



## 2

# Software processes

### Objectives

The objective of this chapter is to introduce you to the idea of a software process—a coherent set of activities for software production. When you have read this chapter, you will:

- understand the concepts of software processes and software process models;
- have been introduced to three general software process models and when they might be used;
- know about the fundamental process activities of software requirements engineering, software development, testing, and evolution;
- understand why processes should be organized to cope with changes in the software requirements and design;
- understand the notion of software process improvement and the factors that affect software process quality.

### Contents

- 2.1 Software process models
- 2.2 Process activities
- 2.3 Coping with change
- 2.4 Process improvement

A software process is a set of related activities that leads to the production of a software system. As I discussed in Chapter 1, there are many different types of software systems, and there is no universal software engineering method that is applicable to all of them. Consequently, there is no universally applicable software process. The process used in different companies depends on the type of software being developed, the requirements of the software customer, and the skills of the people writing the software.

However, although there are many different software processes, they all must include, in some form, the four fundamental software engineering activities that I introduced in Chapter 1:

1. *Software specification* The functionality of the software and constraints on its operation must be defined.
2. *Software development* The software to meet the specification must be produced.
3. *Software validation* The software must be validated to ensure that it does what the customer wants.
4. *Software evolution* The software must evolve to meet changing customer needs.

These activities are complex activities in themselves, and they include subactivities such as requirements validation, architectural design, and unit testing. Processes also include other activities, such as software configuration management and project planning that support production activities.

When we describe and discuss processes, we usually talk about the activities in these processes, such as specifying a data model and designing a user interface, and the ordering of these activities. We can all relate to what people do to develop software. However, when describing processes, it is also important to describe who is involved, what is produced, and conditions that influence the sequence of activities:

1. Products or deliverables are the outcomes of a process activity. For example, the outcome of the activity of architectural design may be a model of the software architecture.
2. Roles reflect the responsibilities of the people involved in the process. Examples of roles are project manager, configuration manager, and programmer.
3. Pre- and postconditions are conditions that must hold before and after a process activity has been enacted or a product produced. For example, before architectural design begins, a precondition may be that the consumer has approved all requirements; after this activity is finished, a postcondition might be that the UML models describing the architecture have been reviewed.

Software processes are complex and, like all intellectual and creative processes, rely on people making decisions and judgments. As there is no universal process that is right for all kinds of software, most software companies have developed their own

development processes. Processes have evolved to take advantage of the capabilities of the software developers in an organization and the characteristics of the systems that are being developed. For safety-critical systems, a very structured development process is required where detailed records are maintained. For business systems, with rapidly changing requirements, a more flexible, agile process is likely to be better.

As I discussed in Chapter 1, professional software development is a managed activity, so planning is an inherent part of all processes. Plan-driven processes are processes where all of the process activities are planned in advance and progress is measured against this plan. In agile processes, which I discuss in Chapter 3, planning is incremental and continual as the software is developed. It is therefore easier to change the process to reflect changing customer or product requirements. As Boehm and Turner (Boehm and Turner 2004) explain, each approach is suitable for different types of software. Generally, for large systems, you need to find a balance between plan-driven and agile processes.

Although there is no universal software process, there is scope for process improvement in many organizations. Processes may include outdated techniques or may not take advantage of the best practice in industrial software engineering. Indeed, many organizations still do not take advantage of software engineering methods in their software development. They can improve their process by introducing techniques such as UML modeling and test-driven development. I discuss software process improvement briefly later in this chapter text and in more detail in web Chapter 26.

## 2.1 Software process models

As I explained in Chapter 1, a software process model (sometimes called a Software Development Life Cycle or SDLC model) is a simplified representation of a software process. Each process model represents a process from a particular perspective and thus only provides partial information about that process. For example, a process activity model shows the activities and their sequence but may not show the roles of the people involved in these activities. In this section, I introduce a number of very general process models (sometimes called *process paradigms*) and present these from an architectural perspective. That is, we see the framework of the process but not the details of process activities.

These generic models are high-level, abstract descriptions of software processes that can be used to explain different approaches to software development. You can think of them as process frameworks that may be extended and adapted to create more specific software engineering processes.

The general process models that I cover here are:

1. *The waterfall model* This takes the fundamental process activities of specification, development, validation, and evolution and represents them as separate process phases such as requirements specification, software design, implementation, and testing.



### The Rational Unified Process

The Rational Unified Process (RUP) brings together elements of all of the general process models discussed here and supports prototyping and incremental delivery of software (Krutchen 2003). The RUP is normally described from three perspectives: a dynamic perspective that shows the phases of the model in time, a static perspective that shows process activities, and a practice perspective that suggests good practices to be used in the process. Phases of the RUP are inception, where a business case for the system is established; elaboration, where requirements and architecture are developed; construction where the software is implemented; and transition, where the system is deployed.

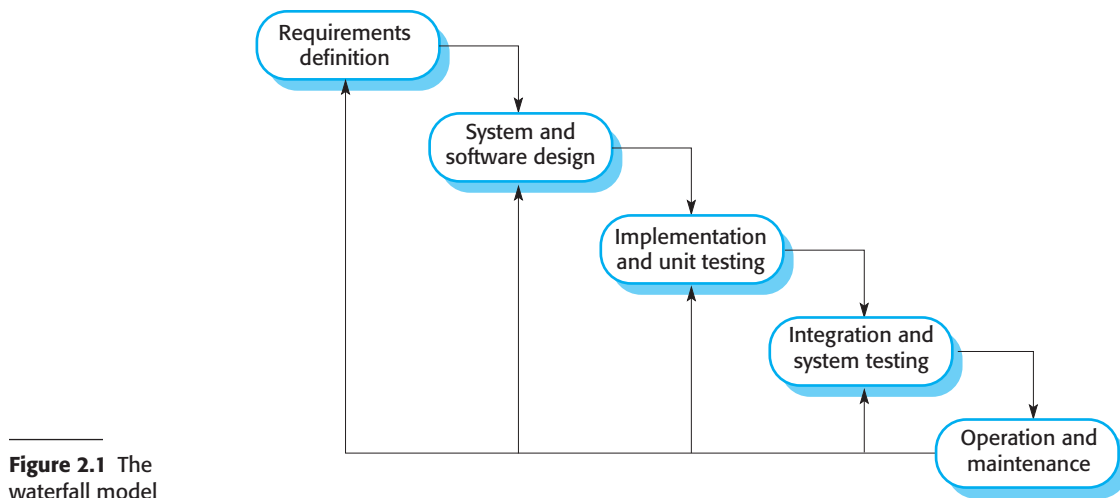
<http://software-engineering-book.com/web/rup/>

2. *Incremental development* This approach interleaves the activities of specification, development, and validation. The system is developed as a series of versions (increments), with each version adding functionality to the previous version.
3. *Integration and configuration* This approach relies on the availability of reusable components or systems. The system development process focuses on configuring these components for use in a new setting and integrating them into a system.

As I have said, there is no universal process model that is right for all kinds of software development. The right process depends on the customer and regulatory requirements, the environment where the software will be used, and the type of software being developed. For example, safety-critical software is usually developed using a waterfall process as lots of analysis and documentation is required before implementation begins. Software products are now always developed using an incremental process model. Business systems are increasingly being developed by configuring existing systems and integrating these to create a new system with the functionality that is required.

The majority of practical software processes are based on a general model but often incorporate features of other models. This is particularly true for large systems engineering. For large systems, it makes sense to combine some of the best features of all of the general processes. You need to have information about the essential system requirements to design a software architecture to support these requirements. You cannot develop this incrementally. Subsystems within a larger system may be developed using different approaches. Parts of the system that are well understood can be specified and developed using a waterfall-based process or may be bought in as off-the-shelf systems for configuration. Other parts of the system, which are difficult to specify in advance, should always be developed using an incremental approach. In both cases, software components are likely to be reused.

Various attempts have been made to develop “universal” process models that draw on all of these general models. One of the best known of these universal models is the Rational Unified Process (RUP) (Krutchen 2003), which was developed by Rational, a U.S. software engineering company. The RUP is a flexible model that



**Figure 2.1** The waterfall model

can be instantiated in different ways to create processes that resemble any of the general process models discussed here. The RUP has been adopted by some large software companies (notably IBM), but it has not gained widespread acceptance.

### 2.1.1 The waterfall model

The first published model of the software development process was derived from engineering process models used in large military systems engineering (Royce 1970). It presents the software development process as a number of stages, as shown in Figure 2.1. Because of the cascade from one phase to another, this model is known as the waterfall model or software life cycle. The waterfall model is an example of a plan-driven process. In principle at least, you plan and schedule all of the process activities before starting software development.

The stages of the waterfall model directly reflect the fundamental software development activities:

1. *Requirements analysis and definition* The system's services, constraints, and goals are established by consultation with system users. They are then defined in detail and serve as a system specification.
2. *System and software design* The systems design process allocates the requirements to either hardware or software systems. It establishes an overall system architecture. Software design involves identifying and describing the fundamental software system abstractions and their relationships.
3. *Implementation and unit testing* During this stage, the software design is realized as a set of programs or program units. Unit testing involves verifying that each unit meets its specification.



### Boehm's spiral process model

Barry Boehm, one of the pioneers in software engineering, proposed an incremental process model that was risk-driven. The process is represented as a spiral rather than a sequence of activities (Boehm 1988).

Each loop in the spiral represents a phase of the software process. Thus, the innermost loop might be concerned with system feasibility, the next loop with requirements definition, the next loop with system design, and so on. The spiral model combines change avoidance with change tolerance. It assumes that changes are a result of project risks and includes explicit risk management activities to reduce these risks.

<http://software-engineering-book.com/web/spiral-model/>

4. *Integration and system testing* The individual program units or programs are integrated and tested as a complete system to ensure that the software requirements have been met. After testing, the software system is delivered to the customer.
5. *Operation and maintenance* Normally, this is the longest life-cycle phase. The system is installed and put into practical use. Maintenance involves correcting errors that were not discovered in earlier stages of the life cycle, improving the implementation of system units, and enhancing the system's services as new requirements are discovered.

