priorities. The highest-priority items are refined by the development team to create a "sprint backlog," which is a list of more detailed implementation items. In situations such as speculative product development or research system development, where there is no specified product owner, the team should collectively prioritize the items.

Items that are ready for implementation should always have an associated estimate of how much effort is needed to implement them. Estimates are essential for sprint planning because a team uses them to decide how much work they can take on for an individual sprint. This effort estimate is an input to the prioritization activity. Sometimes it makes sense to place a higher priority on the items that deliver the most value for the least effort.

PBI estimates provide an indication of the effort required to complete each item. Two metrics are commonly used:

1. *Effort required* The amount of effort may be expressed in person-hours or person-days—that is, the number of hours or days it would take one person to implement that PBI. This is not the same as calendar time. Several people may work on an item, which may shorten the calendar time required. Alternatively, a developer may have other responsibilities that prevent full-time work on a project. Then the calendar time required is longer than the effort estimate.
2. *Story points* Story points are an arbitrary estimate of the effort involved in implementing a PBI, taking into account the size of the task, its complexity, the technology that may be required, and the "unknown" characteristics of the work. Story points were derived originally by comparing user stories, but they can be used for estimating any kind of PBI. Story points are estimated relatively. The team agrees on the story points for a baseline task. Other tasks are then estimated by comparison with this baseline—for example, more or less complex, larger or smaller, and so on. The advantage of story points is that they are more abstract than effort required because all story points should be the same, irrespective of individual abilities.

Effort estimation is hard, especially at the beginning of a project when a team has little or no previous experience with this type of work or when technologies new to the team are used. Estimates are based on the subjective judgment of the team members, and initial estimates are inevitably wrong. Estimates usually improve, however, as the team gains experience with the product and its development process.

The Scrum method recommends a team-based estimation approach called "Planning Poker," which I don't go into here. The rationale is that teams should be able to make better estimates than individuals. However, there is no convincing empirical evidence showing that collective estimation is better than estimates made by experienced, individual developers.

After a number of sprints have been completed, it becomes possible for a team to estimate its "velocity." Simplistically, a team's velocity is the sum of the size estimates of the items that have been completed during a fixed-time sprint. For example, assume that PBIs are estimated in story points and, in consecutive sprints, the team implements 17, 14, 16, and 19 story points. The team's velocity is therefore between 16 and 17 story points per sprint.

Velocity is used to decide how many PBIs a team can realistically commit to in each sprint. In the above example, the team should commit to about 17 story points. Velocity may also be used as a measure of productivity. Teams should try to refine how they work so that their velocity improves over the course of a project.

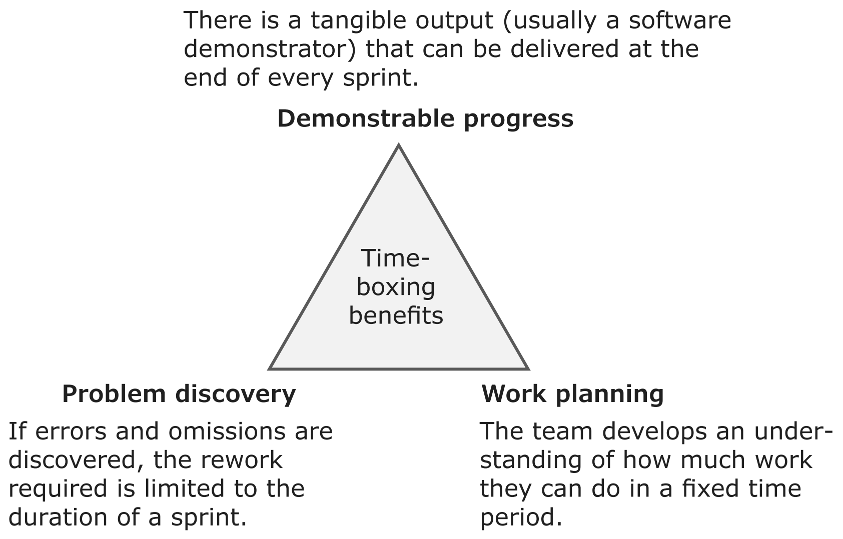
The product backlog is a shared, "living" document that is regularly updated during product development. It is usually too large to fit on a white-board, so it makes sense to maintain it as a shared digital document. Several specialized tools that support Scrum include facilities to share and revise product backlogs. Some companies may decide to buy these tools for their software developers.

