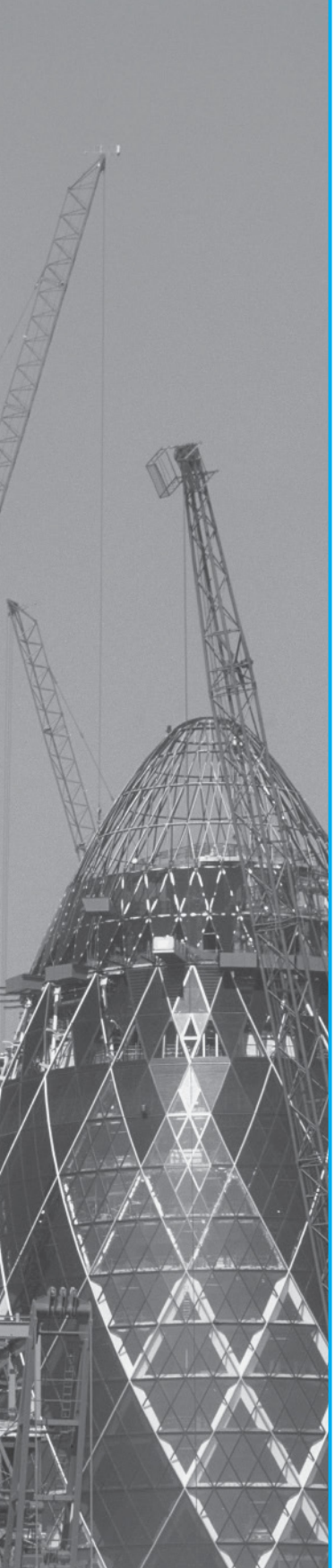

3 Agile software development

Objectives

The objective of this chapter is to introduce you to agile software development methods. When you have read the chapter, you will:

- understand the rationale for agile software development methods, the agile manifesto, and the differences between agile and plan-driven development;
- know about important agile development practices such as user stories, refactoring, pair programming and test-first development;
- understand the Scrum approach to agile project management;
- understand the issues of scaling agile development methods and combining agile approaches with plan-driven approaches in the development of large software systems.

Contents

Businesses now operate in a global, rapidly changing environment. They have to respond to new opportunities and markets, changing economic conditions and the emergence of competing products and services. Software is part of almost all business operations, so new software has to be developed quickly to take advantage of new opportunities and to respond to competitive pressure. Rapid software development and delivery is therefore the most critical requirement for most business systems. In fact, businesses may be willing to trade off software quality and compromise on requirements if they can deploy essential new software quickly.

Because these businesses are operating in a changing environment, it is practically impossible to derive a complete set of stable software requirements. Requirements change because customers find it impossible to predict how a system will affect working practices, how it will interact with other systems, and what user operations should be automated. It may only be after a system has been delivered and users gain experience with it that the real requirements become clear. Even then, external factors drive requirements change.

Plan-driven software development processes that completely specify the requirements and then design, build, and test a system are not geared to rapid software development. As the requirements change or as requirements problems are discovered, the system design or implementation has to be reworked and retested. As a consequence, a conventional waterfall or specification-based process is usually a lengthy one, and the final software is delivered to the customer long after it was originally specified.

For some types of software, such as safety-critical control systems, where a complete analysis of the system is essential, this plan-driven approach is the right one. However, in a fast-moving business environment, it can cause real problems. By the time the software is available for use, the original reason for its procurement may have changed so radically that the software is effectively useless. Therefore, for business systems in particular, development processes that focus on rapid software development and delivery are essential.

The need for rapid software development and processes that can handle changing requirements has been recognized for many years (Larman and Basili 2003). However, faster software development really took off in the late 1990s with the development of the idea of "agile methods" such as Extreme Programming (Beck 1999), Scrum (Schwaber and Beedle 2001), and DSDM (Stapleton 2003).

Rapid software development became known as agile development or agile methods. These agile methods are designed to produce useful software quickly. All of the agile methods that have been proposed share a number of common characteristics:

1. The processes of specification, design and implementation are interleaved. There is no detailed system specification, and design documentation is minimized or generated automatically by the programming environment used to implement the system. The user requirements document is an outline definition of the most important characteristics of the system.

2. The system is developed in a series of increments. End-users and other system stakeholders are involved in specifying and evaluating each increment.

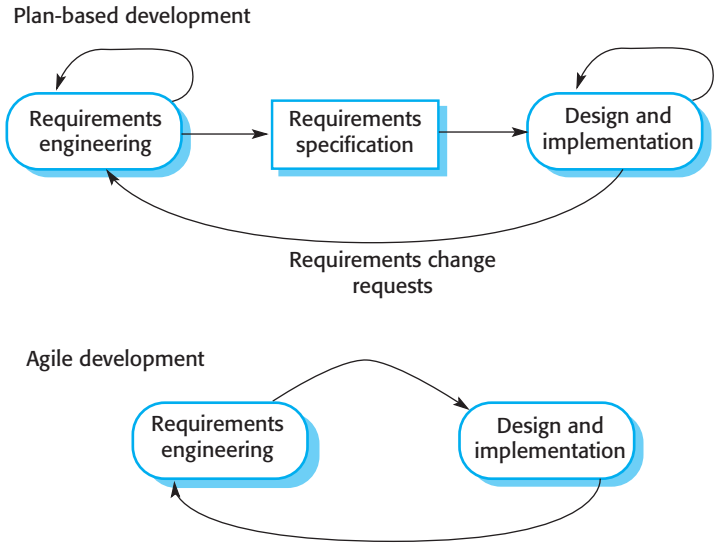

Figure 3.1 Plan-driven and agile development

They may propose changes to the software and new requirements that should be implemented in a later version of the system.

3. Extensive tool support is used to support the development process. Tools that may be used include automated testing tools, tools to support configuration management, and system integration and tools to automate user interface production.

Agile methods are incremental development methods in which the increments are small, and, typically, new releases of the system are created and made available to customers every two or three weeks. They involve customers in the development process to get rapid feedback on changing requirements. They minimize documentation by using informal communications rather than formal meetings with written documents.

Agile approaches to software development consider design and implementation to be the central activities in the software process. They incorporate other activities, such as requirements elicitation and testing, into design and implementation. By contrast, a plan-driven approach to software engineering identifies separate stages in the software process with outputs associated with each stage. The outputs from one stage are used as a basis for planning the following process activity.

Figure 3.1 shows the essential distinctions between plan-driven and agile approaches to system specification. In a plan-driven software development process, iteration occurs within activities, with formal documents used to communicate between stages of the process. For example, the requirements will evolve, and, ultimately, a requirements specification will be produced. This is then an input to the design and implementation process. In an agile approach, iteration occurs across activities. Therefore, the requirements and the design are developed together rather than separately.

In practice, as I explain in Section 3.4.1, plan-driven processes are often used along with agile programming practices, and agile methods may incorporate some planned

activities apart from programming and testing. It is perfectly feasible, in a plan-driven process, to allocate requirements and plan the design and development phase as a series of increments. An agile process is not inevitably code-focused, and it may produce some design documentation. Agile developers may decide that an iteration should not produce new code but rather should produce system models and documentation.