In principle, the result of each phase in the waterfall model is one or more documents that are approved ("signed off"). The following phase should not start until the previous phase has finished. For hardware development, where high manufacturing costs are involved, this makes sense. However, for software development, these stages overlap and feed information to each other. During design, problems with requirements are identified; during coding design problems are found, and so on. The software process, in practice, is never a simple linear model but involves feedback from one phase to another.

As new information emerges in a process stage, the documents produced at previous stages should be modified to reflect the required system changes. For example, if it is discovered that a requirement is too expensive to implement, the requirements document should be changed to remove that requirement. However, this requires customer approval and delays the overall development process.

As a result, both customers and developers may prematurely freeze the software specification so that no further changes are made to it. Unfortunately, this means that problems are left for later resolution, ignored, or programmed around. Premature freezing of requirements may mean that the system won't do what the user wants. It may also lead to badly structured systems as design problems are circumvented by implementation tricks.

During the final life-cycle phase (operation and maintenance) the software is put into use. Errors and omissions in the original software requirements are discovered.

Program and design errors emerge, and the need for new functionality is identified. The system must therefore evolve to remain useful. Making these changes (software maintenance) may involve repeating previous process stages.

In reality, software has to be flexible and accommodate change as it is being developed. The need for early commitment and system rework when changes are made means that the waterfall model is only appropriate for some types of system:

1. Embedded systems where the software has to interface with hardware systems. Because of the inflexibility of hardware, it is not usually possible to delay decisions on the software's functionality until it is being implemented.
2. Critical systems where there is a need for extensive safety and security analysis of the software specification and design. In these systems, the specification and design documents must be complete so that this analysis is possible. Safety-related problems in the specification and design are usually very expensive to correct at the implementation stage.
3. Large software systems that are part of broader engineering systems developed by several partner companies. The hardware in the systems may be developed using a similar model, and companies find it easier to use a common model for hardware and software. Furthermore, where several companies are involved, complete specifications may be needed to allow for the independent development of different subsystems.

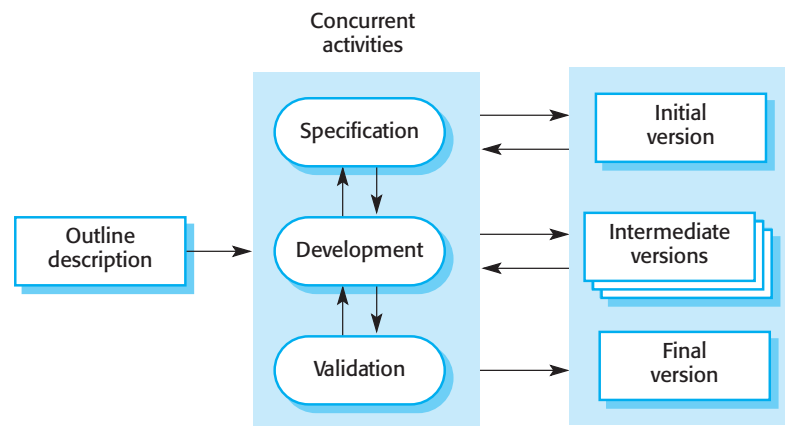
The waterfall model is not the right process model in situations where informal team communication is possible and software requirements change quickly. Iterative development and agile methods are better for these systems.

An important variant of the waterfall model is formal system development, where a mathematical model of a system specification is created. This model is then refined, using mathematical transformations that preserve its consistency, into executable code. Formal development processes, such as that based on the B method (Abrial 2005, 2010), are mostly used in the development of software systems that have stringent safety, reliability, or security requirements. The formal approach simplifies the production of a safety or security case. This demonstrates to customers or regulators that the system actually meets its safety or security requirements. However, because of the high costs of developing a formal specification, this development model is rarely used except for critical systems engineering.

### 2.1.2 Incremental development

Incremental development is based on the idea of developing an initial implementation, getting feedback from users and others, and evolving the software through several versions until the required system has been developed (Figure 2.2). Specification, development, and validation activities are interleaved rather than separate, with rapid feedback across activities.





**Figure 2.2** Incremental development

Incremental development in some form is now the most common approach for the development of application systems and software products. This approach can be either plan-driven, agile or, more usually, a mixture of these approaches. In a plan-driven approach, the system increments are identified in advance; if an agile approach is adopted, the early increments are identified, but the development of later increments depends on progress and customer priorities.

Incremental software development, which is a fundamental part of agile development methods, is better than a waterfall approach for systems whose requirements are likely to change during the development process. This is the case for most business systems and software products. Incremental development reflects the way that we solve problems. We rarely work out a complete problem solution in advance but move toward a solution in a series of steps, backtracking when we realize that we have made a mistake. By developing the software incrementally, it is cheaper and easier to make changes in the software as it is being developed.

Each increment or version of the system incorporates some of the functionality that is needed by the customer. Generally, the early increments of the system include the most important or most urgently required functionality. This means that the customer or user can evaluate the system at a relatively early stage in the development to see if it delivers what is required. If not, then only the current increment has to be changed and, possibly, new functionality defined for later increments.

Incremental development has three major advantages over the waterfall model:

1. The cost of implementing requirements changes is reduced. The amount of analysis and documentation that has to be redone is significantly less than is required with the waterfall model.
2. It is easier to get customer feedback on the development work that has been done. Customers can comment on demonstrations of the software and see how





### Problems with incremental development

Although incremental development has many advantages, it is not problem free. The primary cause of the difficulty is the fact that large organizations have bureaucratic procedures that have evolved over time and there may be a mismatch between these procedures and a more informal iterative or agile process.

Sometimes these procedures are there for good reasons. For example, there may be procedures to ensure that the software meets properly implements external regulations (e.g., in the United States, the Sarbanes Oxley accounting regulations). Changing these procedures may not be possible, so process conflicts may be unavoidable.

<http://software-engineering-book.com/web/incremental-development/>

much has been implemented. Customers find it difficult to judge progress from software design documents.

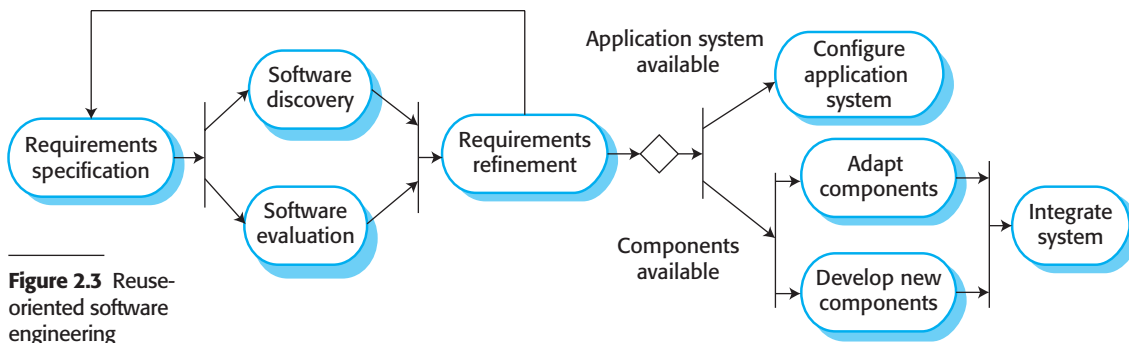
3. Early delivery and deployment of useful software to the customer is possible, even if all of the functionality has not been included. Customers are able to use and gain value from the software earlier than is possible with a waterfall process.

From a management perspective, the incremental approach has two problems:

1. The process is not visible. Managers need regular deliverables to measure progress. If systems are developed quickly, it is not cost effective to produce documents that reflect every version of the system.
2. System structure tends to degrade as new increments are added. Regular change leads to messy code as new functionality is added in whatever way is possible. It becomes increasingly difficult and costly to add new features to a system. To reduce structural degradation and general code messiness, agile methods suggest that you should regularly refactor (improve and restructure) the software.

The problems of incremental development become particularly acute for large, complex, long-lifetime systems, where different teams develop different parts of the system. Large systems need a stable framework or architecture, and the responsibilities of the different teams working on parts of the system need to be clearly defined with respect to that architecture. This has to be planned in advance rather than developed incrementally.

Incremental development does not mean that you have to deliver each increment to the system customer. You can develop a system incrementally and expose it to customers and other stakeholders for comment, without necessarily delivering it and deploying it in the customer's environment. Incremental delivery (covered in Section 2.3.2) means that the software is used in real, operational processes, so user feedback is likely to be realistic. However, providing feedback is not always possible as experimenting with new software can disrupt normal business processes.



**Figure 2.3** Reuse-oriented software engineering

### 2.1.3 Integration and configuration

In the majority of software projects, there is some software reuse. This often happens informally when people working on the project know of or search for code that is similar to what is required. They look for these, modify them as needed, and integrate them with the new code that they have developed.

This informal reuse takes place regardless of the development process that is used. However, since 2000, software development processes that focus on the reuse of existing software have become widely used. Reuse-oriented approaches rely on a base of reusable software components and an integrating framework for the composition of these components.

Three types of software components are frequently reused:

1. Stand-alone application systems that are configured for use in a particular environment. These systems are general-purpose systems that have many features, but they have to be adapted for use in a specific application.
2. Collections of objects that are developed as a component or as a package to be integrated with a component framework such as the Java Spring framework (Wheeler and White 2013).
3. Web services that are developed according to service standards and that are available for remote invocation over the Internet.

Figure 2.3 shows a general process model for reuse-based development, based on integration and configuration. The stages in this process are:

1. *Requirements specification* The initial requirements for the system are proposed. These do not have to be elaborated in detail but should include brief descriptions of essential requirements and desirable system features.
2. *Software discovery and evaluation* Given an outline of the software requirements, a search is made for components and systems that provide the functionality required. Candidate components and systems are evaluated to see if



### Software development tools

Software development tools are programs that are used to support software engineering process activities. These tools include requirements management tools, design editors, refactoring support tools, compilers, debuggers, bug trackers, and system building tools.

