The requirements for a system are the descriptions of what the system should do—the services that it provides and the constraints on its operation. These requirements reflect the needs of customers for a system that serves a certain purpose such as controlling a device, placing an order, or finding information. The process of finding out, analyzing, documenting and checking these services and constraints is called requirements engineering (RE).

The term 'requirement' is not used consistently in the software industry. In some cases, a requirement is simply a high-level, abstract statement of a service that a system should provide or a constraint on a system. At the other extreme, it is a detailed, formal definition of a system function. Davis (1993) explains why these differences exist:

> *If a company wishes to let a contract for a large software development project, it must define its needs in a sufficiently abstract way that a solution is not predefined. The requirements must be written so that several contractors can bid for the contract, offering, perhaps, different ways of meeting the client organization's needs. Once a contract has been awarded, the contractor must write a system definition for the client in more detail so that the client understands and can validate what the software will do. Both of these documents may be called the requirements document for the system.*

Some of the problems that arise during the requirements engineering process are a result of failing to make a clear separation between these different levels of description. I distinguish between them by using the term 'user requirements' to mean the high-level abstract requirements and 'system requirements' to mean the detailed description of what the system should do. User requirements and system requirements may be defined as follows:

1. User requirements are statements, in a natural language plus diagrams, of what services the system is expected to provide to system users and the constraints under which it must operate.
2. System requirements are more detailed descriptions of the software system's functions, services, and operational constraints. The system requirements document (sometimes called a functional specification) should define exactly what is to be implemented. It may be part of the contract between the system buyer and the software developers.

Different levels of requirements are useful because they communicate information about the system to different types of reader. Figure 4.1 illustrates the distinction between user and system requirements. This example from a mental health care patient management system (MHC-PMS) shows how a user requirement may be expanded into several system requirements. You can see from Figure 4.1 that the user requirement is quite general. The system requirements provide more specific information about the services and functions of the system that is to be implemented.

User Requirement Definition

1. The MHC-PMS shall generate monthly management reports showing the cost of drugs prescribed by each clinic during that month.

System Requirements Specification

1.1 On the last working day of each month, a summary of the drugs prescribed, their cost, and the prescribing clinics shall be generated.

1.2 The system shall automatically generate the report for printing after 17.30 on the last working day of the month.

1.3 A report shall be created for each clinic and shall list the individual drug names, the total number of prescriptions, the number of doses prescribed, and the total cost of the prescribed drugs.

1.4 If drugs are available in different dose units (e.g., 10 mg, 20 mg) separate reports shall be created for each dose unit.

1.5 Access to all cost reports shall be restricted to authorized users listed on a management access control list.

Figure 4.1 User and system requirements

You need to write requirements at different levels of detail because different readers use them in different ways. Figure 4.2 shows possible readers of the user and system requirements. The readers of the user requirements are not usually concerned with how the system will be implemented and may be managers who are not interested in the detailed facilities of the system. The readers of the system requirements need to know more precisely what the system will do because they are concerned with how it will support the business processes or because they are involved in the system implementation.

In this chapter, I present a 'traditional' view of requirements rather than requirements in agile processes. For most large systems, it is still the case that there is a clearly identifiable requirements engineering phase before the implementation of the system begins. The outcome is a requirements document, which may be part of the system development contract. Of course, there are usually subsequent changes to the requirements and user requirements may be expanded into more detailed system requirements. However, the agile approach of concurrently eliciting the requirements as the system is developed is rarely used for large systems development.

4.1 Functional and non-functional requirements

Software system requirements are often classified as functional requirements or non-functional requirements:

1. *Functional requirements* These are statements of services the system should provide, how the system should react to particular inputs, and how the system

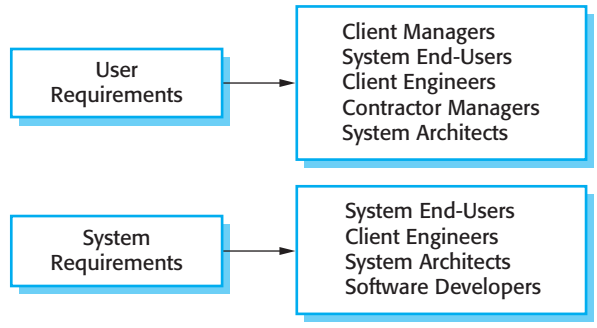


Figure 4.2 Readers of different types of requirements specification

should behave in particular situations. In some cases, the functional requirements may also explicitly state what the system should not do.

2. *Non-functional requirements* These are constraints on the services or functions offered by the system. They include timing constraints, constraints on the development process, and constraints imposed by standards. Non-functional requirements often apply to the system as a whole, rather than individual system features or services.

In reality, the distinction between different types of requirement is not as clear-cut as these simple definitions suggest. A user requirement concerned with security, such as a statement limiting access to authorized users, may appear to be a non-functional requirement. However, when developed in more detail, this requirement may generate other requirements that are clearly functional, such as the need to include user authentication facilities in the system.

This shows that requirements are not independent and that one requirement often generates or constrains other requirements. The system requirements therefore do not just specify the services or the features of the system that are required; they also specify the necessary functionality to ensure that these services/features are delivered properly.

4.1.1 Functional requirements

The functional requirements for a system describe what the system should do. These requirements depend on the type of software being developed, the expected users of the software, and the general approach taken by the organization when writing requirements. When expressed as user requirements, functional requirements are usually described in an abstract way that can be understood by system users. However, more specific functional system requirements describe the system functions, its inputs and outputs, exceptions, etc., in detail.

Functional system requirements vary from general requirements covering what the system should do to very specific requirements reflecting local ways of working or an organization's existing systems. For example, here are examples of functional

Domain requirements

Domain requirements are derived from the application domain of the system rather than from the specific needs of system users. They may be new functional requirements in their own right, constrain existing functional requirements, or set out how particular computations must be carried out.

The problem with domain requirements is that software engineers may not understand the characteristics of the domain in which the system operates. They often cannot tell whether or not a domain requirement has been missed out or conflicts with other requirements.

http://www.SoftwareEngineering-9.com/Web/Requirements/DomainReq.html

requirements for the MHC-PMS system, used to maintain information about patients receiving treatment for mental health problems:

1. A user shall be able to search the appointments lists for all clinics.
2. The system shall generate each day, for each clinic, a list of patients who are expected to attend appointments that day.
3. Each staff member using the system shall be uniquely identified by his or her eight-digit employee number.

These functional user requirements define specific facilities to be provided by the system. These have been taken from the user requirements document and they show that functional requirements may be written at different levels of detail (contrast requirements 1 and 3).

Imprecision in the requirements specification is the cause of many software engineering problems. It is natural for a system developer to interpret an ambiguous requirement in a way that simplifies its implementation. Often, however, this is not what the customer wants. New requirements have to be established and changes made to the system. Of course, this delays system delivery and increases costs.

For example, the first example requirement for the MHC-PMS states that a user shall be able to search the appointments lists for all clinics. The rationale for this requirement is that patients with mental health problems are sometimes confused. They may have an appointment at one clinic but actually go to a different clinic. If they have an appointment, they will be recorded as having attended, irrespective of the clinic.

The medical staff member specifying this may expect 'search' to mean that, given a patient name, the system looks for that name in all appointments at all clinics. However, this is not explicit in the requirement. System developers may interpret the requirement in a different way and may implement a search so that the user has to choose a clinic then carry out the search. This obviously will involve more user input and so take longer.

In principle, the functional requirements specification of a system should be both complete and consistent. Completeness means that all services required by the user should be defined. Consistency means that requirements should not have contradictory

definitions. In practice, for large, complex systems, it is practically impossible to achieve requirements consistency and completeness. One reason for this is that it is easy to make mistakes and omissions when writing specifications for complex systems. Another reason is that there are many stakeholders in a large system. A stakeholder is a person or role that is affected by the system in some way. Stakeholders have different—and often inconsistent—needs. These inconsistencies may not be obvious when the requirements are first specified, so inconsistent requirements are included in the specification. The problems may only emerge after deeper analysis or after the system has been delivered to the customer.