Small companies or groups with limited resources can use a shared document system such as Office 365 or Google docs. These low-cost systems don't require new software to be bought and installed. If you are starting out using product backlogs in your development process, I recommend this general approach to gain experience before you decide whether you need specialized tools for backlog management.

2.3.2 Timeboxed sprints

A Scrum concept that is useful in any agile development process is timeboxed sprints. Timeboxing means that a fixed time is allocated for completing an activity. At the end of the timebox, work on the activity stops whether or not the planned work has been completed. Sprints are short activities (one to four weeks) and take place between defined start and end dates. During a sprint, the team works on the items from the product backlog. The product is therefore developed in a series of sprints, each of which delivers an increment of the product or supporting software.

Figure 2.6 Benefits of using timeboxed sprints


There is a tangible output (usually a software demonstrator) that can be delivered at the end of every sprint.
Demonstrable progress
Time-boxing benefits
Problem discovery
If errors and omissions are discovered, the rework required is limited to the duration of a sprint.
Work planning
The team develops an understanding of how much work they can do in a fixed time period.

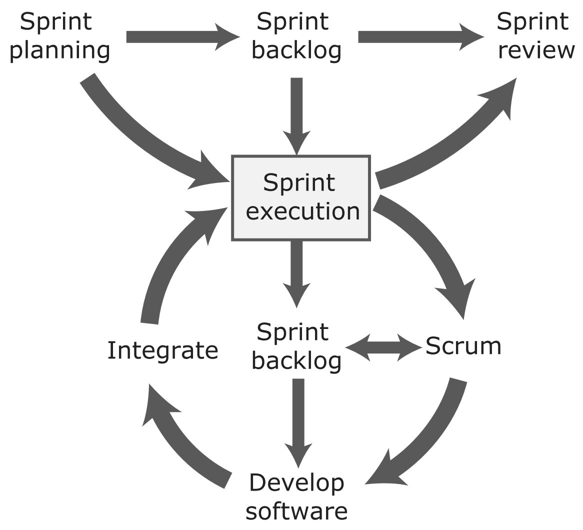
Incremental development is a fundamental part of all agile methods, and I think Scrum has got it right in insisting that the time spent on each increment should be the same. In Figure 2.6, I show three important benefits that arise from using timeboxed sprints.

Every sprint involves three fundamental activities:

1. *Sprint planning* Work items to be completed during that sprint are selected and, if necessary, refined to create a sprint backlog. This should not last more than a day at the beginning of the sprint.
2. *Sprint execution* The team works to implement the sprint backlog items that have been chosen for that sprint. If it is impossible to complete all of the sprint backlog items, the time for the sprint is not extended. Rather, the unfinished items are returned to the product backlog and queued for a future sprint.
3. *Sprint reviewing* The work done during the sprint is reviewed by the team and (possibly) external stakeholders. The team reflects on what went well and what went wrong during the sprint, with a view to improving the work process.

Figure 2.7 shows the cycle of these activities and a more detailed breakdown of sprint execution. The sprint backlog is created during the planning process and drives the development activities when the sprint is executed.

Figure 2.7 Sprint activities



Each sprint should start with a planning meeting at which team members collectively decide on the PBIs to be implemented during the sprint. The inputs to this activity are the product backlog items that are ready for implementation and information from the Product Owner about which of these PBIs has the highest priority.

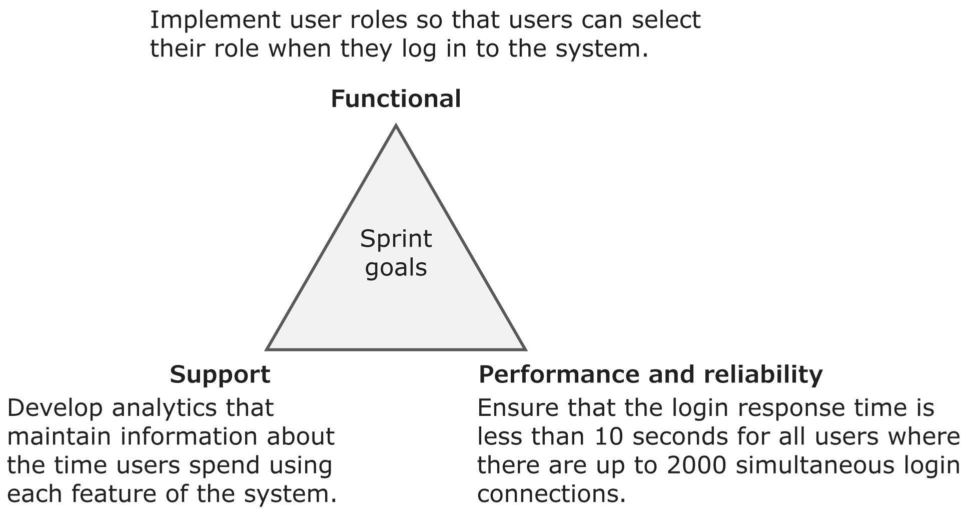
When planning a sprint, the team do three things:

- agree on a sprint goal;
- decide on the list of items from the product backlog that should be implemented;
- create a sprint backlog, a more detailed version of the product backlog that records the work to be done during the sprint.

The sprint goal is a succinct statement of what the team plans to achieve during a sprint. It could be the implementation of a product feature, the development of some essential product infrastructure, or the improvement of some product attribute, such as its performance. It should be possible to decide objectively whether or not the goal has been achieved by the end of the sprint. Figure 2.8 shows the three types of sprint goals and gives an example of each type.

Functional sprint goals relate to the implementation of system features for end-users. Performance and reliability goals relate to improvements in the performance, efficiency, reliability, and security of the system. Support goals cover ancillary activities such as developing infrastructure software or designing the system architecture.

Figure 2.8 Sprint goals


Implement user roles so that users can select their role when they log in to the system.
Functional
Sprint goals
Support
Develop analytics that maintain information about the time users spend using each feature of the system.
Performance and reliability
Ensure that the login response time is less than 10 seconds for all users where there are up to 2000 simultaneous login connections.

You should always consider the highest-priority items on the product backlog when deciding on a sprint goal. The team chooses these items for implementation at the same time as the sprint goal is being set. It makes sense to choose a coherent set of high-priority items that are consistent with the sprint goal. Sometimes items of lower priority in the product backlog are chosen because they are closely related to other items that are part of the overall sprint goal.

As a general rule, the sprint goal should not be changed during the sprint. Sometimes, however, the sprint goal has to be changed if unexpected problems are discovered or if the team finds a way to implement a feature more quickly than originally estimated. In these cases, the scope of the goal may be reduced or extended.

Once a sprint goal has been established, the team should discuss and decide on a sprint plan. As I explained, PBIs should have an associated effort estimate, which is a critical input to the sprint planning process. It's important that a team does not try to implement too many items during the sprint. Overcommitment may make it impossible to achieve the sprint goal.

The velocity of the team is another important input to the sprint planning process. The velocity reflects how much work the team can normally cover in a sprint. You may estimate story points as I have explained, where the team's velocity is the number of story points it can normally implement in a two-week or four-week sprint. This approach obviously makes sense for a team that has a stable velocity.

A team's velocity might be unstable, however, which means that the number of PBIs completed varies from one sprint to another. Velocity may

Table 2.8 Scrums

A scrum is a short, daily meeting that is usually held at the beginning of the day. During a scrum, all team members share information, describe their progress since the previous day's scrum, and present problems that have arisen and plans for the coming day. This means that everyone on the team knows what is going on and, if problems arise, can re-plan short-term work to cope with them.

Scrum meetings should be short and focused. To dissuade team members from getting involved in long discussions, scrums are sometimes organized as "stand-up" meetings where there are no chairs in the meeting room.

During a scrum, the sprint backlog is reviewed. Completed items are removed from it. New items may be added to the backlog as new information emerges. The team then decides who should work on sprint backlog items that day.

be unstable if the team membership changes, if easier items are assigned a higher priority than items that are harder to implement, or if a team includes inexperienced members who improve as the project progresses. If a team's velocity is unstable or unknown, then you have to take a more intuitive approach to choosing the number of PBIs to be implemented during a sprint.

The sprint backlog is a list of work items to be completed during the sprint. Sometimes, a PBI can be transferred directly to the sprint backlog. However, the team normally breaks down each PBI into smaller tasks that are added to the sprint backlog. All team members then discuss how these tasks will be allocated. Each task should have a relatively short duration—one or two days at most—so that the team can assess its progress during the daily sprint meeting. The sprint backlog should be much shorter than the product backlog, so it can be maintained on a shared whiteboard. The whole team can see what items are to be implemented and what items have been completed.

The focus of a sprint is the development of product features or infrastructure and the team works to create the planned software increment. To facilitate cooperation, team members coordinate their work every day in a short meeting called a scrum (Table 2.8). The Scrum method is named after these meetings, which are an essential part of the method. They are a way for teams to communicate—nothing like scrums in the game of rugby.