Software tools provide process support by automating some process activities and by providing information about the software that is being developed. For example:

- The development of graphical system models as part of the requirements specification or the software design
- The generation of code from these graphical models
- The generation of user interfaces from a graphical interface description that is created interactively by the user
- Program debugging through the provision of information about an executing program
- The automated translation of programs written using an old version of a programming language to a more recent version

Tools may be combined within a framework called an Interactive Development Environment or IDE. This provides a common set of facilities that tools can use so that it is easier for tools to communicate and operate in an integrated way.

<http://software-engineering-book.com/web/software-tools/>

they meet the essential requirements and if they are generally suitable for use in the system.

3. *Requirements refinement* During this stage, the requirements are refined using information about the reusable components and applications that have been discovered. The requirements are modified to reflect the available components, and the system specification is re-defined. Where modifications are impossible, the component analysis activity may be reentered to search for alternative solutions.
4. *Application system configuration* If an off-the-shelf application system that meets the requirements is available, it may then be configured for use to create the new system.
5. *Component adaptation and integration* If there is no off-the-shelf system, individual reusable components may be modified and new components developed. These are then integrated to create the system.

Reuse-oriented software engineering, based around configuration and integration, has the obvious advantage of reducing the amount of software to be developed and so reducing cost and risks. It usually also leads to faster delivery of the software. However, requirements compromises are inevitable, and this may lead to a system

that does not meet the real needs of users. Furthermore, some control over the system evolution is lost as new versions of the reusable components are not under the control of the organization using them.

Software reuse is very important, and so several chapters in the third I have dedicated several chapters in the 3rd part of the book to this topic. General issues of software reuse are covered in Chapter 15, component-based software engineering in Chapters 16 and 17, and service-oriented systems in Chapter 18.

## 2.2 Process activities

Real software processes are interleaved sequences of technical, collaborative, and managerial activities with the overall goal of specifying, designing, implementing, and testing a software system. Generally, processes are now tool-supported. This means that software developers may use a range of software tools to help them, such as requirements management systems, design model editors, program editors, automated testing tools, and debuggers.

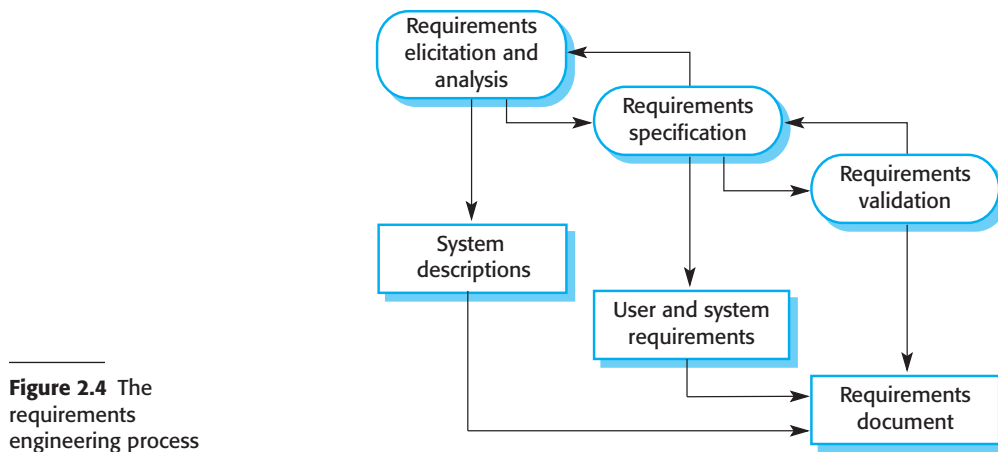
The four basic process activities of specification, development, validation, and evolution are organized differently in different development processes. In the water-fall model, they are organized in sequence, whereas in incremental development they are interleaved. How these activities are carried out depends on the type of software being developed, the experience and competence of the developers, and the type of organization developing the software.

### 2.2.1 Software specification

Software specification or requirements engineering is the process of understanding and defining what services are required from the system and identifying the constraints on the system's operation and development. Requirements engineering is a particularly critical stage of the software process, as mistakes made at this stage inevitably lead to later problems in the system design and implementation.

Before the requirements engineering process starts, a company may carry out a feasibility or marketing study to assess whether or not there is a need or a market for the software and whether or not it is technically and financially realistic to develop the software required. Feasibility studies are short-term, relatively cheap studies that inform the decision of whether or not to go ahead with a more detailed analysis.

The requirements engineering process (Figure 2.4) aims to produce an agreed requirements document that specifies a system satisfying stakeholder requirements. Requirements are usually presented at two levels of detail. End-users and customers need a high-level statement of the requirements; system developers need a more detailed system specification.



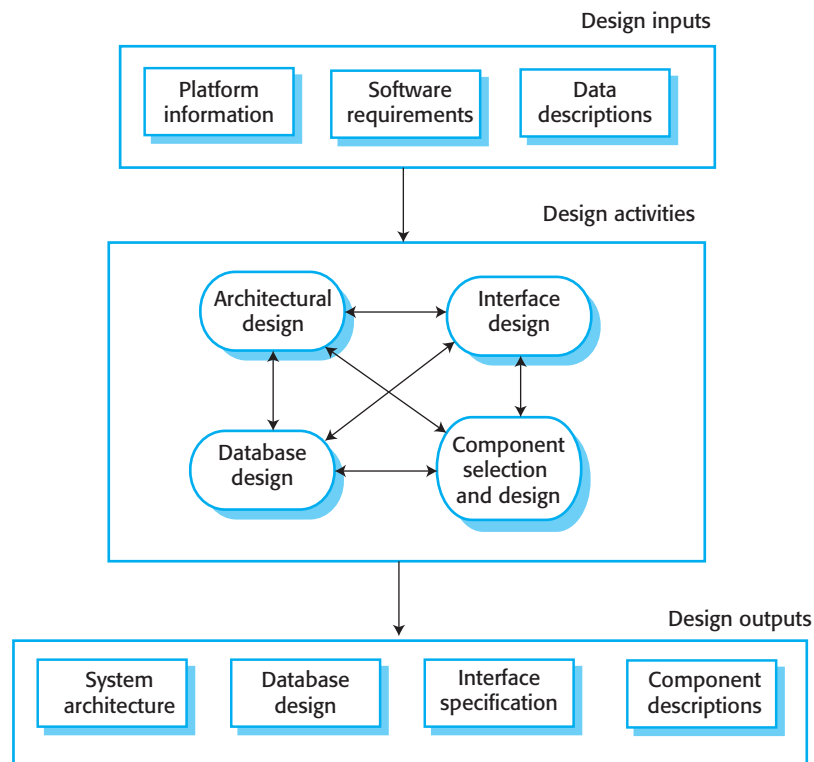
**Figure 2.4** The requirements engineering process

There are three main activities in the requirements engineering process:

1. *Requirements elicitation and analysis* This is the process of deriving the system requirements through observation of existing systems, discussions with potential users and procurers, task analysis, and so on. This may involve the development of one or more system models and prototypes. These help you understand the system to be specified.
2. *Requirements specification* Requirements specification is the activity of translating the information gathered during requirements analysis into a document that defines a set of requirements. Two types of requirements may be included in this document. User requirements are abstract statements of the system requirements for the customer and end-user of the system; system requirements are a more detailed description of the functionality to be provided.
3. *Requirements validation* This activity checks the requirements for realism, consistency, and completeness. During this process, errors in the requirements document are inevitably discovered. It must then be modified to correct these problems.

Requirements analysis continues during definition and specification, and new requirements come to light throughout the process. Therefore, the activities of analysis, definition, and specification are interleaved.

In agile methods, requirements specification is not a separate activity but is seen as part of system development. Requirements are informally specified for each increment of the system just before that increment is developed. Requirements are specified according to user priorities. The elicitation of requirements comes from users who are part of or work closely with the development team.



**Figure 2.5** A general model of the design process

### 2.2.2 Software design and implementation

The implementation stage of software development is the process of developing an executable system for delivery to the customer. Sometimes this involves separate activities of software design and programming. However, if an agile approach to development is used, design and implementation are interleaved, with no formal design documents produced during the process. Of course, the software is still designed, but the design is recorded informally on whiteboards and programmer's notebooks.

A software design is a description of the structure of the software to be implemented, the data models and structures used by the system, the interfaces between system components and, sometimes, the algorithms used. Designers do not arrive at a finished design immediately but develop the design in stages. They add detail as they develop their design, with constant backtracking to modify earlier designs.

Figure 2.5 is an abstract model of the design process showing the inputs to the design process, process activities, and the process outputs. The design process activities are both interleaved and interdependent. New information about the design is constantly being generated, and this affects previous design decisions. Design rework is therefore inevitable.

Most software interfaces with other software systems. These other systems include the operating system, database, middleware, and other application systems. These make up the “software platform,” the environment in which the software will execute. Information about this platform is an essential input to the design process, as designers must decide how best to integrate it with its environment. If the system is to process existing data, then the description of that data may be included in the platform specification. Otherwise, the data description must be an input to the design process so that the system data organization can be defined.

The activities in the design process vary, depending on the type of system being developed. For example, real-time systems require an additional stage of timing design but may not include a database, so there is no database design involved. Figure 2.5 shows four activities that may be part of the design process for information systems:

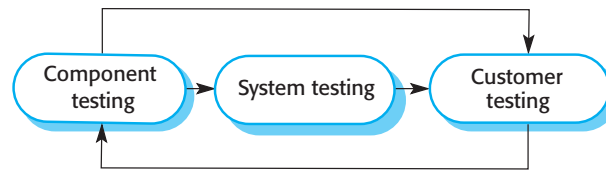
1. *Architectural design*, where you identify the overall structure of the system, the principal components (sometimes called subsystems or modules), their relationships, and how they are distributed.
2. *Database design*, where you design the system data structures and how these are to be represented in a database. Again, the work here depends on whether an existing database is to be reused or a new database is to be created.
3. *Interface design*, where you define the interfaces between system components. This interface specification must be unambiguous. With a precise interface, a component may be used by other components without them having to know how it is implemented. Once interface specifications are agreed, the components can be separately designed and developed.
4. *Component selection and design*, where you search for reusable components and, if no suitable components are available, design new software components. The design at this stage may be a simple component description with the implementation details left to the programmer. Alternatively, it may be a list of changes to be made to a reusable component or a detailed design model expressed in the UML. The design model may then be used to automatically generate an implementation.