3.1 Agile methods

In the 1980s and early 1990s, there was a widespread view that the best way to achieve better software was through careful project planning, formalized quality assurance, use of analysis and design methods supported by software tools, and controlled and rigorous software development processes. This view came from the software engineering community that was responsible for developing large, long-lived software systems such as aerospace and government systems.

This plan-driven approach was developed for software developed by large teams, working for different companies. Teams were often geographically dispersed and worked on the software for long periods of time. An example of this type of software is the control systems for a modern aircraft, which might take up to 10 years from initial specification to deployment. Plan-driven approaches involve a significant overhead in planning, designing, and documenting the system. This overhead is justified when the work of multiple development teams has to be coordinated, when the system is a critical system, and when many different people will be involved in maintaining the software over its lifetime.

However, when this heavyweight, plan-driven development approach is applied to small and medium-sized business systems, the overhead involved is so large that it dominates the software development process. More time is spent on how the system should be developed than on program development and testing. As the system requirements change, rework is essential and, in principle at least, the specification and design have to change with the program.

Dissatisfaction with these heavyweight approaches to software engineering led to the development of agile methods in the late 1990s. These methods allowed the development team to focus on the software itself rather than on its design and documentation. They are best suited to application development where the system requirements usually change rapidly during the development process. They are intended to deliver working software quickly to customers, who can then propose new and changed requirements to be included in later iterations of the system. They aim to cut down on 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 (http://agilemanifesto.org) issued by the leading developers of these methods. This manifesto states:

Principle	Description
Customer involvement	Customers should be closely involved throughout the development process. Their role is provide and prioritize new system requirements and to evaluate the iterations of the system.
Embrace change	Expect the system requirements to change, and so design the system to accommodate these changes.
Incremental delivery	The software is developed in increments, with the customer specifying the requirements to be included in each increment.
Maintain simplicity	Focus on simplicity in both the software being developed and in the development process. Wherever possible, actively work to eliminate complexity from the system.
People, not process	The skills of the development team should be recognized and exploited. Team members should be left to develop their own ways of working without prescriptive processes.

Figure 3.2 The principles of agile methods

We are uncovering better ways of developing software by doing it and helping others 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

That is, while there is value in the items on the right, we value the items on the left more[†].

All agile methods suggest that software should be developed and delivered incrementally. These methods are based on different agile processes but they share a set of principles, based on the agile manifesto, and so they have much in common. I have listed these principles in Figure 3.2.

Agile methods have been particularly successful for two kinds of system development.

1. Product development where a software company is developing a small or medium-sized product for sale. Virtually all software products and apps are now developed using an agile approach.
2. Custom system development within an organization, where there is a clear commitment from the customer to become involved in the development process and where there are few external stakeholders and regulations that affect the software.

Agile methods work well in these situations because it is possible to have continuous communications between the product manager or system customer and the development team. The software itself is a stand-alone system rather than tightly integrated with other systems being developed at the same time. Consequently, there is no need to coordinate parallel development streams. Small and medium-sized

[†]http://agilemanifesto.org/

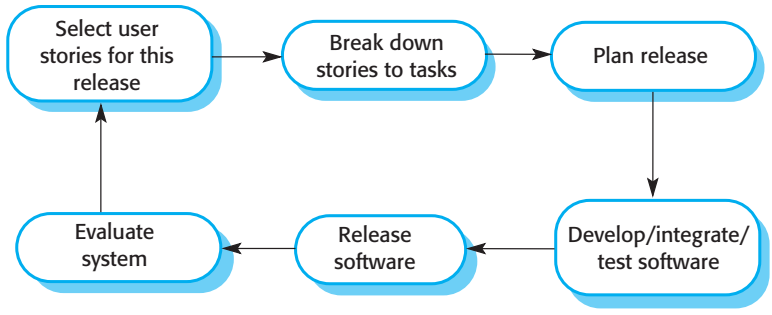

Figure 3.3 The XP release cycle

systems can be developed by co-located teams, so informal communications among team members work well.

3.2 Agile development techniques

The ideas underlying agile methods were developed around the same time by a number of different people in the 1990s. However, perhaps the most significant approach to changing software development culture was the development of Extreme Programming (XP). The name was coined by Kent Beck (Beck 1998) because the approach was developed by pushing recognized good practice, such as iterative development, to "extreme" levels. For example, in XP, several new versions of a system may be developed by different programmers, integrated, and tested in a day. Figure 3.3 illustrates the XP process to produce an increment of the system that is being developed.

In XP, requirements are expressed as scenarios (called user stories), which are implemented directly as a series of tasks. Programmers work in pairs and develop tests for each task before writing the code. All tests must be successfully executed when new code is integrated into the system. There is a short time gap between releases of the system.

Extreme programming was controversial as it introduced a number of agile practices that were quite different from the development practice of that time. These practices are summarized in Figure 3.4 and reflect the principles of the agile manifesto:

1. Incremental development is supported through small, frequent releases of the system. Requirements are based on simple customer stories or scenarios that are used as a basis for deciding what functionality should be included in a system increment.
2. Customer involvement is supported through the continuous engagement of the customer in the development team. The customer representative takes part in the development and is responsible for defining acceptance tests for the system.
3. People, not process, are supported through pair programming, collective ownership of the system code, and a sustainable development process that does not involve excessively long working hours.

Principle or practice	Description
Collective ownership	The pairs of developers work on all areas of the system, so that no islands of expertise develop and all the developers take responsibility for all of the code. Anyone can change anything.
Continuous integration	As soon as the work on a task is complete, it is integrated into the whole system. After any such integration, all the unit tests in the system must pass.
Incremental planning	Requirements are recorded on "story cards," and the stories to be included in a release are determined by the time available and their relative priority. The developers break these stories into development "tasks." See Figures 3.5 and 3.6.
On-site customer	A representative of the end-user of the system (the Customer) should be available full time for the use of the XP team. In an extreme programming process, the customer is a member of the development team and is responsible for bringing system requirements to the team for implementation.
Pair programming	Developers work in pairs, checking each other's work and providing the support to always do a good job.
Refactoring	All developers are expected to refactor the code continuously as soon as potential code improvements are found. This keeps the code simple and maintainable.
Simple design	Enough design is carried out to meet the current requirements and no more.
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 first release.
Sustainable pace	Large amounts of overtime are not considered acceptable, as the net effect is often to reduce code quality and medium-term productivity.
Test first development	An automated unit test framework is used to write tests for a new piece of functionality before that functionality itself is implemented.

Figure 3.4 Extreme programming practices

4. Change is embraced through regular system releases to customers, test-first development, refactoring to avoid code degeneration, and continuous integration of new functionality.

5. Maintaining simplicity is supported by constant refactoring that improves code quality and by using simple designs that do not unnecessarily anticipate future changes to the system.

In practice, the application of Extreme Programming as originally proposed has proved to be more difficult than anticipated. It cannot be readily integrated with the management practices and culture of most businesses. Therefore, companies adopting agile methods pick and choose those XP practices that are most appropriate for their way of working. Sometimes these are incorporated into their own development processes but, more commonly, they are used in conjunction with a management-focused agile method such as Scrum (Rubin 2013).