The Scrum method does not include specific technical development practices; the team may use any agile practices they think are appropriate. Some teams like pair programming; others prefer that members work individually. However, I recommend that two practices always be used in code development sprints:

Table 2.9 Code completeness checklist

State	Description
Reviewed	The code has been reviewed by another team member who has checked that it meets agreed coding standards, is understandable, includes appropriate comments, and has been refactored if necessary.
Unit tested	All unit tests have been run automatically and all tests have executed successfully.
Integrated	The code has been integrated with the project codebase and no integration errors have been reported.
Integration tested	All integration tests have been run automatically and all tests have been executed successfully.
Accepted	Acceptance tests have been run if appropriate and the Product Owner or the development team has confirmed that the product backlog item has been completed.

1. *Test automation* As far as possible, product testing should be automated. You should develop a suite of executable tests that can be run at any time. I explain how to do this in Chapter 9.
2. *Continuous integration* Whenever anyone makes changes to the software components they are developing, these components should be immediately integrated with other components to create a system. This system should then be tested to check for unanticipated component interaction problems. I explain continuous integration in Chapter 10.

The aim of a sprint is to develop a "potentially shippable product increment." Of course, the software will not necessarily be released to customers, but it should not require further work before it can be released. This means different things for different types of software, so it is important that a team establish a "definition of done," which specifies what has to be completed for code that is developed during a sprint.

For example, for a software product that is being developed for external customers, the team may create a checklist that applies to all the software that is being developed. Table 2.9 is an example of a checklist that can be used to judge the completeness of an implemented feature.

If it is not possible to complete all items on this checklist during a sprint, the unfinished items should be added to the product backlog for future implementation. The sprint should never be extended to complete unfinished items.

At the end of each sprint, there is a review meeting that involves the whole team. This meeting has three purposes. First, it reviews whether or not the sprint has met its goal. Second, it sets out any new problems and issues that have emerged during the sprint. Finally, it is a way for a team to reflect on how they can improve the way they work. Members discuss what has gone well, what has gone badly, and what improvements could be made.

The review may involve external stakeholders as well as the development team. The team should be honest about what has and hasn't been achieved during the sprint so that the output of the review is a definitive assessment of the state of the product being developed. If items are unfinished or if new items have been identified, these should be added to the product backlog. The Product Owner has the ultimate authority to decide whether or not the goal of the sprint has been achieved. They should confirm that the implementation of the selected product backlog items is complete.

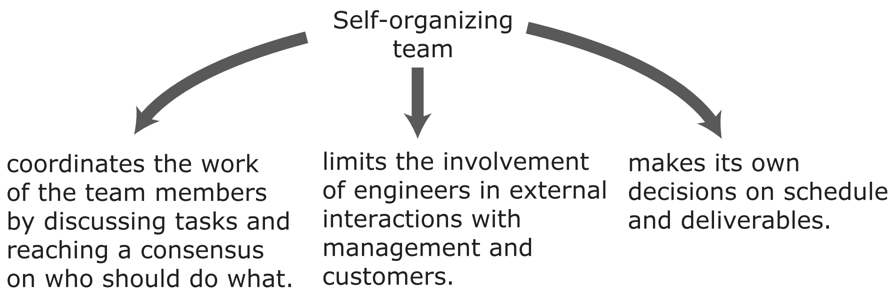
An important part of a sprint review is a process review, in which the team reflects on its own way of working and how Scrum has been used. The aim of a process review is to identify ways to improve and to discuss how to use Scrum more productively. Over the course of a development process, a Scrum team should try to continually improve its effectiveness.

During the review, the team may discuss communication breakdowns, good and bad experiences with tools and the development environment, technical practices that have been adopted, reusable software and libraries that have been discovered, and other issues. If problems have been identified, the team should discuss how they should be addressed in future sprints. For example, a decision may be made to investigate alternative tools to those being used by the team. If aspects of the work have been successful, the team may explicitly schedule time so that experience can be shared and good practice adopted across the team.

2.3.3 Self-organizing teams

A fundamental principle of all agile development methods is that the software development team should be self-organizing. Self-organizing teams don't have a project manager who assigns tasks and makes decisions for the team. Rather, as shown in Figure 2.9, they make their own decisions. Self-organizing teams work by discussing issues and making decisions by consensus.