These activities lead to the design outputs, which are also shown in Figure 2.5. For critical systems, the outputs of the design process are detailed design documents setting out precise and accurate descriptions of the system. If a model-driven approach is used (Chapter 5), the design outputs are design diagrams. Where agile methods of development are used, the outputs of the design process may not be separate specification documents but may be represented in the code of the program.

The development of a program to implement a system follows naturally from system design. Although some classes of program, such as safety-critical systems, are usually designed in detail before any implementation begins, it is more common for design and program development to be interleaved. Software development tools may be used to generate a skeleton program from a design. This includes code to



**Figure 2.6** Stages of testing



define and implement interfaces, and, in many cases, the developer need only add details of the operation of each program component.

Programming is an individual activity, and there is no general process that is usually followed. Some programmers start with components that they understand, develop these, and then move on to less understood components. Others take the opposite approach, leaving familiar components till last because they know how to develop them. Some developers like to define data early in the process and then use this to drive the program development; others leave data unspecified for as long as possible.

Normally, programmers carry out some testing of the code they have developed. This often reveals program defects (bugs) that must be removed from the program. Finding and fixing program defects is called *debugging*. Defect testing and debugging are different processes. Testing establishes the existence of defects. Debugging is concerned with locating and correcting these defects.

When you are debugging, you have to generate hypotheses about the observable behavior of the program and then test these hypotheses in the hope of finding the fault that caused the output anomaly. Testing the hypotheses may involve tracing the program code manually. It may require new test cases to localize the problem. Interactive debugging tools, which show the intermediate values of program variables and a trace of the statements executed, are usually used to support the debugging process.

### 2.2.3 Software validation

Software validation or, more generally, verification and validation (V & V) is intended to show that a system both conforms to its specification and meets the expectations of the system customer. Program testing, where the system is executed using simulated test data, is the principal validation technique. Validation may also involve checking processes, such as inspections and reviews, at each stage of the software process from user requirements definition to program development. However, most V & V time and effort is spent on program testing.

Except for small programs, systems should not be tested as a single, monolithic unit. Figure 2.6 shows a three-stage testing process in which system components are individually tested, then the integrated system is tested. For custom software, customer testing involves testing the system with real customer data. For products that are sold as applications, customer testing is sometimes called beta testing where selected users try out and comment on the software.

The stages in the testing process are:

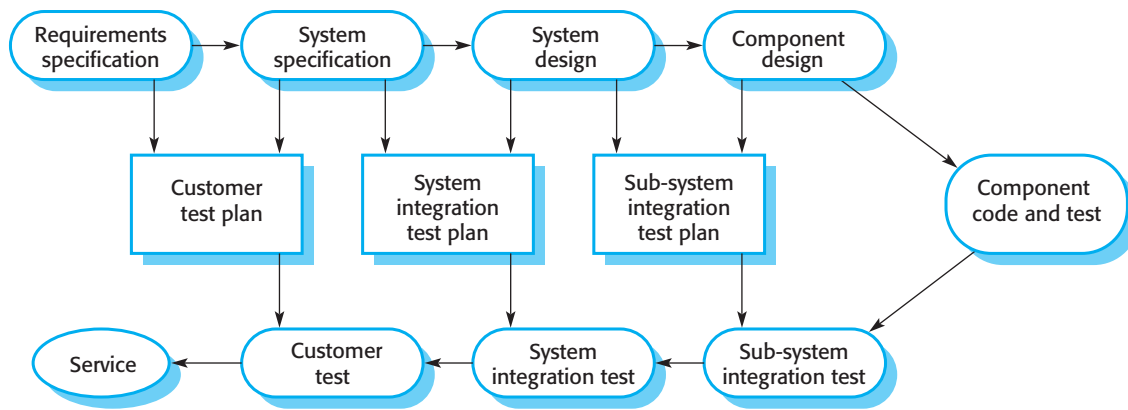
1. *Component testing* The components making up the system are tested by the people developing the system. Each component is tested independently, without other system components. Components may be simple entities such as functions or object classes or may be coherent groupings of these entities. Test automation tools, such as JUnit for Java, that can rerun tests when new versions of the component are created, are commonly used (Koskela 2013).
2. *System testing* System components are integrated to create a complete system. This process is concerned with finding errors that result from unanticipated interactions between components and component interface problems. It is also concerned with showing that the system meets its functional and non-functional requirements, and testing the emergent system properties. For large systems, this may be a multistage process where components are integrated to form subsystems that are individually tested before these subsystems are integrated to form the final system.
3. *Customer testing* This is the final stage in the testing process before the system is accepted for operational use. The system is tested by the system customer (or potential customer) rather than with simulated test data. For custom-built software, customer testing may reveal errors and omissions in the system requirements definition, because the real data exercise the system in different ways from the test data. Customer testing may also reveal requirements problems where the system's facilities do not really meet the users' needs or the system performance is unacceptable. For products, customer testing shows how well the software product meets the customer's needs.

Ideally, component defects are discovered early in the testing process, and interface problems are found when the system is integrated. However, as defects are discovered, the program must be debugged, and this may require other stages in the testing process to be repeated. Errors in program components, say, may come to light during system testing. The process is therefore an iterative one with information being fed back from later stages to earlier parts of the process.

Normally, component testing is simply part of the normal development process. Programmers make up their own test data and incrementally test the code as it is developed. The programmer knows the component and is therefore the best person to generate test cases.

If an incremental approach to development is used, each increment should be tested as it is developed, with these tests based on the requirements for that increment. In test-driven development, which is a normal part of agile processes, tests are developed along with the requirements before development starts. This helps the testers and developers to understand the requirements and ensures that there are no delays as test cases are created.

When a plan-driven software process is used (e.g., for critical systems development), testing is driven by a set of test plans. An independent team of testers works



**Figure 2.7** Testing phases in a plan-driven software process

from these test plans, which have been developed from the system specification and design. Figure 2.7 illustrates how test plans are the link between testing and development activities. This is sometimes called the V-model of development (turn it on its side to see the V). The V-model shows the software validation activities that correspond to each stage of the waterfall process model.

When a system is to be marketed as a software product, a testing process called beta testing is often used. Beta testing involves delivering a system to a number of potential customers who agree to use that system. They report problems to the system developers. This exposes the product to real use and detects errors that may not have been anticipated by the product developers. After this feedback, the software product may be modified and released for further beta testing or general sale.

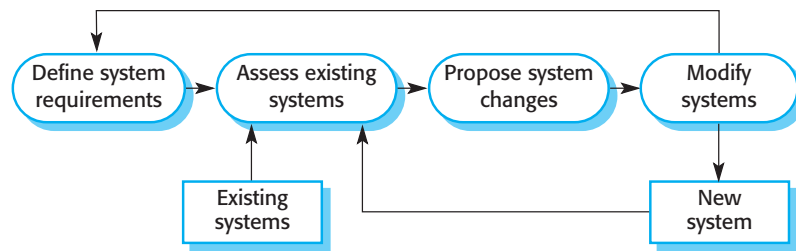
### 2.2.4 Software evolution

The flexibility of software is one of the main reasons why more and more software is being incorporated into large, complex systems. Once a decision has been made to manufacture hardware, it is very expensive to make changes to the hardware design. However, changes can be made to software at any time during or after the system development. Even extensive changes are still much cheaper than corresponding changes to system hardware.

Historically, there has always been a split between the process of software development and the process of software evolution (software maintenance). People think of software development as a creative activity in which a software system is developed from an initial concept through to a working system. However, they sometimes think of software maintenance as dull and uninteresting. They think that software maintenance is less interesting and challenging than original software development.

This distinction between development and maintenance is increasingly irrelevant. Very few software systems are completely new systems, and it makes much more

**Figure 2.8** Software system evolution



sense to see development and maintenance as a continuum. Rather than two separate processes, it is more realistic to think of software engineering as an evolutionary process (Figure 2.8) where software is continually changed over its lifetime in response to changing requirements and customer needs.

## 2.3 Coping with change

Change is inevitable in all large software projects. The system requirements change as businesses respond to external pressures, competition, and changed management priorities. As new technologies become available, new approaches to design and implementation become possible. Therefore whatever software process model is used, it is essential that it can accommodate changes to the software being developed.

Change adds to the costs of software development because it usually means that work that has been completed has to be redone. This is called rework. For example, if the relationships between the requirements in a system have been analyzed and new requirements are then identified, some or all of the requirements analysis has to be repeated. It may then be necessary to redesign the system to deliver the new requirements, change any programs that have been developed, and retest the system.

Two related approaches may be used to reduce the costs of rework:

1. *Change anticipation*, where the software process includes activities that can anticipate or predict possible changes before significant rework is required. For example, a prototype system may be developed to show some key features of the system to customers. They can experiment with the prototype and refine their requirements before committing to high software production costs.
2. *Change tolerance*, where the process and software are designed so that changes can be easily made to the system. This normally involves some form of incremental development. Proposed changes may be implemented in increments that have not yet been developed. If this is impossible, then only a single increment (a small part of the system) may have to be altered to incorporate the change.

In this section, I discuss two ways of coping with change and changing system requirements:

1. *System prototyping*, where a version of the system or part of the system is developed quickly to check the customer's requirements and the feasibility of design decisions. This is a method of change anticipation as it allows users to experiment with the system before delivery and so refine their requirements. The number of requirements change proposals made after delivery is therefore likely to be reduced.
2. *Incremental delivery*, where system increments are delivered to the customer for comment and experimentation. This supports both change avoidance and change tolerance. It avoids the premature commitment to requirements for the whole system and allows changes to be incorporated into later increments at relatively low cost.