4.1.2 Non-functional requirements

Non-functional requirements, as the name suggests, are requirements that are not directly concerned with the specific services delivered by the system to its users. They may relate to emergent system properties such as reliability, response time, and store occupancy. Alternatively, they may define constraints on the system implementation such as the capabilities of I/O devices or the data representations used in interfaces with other systems.

Non-functional requirements, such as performance, security, or availability, usually specify or constrain characteristics of the system as a whole. Non-functional requirements are often more critical than individual functional requirements. System users can usually find ways to work around a system function that doesn't really meet their needs. However, failing to meet a non-functional requirement can mean that the whole system is unusable. For example, if an aircraft system does not meet its reliability requirements, it will not be certified as safe for operation; if an embedded control system fails to meet its performance requirements, the control functions will not operate correctly.

Although it is often possible to identify which system components implement specific functional requirements (e.g., there may be formatting components that implement reporting requirements), it is often more difficult to relate components to non-functional requirements. The implementation of these requirements may be diffused throughout the system. There are two reasons for this:

1. Non-functional requirements may affect the overall architecture of a system rather than the individual components. For example, to ensure that performance requirements are met, you may have to organize the system to minimize communications between components.
2. A single non-functional requirement, such as a security requirement, may generate a number of related functional requirements that define new system services that are required. In addition, it may also generate requirements that restrict existing requirements.

Non-functional requirements arise through user needs, because of budget constraints, organizational policies, the need for interoperability with other software or

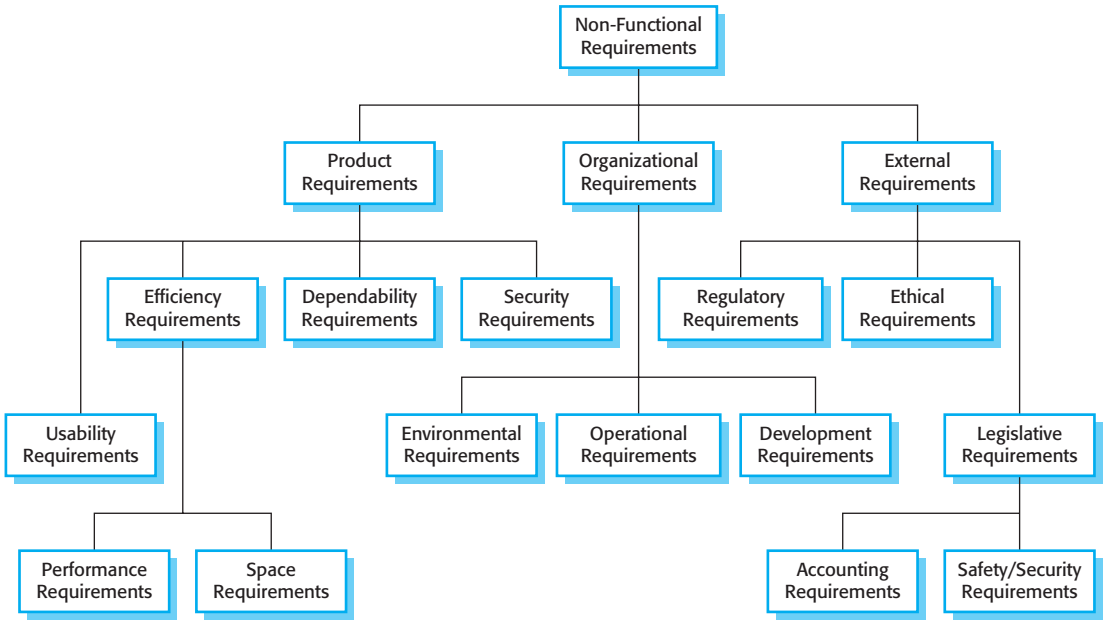


Figure 4.3 Types of non-functional requirement

hardware systems, or external factors such as safety regulations or privacy legislation. Figure 4.3 is a classification of non-functional requirements. You can see from this diagram that the non-functional requirements may come from required characteristics of the software (product requirements), the organization developing the software (organizational requirements), or from external sources:

1. *Product requirements* These requirements specify or constrain the behavior of the software. Examples include performance requirements on how fast the system must execute and how much memory it requires, reliability requirements that set out the acceptable failure rate, security requirements, and usability requirements.
2. *Organizational requirements* These requirements are broad system requirements derived from policies and procedures in the customer's and developer's organization. Examples include operational process requirements that define how the system will be used, development process requirements that specify the programming language, the development environment or process standards to be used, and environmental requirements that specify the operating environment of the system.
3. *External requirements* This broad heading covers all requirements that are derived from factors external to the system and its development process. These may include regulatory requirements that set out what must be done for the system to be approved for use by a regulator, such as a central bank; legislative requirements that must be followed to ensure that the system operates within the law; and ethical requirements that ensure that the system will be acceptable to its users and the general public.

PRODUCT REQUIREMENT
The MHC-PMS shall be available to all clinics during normal working hours (Mon–Fri, 08.30–17.30). Downtime within normal working hours shall not exceed five seconds in any one day.

ORGANIZATIONAL REQUIREMENT
Users of the MHC-PMS system shall authenticate themselves using their health authority identity card.

EXTERNAL REQUIREMENT
The system shall implement patient privacy provisions as set out in HStan-03-2006-priv.

Figure 4.4 Examples of non-functional requirements in the MHC-PMS

Figure 4.4 shows examples of product, organizational, and external requirements taken from the MHC-PMS whose user requirements were introduced in Section 4.1.1. The product requirement is an availability requirement that defines when the system has to be available and the allowed down time each day. It says nothing about the functionality of MHC-PMS and clearly identifies a constraint that has to be considered by the system designers.

The organizational requirement specifies how users authenticate themselves to the system. The health authority that operates the system is moving to a standard authentication procedure for all software where, instead of users having a login name, they swipe their identity card through a reader to identify themselves. The external requirement is derived from the need for the system to conform to privacy legislation. Privacy is obviously a very important issue in healthcare systems and the requirement specifies that the system should be developed in accordance with a national privacy standard.

A common problem with non-functional requirements is that users or customers often propose these requirements as general goals, such as ease of use, the ability of the system to recover from failure, or rapid user response. Goals set out good intentions but cause problems for system developers as they leave scope for interpretation and subsequent dispute once the system is delivered. For example, the following system goal is typical of how a manager might express usability requirements:

> *The system should be easy to use by medical staff and should be organized in such a way that user errors are minimized.*

I have rewritten this to show how the goal could be expressed as a 'testable' non-functional requirement. It is impossible to objectively verify the system goal, but in the description below you can at least include software instrumentation to count the errors made by users when they are testing the system.

> *Medical staff shall be able to use all the system functions after four hours of training. After this training, the average number of errors made by experienced users shall not exceed two per hour of system use.*

Whenever possible, you should write non-functional requirements quantitatively so that they can be objectively tested. Figure 4.5 shows metrics that you can use to specify non-functional system properties. You can measure these characteristics

Property	Measure
Speed	Processed transactions/second User/event response time Screen refresh time
Size	Mbytes Number of ROM chips
Ease of use	Training time Number of help frames
Reliability	Mean time to failure Probability of unavailability Rate of failure occurrence Availability
Robustness	Time to restart after failure Percentage of events causing failure Probability of data corruption on failure
Portability	Percentage of target dependent statements Number of target systems

Figure 4.5 Metrics for specifying non-functional requirements

when the system is being tested to check whether or not the system has met its non-functional requirements.

In practice, customers for a system often find it difficult to translate their goals into measurable requirements. For some goals, such as maintainability, there are no metrics that can be used. In other cases, even when quantitative specification is possible, customers may not be able to relate their needs to these specifications. They don't understand what some number defining the required reliability (say) means in terms of their everyday experience with computer systems. Furthermore, the cost of objectively verifying measurable, non-functional requirements can be very high and the customers paying for the system may not think these costs are justified.

Non-functional requirements often conflict and interact with other functional or non-functional requirements. For example, the authentication requirement in Figure 4.4 obviously requires a card reader to be installed with each computer attached to the system. However, there may be another requirement that requests mobile access to the system from doctors' or nurses' laptops. These are not normally equipped with card readers so, in these circumstances, some alternative authentication method may have to be allowed.

