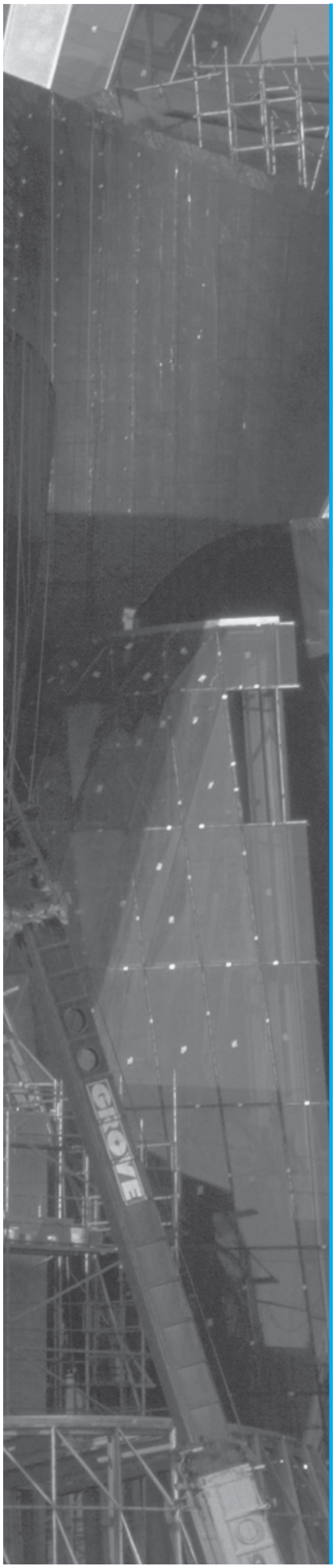


# 2

# 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 generic 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 how the Rational Unified Process integrates good software engineering practice to create adaptable software processes.

## Contents

A software process is a set of related activities that leads to the production of a software product. These activities may involve the development of software from scratch in a standard programming language like Java or C. However, business applications are not necessarily developed in this way. New business software is now often developed by extending and modifying existing systems or by configuring and integrating off-the-shelf software or system components.

There are many different software processes but all must include four activities that are fundamental to software engineering:

1. *Software specification* The functionality of the software and constraints on its operation must be defined.
2. *Software design and implementation* 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.

In some form, these activities are part of all software processes. In practice, of course, they are complex activities in themselves and include sub-activities such as requirements validation, architectural design, unit testing, etc. There are also supporting process activities such as documentation and software configuration management.

When we describe and discuss processes, we usually talk about the activities in these processes such as specifying a data model, designing a user interface, etc., and the ordering of these activities. However, as well as activities, process descriptions may also include:

1. Products, which 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, which reflect the responsibilities of the people involved in the process. Examples of roles are project manager, configuration manager, programmer, etc.
3. Pre- and post-conditions, which are statements that are true before and after a process activity has been enacted or a product produced. For example, before architectural design begins, a pre-condition may be that all requirements have been approved by the customer; after this activity is finished, a post-condition 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. There is no ideal process and most organizations have developed their own software development processes. Processes have evolved to take advantage of the capabilities of the people in an organization and the specific characteristics of the systems that are being developed. For some systems, such as critical systems, a very structured development process is required. For business systems, with rapidly changing requirements, a less formal, flexible process is likely to be more effective.

Sometimes, software processes are categorized as either plan-driven or agile 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 it is easier to change the process to reflect changing customer requirements. As Boehm and Turner (2003) discuss, each approach is suitable for different types of software. Generally, you need to find a balance between plan-driven and agile processes.

Although there is no 'ideal' software process, there is scope for improving the software process 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.

Software processes can be improved by process standardization where the diversity in software processes across an organization is reduced. This leads to improved communication and a reduction in training time, and makes automated process support more economical. Standardization is also an important first step in introducing new software engineering methods and techniques and good software engineering practice. I discuss software process improvement in more detail in Chapter 26.

## 2.1 Software process models

As I explained in Chapter 1, a software process model is a simplified representation of a software process. Each process model represents a process from a particular perspective, and thus provides only 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 specific activities.

These generic models are not definitive descriptions of software processes. Rather, they are abstractions of the process 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 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, testing, and so on.