Prescribing medication

Kate is a doctor who wishes to prescribe medication for a patient attending a clinic. The patient record is already displayed on her computer so she clicks on the medication field and can select 'current medication', 'new medication' or 'formulary'.

If she selects 'current medication', the system asks her to check the dose; If she wants to change the dose, she enters the new dose then confirms the prescription.

If she chooses 'new medication', the system assumes that she knows which medication to prescribe. She types the first few letters of the drug name. The system displays a list of possible drugs starting with these letters. She chooses the required medication and the system responds by asking her to check that the medication selected is correct. She enters the dose then confirms the prescription.

If she chooses 'formulary', the system displays a search box for the approved formulary. She can then search for the drug required. She selects a drug and is asked to check that the medication is correct. She enters the dose then confirms the prescription.

The system always checks that the dose is within the approved range. If it isn't, Kate is asked to change the dose.

After Kate has confirmed the prescription, it will be displayed for checking. She either clicks 'OK' or 'Change'. If she clicks 'OK', the prescription is recorded on the audit database. If she clicks on 'Change', she reenters the 'Prescribing medication' process.

Figure 3.5 A "prescribing medication" story

I am not convinced that XP on its own is a practical agile method for most companies, but its most significant contribution is probably the set of agile development practices that it introduced to the community. I discuss the most important of these practices in this section.

3.2.1 User stories

Software requirements always change. To handle these changes, agile methods do not have a separate requirements engineering activity. Rather, they integrate requirements elicitation with development. To make this easier, the idea of "user stories" was developed where a user story is a scenario of use that might be experienced by a system user.

As far as possible, the system customer works closely with the development team and discusses these scenarios with other team members. Together, they develop a "story card" that briefly describes a story that encapsulates the customer needs. The development team then aims to implement that scenario in a future release of the software. An example of a story card for the Mentcare system is shown in Figure 3.5. This is a short description of a scenario for prescribing medication for a patient.

User stories may be used in planning system iterations. Once the story cards have been developed, the development team breaks these down into tasks (Figure 3.6) and estimates the effort and resources required for implementing each task. This usually involves discussions with the customer to refine the requirements. The customer then prioritizes the stories for implementation, choosing those stories that can be

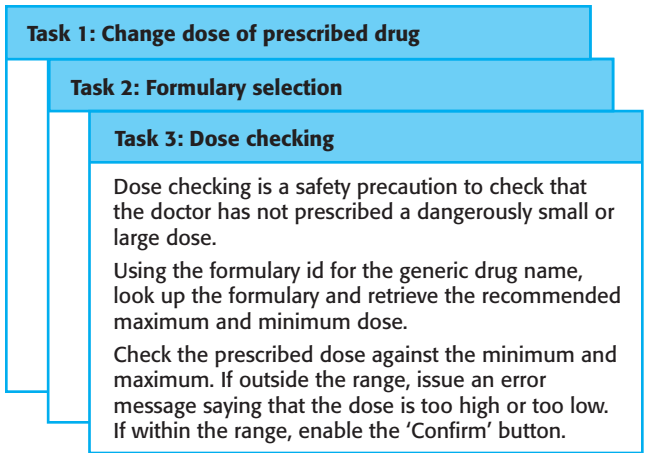

Figure 3.6 Examples of task cards for prescribing medication

used immediately to deliver useful business support. The intention is to identify useful functionality that can be implemented in about two weeks, when the next release of the system is made available to the customer.

Of course, as requirements change, the unimplemented stories change or may be discarded. If changes are required for a system that has already been delivered, new story cards are developed and again, the customer decides whether these changes should have priority over new functionality.

The idea of user stories is a powerful one—people find it much easier to relate to these stories than to a conventional requirements document or use cases. User stories can be helpful in getting users involved in suggesting requirements during an initial predevelopment requirements elicitation activity. I discuss this in more detail in Chapter 4.

The principal problem with user stories is completeness. It is difficult to judge if enough user stories have been developed to cover all of the essential requirements of a system. It is also difficult to judge if a single story gives a true picture of an activity. Experienced users are often so familiar with their work that they leave things out when describing it.

3.2.2 Refactoring

A fundamental precept of traditional software engineering is that you should design for change. That is, you should anticipate future changes to the software and design it so that these changes can be easily implemented. Extreme programming, however, has discarded this principle on the basis that designing for change is often wasted effort. It isn't worth taking time to add generality to a program to cope with change. Often the changes anticipated never materialize, or completely different change requests may actually be made.

Of course, in practice, changes will always have to be made to the code being developed. To make these changes easier, the developers of XP suggested that the code being developed should be constantly refactored. Refactoring (Fowler et al. 1999) means that the programming team look for possible improvements to the software and implements

them immediately. When team members see code that can be improved, they make these improvements even in situations where there is no immediate need for them.

A fundamental problem of incremental development is that local changes tend to degrade the software structure. Consequently, further changes to the software become harder and harder to implement. Essentially, the development proceeds by finding workarounds to problems, with the result that code is often duplicated, parts of the software are reused in inappropriate ways, and the overall structure degrades as code is added to the system. Refactoring improves the software structure and readability and so avoids the structural deterioration that naturally occurs when software is changed.

Examples of refactoring include the reorganization of a class hierarchy to remove duplicate code, the tidying up and renaming of attributes and methods, and the replacement of similar code sections, with calls to methods defined in a program library. Program development environments usually include tools for refactoring. These simplify the process of finding dependencies between code sections and making global code modifications.

In principle, when refactoring is part of the development process, the software should always be easy to understand and change as new requirements are proposed. In practice, this is not always the case. Sometimes development pressure means that refactoring is delayed because the time is devoted to the implementation of new functionality. Some new features and changes cannot readily be accommodated by code-level refactoring and require that the architecture of the system be modified.

3.2.3 Test-first development

As I discussed in the introduction to this chapter, one of the important differences between incremental development and plan-driven development is in the way that the system is tested. With incremental development, there is no system specification that can be used by an external testing team to develop system tests. As a consequence, some approaches to incremental development have a very informal testing process, in comparison with plan-driven testing.

Extreme Programming developed a new approach to program testing to address the difficulties of testing without a specification. Testing is automated and is central to the development process, and development cannot proceed until all tests have been successfully executed. The key features of testing in XP are:

1. test-first development,
2. incremental test development from scenarios,
3. user involvement in the test development and validation, and
4. the use of automated testing frameworks.

XP's test-first philosophy has now evolved into more general test-driven development techniques (Jeffries and Melnik 2007). I believe that test-driven development is one of the most important innovations in software engineering. Instead of writing code and then writing tests for that code, you write the tests before you write the code. This

Test 4: Dose checking

Input:
1. A number in mg representing a single dose of the drug.
2. A number representing the number of single doses per day.

Tests:
1. Test for inputs where the single dose is correct but the frequency is too high.
2. Test for inputs where the single dose is too high and too low.
3. Test for inputs where the single dose * frequency is too high and too low.
4. Test for inputs where single dose * frequency is in the permitted range.

Output:
OK or error message indicating that the dose is outside the safe range.

Figure 3.7 Test case description for dose checking

means that you can run the test as the code is being written and discover problems during development. I discuss test-driven development in more depth in Chapter 8.