It is difficult, in practice, to separate functional and non-functional requirements in the requirements document. If the non-functional requirements are stated separately from the functional requirements, the relationships between them may be hard to understand. However, you should explicitly highlight requirements that are clearly related to emergent system properties, such as performance or reliability. You can do this by putting them in a separate section of the requirements document or by distinguishing them, in some way, from other system requirements.

Requirements document standards

A number of large organizations, such as the U.S. Department of Defense and the IEEE, have defined standards for requirements documents. These are usually very generic but are nevertheless useful as a basis for developing more detailed organizational standards. The U.S. Institute of Electrical and Electronic Engineers (IEEE) is one of the best-known standards providers and they have developed a standard for the structure of requirements documents. This standard is most appropriate for systems such as military command and control systems that have a long lifetime and are usually developed by a group of organizations.

http://www.SoftwareEngineering-9.com/Web/Requirements/IEEE-standard.html

Non-functional requirements such as reliability, safety, and confidentiality requirements are particularly important for critical systems. I cover these requirements in Chapter 12, where I describe specific techniques for specifying dependability and security requirements.

4.2 The software requirements document

The software requirements document (sometimes called the software requirements specification or SRS) is an official statement of what the system developers should implement. It should include both the user requirements for a system and a detailed specification of the system requirements. Sometimes, the user and system requirements are integrated into a single description. In other cases, the user requirements are defined in an introduction to the system requirements specification. If there are a large number of requirements, the detailed system requirements may be presented in a separate document.

Requirements documents are essential when an outside contractor is developing the software system. However, agile development methods argue that requirements change so rapidly that a requirements document is out of date as soon as it is written, so the effort is largely wasted. Rather than a formal document, approaches such as Extreme Programming (Beck, 1999) collect user requirements incrementally and write these on cards as user stories. The user then prioritizes requirements for implementation in the next increment of the system.

For business systems where requirements are unstable, I think that this approach is a good one. However, I think that it is still useful to write a short supporting document that defines the business and dependability requirements for the system; it is easy to forget the requirements that apply to the system as a whole when focusing on the functional requirements for the next system release.

The requirements document has a diverse set of users, ranging from the senior management of the organization that is paying for the system to the engineers responsible for developing the software. Figure 4.6, taken from my book with Gerald Kotonya on requirements engineering (Kotonya and Sommerville, 1998) shows possible users of the document and how they use it.

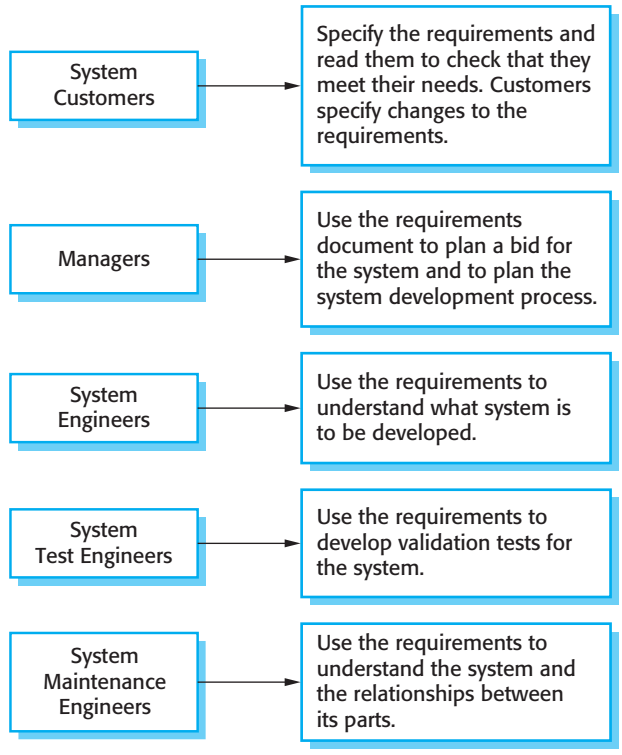


Figure 4.6 Users of a requirements document

The diversity of possible users means that the requirements document has to be a compromise between communicating the requirements to customers, defining the requirements in precise detail for developers and testers, and including information about possible system evolution. Information on anticipated changes can help system designers avoid restrictive design decisions and help system maintenance engineers who have to adapt the system to new requirements.

The level of detail that you should include in a requirements document depends on the type of system that is being developed and the development process used. Critical systems need to have detailed requirements because safety and security have to be analyzed in detail. When the system is to be developed by a separate company (e.g., through outsourcing), the system specifications need to be detailed and precise. If an in-house, iterative development process is used, the requirements document can be much less detailed and any ambiguities can be resolved during development of the system.

Figure 4.7 shows one possible organization for a requirements document that is based on an IEEE standard for requirements documents (IEEE, 1998). This standard is a generic standard that can be adapted to specific uses. In this case, I have extended the standard to include information about predicted system evolution. This information helps the maintainers of the system and allows designers to include support for future system features.

Naturally, the information that is included in a requirements document depends on the type of software being developed and the approach to development that is to be used. If an evolutionary approach is adopted for a software product (say), the

Chapter	Description
Preface	This should define the expected readership of the document and describe its version history, including a rationale for the creation of a new version and a summary of the changes made in each version.
Introduction	This should describe the need for the system. It should briefly describe the system's functions and explain how it will work with other systems. It should also describe how the system fits into the overall business or strategic objectives of the organization commissioning the software.
Glossary	This should define the technical terms used in the document. You should not make assumptions about the experience or expertise of the reader.
User requirements definition	Here, you describe the services provided for the user. The non-functional system requirements should also be described in this section. This description may use natural language, diagrams, or other notations that are understandable to customers. Product and process standards that must be followed should be specified.
System architecture	This chapter should present a high-level overview of the anticipated system architecture, showing the distribution of functions across system modules. Architectural components that are reused should be highlighted.
System requirements specification	This should describe the functional and non-functional requirements in more detail. If necessary, further detail may also be added to the non-functional requirements. Interfaces to other systems may be defined.
System models	This might include graphical system models showing the relationships between the system components, the system, and its environment. Examples of possible models are object models, data-flow models, or semantic data models.
System evolution	This should describe the fundamental assumptions on which the system is based, and any anticipated changes due to hardware evolution, changing user needs, and so on. This section is useful for system designers as it may help them avoid design decisions that would constrain likely future changes to the system.
Appendices	These should provide detailed, specific information that is related to the application being developed; for example, hardware and database descriptions. Hardware requirements define the minimal and optimal configurations for the system. Database requirements define the logical organization of the data used by the system and the relationships between data.
Index	Several indexes to the document may be included. As well as a normal alphabetic index, there may be an index of diagrams, an index of functions, and so on.

Figure 4.7 The structure of a requirements document

requirements document will leave out many of detailed chapters suggested above. The focus will be on defining the user requirements and high-level, non-functional system requirements. In this case, the designers and programmers use their judgment to decide how to meet the outline user requirements for the system.

However, when the software is part of a large system project that includes interacting hardware and software systems, it is usually necessary to define the requirements

Problems with using natural language for requirements specification

The flexibility of natural language, which is so useful for specification, often causes problems. There is scope for writing unclear requirements, and readers (the designers) may misinterpret requirements because they have a different background to the user. It is easy to amalgamate several requirements into a single sentence and structuring natural language requirements can be difficult.

http://www.SoftwareEngineering-9.com/Web/Requirements/NL-problems.html

to a fine level of detail. This means that the requirements documents are likely to be very long and should include most if not all of the chapters shown in Figure 4.7. For long documents, it is particularly important to include a comprehensive table of contents and document index so that readers can find the information that they need.

4.3 Requirements specification

Requirements specification is the process of writing down the user and system requirements in a requirements document. Ideally, the user and system requirements should be clear, unambiguous, easy to understand, complete, and consistent. In practice, this is difficult to achieve as stakeholders interpret the requirements in different ways and there are often inherent conflicts and inconsistencies in the requirements.