The notion of refactoring, namely, improving the structure and organization of a program, is also an important mechanism that supports change tolerance. I discuss this in Chapter 3 (Agile methods).

### 2.3.1 Prototyping

---

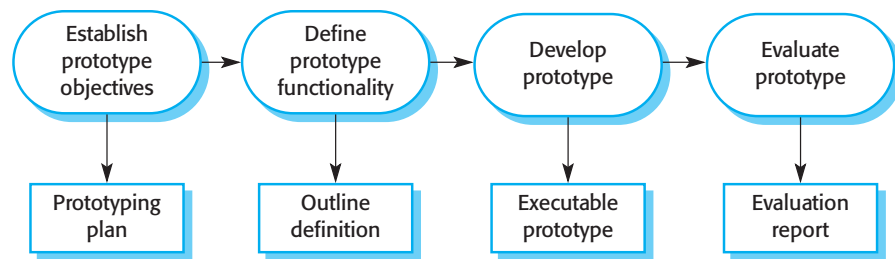
A prototype is an early version of a software system that is used to demonstrate concepts, try out design options, and find out more about the problem and its possible solutions. Rapid, iterative development of the prototype is essential so that costs are controlled and system stakeholders can experiment with the prototype early in the software process.

A software prototype can be used in a software development process to help anticipate changes that may be required:

1. In the requirements engineering process, a prototype can help with the elicitation and validation of system requirements.
2. In the system design process, a prototype can be used to explore software solutions and in the development of a user interface for the system.

System prototypes allow potential users to see how well the system supports their work. They may get new ideas for requirements and find areas of strength and weakness in the software. They may then propose new system requirements. Furthermore, as the prototype is developed, it may reveal errors and omissions in the system requirements. A feature described in a specification may seem to be clear and useful. However, when that function is combined with other functions, users often find that their initial view was incorrect or incomplete. The system specification can then be modified to reflect the changed understanding of the requirements.

**Figure 2.9** Prototype development

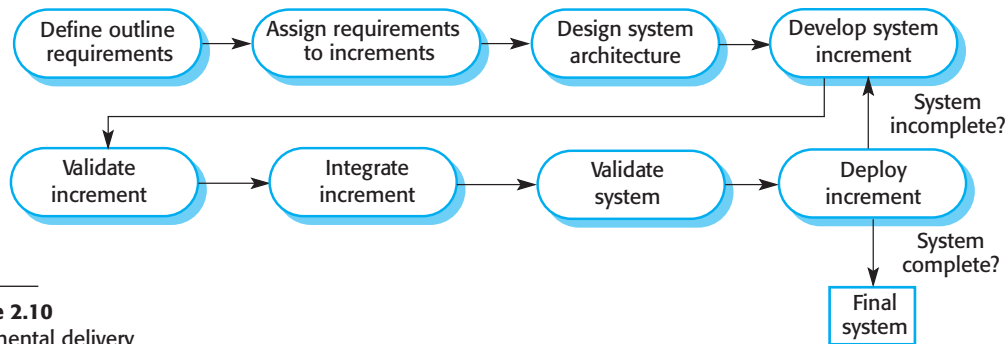


A system prototype may be used while the system is being designed to carry out design experiments to check the feasibility of a proposed design. For example, a database design may be prototyped and tested to check that it supports efficient data access for the most common user queries. Rapid prototyping with end-user involvement is the only sensible way to develop user interfaces. Because of the dynamic nature of user interfaces, textual descriptions and diagrams are not good enough for expressing the user interface requirements and design.

A process model for prototype development is shown in Figure 2.9. The objectives of prototyping should be made explicit from the start of the process. These may be to develop the user interface, to develop a system to validate functional system requirements, or to develop a system to demonstrate the application to managers. The same prototype usually cannot meet all objectives. If the objectives are left unstated, management or end-users may misunderstand the function of the prototype. Consequently, they may not get the benefits that they expected from the prototype development.

The next stage in the process is to decide what to put into and, perhaps more importantly, what to leave out of the prototype system. To reduce prototyping costs and accelerate the delivery schedule, you may leave some functionality out of the prototype. You may decide to relax non-functional requirements such as response time and memory utilization. Error handling and management may be ignored unless the objective of the prototype is to establish a user interface. Standards of reliability and program quality may be reduced.

The final stage of the process is prototype evaluation. Provision must be made during this stage for user training, and the prototype objectives should be used to derive a plan for evaluation. Potential users need time to become comfortable with a new system and to settle into a normal pattern of usage. Once they are using the system normally, they then discover requirements errors and omissions. A general problem with prototyping is that users may not use the prototype in the same way as they use the final system. Prototype testers may not be typical of system users. There may not be enough time to train users during prototype evaluation. If the prototype is slow, the evaluators may adjust their way of working and avoid those system features that have slow response times. When provided with better response in the final system, they may use it in a different way.



**Figure 2.10**  
Incremental delivery

### 2.3.2 Incremental delivery

Incremental delivery (Figure 2.10) is an approach to software development where some of the developed increments are delivered to the customer and deployed for use in their working environment. In an incremental delivery process, customers define which of the services are most important and which are least important to them. A number of delivery increments are then defined, with each increment providing a subset of the system functionality. The allocation of services to increments depends on the service priority, with the highest priority services implemented and delivered first.

Once the system increments have been identified, the requirements for the services to be delivered in the first increment are defined in detail and that increment is developed. During development, further requirements analysis for later increments can take place, but requirements changes for the current increment are not accepted.

Once an increment is completed and delivered, it is installed in the customer's normal working environment. They can experiment with the system, and this helps them clarify their requirements for later system increments. As new increments are completed, they are integrated with existing increments so that system functionality improves with each delivered increment.

Incremental delivery has a number of advantages:

1. Customers can use the early increments as prototypes and gain experience that informs their requirements for later system increments. Unlike prototypes, these are part of the real system, so there is no relearning when the complete system is available.
2. Customers do not have to wait until the entire system is delivered before they can gain value from it. The first increment satisfies their most critical requirements, so they can use the software immediately.
3. The process maintains the benefits of incremental development in that it should be relatively easy to incorporate changes into the system.



4. As the highest priority services are delivered first and later increments then integrated, the most important system services receive the most testing. This means that customers are less likely to encounter software failures in the most important parts of the system.

However, there are problems with incremental delivery. In practice, it only works in situations where a brand-new system is being introduced and the system evaluators are given time to experiment with the new system. Key problems with this approach are:

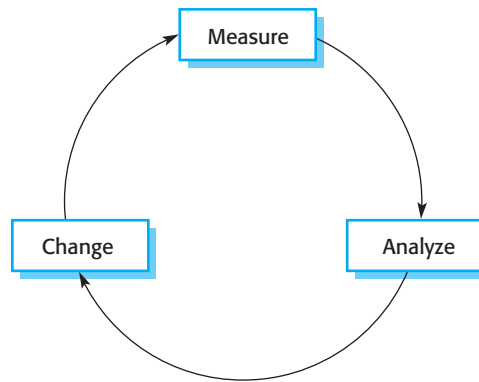
1. Iterative delivery is problematic when the new system is intended to replace an existing system. Users need all of the functionality of the old system and are usually unwilling to experiment with an incomplete new system. It is often impractical to use the old and the new systems alongside each other as they are likely to have different databases and user interfaces.
2. Most systems require a set of basic facilities that are used by different parts of the system. As requirements are not defined in detail until an increment is to be implemented, it can be hard to identify common facilities that are needed by all increments.
3. The essence of iterative processes is that the specification is developed in conjunction with the software. However, this conflicts with the procurement model of many organizations, where the complete system specification is part of the system development contract. In the incremental approach, there is no complete system specification until the final increment is specified. This requires a new form of contract, which large customers such as government agencies may find difficult to accommodate.

For some types of systems, incremental development and delivery is not the best approach. These are very large systems where development may involve teams working in different locations, some embedded systems where the software depends on hardware development, and some critical systems where all the requirements must be analyzed to check for interactions that may compromise the safety or security of the system.

These large systems, of course, suffer from the same problems of uncertain and changing requirements. Therefore, to address these problems and get some of the benefits of incremental development, a system prototype may be developed and used as a platform for experiments with the system requirements and design. With the experience gained from the prototype, definitive requirements can then be agreed.

## 2.4 Process improvement

Nowadays, there is a constant demand from industry for cheaper, better software, which has to be delivered to ever-tighter deadlines. Consequently, many software companies have turned to software process improvement as a way of enhancing the



**Figure 2.11** The process improvement cycle

quality of their software, reducing costs, or accelerating their development processes. Process improvement means understanding existing processes and changing these processes to increase product quality and/or reduce costs and development time. I cover general issues of process measurement and process improvement in detail in web Chapter 26.

Two quite different approaches to process improvement and change are used:

1. The process maturity approach, which has focused on improving process and project management and introducing good software engineering practice into an organization. The level of process maturity reflects the extent to which good technical and management practice has been adopted in organizational software development processes. The primary goals of this approach are improved product quality and process predictability.
2. The agile approach, which has focused on iterative development and the reduction of overheads in the software process. The primary characteristics of agile methods are rapid delivery of functionality and responsiveness to changing customer requirements. The improvement philosophy here is that the best processes are those with the lowest overheads and agile approaches can achieve this. I describe agile approaches in Chapter 3.

People who are enthusiastic about and committed to each of these approaches are generally skeptical of the benefits of the other. The process maturity approach is rooted in plan-driven development and usually requires increased “overhead,” in the sense that activities are introduced that are not directly relevant to program development. Agile approaches focus on the code being developed and deliberately minimize formality and documentation.