Writing tests implicitly defines both an interface and a specification of behavior for the functionality being developed. Problems of requirements and interface misunderstandings are reduced. Test-first development requires there to be a clear relationship between system requirements and the code implementing the corresponding requirements. In XP, this relationship is clear because the story cards representing the requirements are broken down into tasks and the tasks are the principal unit of implementation.

In test-first development, the task implementers have to thoroughly understand the specification so that they can write tests for the system. This means that ambiguities and omissions in the specification have to be clarified before implementation begins. Furthermore, it also avoids the problem of "test-lag." This may happen when the developer of the system works at a faster pace than the tester. The implementation gets further and further ahead of the testing and there is a tendency to skip tests, so that the development schedule can be maintained.

XP's test-first approach assumes that user stories have been developed, and these have been broken down into a set of task cards, as shown in Figure 3.6. Each task generates one or more unit tests that check the implementation described in that task. Figure 3.7 is a shortened description of a test case that has been developed to check that the prescribed dose of a drug does not fall outside known safe limits.

The role of the customer in the testing process is to help develop acceptance tests for the stories that are to be implemented in the next release of the system. As I explain in Chapter 8, acceptance testing is the process whereby the system is tested using customer data to check that it meets the customer's real needs.

Test automation is essential for test-first development. Tests are written as executable components before the task is implemented. These testing components should be stand-alone, should simulate the submission of input to be tested, and should check that the result meets the output specification. An automated test framework is a system that makes it easy to write executable tests and submit a set of tests for execution. Junit (Tahchiev et al. 2010) is a widely used example of an automated testing framework for Java programs.

As testing is automated, there is always a set of tests that can be quickly and easily executed. Whenever any functionality is added to the system, the tests can be run and problems that the new code has introduced can be caught immediately.

Test-first development and automated testing usually result in a large number of tests being written and executed. However, there are problems in ensuring that test coverage is complete:

1. Programmers prefer programming to testing, and sometimes they take shortcuts when writing tests. For example, they may write incomplete tests that do not check for all possible exceptions that may occur.
2. Some tests can be very difficult to write incrementally. For example, in a complex user interface, it is often difficult to write unit tests for the code that implements the "display logic" and workflow between screens.

It is difficult to judge the completeness of a set of tests. Although you may have a lot of system tests, your test set may not provide complete coverage. Crucial parts of the system may not be executed and so will remain untested. Therefore, although a large set of frequently executed tests may give the impression that the system is complete and correct, this may not be the case. If the tests are not reviewed and further tests are written after development, then undetected bugs may be delivered in the system release.

3.2.4 Pair programming

Another innovative practice that was introduced in XP is that programmers work in pairs to develop the software. The programming pair sits at the same computer to develop the software. However, the same pair do not always program together. Rather, pairs are created dynamically so that all team members work with each other during the development process.

Pair programming has a number of advantages.

1. It supports the idea of collective ownership and responsibility for the system. This reflects Weinberg's idea of egoless programming (Weinberg 1971) where the software is owned by the team as a whole and individuals are not held responsible for problems with the code. Instead, the team has collective responsibility for resolving these problems.
2. It acts as an informal review process because each line of code is looked at by at least two people. Code inspections and reviews (Chapter 24) are effective in discovering a high percentage of software errors. However, they are time consuming to organize and, typically, introduce delays into the development process. Pair programming is a less formal process that probably doesn't find as many errors as code inspections. However, it is cheaper and easier to organize than formal program inspections.
3. It encourages refactoring to improve the software structure. The problem with asking programmers to refactor in a normal development environment is that effort

involved is expended for long-term benefit. An developer who spends time refactoring may be judged to be less efficient than one who simply carries on developing code. Where pair programming and collective ownership are used, others benefit immediately from the refactoring so they are likely to support the process.

You might think that pair programming would be less efficient than individual programming. In a given time, a pair of developers would produce half as much code as two individuals working alone. Many companies that have adopted agile methods are suspicious of pair programming and do not use it. Other companies mix pair and individual programming with an experienced programmer working with a less experienced colleague when they have problems.

Formal studies of the value of pair programming have had mixed results. Using student volunteers, Williams and her collaborators (Williams et al. 2000) found that productivity with pair programming seems to be comparable to that of two people working independently. The reasons suggested are that pairs discuss the software before development and so probably have fewer false starts and less rework. Furthermore, the number of errors avoided by the informal inspection is such that less time is spent repairing bugs discovered during the testing process.

However, studies with more experienced programmers did not replicate these results (Arisholm et al. 2007). They found that there was a significant loss of productivity compared with two programmers working alone. There were some quality benefits, but these did not fully compensate for the pair-programming overhead. Nevertheless, the sharing of knowledge that happens during pair programming is very important as it reduces the overall risks to a project when team members leave. In itself, this may make pair programming worthwhile.

3.3 Agile project management

In any software business, managers need to know what is going on and whether or not a project is likely to meet its objectives and deliver the software on time with the proposed budget. Plan-driven approaches to software development evolved to meet this need. As I discussed in Chapter 23, managers draw up a plan for the project showing what should be delivered, when it should be delivered, and who will work on the development of the project deliverables. A plan-based approach requires a manager to have a stable view of everything that has to be developed and the development processes.

The informal planning and project control that was proposed by the early adherents of agile methods clashed with this business requirement for visibility. Teams were self-organizing, did not produce documentation, and planned development in very short cycles. While this can and does work for small companies developing software products, it is inappropriate for larger companies who need to know what is going on in their organization.

Like every other professional software development process, agile development has to be managed so that the best use is made of the time and resources available to

Scrum term	Definition
Development team	A self-organizing group of software developers, which should be no more than seven people. They are responsible for developing the software and other essential project documents.
Potentially shippable product increment	The software increment that is delivered from a sprint. The idea is that this should be "potentially shippable," which means that it is in a finished state and no further work, such as testing, is needed to incorporate it into the final product. In practice, this is not always achievable.
Product backlog	This is a list of "to do" items that the Scrum team must tackle. They may be feature definitions for the software, software requirements, user stories, or descriptions of supplementary tasks that are needed, such as architecture definition or user documentation.
Product owner	An individual (or possibly a small group) whose job is to identify product features or requirements, prioritize these for development, and continuously review the product backlog to ensure that the project continues to meet critical business needs. The Product Owner can be a customer but might also be a product manager in a software company or other stakeholder representative.
Scrum	A daily meeting of the Scrum team that reviews progress and prioritizes work to be done that day. Ideally, this should be a short face-to-face meeting that includes the whole team.
ScrumMaster	The ScrumMaster is responsible for ensuring that the Scrum process is followed and guides the team in the effective use of Scrum. He or she is responsible for interfacing with the rest of the company and for ensuring that the Scrum team is not diverted by outside interference. The Scrum developers are adamant that the ScrumMaster should not be thought of as a project manager. Others, however, may not always find it easy to see the difference.
Sprint	A development iteration. Sprints are usually 2 to 4 weeks long.
Velocity	An estimate of how much product backlog effort a team can cover in a single sprint. Understanding a team's velocity helps them estimate what can be covered in a sprint and provides a basis for measuring improving performance.