The user requirements for a system should describe the functional and nonfunctional requirements so that they are understandable by system users who don't have detailed technical knowledge. Ideally, they should specify only the external behavior of the system. The requirements document should not include details of the system architecture or design. Consequently, if you are writing user requirements, you should not use software jargon, structured notations, or formal notations. You should write user requirements in natural language, with simple tables, forms, and intuitive diagrams.

System requirements are expanded versions of the user requirements that are used by software engineers as the starting point for the system design. They add detail and explain how the user requirements should be provided by the system. They may be used as part of the contract for the implementation of the system and should therefore be a complete and detailed specification of the whole system.

Ideally, the system requirements should simply describe the external behavior of the system and its operational constraints. They should not be concerned with how the system should be designed or implemented. However, at the level of detail required to completely specify a complex software system, it is practically impossible to exclude all design information. There are several reasons for this:

1. You may have to design an initial architecture of the system to help structure the requirements specification. The system requirements are organized according to

Notation	Description
Natural language sentences	The requirements are written using numbered sentences in natural language. Each sentence should express one requirement.
Structured natural language	The requirements are written in natural language on a standard form or template. Each field provides information about an aspect of the requirement.
Design description languages	This approach uses a language like a programming language, but with more abstract features to specify the requirements by defining an operational model of the system. This approach is now rarely used although it can be useful for interface specifications.
Graphical notations	Graphical models, supplemented by text annotations, are used to define the functional requirements for the system; UML use case and sequence diagrams are commonly used.
Mathematical specifications	These notations are based on mathematical concepts such as finite-state machines or sets. Although these unambiguous specifications can reduce the ambiguity in a requirements document, most customers don't understand a formal specification. They cannot check that it represents what they want and are reluctant to accept it as a system contract.

Figure 4.8 Ways of writing a system requirements specification

the different sub-systems that make up the system. As I discuss in Chapters 6 and 18, this architectural definition is essential if you want to reuse software components when implementing the system.

2. In most cases, systems must interoperate with existing systems, which constrain the design and impose requirements on the new system.
3. The use of a specific architecture to satisfy non-functional requirements (such as N-version programming to achieve reliability, discussed in Chapter 13) may be necessary. An external regulator who needs to certify that the system is safe may specify that an already certified architectural design be used.

User requirements are almost always written in natural language supplemented by appropriate diagrams and tables in the requirements document. System requirements may also be written in natural language but other notations based on forms, graphical system models, or mathematical system models can also be used. Figure 4.8 summarizes the possible notations that could be used for writing system requirements.

Graphical models are most useful when you need to show how a state changes or when you need to describe a sequence of actions. UML sequence charts and state charts, described in Chapter 5, show the sequence of actions that occur in response to a certain message or event. Formal mathematical specifications are sometimes used to describe the requirements for safety- or security-critical systems, but are rarely used in other circumstances. I explain this approach to writing specifications in Chapter 12.

3.2 The system shall measure the blood sugar and deliver insulin, if required, every 10 minutes. (*Changes in blood sugar are relatively slow so more frequent measurement is unnecessary; less frequent measurement could lead to unnecessarily high sugar levels.*)

3.6 The system shall run a self-test routine every minute with the conditions to be tested and the associated actions defined in Table 1. (*A self-test routine can discover hardware and software problems and alert the user to the fact the normal operation may be impossible.*)

Figure 4.9 Example requirements for the insulin pump software system

4.3.1 Natural language specification

Natural language has been used to write requirements for software since the beginning of software engineering. It is expressive, intuitive, and universal. It is also potentially vague, ambiguous, and its meaning depends on the background of the reader. As a result, there have been many proposals for alternative ways to write requirements. However, none of these have been widely adopted and natural language will continue to be the most widely used way of specifying system and software requirements.

To minimize misunderstandings when writing natural language requirements, I recommend that you follow some simple guidelines:

1. Invent a standard format and ensure that all requirement definitions adhere to that format. Standardizing the format makes omissions less likely and requirements easier to check. The format I use expresses the requirement in a single sentence. I associate a statement of rationale with each user requirement to explain why the requirement has been proposed. The rationale may also include information on who proposed the requirement (the requirement source) so that you know whom to consult if the requirement has to be changed.
2. Use language consistently to distinguish between mandatory and desirable requirements. Mandatory requirements are requirements that the system must support and are usually written using 'shall'. Desirable requirements are not essential and are written using 'should'.
3. Use text highlighting (bold, italic, or color) to pick out key parts of the requirement.
4. Do not assume that readers understand technical software engineering language. It is easy for words like 'architecture' and 'module' to be misunderstood. You should, therefore, avoid the use of jargon, abbreviations, and acronyms.
5. Whenever possible, you should try to associate a rationale with each user requirement. The rationale should explain why the requirement has been included. It is particularly useful when requirements are changed as it may help decide what changes would be undesirable.

Figure 4.9 illustrates how these guidelines may be used. It includes two requirements for the embedded software for the automated insulin pump, introduced in Chapter 1. You can download the complete insulin pump requirements specification from the book's web pages.

Insulin Pump/Control Software/SRS/3.3.2

Function	Compute insulin dose: Safe sugar level.
Description	Computes the dose of insulin to be delivered when the current measured sugar level is in the safe zone between 3 and 7 units.
Inputs	Current sugar reading (r2), the previous two readings (r0 and r1).
Source	Current sugar reading from sensor. Other readings from memory.
Outputs	CompDose—the dose in insulin to be delivered.
Destination	Main control loop.
Action	CompDose is zero if the sugar level is stable or falling or if the level is increasing but the rate of increase is decreasing. If the level is increasing and the rate of increase is increasing, then CompDose is computed by dividing the difference between the current sugar level and the previous level by 4 and rounding the result. If the result, is rounded to zero then CompDose is set to the minimum dose that can be delivered.
Requirements	Two previous readings so that the rate of change of sugar level can be computed.
Pre-condition	The insulin reservoir contains at least the maximum allowed single dose of insulin.
Post-condition	r0 is replaced by r1 then r1 is replaced by r2.
Side effects	None.

Figure 4.10 A structured specification of a requirement for an insulin pump

4.3.2 Structured specifications

Structured natural language is a way of writing system requirements where the freedom of the requirements writer is limited and all requirements are written in a standard way. This approach maintains most of the expressiveness and understandability of natural language but ensures that some uniformity is imposed on the specification. Structured language notations use templates to specify system requirements. The specification may use programming language constructs to show alternatives and iteration, and may highlight key elements using shading or different fonts.

The Robertsons (Robertson and Robertson, 1999), in their book on the VOLERE requirements engineering method, recommend that user requirements be initially written on cards, one requirement per card. They suggest a number of fields on each card, such as the requirements rationale, the dependencies on other requirements, the source of the requirements, supporting materials, and so on. This is similar to the approach used in the example of a structured specification shown in Figure 4.10.

To use a structured approach to specifying system requirements, you define one or more standard templates for requirements and represent these templates as structured forms. The specification may be structured around the objects manipulated by the system, the functions performed by the system, or the events processed by the system. An example of a form-based specification, in this case, one that defines how to calculate the dose of insulin to be delivered when the blood sugar is within a safe band, is shown in Figure 4.10.

Condition	Action
Sugar level falling (r2 < r1)	CompDose = 0
Sugar level stable (r2 = r1)	CompDose = 0
Sugar level increasing and rate of increase decreasing ((r2 − r1) < (r1 − r0))	CompDose = 0
Sugar level increasing and rate of increase stable or increasing ((r2 − r1) ≥ (r1 − r0))	CompDose = round ((r2 − r1)/4) If rounded result = 0 then CompDose = MinimumDose

Figure 4.11 Tabular specification of computation for an insulin pump

When a standard form is used for specifying functional requirements, the following information should be included:

1. A description of the function or entity being specified.
2. A description of its inputs and where these come from.
3. A description of its outputs and where these go to.
4. Information about the information that is needed for the computation or other entities in the system that are used (the 'requires' part).
5. A description of the action to be taken.
6. If a functional approach is used, a pre-condition setting out what must be true before the function is called, and a post-condition specifying what is true after the function is called.
7. A description of the side effects (if any) of the operation.