The general process improvement process underlying the process maturity approach is a cyclical process, as shown in Figure 2.11. The stages in this process are:

1. *Process measurement* You measure one or more attributes of the software process or product. These measurements form a baseline that helps you decide if

process improvements have been effective. As you introduce improvements, you re-measure the same attributes, which will hopefully have improved in some way.

2. *Process analysis* The current process is assessed, and process weaknesses and bottlenecks are identified. Process models (sometimes called process maps) that describe the process may be developed during this stage. The analysis may be focused by considering process characteristics such as rapidity and robustness.
3. *Process change* Process changes are proposed to address some of the identified process weaknesses. These are introduced, and the cycle resumes to collect data about the effectiveness of the changes.

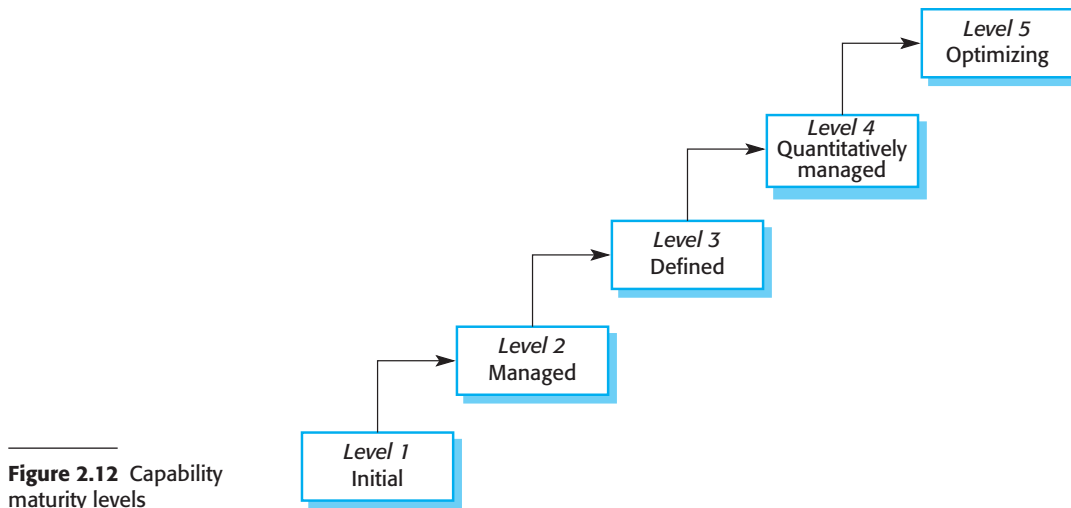
Without concrete data on a process or the software developed using that process, it is impossible to assess the value of process improvement. However, companies starting the process improvement process are unlikely to have process data available as an improvement baseline. Therefore, as part of the first cycle of changes, you may have to collect data about the software process and to measure software product characteristics.

Process improvement is a long-term activity, so each of the stages in the improvement process may last several months. It is also a continuous activity as, whatever new processes are introduced, the business environment will change and the new processes will themselves have to evolve to take these changes into account.

The notion of process maturity was introduced in the late 1980s when the Software Engineering Institute (SEI) proposed their model of process capability maturity (Humphrey 1988). The maturity of a software company's processes reflects the process management, measurement, and use of good software engineering practices in the company. This idea was introduced so that the U.S. Department of Defense could assess the software engineering capability of defense contractors, with a view to limiting contracts to those contractors who had reached a required level of process maturity. Five levels of process maturity were proposed, as shown in Figure 2.12. These have evolved and developed over the last 25 years (Chrissis, Konrad, and Shrum 2011), but the fundamental ideas in Humphrey's model are still the basis of software process maturity assessment.

The levels in the process maturity model are:

1. *Initial* The goals associated with the process area are satisfied, and for all processes the scope of the work to be performed is explicitly set out and communicated to the team members.
2. *Managed* At this level, the goals associated with the process area are met, and organizational policies are in place that define when each process should be used. There must be documented project plans that define the project goals. Resource management and process monitoring procedures must be in place across the institution.
3. *Defined* This level focuses on organizational standardization and deployment of processes. Each project has a managed process that is adapted to the project requirements from a defined set of organizational processes. Process assets and process measurements must be collected and used for future process improvements.



4. *Quantitatively managed* At this level, there is an organizational responsibility to use statistical and other quantitative methods to control subprocesses. That is, collected process and product measurements must be used in process management.
5. *Optimizing* At this highest level, the organization must use the process and product measurements to drive process improvement. Trends must be analyzed and the processes adapted to changing business needs.

The work on process maturity levels has had a major impact on the software industry. It focused attention on the software engineering processes and practices that were used and led to significant improvements in software engineering capability. However, there is too much overhead in formal process improvement for small companies, and maturity estimation with agile processes is difficult. Consequently, only large software companies now use this maturity-focused approach to software process improvement.

## KEY POINTS

- Software processes are the activities involved in producing a software system. Software process models are abstract representations of these processes.
- General process models describe the organization of software processes. Examples of these general models include the waterfall model, incremental development, and reusable component configuration and integration.

- Requirements engineering is the process of developing a software specification. Specifications are intended to communicate the system needs of the customer to the system developers.
- Design and implementation processes are concerned with transforming a requirements specification into an executable software system.
- Software validation is the process of checking that the system conforms to its specification and that it meets the real needs of the users of the system.
- Software evolution takes place when you change existing software systems to meet new requirements. Changes are continuous, and the software must evolve to remain useful.
- Processes should include activities to cope with change. This may involve a prototyping phase that helps avoid poor decisions on requirements and design. Processes may be structured for iterative development and delivery so that changes may be made without disrupting the system as a whole.
- Process improvement is the process of improving existing software processes to improve software quality, lower development costs, or reduce development time. It is a cyclic process involving process measurement, analysis, and change.

## FURTHER READING

“Process Models in Software Engineering.” This is an excellent overview of a wide range of software engineering process models that have been proposed. (W. Scacchi, *Encyclopaedia of Software Engineering*, ed. J. J. Marciniak, John Wiley & Sons, 2001) <http://www.ics.uci.edu/~wscacchi/Papers/SE-Encyc/Process-Models-SE-Encyc.pdf>

*Software Process Improvement: Results and Experience from the Field*. This book is a collection of papers focusing on process improvement case studies in several small and medium-sized Norwegian companies. It also includes a good introduction to the general issues of process improvement. (Conradi, R., Dybå, T., Sjøberg, D., and Ulsund, T. (eds.), Springer, 2006).

“Software Development Life Cycle Models and Methodologies.” This blog post is a succinct summary of several software process models that have been proposed and used. It discusses the advantages and disadvantages of each of these models (M. Sami, 2012). <http://melsatar.wordpress.com/2012/03/15/software-development-life-cycle-models-and-methodologies/>

## WEBSITE

PowerPoint slides for this chapter:

[www.pearsonglobaleditions.com/Sommerville](http://www.pearsonglobaleditions.com/Sommerville)

Links to supporting videos:

<http://software-engineering-book.com/videos/software-engineering/>

## EXERCISES

- 2.1.** Suggest the most appropriate generic software process model that might be used as a basis for managing the development of the following systems. Explain your answer according to the type of system being developed:

A system to control antilock braking in a car

A virtual reality system to support software maintenance

A university accounting system that replaces an existing system

An interactive travel planning system that helps users plan journeys with the lowest environmental impact

- 2.2.** Incremental software development could be very effectively used for customers who do not have a clear idea about the systems needed for their operations. Discuss.
- 2.3.** Consider the integration and configuration process model shown in Figure 2.3. Explain why it is essential to repeat the requirements engineering activity in the process.
- 2.4.** Suggest why it is important to make a distinction between developing the user requirements and developing system requirements in the requirements engineering process.
- 2.5.** Using an example, explain why the design activities of architectural design, database design, interface design, and component design are interdependent.
- 2.6.** Explain why software testing should always be an incremental, staged activity. Are programmers the best people to test the programs that they have developed?
- 2.7.** Imagine that a government wants a software program that helps to keep track of the utilization of the country's vast mineral resources. Although the requirements put forward by the government were not very clear, a software company was tasked with the development of a prototype. The government found the prototype impressive, and asked it be extended to be the actual system that would be used. Discuss the pros and cons of taking this approach.
- 2.8.** You have developed a prototype of a software system and your manager is very impressed by it. She proposes that it should be put into use as a production system, with new features added as required. This avoids the expense of system development and makes the system immediately useful. Write a short report for your manager explaining why prototype systems should not normally be used as production systems.
- 2.9.** Suggest two advantages and two disadvantages of the approach to process assessment and improvement that is embodied in the SEI's Capability Maturity framework.
- 2.10.** Historically, the introduction of technology has caused profound changes in the labor market and, temporarily at least, displaced people from jobs. Discuss whether the introduction of extensive process automation is likely to have the same consequences for software engineers. If you don't think it will, explain why not. If you think that it will reduce job opportunities, is it ethical for the engineers affected to passively or actively resist the introduction of this technology?

## REFERENCES

- Abrial, J. R. 2005. *The B Book: Assigning Programs to Meanings*. Cambridge, UK: Cambridge University Press.
- . 2010. *Modeling in Event-B: System and Software Engineering*. Cambridge, UK: Cambridge University Press.
- Boehm, B. W. (1988). “A Spiral Model of Software Development and Enhancement.” *IEEE Computer*, 21 (5), 61–72. doi:10.1145/12944.12948
- Boehm, B. W., and R. Turner. 2004. “Balancing Agility and Discipline: Evaluating and Integrating Agile and Plan-Driven Methods.” In *26th Int. Conf on Software Engineering*, Edinburgh, Scotland. doi:10.1109/ICSE.2004.1317503.
- Chrissis, M. B., M. Konrad, and S. Shrum. 2011. *CMMI for Development: Guidelines for Process Integration and Product Improvement*, 3rd ed. Boston: Addison-Wesley.
- Humphrey, W. S. 1988. “Characterizing the Software Process: A Maturity Framework.” *IEEE Software* 5 (2): 73–79. doi:10.1109/2.59.
- Koskela, L. 2013. *Effective Unit Testing: A Guide for Java Developers*. Greenwich, CT: Manning Publications.
- Krutchén, P. 2003. *The Rational Unified Process—An Introduction*, 3rd ed. Reading, MA: Addison-Wesley.
- Royce, W. W. 1970. “Managing the Development of Large Software Systems: Concepts and Techniques.” In *IEEE WESTCON*, 1–9. Los Angeles, CA.
- Wheeler, W., and J. White. 2013. *Spring in Practice*. Greenwich, CT: Manning Publications.