Figure 3.8 Scrum terminology

the team. To address this issue, the Scrum agile method was developed (Schwaber and Beedle 2001; Rubin 2013) to provide a framework for organizing agile projects and, to some extent at least, provide external visibility of what is going on. The developers of Scrum wished to make clear that Scrum was not a method for project management in the conventional sense, so they deliberately invented new terminology, such as ScrumMaster, which replaced names such as project manager. Figure 3.8 summarizes Scrum terminology and what it means.

Scrum is an agile method insofar as it follows the principles from the agile manifesto, which I showed in Figure 3.2. However, it focuses on providing a framework for agile project organization, and it does not mandate the use of specific development

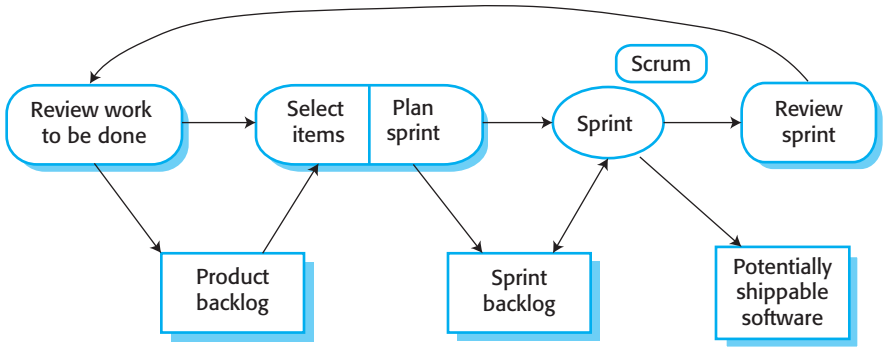

Figure 3.9 The Scrum sprint cycle

practices such as pair programming and test-first development. This means that it can be more easily integrated with existing practice in a company. Consequently, as agile methods have become a mainstream approach to software development, Scrum has emerged as the most widely used method.

The Scrum process or sprint cycle is shown in Figure 3.9. The input to the process is the product backlog. Each process iteration produces a product increment that could be delivered to customers.

The starting point for the Scrum sprint cycle is the product backlog—the list of items such as product features, requirements, and engineering improvement that have to be worked on by the Scrum team. The initial version of the product backlog may be derived from a requirements document, a list of user stories, or other description of the software to be developed.

While the majority of entries in the product backlog are concerned with the implementation of system features, other activities may also be included. Sometimes, when planning an iteration, questions that cannot be easily answered come to light and additional work is required to explore possible solutions. The team may carry out some prototyping or trial development to understand the problem and solution. There may also be backlog items to design the system architecture or to develop system documentation.

The product backlog may be specified at varying levels of detail, and it is the responsibility of the Product Owner to ensure that the level of detail in the specification is appropriate for the work to be done. For example, a backlog item could be a complete user story such as that shown in Figure 3.5, or it could simply be an instruction such as "Refactor user interface code" that leaves it up to the team to decide on the refactoring to be done.

Each sprint cycle lasts a fixed length of time, which is usually between 2 and 4 weeks. At the beginning of each cycle, the Product Owner prioritizes the items on the product backlog to define which are the most important items to be developed in that cycle. Sprints are never extended to take account of unfinished work. Items are returned to the product backlog if these cannot be completed within the allocated time for the sprint.

The whole team is then involved in selecting which of the highest priority items they believe can be completed. They then estimate the time required to complete these items. To make these estimates, they use the velocity attained in previous

sprints, that is, how much of the backlog could be covered in a single sprint. This leads to the creation of a sprint backlog—the work to be done during that sprint. The team self-organizes to decide who will work on what, and the sprint begins.

During the sprint, the team holds short daily meetings (Scrums) to review progress and, where necessary, to re-prioritize work. During the Scrum, all team members share information, describe their progress since the last meeting, bring up problems that have arisen, and state what is planned for the following day. Thus, everyone on the team knows what is going on and, if problems arise, can re-plan short-term work to cope with them. Everyone participates in this short-term planning; there is no top-down direction from the ScrumMaster.

The daily interactions among Scrum teams may be coordinated using a Scrum board. This is an office whiteboard that includes information and post-it notes about the Sprint backlog, work done, unavailability of staff, and so on. This is a shared resource for the whole team, and anyone can change or move items on the board. It means that any team member can, at a glance, see what others are doing and what work remains to be done.

At the end of each sprint, there is a review meeting, which involves the whole team. This meeting has two purposes. First, it is a means of process improvement. The team reviews the way they have worked and reflects on how things could have been done better. Second, it provides input on the product and the product state for the product backlog review that precedes the next sprint.

While the ScrumMaster is not formally a project manager, in practice ScrumMasters take this role in many organizations that have a conventional management structure. They report on progress to senior management and are involved in longer-term planning and project budgeting. They may be involved in project administration (agreeing on holidays for staff, liaising with HR, etc.) and hardware and software purchases.

In various Scrum success stories (Schatz and Abdelshafi 2005; Mulder and van Vliet 2008; Bellouiti 2009), the things that users like about the Scrum method are:

1. The product is broken down into a set of manageable and understandable chunks that stakeholders can relate to.
2. Unstable requirements do not hold up progress.
3. The whole team has visibility of everything, and consequently team communication and morale are improved.
4. Customers see on-time delivery of increments and gain feedback on how the product works. They are not faced with last-minute surprises when a team announces that software will not be delivered as expected.
5. Trust between customers and developers is established, and a positive culture is created in which everyone expects the project to succeed.

Scrum, as originally designed, was intended for use with co-located teams where all team members could get together every day in stand-up meetings. However, much software development now involves distributed teams, with team members located in different places around the world. This allows companies to take advantage

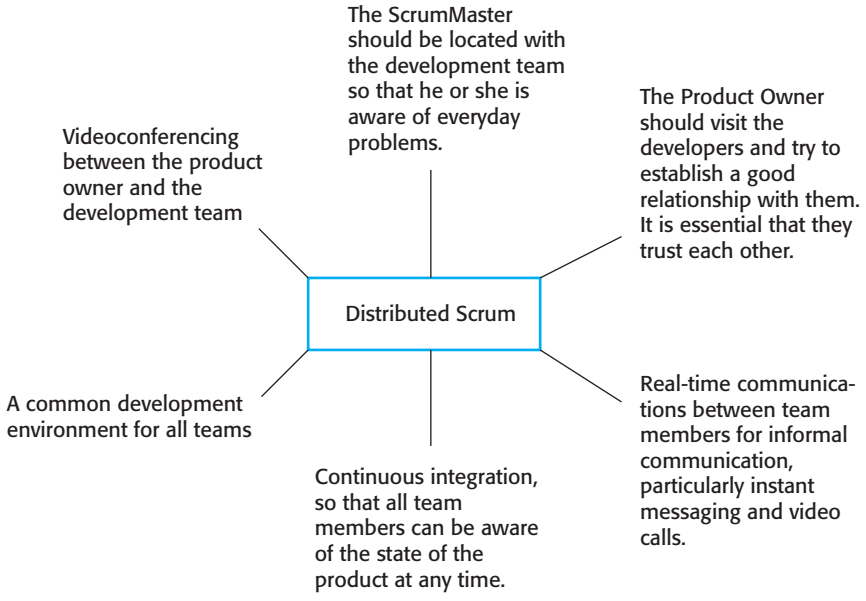

Figure 3.10 Distributed Scrum

of lower cost staff in other countries, makes access to specialist skills possible, and allows for 24-hour development, with work going on in different time zones.