Using structured specifications removes some of the problems of natural language specification. Variability in the specification is reduced and requirements are organized more effectively. However, it is still sometimes difficult to write requirements in a clear and unambiguous way, particularly when complex computations (e.g., how to calculate the insulin dose) are to be specified.

To address this problem, you can add extra information to natural language requirements, for example, by using tables or graphical models of the system. These can show how computations proceed, how the system state changes, how users interact with the system, and how sequences of actions are performed.

Tables are particularly useful when there are a number of possible alternative situations and you need to describe the actions to be taken for each of these. The insulin pump bases its computations of the insulin requirement on the rate of change of blood sugar levels. The rates of change are computed using the current and previous readings. Figure 4.11 is a tabular description of how the rate of change of blood sugar is used to calculate the amount of insulin to be delivered.

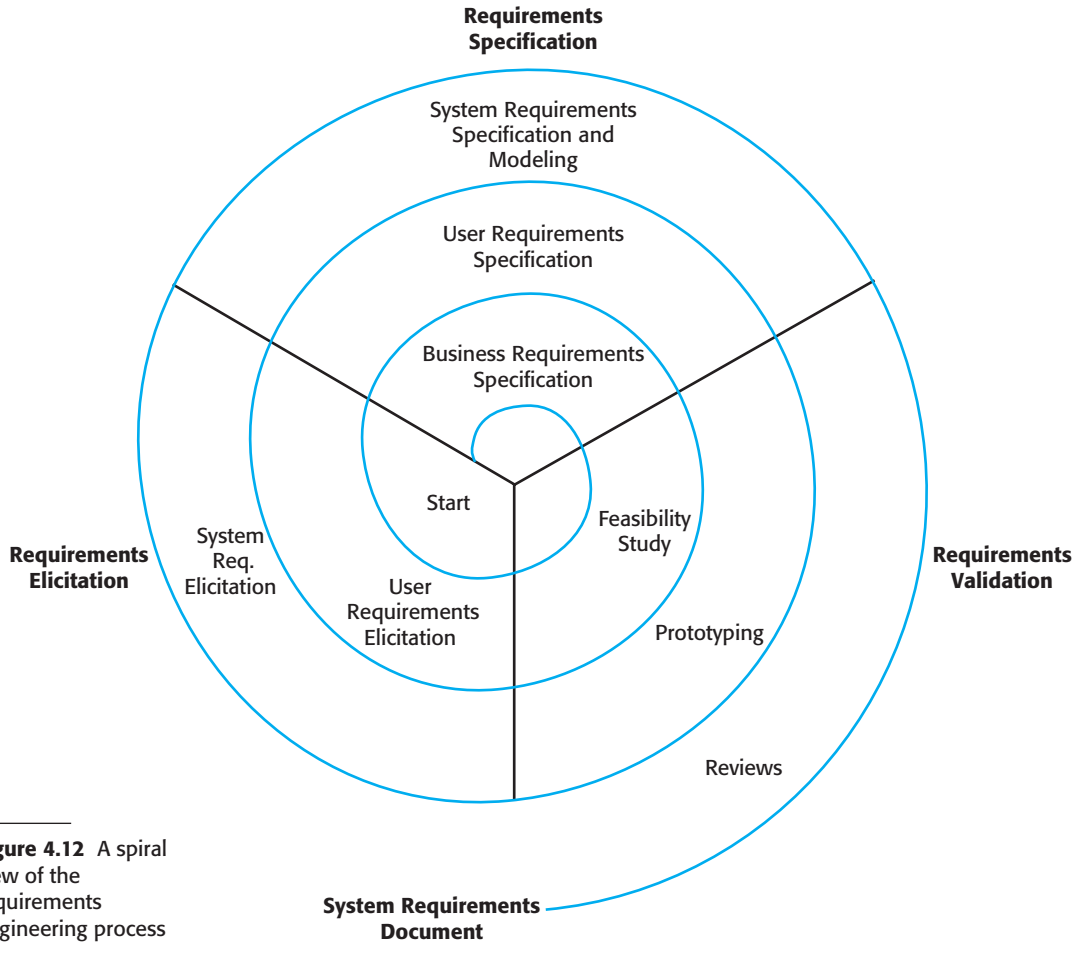


Figure 4.12 A spiral view of the requirements engineering process

4.4 Requirements engineering processes

As I discussed in Chapter 2, requirements engineering processes may include four high-level activities. These focus on assessing if the system is useful to the business (feasibility study), discovering requirements (elicitation and analysis), converting these requirements into some standard form (specification), and checking that the requirements actually define the system that the customer wants (validation). I have shown these as sequential processes in Figure 2.6. However, in practice, requirements engineering is an iterative process in which the activities are interleaved.

Figure 4.12 shows this interleaving. The activities are organized as an iterative process around a spiral, with the output being a system requirements document. The amount of time and effort devoted to each activity in each iteration depends on the stage of the overall process and the type of system being developed. Early in the process, most effort will be spent on understanding high-level business and

Feasibility studies

A feasibility study is a short, focused study that should take place early in the RE process. It should answer three key questions: a) does the system contribute to the overall objectives of the organization? b) can the system be implemented within schedule and budget using current technology? and c) can the system be integrated with other systems that are used?

If the answer to any of these questions is no, you should probably not go ahead with the project.

http://www.SoftwareEngineering-9.com/Web/Requirements/FeasibilityStudy.html

non-functional requirements, and the user requirements for the system. Later in the process, in the outer rings of the spiral, more effort will be devoted to eliciting and understanding the detailed system requirements.

This spiral model accommodates approaches to development where the requirements are developed to different levels of detail. The number of iterations around the spiral can vary so the spiral can be exited after some or all of the user requirements have been elicited. Agile development can be used instead of prototyping so that the requirements and the system implementation are developed together.

Some people consider requirements engineering to be the process of applying a structured analysis method, such as object-oriented analysis (Larman, 2002). This involves analyzing the system and developing a set of graphical system models, such as use case models, which then serve as a system specification. The set of models describes the behavior of the system and is annotated with additional information describing, for example, the system's required performance or reliability.

Although structured methods have a role to play in the requirements engineering process, there is much more to requirements engineering than is covered by these methods. Requirements elicitation, in particular, is a human-centered activity and people dislike the constraints imposed on it by rigid system models.

In virtually all systems, requirements change. The people involved develop a better understanding of what they want the software to do; the organization buying the system changes; modifications are made to the system's hardware, software, and organizational environment. The process of managing these changing requirements is called requirements management, which I cover in Section 4.7.

4.5 Requirements elicitation and analysis

After an initial feasibility study, the next stage of the requirements engineering process is requirements elicitation and analysis. In this activity, software engineers work with customers and system end-users to find out about the application domain, what services the system should provide, the required performance of the system, hardware constraints, and so on.

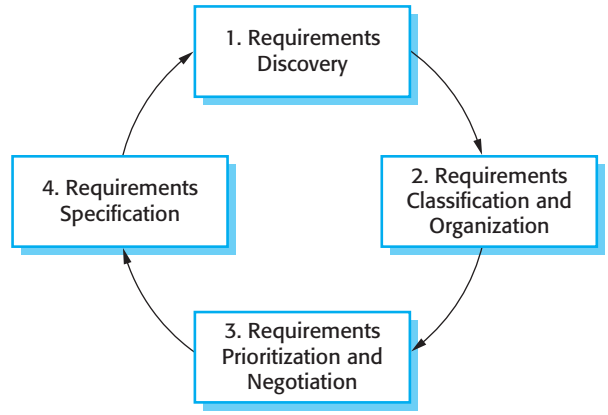


Figure 4.13 The requirements elicitation and analysis process

Requirements elicitation and analysis may involve a variety of different kinds of people in an organization. A system stakeholder is anyone who should have some direct or indirect influence on the system requirements. Stakeholders include end-users who will interact with the system and anyone else in an organization who will be affected by it. Other system stakeholders might be engineers who are developing or maintaining other related systems, business managers, domain experts, and trade union representatives.

A process model of the elicitation and analysis process is shown in Figure 4.13. Each organization will have its own version or instantiation of this general model depending on local factors such as the expertise of the staff, the type of system being developed, the standards used, etc.