# 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

- 3.1** Agile methods
- 3.2** Agile development techniques
- 3.3** Agile project management
- 3.4** Scaling agile methods

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.

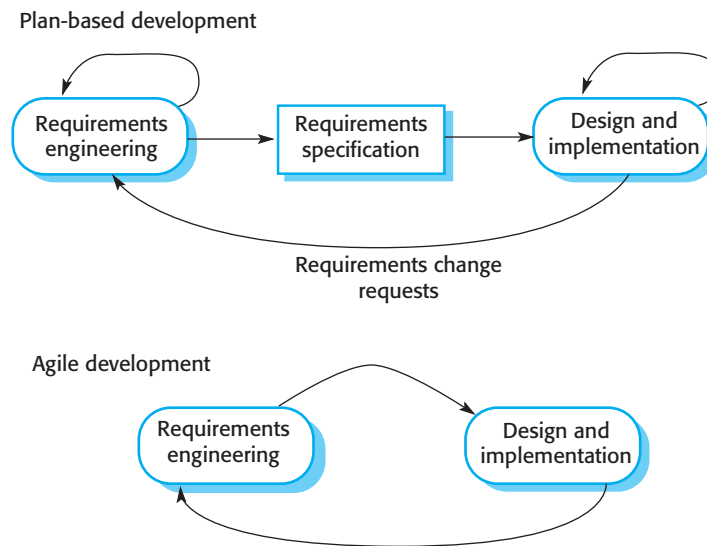
Plan-driven software development processes that completely specify the requirements and then design, build, and test a system are not geared to rapid software development. As the requirements change or as requirements problems are discovered, the system design or implementation has to be reworked and 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<sup>†</sup>.*

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.

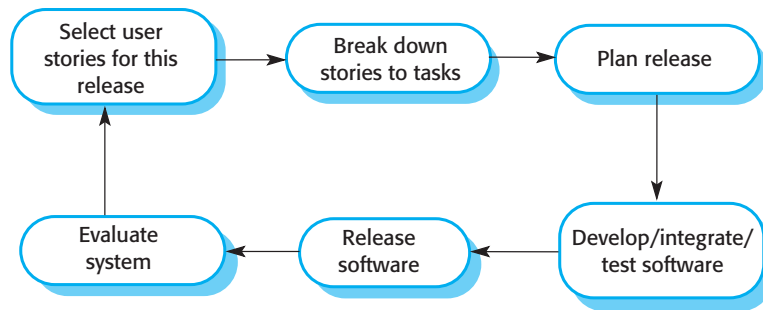
Agile methods have been particularly successful for two kinds of system development.

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

<sup>†</sup><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).



**Figure 3.5** A  
“prescribing medication”  
story

#### 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.

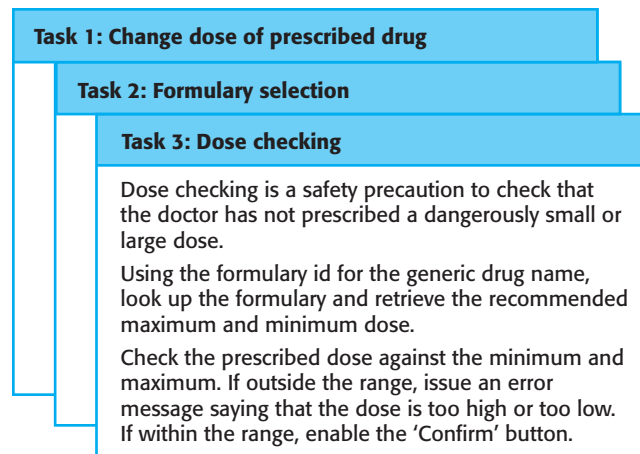
I am not convinced that XP on its own is a practical agile method for most companies, but its most significant contribution is probably the set of agile development practices that it introduced to the community. I discuss the most important of these practices in this section.

### 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

**Figure 3.7** Test case description for dose checking

#### 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.

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. A 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

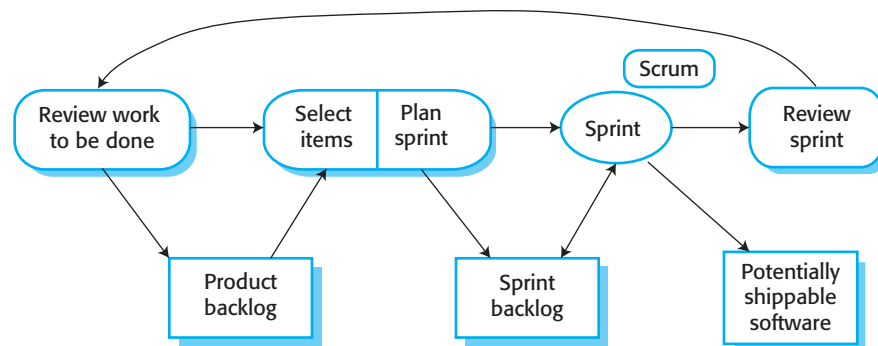
Scrum term	Definition
Development team	A self-organizing group of software developers, which should be no more than seven people. They are responsible for developing the software and other essential project documents.
Potentially shippable product increment	The software increment that is delivered from 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.

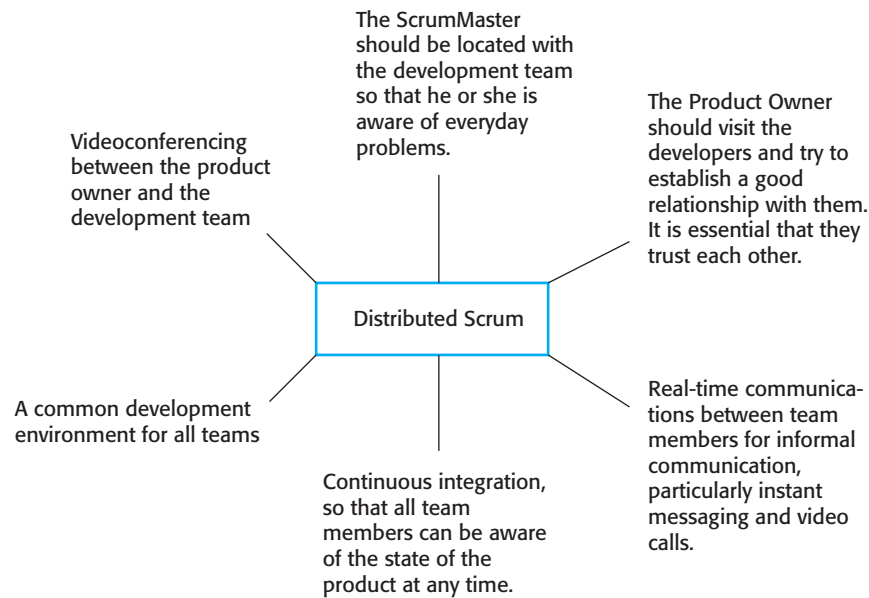
At the end of each sprint, there is a review meeting, which involves the whole team. This meeting has two purposes. First, it is a means of process improvement. The team reviews the way they have worked and reflects on how things could have been done better. Second, it provides input on the product and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

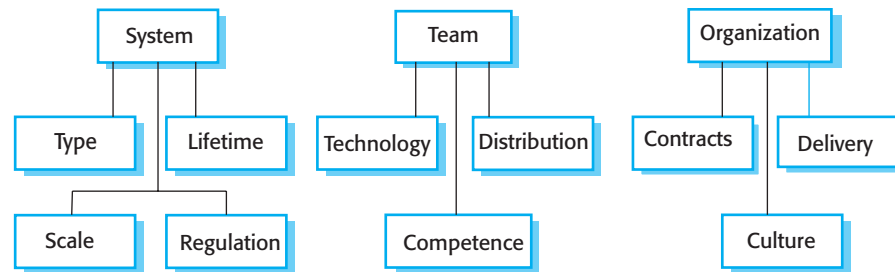
A fundamental requirement of scaling agile methods is to integrate them with plan-driven approaches. Small startup companies can work with informal and short-term planning, but larger companies have to have longer-term plans and budgets for investment, staffing, and business development. Their software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

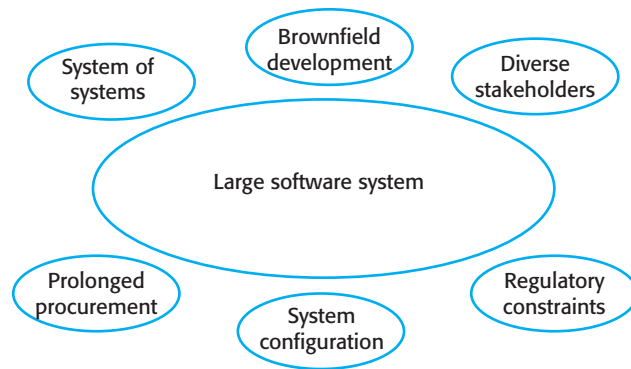
1. Is it important to have a very detailed specification and design before moving to implementation, perhaps for contractual reasons? If so, you probably need to use a plan-driven approach for requirements engineering but may use agile development practices during system implementation.
2. Is an incremental 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:

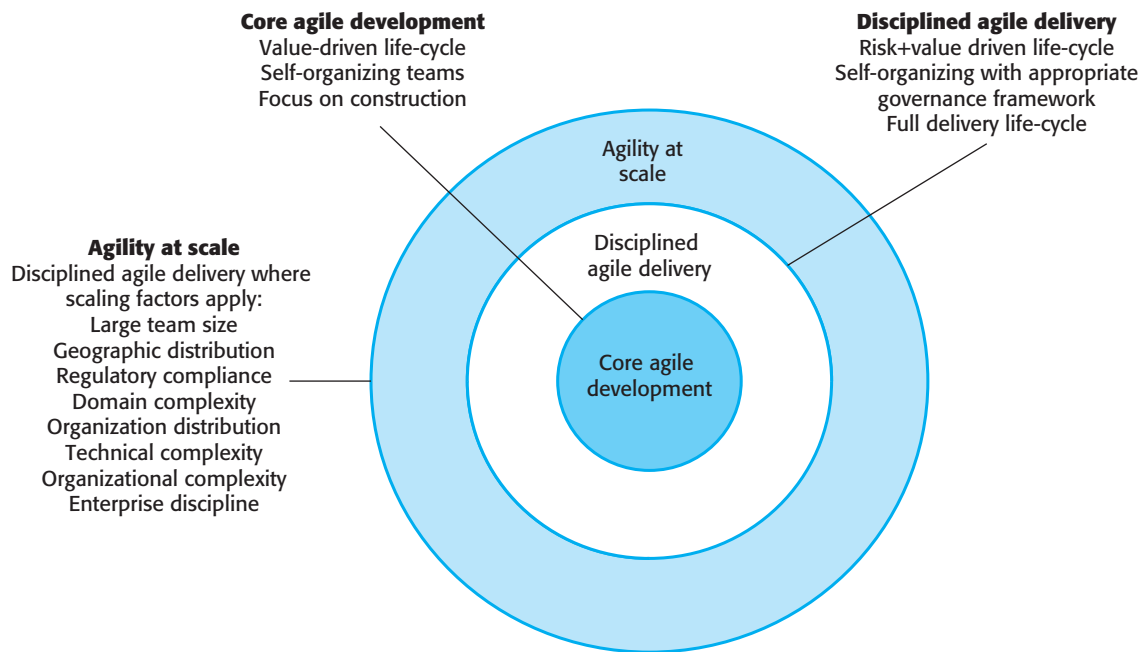




**Figure 3.13** Large project characteristics

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.

## KEY POINTS

- Agile methods are iterative development methods that focus on reducing process overheads and documentation and on incremental software delivery. They involve customer representatives directly in the development process.
- The decision on whether to use an agile or a plan-driven approach to development should depend on the type of software being developed, the capabilities of the development team, and the culture of the company developing the system. In practice, a mix of agile and plan-based techniques may be used.
- Agile development practices include requirements expressed as user stories, pair programming, refactoring, continuous integration, and test-first development.
- Scrum is an agile method that provides a framework for organizing agile projects. It is centered around a set of sprints, which are fixed time periods when a system increment is developed. Planning is based on prioritizing a backlog of work and selecting the highest priority tasks for a sprint.
- To scale agile methods, some plan-based practices have to be integrated with agile practice. These include up-front requirements, multiple customer representatives, more documentation, common tooling across project teams, and the alignment of releases across teams.

## FURTHER READING

“Get Ready for Agile Methods, With Care.” A thoughtful critique of agile methods that discusses their strengths and weaknesses, written by a vastly experienced software engineer. Still very relevant, although almost 15 years old. (B. Boehm, *IEEE Computer*, January 2002) <http://dx.doi.org/10.1109/2.976920>

*Extreme Programming Explained*. This was the first book on XP and is still, perhaps, the most readable. It explains the approach from the perspective of one of its inventors, and his enthusiasm comes through very clearly in the book. (K. Beck and C. Andres, Addison-Wesley, 2004) *Essential Scrum: A Practical Guide to the Most Popular Agile Process*. This is a comprehensive and readable description of the 2011 development of the Scrum method (K.S. Rubin, Addison-Wesley, 2013).

“Agility at Scale: Economic Governance, Measured Improvement and Disciplined Delivery.” This paper discusses IBM's approach to scale agile methods, where they have a systematic approach to integrating plan-based and agile development. It is an excellent and thoughtful discussion of the key issues in scaling agile (A.W. Brown, S.W. Ambler, and W. Royce, *Proc. 35th Int. Conf. on Software Engineering*, 2013) <http://dx.doi.org/10.1145/12944.12948>

## WEBSITE

PowerPoint slides for this chapter:

[www.pearsonglobaleditions.com/Sommerville](http://www.pearsonglobaleditions.com/Sommerville)

Links to supporting videos:

<http://software-engineering-book.com/videos/agile-methods/>

## EXERCISES

- 3.1. At the end of their study program, students in a software engineering course are typically expected to complete a major project. Explain how the agile methodology may be very useful for the students to use in this case.
- 3.2. Explain how the principles underlying agile methods lead to the accelerated development and deployment of software.
- 3.3. Extreme programming expresses user requirements as stories, with each story written on a card. Discuss the advantages and disadvantages of this approach to requirements description.
- 3.4. In test-first development, tests are written before the code. Explain how the test suite may compromise the quality of the software system being developed.
- 3.5. Suggest four reasons why the productivity rate of programmers working as a pair might be more than half that of two programmers working individually.
- 3.6. Compare and contrast the Scrum approach to project management with conventional plan-based approaches as discussed in Chapter 23. Your comparison should be based on the effectiveness of each approach for planning the allocation of people to projects, estimating the cost of projects, maintaining team cohesion, and managing changes in project team membership.
- 3.7. To reduce costs and the environmental impact of commuting, your company decides to close a number of offices and to provide support for staff to work from home. However, the senior management who introduce the policy are unaware that software is developed using Scrum. Explain how you could use technology to support Scrum in a distributed environment to make this possible. What problems are you likely to encounter using this approach?
- 3.8. Why is it necessary to introduce some methods and documentation from plan-based approaches when scaling agile methods to larger projects that are developed by distributed development teams?
- 3.9. Explain why agile methods may not work well in organizations that have teams with a wide range of skills and abilities and well-established processes.
- 3.10. One of the problems of having a user closely involved with a software development team is that they “go native.” That is, they adopt the outlook of the development team and lose sight of the needs of their user colleagues. Suggest three ways how you might avoid this problem, and discuss the advantages and disadvantages of each approach.

## REFERENCES

- Ambler, S. W. 2010. “Scaling Agile: A Executive Guide.” [http://www.ibm.com/developerworks/community/blogs/ambler/entry/scaling\\_agile\\_an\\_executive\\_guide10/](http://www.ibm.com/developerworks/community/blogs/ambler/entry/scaling_agile_an_executive_guide10/)
- Arisholm, E., H. Gallis, T. Dyba, and D. I. K. Sjöberg. 2007. “Evaluating Pair Programming with Respect to System Complexity and Programmer Expertise.” *IEEE Trans. on Software Eng.* 33 (2): 65–86. doi:10.1109/TSE.2007.17.
- Beck, K. 1998. “Chrysler Goes to ‘Extremes.’” *Distributed Computing* (10): 24–28.

- . 1999. "Embracing Change with Extreme Programming." *IEEE Computer* 32 (10): 70–78. doi:10.1109/2.796139.
- Bellouiti, S. 2009. "How Scrum Helped Our A-Team." <http://www.scrumalliance.org/community/articles/2009/2009-june/how-scrum-helped-our-team>
- Bird, J. 2011. "You Can't Be Agile in Maintenance." <http://swreflections.blogspot.co.uk/2011/10/you-cant-be-agile-in-maintenance.html>
- Deemer, P. 2011. "The Distributed Scrum Primer." <http://www.goodagile.com/distributedscrumprimer/>.
- Fowler, M., K. Beck, J. Brant, W. Opdyke, and D. Roberts. 1999. *Refactoring: Improving the Design of Existing Code*. Boston: Addison-Wesley.
- Hopkins, R., and K. Jenkins. 2008. *Eating the IT Elephant: Moving from Greenfield Development to Brownfield*. Boston: IBM Press.
- Jeffries, R., and G. Melnik. 2007. "TDD: The Art of Fearless Programming." *IEEE Software* 24: 24–30. doi:10.1109/MS.2007.75.
- Kilner, S. 2012. "Can Agile Methods Work for Software Maintenance." <http://www.vlegaci.com/can-agile-methods-work-for-software-maintenance-part-1/>
- Larman, C., and V. R. Basili. 2003. "Iterative and Incremental Development: A Brief History." *IEEE Computer* 36 (6): 47–56. doi:10.1109/MC.2003.1204375.
- Leffingwell, D. 2007. *Scaling Software Agility: Best Practices for Large Enterprises*. Boston: Addison-Wesley.
- Leffingwell, D. 2011. *Agile Software Requirements: Lean Requirements Practices for Teams, Programs and the Enterprise*. Boston: Addison-Wesley.
- Mulder, M., and M. van Vliet. 2008. "Case Study: Distributed Scrum Project for Dutch Railways." *InfoQ*. <http://www.infoq.com/articles/dutch-railway-scrum>
- Rubin, K. S. 2013. *Essential Scrum*. Boston: Addison-Wesley.
- Schatz, B., and I. Abdelshafi. 2005. "Primavera Gets Agile: A Successful Transition to Agile Development." *IEEE Software* 22 (3): 36–42. doi:10.1109/MS.2005.74.
- Schwaber, K., and M. Beedle. 2001. *Agile Software Development with Scrum*. Englewood Cliffs, NJ: Prentice-Hall.
- Stapleton, J. 2003. *DSDM: Business Focused Development, 2nd ed.* Harlow, UK: Pearson Education.
- Tahchiev, P., F. Leme, V. Massol, and G. Gregory. 2010. *JUnit in Action, 2/e*. Greenwich, CT: Manning Publications.
- Weinberg, G. 1971. *The Psychology of Computer Programming*. New York: Van Nostrand.
- Williams, L., R. R. Kessler, W. Cunningham, and R. Jeffries. 2000. "Strengthening the Case for Pair Programming." *IEEE Software* 17 (4): 19–25. doi:10.1109/52.854064.