Consequently, there have been developments of Scrum for distributed development environments and multi-team working. Typically, for offshore development, the product owner is in a different country from the development team, which may also be distributed. Figure 3.10 shows the requirements for Distributed Scrum (Deemer 2011).

3.4 Scaling agile methods

Agile methods were developed for use by small programming teams that could work together in the same room and communicate informally. They were originally used by for the development of small and medium-sized systems and software products. Small companies, without formal processes or bureaucracy, were enthusiastic initial adopters of these methods.

Of course, the need for faster delivery of software, which is more suited to customer needs, also applies to both larger systems and larger companies. Consequently, over the past few years, a lot of work has been put into evolving agile methods for both large software systems and for use in large companies.

Scaling agile methods has closely related facets:

1. Scaling up these methods to handle the development of large systems that are too big to be developed by a single small team.
2. Scaling out these methods from specialized development teams to more widespread use in a large company that has many years of software development experience.

Of course, scaling up and scaling out are closely related. Contracts to develop large software systems are usually awarded to large organizations, with multiple teams working on the development project. These large companies have often experimented with agile methods in smaller projects, so they face the problems of scaling up and scaling out at the same time.

There are many anecdotes about the effectiveness of agile methods, and it has been suggested that these can lead to orders of magnitude improvements in productivity and comparable reductions in defects. Ambler (Ambler 2010), an influential agile method developer, suggests that these productivity improvements are exaggerated for large systems and organizations. He suggests that an organization moving to agile methods can expect to see productivity improvement across the organization of about 15% over 3 years, with similar reductions in the number of product defects.

3.4.1 Practical problems with agile methods

In some areas, particularly in the development of software products and apps, agile development has been incredibly successful. It is by far the best approach to use for this type of system. However, agile methods may not be suitable for other types of software development, such as embedded systems engineering or the development of large and complex systems.

For large, long-lifetime systems that are developed by a software company for an external client, using an agile approach presents a number of problems.

1. The informality of agile development is incompatible with the legal approach to contract definition that is commonly used in large companies.
2. Agile methods are most appropriate for new software development rather than for software maintenance. Yet the majority of software costs in large companies come from maintaining their existing software systems.
3. Agile methods are designed for small co-located teams, yet much software development now involves worldwide distributed teams.

Contractual issues can be a major problem when agile methods are used. When the system customer uses an outside organization for system development, a contract for the software development is drawn up between them. The software requirements document is usually part of that contract between the customer and the supplier. Because the interleaved development of requirements and code is fundamental to agile methods, there is no definitive statement of requirements that can be included in the contract.

Consequently, agile methods have to rely on contracts in which the customer pays for the time required for system development rather than the development of a specific set of requirements. As long as all goes well, this benefits both the customer and the developer. However, if problems arise, then there may be difficult disputes over who is to blame and who should pay for the extra time and resources required to resolve the problems.

As I explain in Chapter 9, a huge amount of software engineering effort goes into the maintenance and evolution of existing software systems. Agile practices, such as incremental delivery, design for change, and maintaining simplicity all make sense when software is being changed. In fact, you can think of an agile development process as a process that supports continual change. If agile methods are used for software product development, new releases of the product or app simply involve continuing the agile approach.

However, where maintenance involves a custom system that must be changed in response to new business requirements, there is no clear consensus on the suitability of agile methods for software maintenance (Bird 2011; Kilner 2012). Three types of problems can arise:

- lack of product documentation
- keeping customers involved
- development team continuity

Formal documentation is supposed to describe the system and so make it easier for people changing the system to understand. In practice, however, formal documentation is rarely updated and so does not accurately reflect the program code. For this reason, agile methods enthusiasts argue that it is a waste of time to write this documentation and that the key to implementing maintainable software is to produce high-quality, readable code. The lack of documentation should not be a problem in maintaining systems developed using an agile approach.

However, my experience of system maintenance is that the most important document is the system requirements document, which tells the software engineer what the system is supposed to do. Without such knowledge, it is difficult to assess the impact of proposed system changes. Many agile methods collect requirements informally and incrementally and do not create a coherent requirements document. The use of agile methods may therefore make subsequent system maintenance more difficult and expensive. This is a particular problem if development team continuity cannot be maintained.

A key challenge in using an agile approach to maintenance is keeping customers involved in the process. While a customer may be able to justify the full-time involvement of a representative during system development, this is less likely during maintenance where changes are not continuous. Customer representatives are likely to lose interest in the system. Therefore, it is likely that alternative mechanisms, such as change proposals, discussed in Chapter 25, will have to be adapted to fit in with an agile approach.

Another potential problem that may arise is maintaining continuity of the development team. Agile methods rely on team members understanding aspects of the system without having to consult documentation. If an agile development team is broken up, then this implicit knowledge is lost and it is difficult for new team members to build up the same understanding of the system and its components. Many programmers prefer to work on new development to software maintenance, and so they are unwilling to continue to work on a software system after the first release has been delivered. Therefore, even when the intention is to keep the development team together, people leave if they are assigned maintenance tasks.

Principle	Practice
Customer involvement	This depends on having a customer who is willing and able to spend time with the development team and who can represent all system stakeholders. Often, customer representatives have other demands on their time and cannot play a full part in the software development. Where there are external stakeholders, such as regulators, it is difficult to represent their views to the agile team.
Embrace change	Prioritizing changes can be extremely difficult, especially in systems for which there are many stakeholders. Typically, each stakeholder gives different priorities to different changes.
Incremental delivery	Rapid iterations and short-term planning for development does not always fit in with the longer-term planning cycles of business planning and marketing. Marketing managers may need to know product features several months in advance to prepare an effective marketing campaign.
Maintain simplicity	Under pressure from delivery schedules, team members may not have time to carry out desirable system simplifications.
People, not process	Individual team members may not have suitable personalities for the intense involvement that is typical of agile methods and therefore may not interact well with other team members.

Figure 3.11 Agile principles and organizational practice

3.4.2 Agile and plan-driven methods

A fundamental requirement of scaling agile methods is to integrate them with plan-driven approaches. Small startup companies can work with informal and short-term planning, but larger companies have to have longer-term plans and budgets for investment, staffing, and business development. Their software development must support these plans, so longer-term software planning is essential.

Early adopters of agile methods in the first decade of the 21st century were enthusiasts and deeply committed to the agile manifesto. They deliberately rejected the plan-driven approach to software engineering and were reluctant to change the initial vision of agile methods in any way. However, as organizations saw the value and benefits of an agile approach, they adapted these methods to suit their own culture and ways of working. They had to do this because the principles underlying agile methods are sometimes difficult to realize in practice (Figure 3.11).

To address these problems, most large "agile" software development projects combine practices from plan-driven and agile approaches. Some are mostly agile, and others are mostly plan-driven but with some agile practices. To decide on the balance between a plan-based and an agile approach, you have to answer a range of technical, human and organizational questions. These relate to the system being developed, the development team, and the organizations that are developing and procuring the system (Figure 3.12).