The process activities are:

1. *Requirements discovery* This is the process of interacting with stakeholders of the system to discover their requirements. Domain requirements from stakeholders and documentation are also discovered during this activity. There are several complementary techniques that can be used for requirements discovery, which I discuss later in this section.

2. *Requirements classification and organization* This activity takes the unstructured collection of requirements, groups related requirements, and organizes them into coherent clusters. The most common way of grouping requirements is to use a model of the system architecture to identify sub-systems and to associate requirements with each sub-system. In practice, requirements engineering and architectural design cannot be completely separate activities.

3. *Requirements prioritization and negotiation* Inevitably, when multiple stakeholders are involved, requirements will conflict. This activity is concerned with prioritizing requirements and finding and resolving requirements conflicts through negotiation. Usually, stakeholders have to meet to resolve differences and agree on compromise requirements.

4. *Requirements specification* The requirements are documented and input into the next round of the spiral. Formal or informal requirements documents may be produced, as discussed in Section 4.3.

Figure 4.13 shows that requirements elicitation and analysis is an iterative process with continual feedback from each activity to other activities. The process cycle starts with requirements discovery and ends with the requirements documentation. The analyst's understanding of the requirements improves with each round of the cycle. The cycle ends when the requirements document is complete.

Eliciting and understanding requirements from system stakeholders is a difficult process for several reasons:

1. Stakeholders often don't know what they want from a computer system except in the most general terms; they may find it difficult to articulate what they want the system to do; they may make unrealistic demands because they don't know what is and isn't feasible.
2. Stakeholders in a system naturally express requirements in their own terms and with implicit knowledge of their own work. Requirements engineers, without experience in the customer's domain, may not understand these requirements.
3. Different stakeholders have different requirements and they may express these in different ways. Requirements engineers have to discover all potential sources of requirements and discover commonalities and conflict.
4. Political factors may influence the requirements of a system. Managers may demand specific system requirements because these will allow them to increase their influence in the organization.
5. The economic and business environment in which the analysis takes place is dynamic. It inevitably changes during the analysis process. The importance of particular requirements may change. New requirements may emerge from new stakeholders who were not originally consulted.

Inevitably, different stakeholders have different views on the importance and priority of requirements and, sometimes, these views are conflicting. During the process, you should organize regular stakeholder negotiations so that compromises can be reached. It is impossible to completely satisfy every stakeholder but if some stakeholders feel that their views have not been properly considered then they may deliberately attempt to undermine the RE process.

At the requirements specification stage, the requirements that have been elicited so far are documented in such a way that they can be used to help with requirements discovery. At this stage, an early version of the system requirements document may be produced with missing sections and incomplete requirements. Alternatively, the requirements may be documented in a completely different way (e.g., in a spreadsheet or on cards). Writing requirements on cards can be very effective as these are easy for stakeholders to handle, change, and organize.

Viewpoints

A viewpoint is way of collecting and organizing a set of requirements from a group of stakeholders who have something in common. Each viewpoint therefore includes a set of system requirements. Viewpoints might come from end-users, managers, etc. They help identify the people who can provide information about their requirements and structure the requirements for analysis.

http://www.SoftwareEngineering-9.com/Web/Requirements/Viewpoints.html

4.5.1 Requirements discovery

Requirements discovery (sometime called requirements elicitation) is the process of gathering information about the required system and existing systems, and distilling the user and system requirements from this information. Sources of information during the requirements discovery phase include documentation, system stakeholders, and specifications of similar systems. You interact with stakeholders through interviews and observation and you may use scenarios and prototypes to help stakeholders understand what the system will be like.

Stakeholders range from end-users of a system through managers to external stakeholders such as regulators, who certify the acceptability of the system. For example, system stakeholders for the mental healthcare patient information system include:

1. Patients whose information is recorded in the system.
2. Doctors who are responsible for assessing and treating patients.
3. Nurses who coordinate the consultations with doctors and administer some treatments.
4. Medical receptionists who manage patients' appointments.
5. IT staff who are responsible for installing and maintaining the system.
6. A medical ethics manager who must ensure that the system meets current ethical guidelines for patient care.
7. Healthcare managers who obtain management information from the system.
8. Medical records staff who are responsible for ensuring that system information can be maintained and preserved, and that record keeping procedures have been properly implemented.

In addition to system stakeholders, we have already seen that requirements may also come from the application domain and from other systems that interact with the system being specified. All of these must be considered during the requirements elicitation process.

These different requirements sources (stakeholders, domain, systems) can all be represented as system viewpoints with each viewpoint showing a subset of the

requirements for the system. Different viewpoints on a problem see the problem in different ways. However, their perspectives are not completely independent but usually overlap so that they have common requirements. You can use these viewpoints to structure both the discovery and the documentation of the system requirements.

4.5.2 Interviewing

Formal or informal interviews with system stakeholders are part of most requirements engineering processes. In these interviews, the requirements engineering team puts questions to stakeholders about the system that they currently use and the system to be developed. Requirements are derived from the answers to these questions. Interviews may be of two types:

1. Closed interviews, where the stakeholder answers a pre-defined set of questions.
2. Open interviews, in which there is no pre-defined agenda. The requirements engineering team explores a range of issues with system stakeholders and hence develop a better understanding of their needs.

In practice, interviews with stakeholders are normally a mixture of both of these. You may have to obtain the answer to certain questions but these usually lead on to other issues that are discussed in a less structured way. Completely open-ended discussions rarely work well. You usually have to ask some questions to get started and to keep the interview focused on the system to be developed.

Interviews are good for getting an overall understanding of what stakeholders do, how they might interact with the new system, and the difficulties that they face with current systems. People like talking about their work so are usually happy to get involved in interviews. However, interviews are not so helpful in understanding the requirements from the application domain.

It can be difficult to elicit domain knowledge through interviews for two reasons:

1. All application specialists use terminology and jargon that are specific to a domain. It is impossible for them to discuss domain requirements without using this terminology. They normally use terminology in a precise and subtle way that is easy for requirements engineers to misunderstand.
2. Some domain knowledge is so familiar to stakeholders that they either find it difficult to explain or they think it is so fundamental that it isn't worth mentioning. For example, for a librarian, it goes without saying that all acquisitions are catalogued before they are added to the library. However, this may not be obvious to the interviewer, and so it isn't taken into account in the requirements.

Interviews are also not an effective technique for eliciting knowledge about organizational requirements and constraints because there are subtle power relationships between the different people in the organization. Published organizational structures

rarely match the reality of decision making in an organization but interviewees may not wish to reveal the actual rather than the theoretical structure to a stranger. In general, most people are generally reluctant to discuss political and organizational issues that may affect the requirements.

Effective interviewers have two characteristics:

1. They are open-minded, avoid pre-conceived ideas about the requirements, and are willing to listen to stakeholders. If the stakeholder comes up with surprising requirements, then they are willing to change their mind about the system.
2. They prompt the interviewee to get discussions going using a springboard question, a requirements proposal, or by working together on a prototype system. Saying to people 'tell me what you want' is unlikely to result in useful information. They find it much easier to talk in a defined context rather than in general terms.

Information from interviews supplements other information about the system from documentation describing business processes or existing systems, user observations, etc. Sometimes, apart from the information in the system documents, the interview information may be the only source of information about the system requirements. However, interviewing on its own is liable to miss essential information and so it should be used in conjunction with other requirements elicitation techniques.

4.5.3 Scenarios

People usually find it easier to relate to real-life examples rather than abstract descriptions. They can understand and criticize a scenario of how they might interact with a software system. Requirements engineers can use the information gained from this discussion to formulate the actual system requirements.

Scenarios can be particularly useful for adding detail to an outline requirements description. They are descriptions of example interaction sessions. Each scenario usually covers one or a small number of possible interactions. Different forms of scenarios are developed and they provide different types of information at different levels of detail about the system. The stories used in extreme programming, discussed in Chapter 3, are a type of requirements scenario.

A scenario starts with an outline of the interaction. During the elicitation process, details are added to this to create a complete description of that interaction. At its most general, a scenario may include:

1. A description of what the system and users expects when the scenario starts.
2. A description of the normal flow of events in the scenario.
3. A description of what can go wrong and how this is handled.
4. Information about other activities that might be going on at the same time.
5. A description of the system state when the scenario finishes.