Figure 2.9 Self-organizing teams


Self-organizing team
coordinates the work of the team members by discussing tasks and reaching a consensus on who should do what.
limits the involvement of engineers in external interactions with management and customers.
makes its own decisions on schedule and deliverables.

The ideal Scrum team size is between five and eight people—large enough to be diverse yet small enough to communicate informally and effectively and to agree on the priorities of the team. Because teams have to tackle diverse tasks, it's important to have a range of expertise in a Scrum team such as networking, user experience, database design and so on.

In reality, it may not be possible to form *ideal* teams. In a non-commercial setting such as a university, teams are smaller and made up of people who have largely the same skill set. There is a worldwide shortage of software engineers, so it is sometimes impossible to find people with the right mix of skills and experience. A team may change during a project as people leave and new members are hired. Some team members may work part-time or from home.

The advantage of an effective self-organizing team is that it can be cohesive and can adapt to change. Because the team rather than individuals takes responsibility for the work, the team can cope with people leaving and joining the group. Good team communication means that team members inevitably learn something about each other's areas. They can therefore compensate, to some extent, when people leave the team.

In a managed team, the project manager coordinates the work. Managers look at the work to be done and assign tasks to team members. Project managers have to arrange things so that work is not delayed because one team member is waiting for others to finish their work. They have to tell all team members about problems and other factors that may delay the work. Team members are not encouraged to take responsibility for coordination and communication.

In a self-organizing team, the team itself has to put in place ways to coordinate the work and communicate issues to all team members. The developers of Scrum assumed that team members are co-located. They work in the same office and can communicate informally. If one team member needs to know something about what another has done, they simply talk to each other to find out. There is no need for people to document their work for others to read. Daily scrums mean that the team members know what's been done and what others are doing.

The Scrum approach embodies the essentials for coordination in a self-managed team—namely, good informal communication and regular meetings to ensure that everyone is up to speed. Team members explain their work and are aware of team progress and possible risks that may affect the team. However, there are practical reasons why informal verbal communication may not always work:

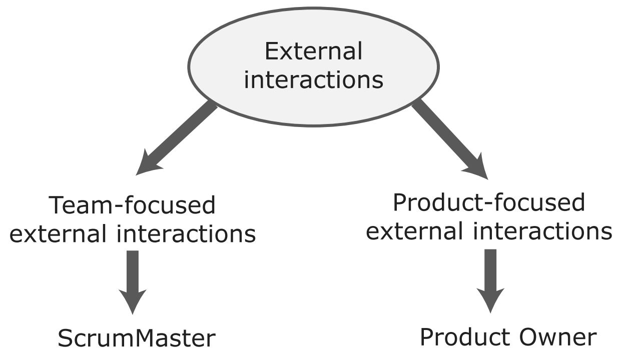
1. Scrum assumes that the team is made up of full-time workers who share a workspace. In reality, team members may be part-time and may work in different places. For a student project, team members may take different classes at different times, so it may be difficult to find a time slot where all team members can meet.
2. Scrum assumes that all team members can attend a morning meeting to coordinate the work for the day. This does not take into account that team members may work flexible hours (for example, because of child care responsibilities) or may work part-time on several projects. They are, therefore, not available every morning.

If co-located working with daily meetings is impractical, then the team must work out other ways to communicate. Messaging systems, such as Slack, can be effective for informal communications. The benefit of messaging is that all messages are recorded so that people can catch up on conversations that they missed. Messaging does not have the immediacy of face-to-face communication, but it is better than email or shared documents for coordination.

Talking to each other is the best way for team members to coordinate work and to communicate what has gone well and what problems have arisen. Daily meetings may be impossible, but agile teams really have to schedule progress meetings regularly even if all members can't attend or have to attend virtually using teleconferencing. Members who can't attend should submit a short summary of their own progress so that the team can assess how well the work is going.

All development teams, even those working in small startups or non-commercial developments, have some external interactions. Some interactions will help the team understand what customers require from the software

Figure 2.10 Managing external interactions


External interactions
Team-focused external interactions
Product-focused external interactions
ScrumMaster
Product Owner

product being developed. Others will be with company management and other parts of the company, such as human resources and marketing.