Agile methods were developed and refined in projects to develop small to medium-sized business systems and software products, where the software developer controls the specification of the system. Other types of system have attributes such as size, complexity, real-time response, and external regulation that mean a "pure" agile approach is

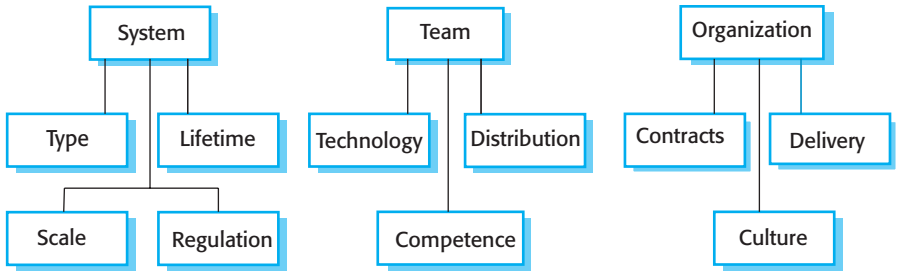

Figure 3.12 Factors influencing the choice of plan-based or agile development

unlikely to work. There needs to be some up-front planning, design, and documentation in the systems engineering process. Some of the key issues are as follows:

1. How large is the system that is being developed? Agile methods are most effective when the system can be developed with a relatively small co-located team who can communicate informally. This may not be possible for large systems that require larger development teams, so a plan-driven approach may have to be used.
2. What type of system is being developed? Systems that require a lot of analysis before implementation (e.g., real-time system with complex timing requirements) usually need a fairly detailed design to carry out this analysis. A plan-driven approach may be best in those circumstances.
3. What is the expected system lifetime? Long-lifetime systems may require more design documentation to communicate the original intentions of the system developers to the support team. However, supporters of agile methods rightly argue that documentation is frequently not kept up to date and is not of much use for long-term system maintenance.
4. Is the system subject to external regulation? If a system has to be approved by an external regulator (e.g., the Federal Aviation Administration approves software that is critical to the operation of an aircraft), then you will probably be required to produce detailed documentation as part of the system safety case.

Agile methods place a great deal of responsibility on the development team to cooperate and communicate during the development of the system. They rely on individual engineering skills and software support for the development process. However, in reality, not everyone is a highly skilled engineer, people do not communicate effectively, and it is not always possible for teams to work together. Some planning may be required to make the most effective use of the people available. Key issues are:

1. How good are the designers and programmers in the development team? It is sometimes argued that agile methods require higher skill levels than plan-based approaches in which programmers simply translate a detailed design into code. If you have a team with relatively low skill levels, you may need to use the best people to develop the design, with others responsible for programming.

2. How is the development team organized? If the development team is distributed or if part of the development is being outsourced, then you may need to develop design documents to communicate across the development teams.
3. What technologies are available to support system development? Agile methods often rely on good tools to keep track of an evolving design. If you are developing a system using an IDE that does not have good tools for program visualization and analysis, then more design documentation may be required.

Television and films have created a popular vision of software companies as informal organizations run by young men (mostly) who provide a fashionable working environment, with a minimum of bureaucracy and organizational procedures. This is far from the truth. Most software is developed in large companies that have established their own working practices and procedures. Management in these companies may be uncomfortable with the lack of documentation and the informal decision making in agile methods. Key issues are:

1. Is it important to have a very detailed specification and design before moving to implementation, perhaps for contractual reasons? If so, you probably need to use a plan-driven approach for requirements engineering but may use agile development practices during system implementation.
2. Is an incremental delivery strategy, where you deliver the software to customers or other system stakeholders and get rapid feedback from them, realistic? Will customer representatives be available, and are they willing to participate in the development team?
3. Are there cultural issues that may affect system development? Traditional engineering organizations have a culture of plan-based development, as this is the norm in engineering. This usually requires extensive design documentation rather than the informal knowledge used in agile processes.

In reality, the issue of whether a project can be labeled as plan-driven or agile is not very important. Ultimately, the primary concern of buyers of a software system is whether or not they have an executable software system that meets their needs and does useful things for the individual user or the organization. Software developers should be pragmatic and should choose those methods that are most effective for the type of system being developed, whether or not these are labeled agile or plan-driven.

3.4.3 Agile methods for large systems

Agile methods have to evolve to be used for large-scale software development. The fundamental reason for this is that large-scale software systems are much more complex and difficult to understand and manage than small-scale systems or software products. Six principal factors (Figure 3.13) contribute to this complexity:

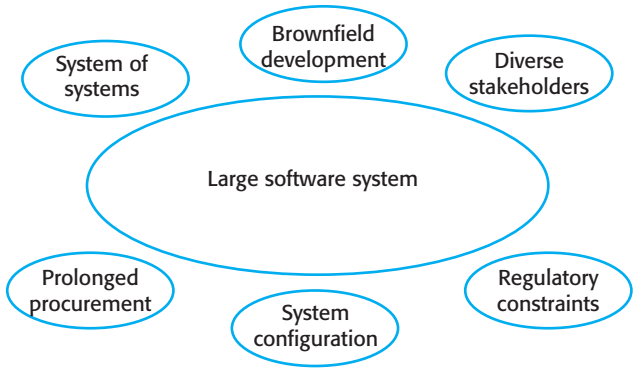

Figure 3.13 Large project characteristics

1. Large systems are usually systems of systems—collections of separate, communicating systems, where separate teams develop each system. Frequently, these teams are working in different places, sometimes in different time zones. It is practically impossible for each team to have a view of the whole system. Consequently, their priorities are usually to complete their part of the system without regard for wider systems issues.

2. Large systems are brownfield systems (Hopkins and Jenkins 2008); that is, they include and interact with a number of existing systems. Many of the system requirements are concerned with this interaction and so don't really lend themselves to flexibility and incremental development. Political issues can also be significant here—often the easiest solution to a problem is to change an existing system. However, this requires negotiation with the managers of that system to convince them that the changes can be implemented without risk to the system's operation.

3. Where several systems are integrated to create a system, a significant fraction of the development is concerned with system configuration rather than original code development. This is not necessarily compatible with incremental development and frequent system integration.

4. Large systems and their development processes are often constrained by external rules and regulations limiting the way that they can be developed, that require certain types of system documentation to be produced, and so on. Customers may have specific compliance requirements that may have to be followed, and these may require process documentation to be completed.

5. Large systems have a long procurement and development time. It is difficult to maintain coherent teams who know about the system over that period as, inevitably, people move on to other jobs and projects.

6. Large systems usually have a diverse set of stakeholders with different perspectives and objectives. For example, nurses and administrators may be the end-users of a medical system, but senior medical staff, hospital managers, and others, are also stakeholders in the system. It is practically impossible to involve all of these different stakeholders in the development process.