INITIAL ASSUMPTION:
The patient has seen a medical receptionist who has created a record in the system and collected the patient's personal information (name, address, age, etc.). A nurse is logged on to the system and is collecting medical history.

NORMAL:
The nurse searches for the patient by family name. If there is more than one patient with the same surname, the given name (first name in English) and date of birth are used to identify the patient.

The nurse chooses the menu option to add medical history.

The nurse then follows a series of prompts from the system to enter information about consultations elsewhere on mental health problems (free text input), existing medical conditions (nurse selects conditions from menu), medication currently taken (selected from menu), allergies (free text), and home life (form).

WHAT CAN GO WRONG:
The patient's record does not exist or cannot be found. The nurse should create a new record and record personal information.

Patient conditions or medication are not entered in the menu. The nurse should choose the 'other' option and enter free text describing the condition/medication.

Patient cannot/will not provide information on medical history. The nurse should enter free text recording the patient's inability/unwillingness to provide information. The system should print the standard exclusion form stating that the lack of information may mean that treatment will be limited or delayed. This should be signed and handed to the patient.

OTHER ACTIVITIES:
Record may be consulted but not edited by other staff while information is being entered.

SYSTEM STATE ON COMPLETION:
User is logged on. The patient record including medical history is entered in the database, a record is added to the system log showing the start and end time of the session and the nurse involved.

Figure 4.14 Scenario for collecting medical history in MHC-PMS

Scenario-based elicitation involves working with stakeholders to identify scenarios and to capture details to be included in these scenarios. Scenarios may be written as text, supplemented by diagrams, screen shots, etc. Alternatively, a more structured approach such as event scenarios or use cases may be used.

As an example of a simple text scenario, consider how the MHC-PMS may be used to enter data for a new patient (Figure 4.14). When a new patient attends a clinic, a new record is created by a medical receptionist and personal information (name, age, etc.) is added to it. A nurse then interviews the patient and collects medical history. The patient then has an initial consultation with a doctor who makes a diagnosis and, if appropriate, recommends a course of treatment. The scenario shows what happens when medical history is collected.

4.5.4 Use cases

Use cases are a requirements discovery technique that were first introduced in the Objectory method (Jacobson et al., 1993). They have now become a fundamental feature of the unified modeling language. In their simplest form, a use case identifies

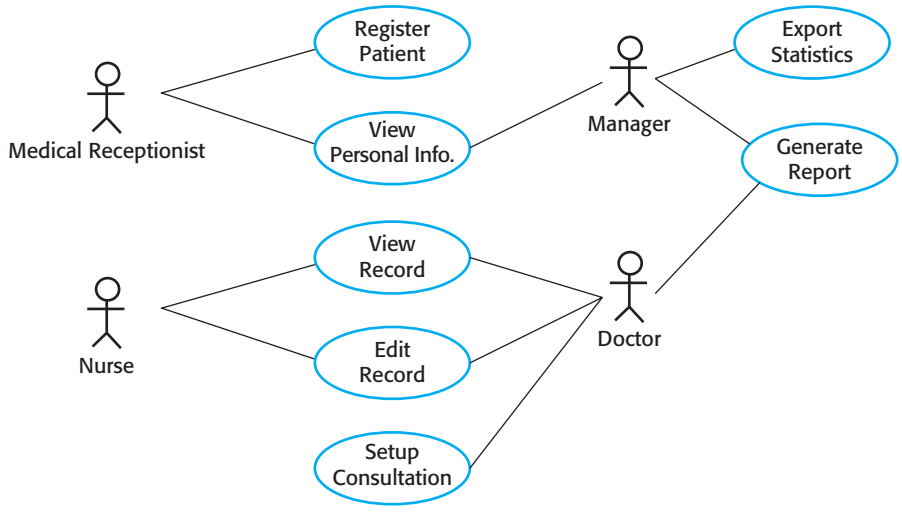


Figure 4.15 Use cases for the MHC-PMS

the actors involved in an interaction and names the type of interaction. This is then supplemented by additional information describing the interaction with the system. The additional information may be a textual description or one or more graphical models such as UML sequence or state charts.

Use cases are documented using a high-level use case diagram. The set of use cases represents all of the possible interactions that will be described in the system requirements. Actors in the process, who may be human or other systems, are represented as stick figures. Each class of interaction is represented as a named ellipse. Lines link the actors with the interaction. Optionally, arrowheads may be added to lines to show how the interaction is initiated. This is illustrated in Figure 4.15, which shows some of the use cases for the patient information system.

There is no hard and fast distinction between scenarios and use cases. Some people consider that each use case is a single scenario; others, as suggested by Stevens and Pooley (2006), encapsulate a set of scenarios in a single use case. Each scenario is a single thread through the use case. Therefore, there would be a scenario for the normal interaction plus scenarios for each possible exception. You can, in practice, use them in either way.

Use cases identify the individual interactions between the system and its users or other systems. Each use case should be documented with a textual description. These can then be linked to other models in the UML that will develop the scenario in more detail. For example, a brief description of the Setup Consultation use case from Figure 4.15 might be:

> *Setup consultation allows two or more doctors, working in different offices, to view the same record at the same time. One doctor initiates the consultation by choosing the people involved from a drop-down menu of doctors who are online. The patient record is then displayed on their screens but only the initiating doctor can edit the record. In addition, a text chat window is created to help*

coordinate actions. It is assumed that a phone conference for voice communication will be separately set up.

Scenarios and use cases are effective techniques for eliciting requirements from stakeholders who interact directly with the system. Each type of interaction can be represented as a use case. However, because they focus on interactions with the system, they are not as effective for eliciting constraints or high-level business and nonfunctional requirements or for discovering domain requirements.

The UML is a de facto standard for object-oriented modeling, so use cases and use case–based elicitation are now widely used for requirements elicitation. I discuss use cases further in Chapter 5 and show how they are used alongside other system models to document a system design.

4.5.5 Ethnography

Software systems do not exist in isolation. They are used in a social and organizational context and software system requirements may be derived or constrained by that context. Satisfying these social and organizational requirements is often critical for the success of the system. One reason why many software systems are delivered but never used is that their requirements do not take proper account of how the social and organizational context affects the practical operation of the system.

Ethnography is an observational technique that can be used to understand operational processes and help derive support requirements for these processes. An analyst immerses himself or herself in the working environment where the system will be used. The day-to-day work is observed and notes made of the actual tasks in which participants are involved. The value of ethnography is that it helps discover implicit system requirements that reflect the actual ways that people work, rather than the formal processes defined by the organization.

People often find it very difficult to articulate details of their work because it is second nature to them. They understand their own work but may not understand its relationship to other work in the organization. Social and organizational factors that affect the work, but which are not obvious to individuals, may only become clear when noticed by an unbiased observer. For example, a work group may self-organize so that members know of each other's work and can cover for each other if someone is absent. This may not be mentioned during an interview as the group might not see it as an integral part of their work.

Suchman (1987) pioneered the use of ethnography to study office work. She found that the actual work practices were far richer, more complex, and more dynamic than the simple models assumed by office automation systems. The difference between the assumed and the actual work was the most important reason why these office systems had no significant effect on productivity. Crabtree (2003) discusses a wide range of studies since then and describes, in general, the use of ethnography in systems design. In my own research, I have investigated methods of

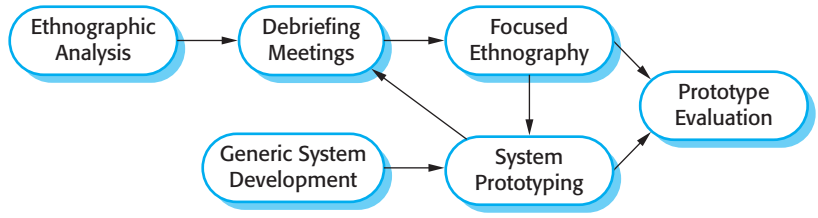


Figure 4.16 Ethnography and prototyping for requirements analysis

integrating ethnography into the software engineering process by linking it with requirements engineering methods (Viller and Sommerville, 1999; Viller and Sommerville, 2000) and documenting patterns of interaction in cooperative systems (Martin et al., 2001; Martin et al., 2002; Martin and Sommerville, 2004).

