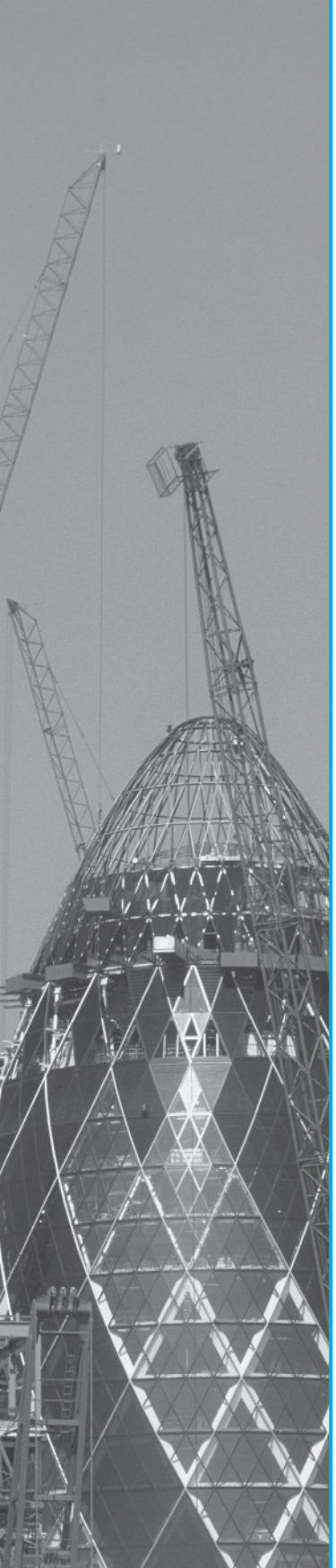

2 Software processes

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

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 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 thischapter 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

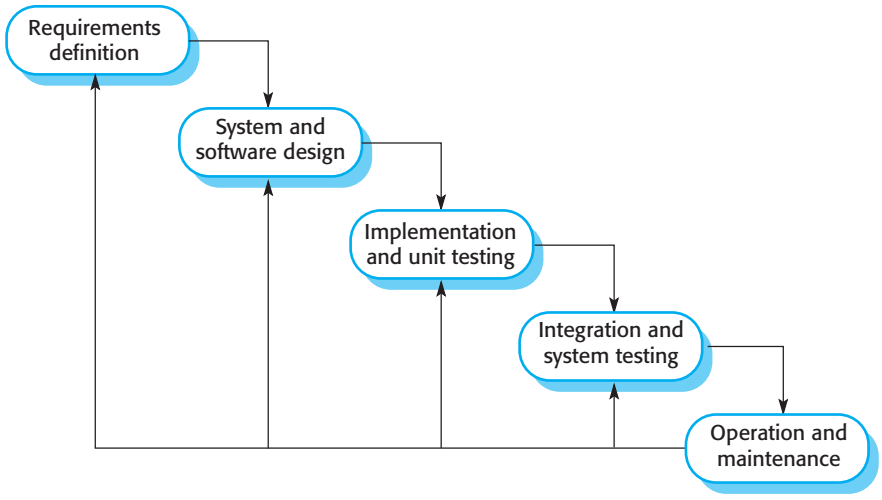

Figure 2.1 The waterfall model

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.

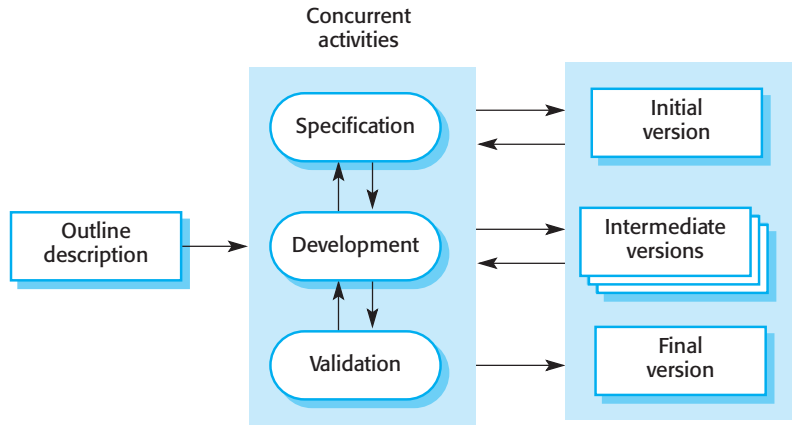

Figure 2.2 Incremental development

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.