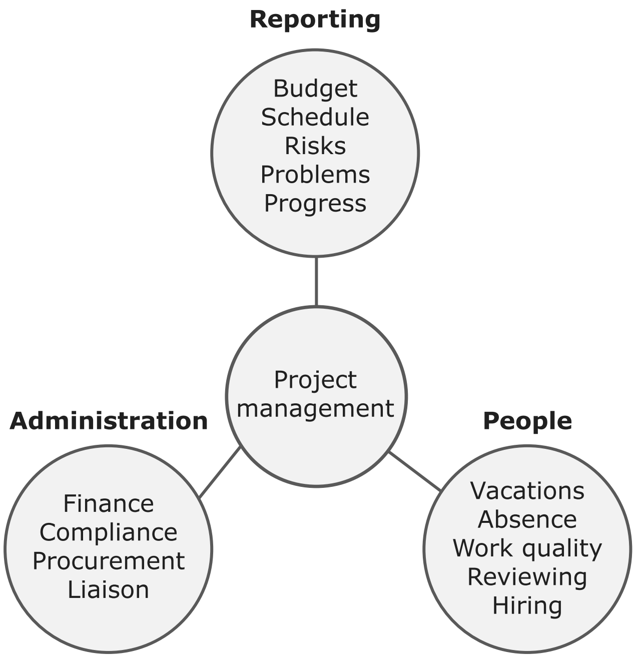
In a Scrum project, the ScrumMaster and the Product Owner should be jointly responsible for managing interactions with people outside the team (Figure 2.10).

Product Owners are responsible for interactions with current and potential customers as well as the company's sales and marketing staff. Their job is to understand what customers are looking for in a software product and to identify possible barriers to the adoption and use of the product being developed. They should understand the innovative features of the product to establish how customers can benefit from them. Product Owners use this knowledge to help develop the product backlog and to prioritize backlog items for implementation.

The ScrumMaster role has a dual function. Part of the role is to work closely with the team, coaching them in the use of Scrum and working on the development of the product backlog. *The Scrum Guide* states that the ScrumMaster should also work with people outside of the team to "remove impediments"; that is, they should deal with external problems and queries and represent the team to the wider organization. The intention is for the team to be able to work on software development without external interference or distractions.

Whether or not a team is using Scrum or some other agile approach, you need to pay attention to these issues. In small teams, it may be impossible to have different people take care of interactions with customers and interactions with managers. The best approach may be for one person to take on both of these roles and to work part-time on software development. The key requirement for "external communicators" is good communication and people skills

Figure 2.11 Project management responsibilities



so that they can talk about the team's work in a way that people outside the team can understand and relate to.

The ScrumMaster is not a conventional project manager. The job is to help team members use the Scrum method effectively and to ensure that they are not distracted by external considerations. However, in all commercial projects, someone has to take on essential project management responsibilities (Figure 2.11).

The Scrum Guide and many Scrum books (although not Rubin's book that I've included in Recommended Reading) simply ignore these issues. But they are a reality of working in all but the smallest companies. A self-organizing team has to appoint a team member to take on management tasks. Because of the need to maintain continuity of communication with people outside of the group, sharing the management tasks among team members is not a viable approach.

In response to this issue, Rubin suggests that it may sometimes be appropriate for a project manager outside of the team to act for several Scrum teams. I think this idea is unworkable for three reasons:

1. Small companies may not have the resources to support dedicated project managers.

2. Many project management tasks require detailed knowledge of a team's work. If a project manager is working across several teams, it may be impossible to know the work of each team in detail.
3. Self-organizing teams are cohesive and tend to resent being told what to do by people outside of the team. Members are liable to obstruct, rather than support, an external project manager.

In my opinion it is unrealistic for the ScrumMaster role to exclude project management responsibilities. ScrumMasters know the work going on and are in by far the best position to provide accurate information and project plans and progress.

KEY POINTS

- The best way to develop software products is to use agile software engineering methods that are geared to rapid product development and delivery.
- Agile methods are based on iterative development and the minimization of overheads during the development process.
- Extreme Programming (XP) is an influential agile method that introduced agile development practices such as user stories, test-first development, and continuous integration. These are now mainstream software development activities.
- Scrum is an agile method that focuses on agile planning and management. Unlike XP, it does not define the engineering practices to be used. The development team may use any technical practices they consider appropriate for the product being developed.
- In Scrum, work to be done is maintained in a product backlog, a list of work items to be completed. Each increment of the software implements some of the work items from the product backlog.
- Sprints are fixed-time activities (usually two to four weeks) in which a product increment is developed. Increments should be potentially shippable; that is, they should not need further work before they are delivered.
- A self-organizing team is a development team that organizes the work to be done by discussion and agreement among team members.
- Scrum practices, such as the product backlog, sprints, and self-organizing teams, can be used in any agile development process, even if other aspects of Scrum are not used.