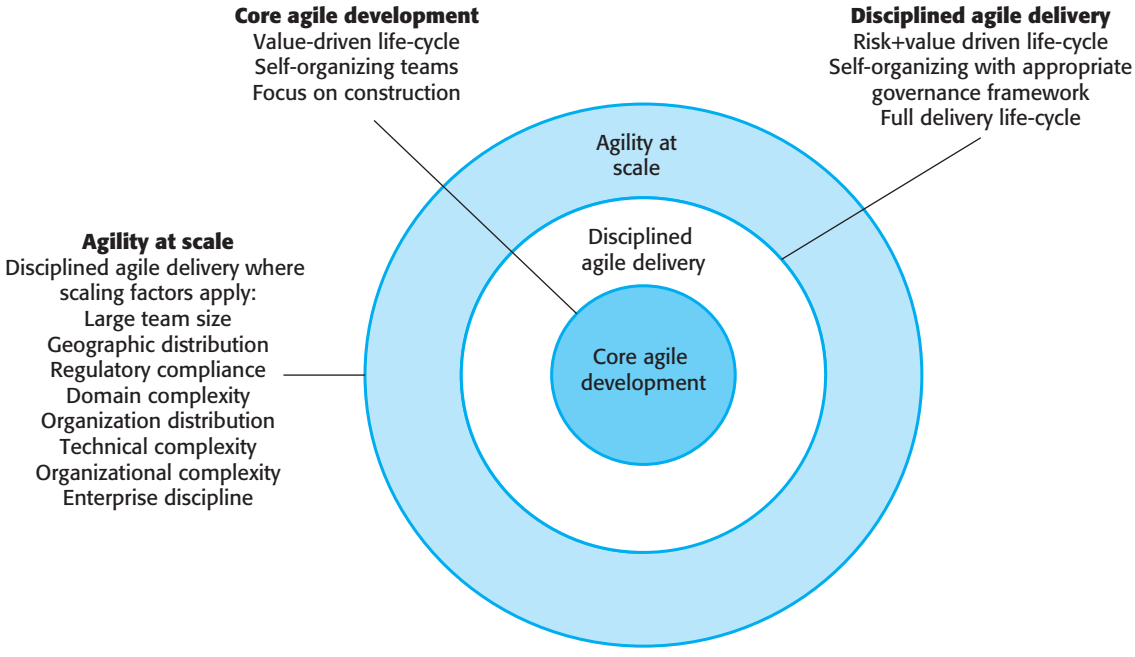

Figure 3.14 IBM's Agility at Scale model (© IBM 2010)

Dean Leffingwell, who has a great deal of experience in scaling agile methods, has developed the Scaled Agile Framework (Leffingwell 2007, 2011) to support large-scale, multi-team software development. He reports how this method has been used successfully in a number of large companies. IBM has also developed a framework for the large-scale use of agile methods called the Agile Scaling Model (ASM). Figure 3.14, taken from Ambler's white paper that discusses ASM (Ambler 2010), shows an overview of this model.

The ASM recognizes that scaling is a staged process where development teams move from the core agile practices discussed here to what is called Disciplined Agile Delivery. Essentially, this stage involves adapting these practices to a disciplined organizational setting and recognizing that teams cannot simply focus on development but must also take into account other stages of the software engineering process, such as requirements and architectural design.

The final scaling stage in ASM is to move to Agility at Scale where the complexity that is inherent in large projects is recognized. This involves taking account of factors such as distributed development, complex legacy environments, and regulatory compliance requirements. The practices used for disciplined agile delivery may have to be modified on a project-by-project basis to take these into account and, sometimes, additional plan-based practices added to the process.

No single model is appropriate for all large-scale agile products as the type of product, the customer requirements, and the people available are all different. However, approaches to scaling agile methods have a number of things in common:

1. A completely incremental approach to requirements engineering is impossible. Some early work on initial software requirements is essential. You need this work to identify the different parts of the system that may be developed by different teams and, often, to be part of the contract for the system development. However, these requirements should not normally be specified in detail; details are best developed incrementally.
2. There cannot be a single product owner or customer representative. Different people have to be involved for different parts of the system, and they have to continuously communicate and negotiate throughout the development process.
3. It is not possible to focus only on the code of the system. You need to do more up-front design and system documentation. The software architecture has to be designed, and there has to be documentation produced to describe critical aspects of the system, such as database schemas and the work breakdown across teams.
4. Cross-team communication mechanisms have to be designed and used. This should involve regular phone and videoconferences between team members and frequent, short electronic meetings where teams update each other on progress. A range of communication channels such as email, instant messaging, wikis, and social networking systems should be provided to facilitate communications.
5. Continuous integration, where the whole system is built every time any developer checks in a change, is practically impossible when several separate programs have to be integrated to create the system. However, it is essential to maintain frequent system builds and regular releases of the system. Configuration management tools that support multi-team software development are essential.

Scrum has been adapted for large-scale development. In essence, the Scrum team model described in Section 3.3 is maintained, but multiple Scrum teams are set up. The key characteristics of multi-team Scrum are:

1. *Role replication* Each team has a Product Owner for its work component and ScrumMaster. There may be a chief Product Owner and ScrumMaster for the entire project.
2. *Product architects* Each team chooses a product architect, and these architects collaborate to design and evolve the overall system architecture.
3. *Release alignment* The dates of product releases from each team are aligned so that a demonstrable and complete system is produced.
4. *Scrum of Scrums* There is a daily Scrum of Scrums where representatives from each team meet to discuss progress, identify problems, and plan the work to be done that day. Individual team Scrums may be staggered in time so that representatives from other teams can attend if necessary.

3.4.4 Agile methods across organizations

Small software companies that develop software products have been among the most enthusiastic adopters of agile methods. These companies are not constrained by organizational bureaucracies or process standards, and they can change quickly to adopt new ideas. Of course, larger companies have also experimented with agile methods in specific projects, but it is much more difficult for them to "scale out" these methods across the organization.

It can be difficult to introduce agile methods into large companies for a number of reasons:

1. Project managers who do not have experience of agile methods may be reluctant to accept the risk of a new approach, as they do not know how this will affect their particular projects.
2. Large organizations often have quality procedures and standards that all projects are expected to follow, and, because of their bureaucratic nature, these are likely to be incompatible with agile methods. Sometimes, these are supported by software tools (e.g., requirements management tools), and the use of these tools is mandated for all projects.
3. Agile methods seem to work best when team members have a relatively high skill level. However, within large organizations, there are likely to be a wide range of skills and abilities, and people with lower skill levels may not be effective team members in agile processes.
4. There may be cultural resistance to agile methods, especially in those organizations that have a long history of using conventional systems engineering processes.

Change management and testing procedures are examples of company procedures that may not be compatible with agile methods. Change management is the process of controlling changes to a system, so that the impact of changes is predictable and costs are controlled. All changes have to be approved in advance before they are made, and this conflicts with the notion of refactoring. When refactoring is part of an agile process, any developer can improve any code without getting external approval. For large systems, there are also testing standards where a system build is handed over to an external testing team. This may conflict with test-first approaches used in agile development methods.

Introducing and sustaining the use of agile methods across a large organization is a process of cultural change. Cultural change takes a long time to implement and often requires a change of management before it can be accomplished. Companies wishing to use agile methods need evangelists to promote change. Rather than trying to force agile methods onto unwilling developers, companies have found that the best way to introduce agile is bit by bit, starting with an enthusiastic group of developers. A successful agile project can act as a starting point, with the project team spreading agile practice across the organization. Once the notion of agile is widely known, explicit actions can then be taken to spread it across the organization.