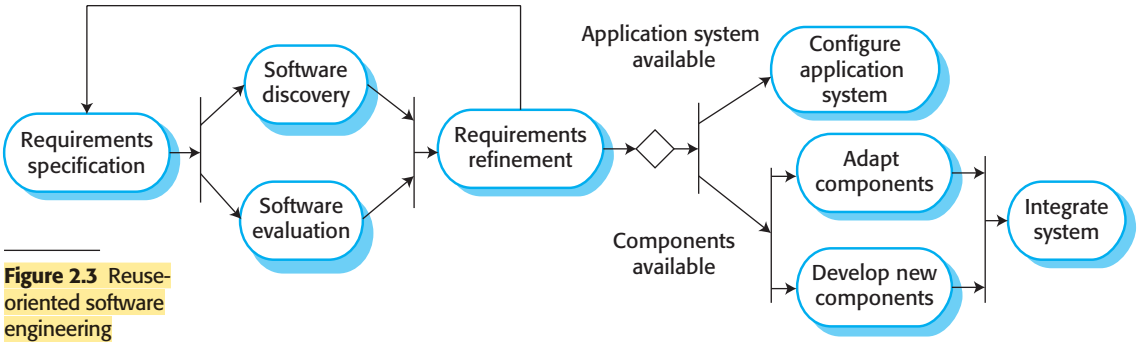

Figure 2.3 Reuse-oriented software engineering

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 waterfall 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.

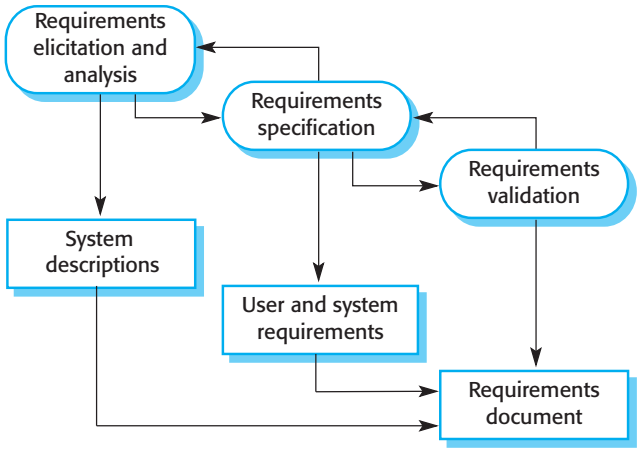

Figure 2.4 The requirements engineering process

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.