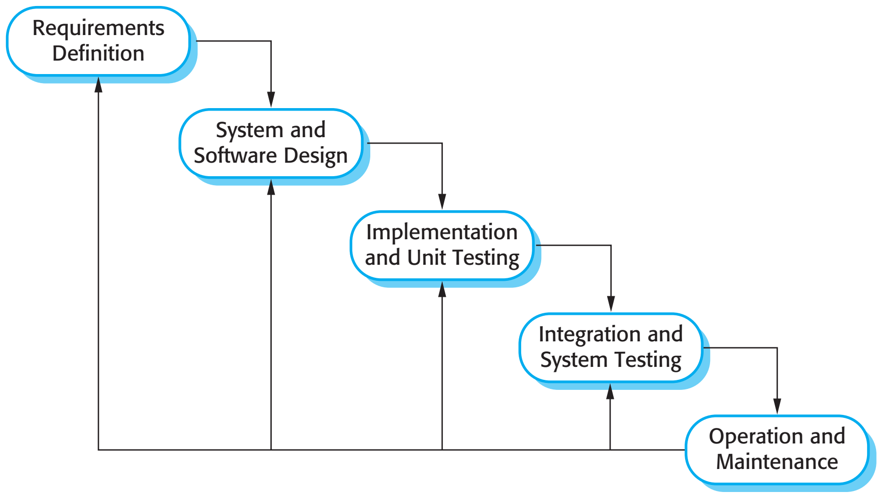


**Figure 2.1** The waterfall model

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. *Reuse-oriented software engineering* This approach is based on the existence of a significant number of reusable components. The system development process focuses on integrating these components into a system rather than developing them from scratch.

These models are not mutually exclusive and are often used together, especially for large systems development. For large systems, it makes sense to combine some of the best features of the waterfall and the incremental development models. You need to have information about the essential system requirements to design a software architecture to support these requirements. You cannot develop this incrementally. Sub-systems 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. Parts of the system which are difficult to specify in advance, such as the user interface, should always be developed using an incremental approach.

### 2.1.1 The waterfall model

The first published model of the software development process was derived from more general system engineering processes (Royce, 1970). This model is illustrated 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, you must plan and schedule all of the process activities before starting work on them.

The principal stages of the waterfall model directly reflect the fundamental 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 by establishing 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.
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 (although not necessarily), this is the longest life cycle phase. The system is installed and put into practical use. Maintenance involves correcting errors which 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 is one or more documents that are approved ('signed off'). The following phase should not start until the previous phase has finished. In practice, 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 is not a simple linear model but involves feedback from one phase to another. Documents produced in each phase may then have to be modified to reflect the changes made.

Because of the costs of producing and approving documents, iterations can be costly and involve significant rework. Therefore, after a small number of iterations, it is normal to freeze parts of the development, such as the specification, and to continue with the later development stages. Problems are left for later resolution, ignored, or programmed around. This 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.

**Cleanroom software engineering**

An example of a formal development process, originally developed by IBM, is the Cleanroom process. In the Cleanroom process each software increment is formally specified and this specification is transformed into an implementation. Software correctness is demonstrated using a formal approach. There is no unit testing for defects in the process and the system testing is focused on assessing the system's reliability.

The objective of the Cleanroom process is zero-defects software so that delivered systems have a high level of reliability.

**http://www.SoftwareEngineering-9.com/Web/Cleanroom/**

The waterfall model is consistent with other engineering process models and documentation is produced at each phase. This makes the process visible so managers can monitor progress against the development plan. Its major problem is the inflexible partitioning of the project into distinct stages. Commitments must be made at an early stage in the process, which makes it difficult to respond to changing customer requirements.

In principle, the waterfall model should only be used when the requirements are well understood and unlikely to change radically during system development. However, the waterfall model reflects the type of process used in other engineering projects. As is easier to use a common management model for the whole project, software processes based on the waterfall model are still commonly used.

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. Based on the assumption that your mathematical transformations are correct, you can therefore make a strong argument that a program generated in this way is consistent with its specification.

Formal development processes, such as that based on the B method (Schneider, 2001; Wordsworth, 1996) are particularly suited to the development of 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.

Processes based on formal transformations are generally only used in the development of safety-critical or security-critical systems. They require specialized expertise. For the majority of systems this process does not offer significant cost-benefits over other approaches to system development.

### 2.1.2 Incremental development

Incremental development is based on the idea of developing an initial implementation, exposing this to user comment and evolving it through several versions until an adequate system has been developed (Figure 2.2). Specification, development, and

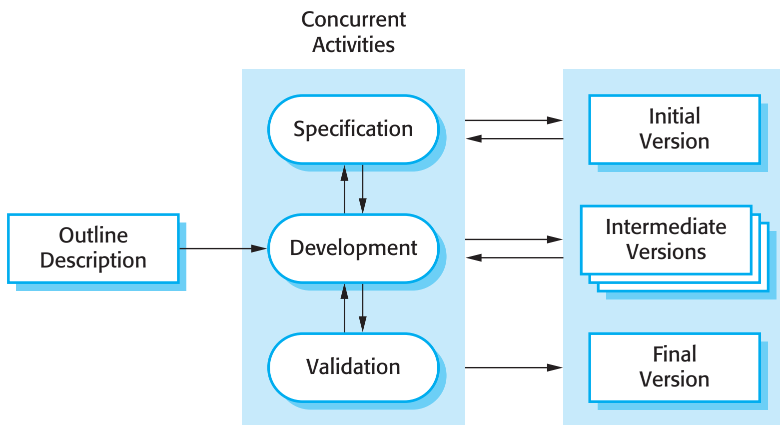


**Figure 2.2** Incremental development

validation activities are interleaved rather than separate, with rapid feedback across activities.