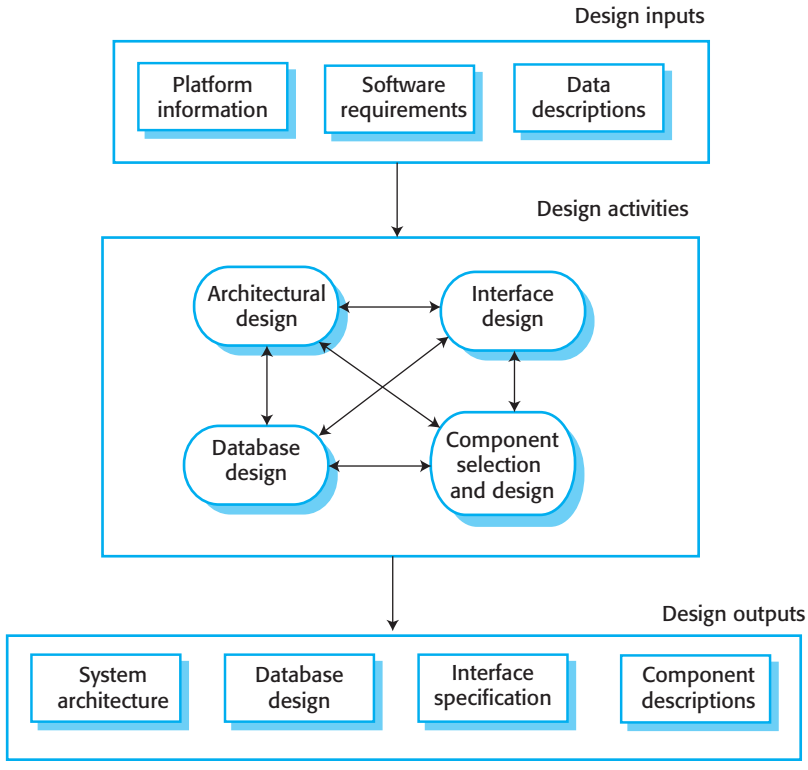

Figure 2.5 A general model of the design process

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

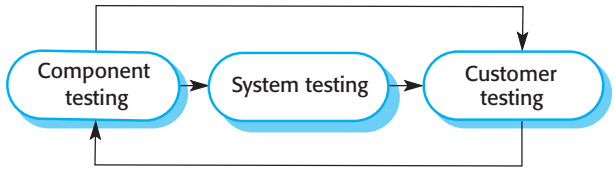

Figure 2.6 Stages of testing

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

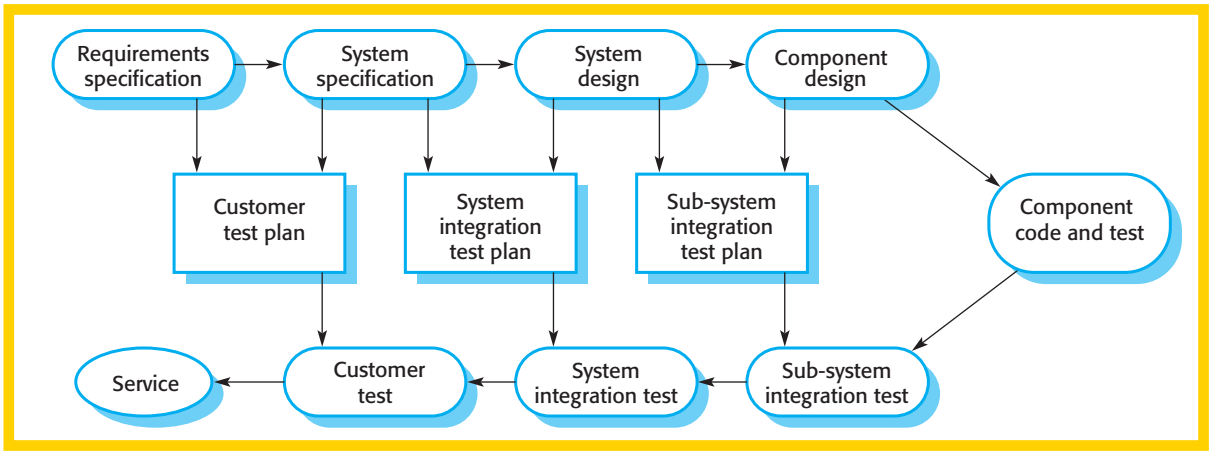

Figure 2.7 Testing phases in a plan-driven software process

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

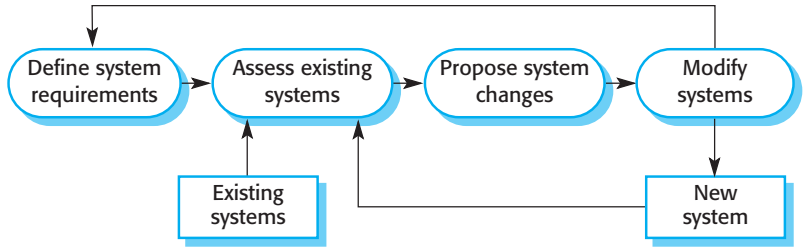

Figure 2.8 Software system evolution

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.

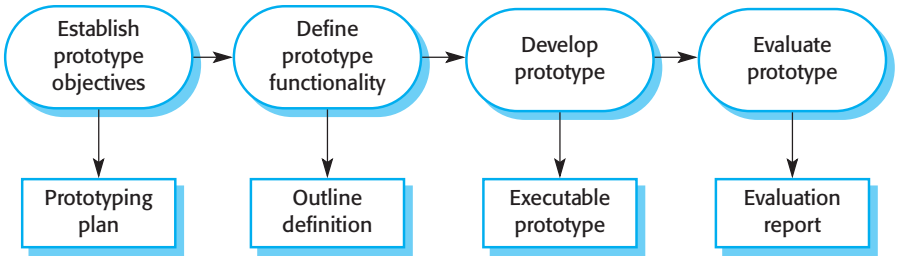

Figure 2.9 Prototype development

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.

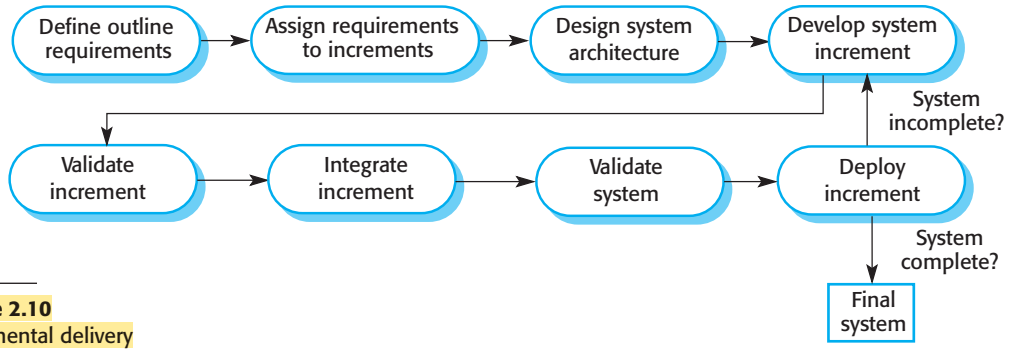

Figure 2.10
Incremental delivery

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

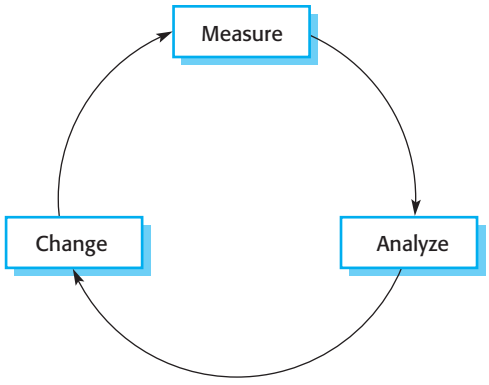

Figure 2.11 The process improvement cycle

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.

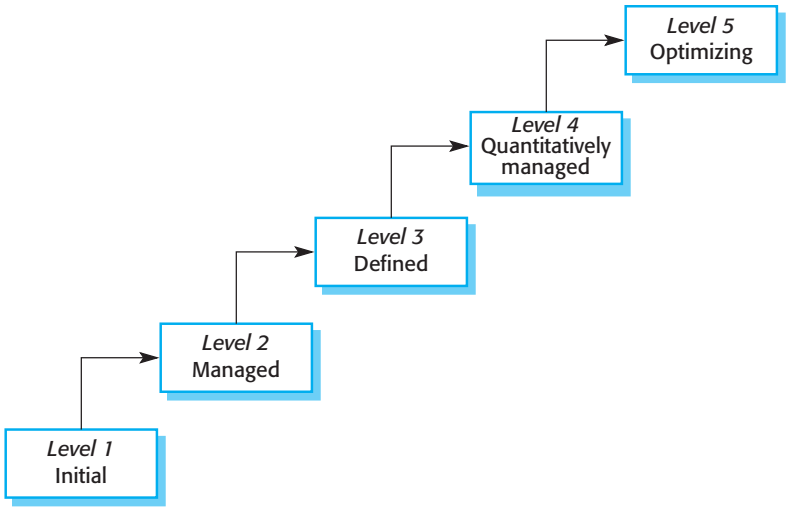

Figure 2.12 Capability maturity levels

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

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.

Krutchen, 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.