Incremental software development, which is a fundamental part of agile approaches, is better than a waterfall approach for most business, e-commerce, and personal systems. 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 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 important benefits, compared to the waterfall model:

1. The cost of accommodating changing customer requirements is reduced. The amount of analysis and documentation that has to be redone is much 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 much has been implemented. Customers find it difficult to judge progress from software design documents.

3. More rapid 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.

**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 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://www.SoftwareEngineering-9.com/Web/IncrementalDev/**

Incremental development in some form is now the most common approach for the development of application systems. 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.

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. Unless time and money is spent on refactoring to improve the software, regular change tends to corrupt its structure. Incorporating further software changes becomes increasingly difficult and costly.

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.

You can develop a system incrementally and expose it to customers for comment, without actually delivering it and deploying it in the customer's environment. Incremental delivery and deployment means that the software is used in real, operational processes. This is not always possible as experimenting with new software can disrupt normal business processes. I discuss the advantages and disadvantages of incremental delivery in Section 2.3.2.

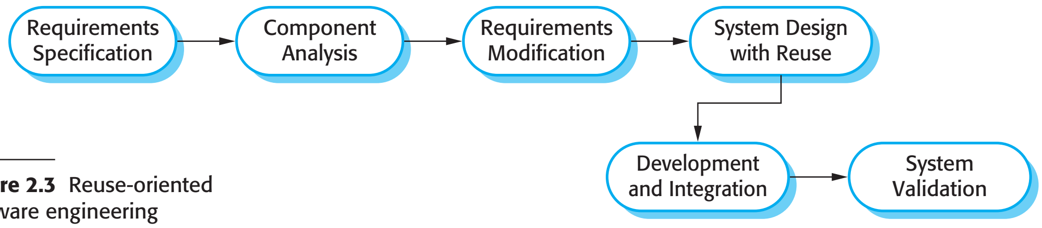


**Figure 2.3** Reuse-oriented software engineering

## 2.1.3 Reuse-oriented software engineering

In the majority of software projects, there is some software reuse. This often happens informally when people working on the project know of designs or code that are similar to what is required. They look for these, modify them as needed, and incorporate them into their system.

This informal reuse takes place irrespective of the development process that is used. However, in the 21st century, software development processes that focus on the reuse of existing software have become widely used. Reuse-oriented approaches rely on a large base of reusable software components and an integrating framework for the composition of these components. Sometimes, these components are systems in their own right (COTS or commercial off-the-shelf systems) that may provide specific functionality such as word processing or a spreadsheet.

A general process model for reuse-based development is shown in Figure 2.3. Although the initial requirements specification stage and the validation stage are comparable with other software processes, the intermediate stages in a reuse-oriented process are different. These stages are:

1. *Component analysis* Given the requirements specification, a search is made for components to implement that specification. Usually, there is no exact match and the components that may be used only provide some of the functionality required.

2. *Requirements modification* During this stage, the requirements are analyzed using information about the components that have been discovered. They are then modified to reflect the available components. Where modifications are impossible, the component analysis activity may be re-entered to search for alternative solutions.

3. *System design with reuse* During this phase, the framework of the system is designed or an existing framework is reused. The designers take into account the components that are reused and organize the framework to cater for this. Some new software may have to be designed if reusable components are not available.

4. *Development and integration* Software that cannot be externally procured is developed, and the components and COTS systems are integrated to create the new system. System integration, in this model, may be part of the development process rather than a separate activity.

There are three types of software component that may be used in a reuse-oriented process:

1. Web services that are developed according to service standards and which are available for remote invocation.
2. Collections of objects that are developed as a package to be integrated with a component framework such as .NET or J2EE.
3. Stand-alone software systems that are configured for use in a particular environment.

Reuse-oriented software engineering 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 I have dedicated several chapters in the third part of the book to this topic. General issues of software reuse and COTS reuse are covered in Chapter 16, component-based software engineering in Chapters 17 and 18, and service-oriented systems in Chapter 19.

## 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. Software developers use a variety of different software tools in their work. Tools are particularly useful for supporting the editing of different types of document and for managing the immense volume of detailed information that is generated in a large software project.

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, people, and organizational structures involved. In extreme programming, for example, specifications are written on cards. Tests are executable and developed before the program itself. Evolution may involve substantial system restructuring or refactoring.

### 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