Ethnography is particularly effective for discovering two types of requirements:

1. Requirements that are derived from the way in which people actually work, rather than the way in which process definitions say they ought to work. For example, air traffic controllers may switch off a conflict alert system that detects aircraft with intersecting flight paths, even though normal control procedures specify that it should be used. They deliberately put the aircraft on conflicting paths for a short time to help manage the airspace. Their control strategy is designed to ensure that these aircrafts are moved apart before problems occur and they find that the conflict alert alarm distracts them from their work.

2. Requirements that are derived from cooperation and awareness of other people's activities. For example, air traffic controllers may use an awareness of other controllers' work to predict the number of aircrafts that will be entering their control sector. They then modify their control strategies depending on that predicted workload. Therefore, an automated ATC system should allow controllers in a sector to have some visibility of the work in adjacent sectors.

Ethnography can be combined with prototyping (Figure 4.16). The ethnography informs the development of the prototype so that fewer prototype refinement cycles are required. Furthermore, the prototyping focuses the ethnography by identifying problems and questions that can then be discussed with the ethnographer. He or she should then look for the answers to these questions during the next phase of the system study (Sommerville et al., 1993).

Ethnographic studies can reveal critical process details that are often missed by other requirements elicitation techniques. However, because of its focus on the end-user, this approach is not always appropriate for discovering organizational or domain requirements. They cannot always identify new features that should be added to a system. Ethnography is not, therefore, a complete approach to elicitation on its own and it should be used to complement other approaches, such as use case analysis.

Requirements reviews

A requirements review is a process where a group of people from the system customer and the system developer read the requirements document in detail and check for errors, anomalies, and inconsistencies. Once these have been detected and recorded, it is then up to the customer and the developer to negotiate how the identified problems should be solved.

http://www.SoftwareEngineering-9.com/Web/Requirements/Reviews.html

4.6 Requirements validation

Requirements validation is the process of checking that requirements actually define the system that the customer really wants. It overlaps with analysis as it is concerned with finding problems with the requirements. Requirements validation is important because errors in a requirements document can lead to extensive rework costs when these problems are discovered during development or after the system is in service.

The cost of fixing a requirements problem by making a system change is usually much greater than repairing design or coding errors. The reason for this is that a change to the requirements usually means that the system design and implementation must also be changed. Furthermore the system must then be re-tested.

During the requirements validation process, different types of checks should be carried out on the requirements in the requirements document. These checks include:

1. *Validity checks* A user may think that a system is needed to perform certain functions. However, further thought and analysis may identify additional or different functions that are required. Systems have diverse stakeholders with different needs and any set of requirements is inevitably a compromise across the stakeholder community.
2. *Consistency checks* Requirements in the document should not conflict. That is, there should not be contradictory constraints or different descriptions of the same system function.
3. *Completeness checks* The requirements document should include requirements that define all functions and the constraints intended by the system user.
4. *Realism checks* Using knowledge of existing technology, the requirements should be checked to ensure that they can actually be implemented. These checks should also take account of the budget and schedule for the system development.
5. *Verifiability* To reduce the potential for dispute between customer and contractor, system requirements should always be written so that they are verifiable. This means that you should be able to write a set of tests that can demonstrate that the delivered system meets each specified requirement.

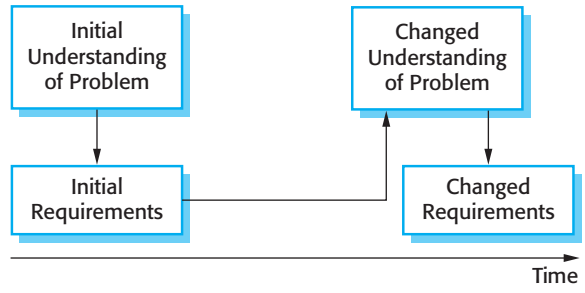


Figure 4.17 Requirements evolution

There are a number of requirements validation techniques that can be used individually or in conjunction with one another:

1. *Requirements reviews* The requirements are analyzed systematically by a team of reviewers who check for errors and inconsistencies.
2. *Prototyping* In this approach to validation, an executable model of the system in question is demonstrated to end-users and customers. They can experiment with this model to see if it meets their real needs.
3. *Test-case generation* Requirements should be testable. If the tests for the requirements are devised as part of the validation process, this often reveals requirements problems. If a test is difficult or impossible to design, this usually means that the requirements will be difficult to implement and should be reconsidered. Developing tests from the user requirements before any code is written is an integral part of extreme programming.

You should not underestimate the problems involved in requirements validation. Ultimately, it is difficult to show that a set of requirements does in fact meet a user's needs. Users need to picture the system in operation and imagine how that system would fit into their work. It is hard even for skilled computer professionals to perform this type of abstract analysis and harder still for system users. As a result, you rarely find all requirements problems during the requirements validation process. It is inevitable that there will be further requirements changes to correct omissions and misunderstandings after the requirements document has been agreed upon.

4.7 Requirements management

The requirements for large software systems are always changing. One reason for this is that these systems are usually developed to address 'wicked' problems—problems that cannot be completely defined. Because the problem cannot be fully defined, the software requirements are bound to be incomplete. During the software process, the stakeholders' understanding of the problem is constantly changing (Figure 4.17). The system requirements must then also evolve to reflect this changed problem view.

Enduring and volatile requirements

Some requirements are more susceptible to change than others. Enduring requirements are the requirements that are associated with the core, slow-to-change activities of an organization. Enduring requirements are associated with fundamental work activities. Volatile requirements are more likely to change. They are usually associated with supporting activities that reflect how the organization does its work rather than the work itself.

http://www.SoftwareEngineering-9.com/Web/Requirements/EnduringReq.html

Once a system has been installed and is regularly used, new requirements inevitably emerge. It is hard for users and system customers to anticipate what effects the new system will have on their business processes and the way that work is done. Once end-users have experience of a system, they will discover new needs and priorities. There are several reasons why change is inevitable:

1. The business and technical environment of the system always changes after installation. New hardware may be introduced, it may be necessary to interface the system with other systems, business priorities may change (with consequent changes in the system support required), and new legislation and regulations may be introduced that the system must necessarily abide by.
2. The people who pay for a system and the users of that system are rarely the same people. System customers impose requirements because of organizational and budgetary constraints. These may conflict with end-user requirements and, after delivery, new features may have to be added for user support if the system is to meet its goals.
3. Large systems usually have a diverse user community, with many users having different requirements and priorities that may be conflicting or contradictory. The final system requirements are inevitably a compromise between them and, with experience, it is often discovered that the balance of support given to different users has to be changed.

Requirements management is the process of understanding and controlling changes to system requirements. You need to keep track of individual requirements and maintain links between dependent requirements so that you can assess the impact of requirements changes. You need to establish a formal process for making change proposals and linking these to system requirements. The formal process of requirements management should start as soon as a draft version of the requirements document is available. However, you should start planning how to manage changing requirements during the requirements elicitation process.

4.7.1 Requirements management planning

Planning is an essential first stage in the requirements management process. The planning stage establishes the level of requirements management detail that is required. During the requirements management stage, you have to decide on:

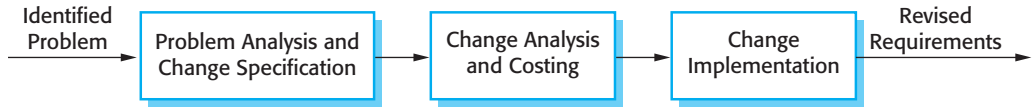


Figure 4.18 Requirements change management

1. *Requirements identification* Each requirement must be uniquely identified so that it can be cross-referenced with other requirements and used in traceability assessments.
2. *A change management process* This is the set of activities that assess the impact and cost of changes. I discuss this process in more detail in the following section.
3. *Traceability policies* These policies define the relationships between each requirement and between the requirements and the system design that should be recorded. The traceability policy should also define how these records should be maintained.
4. *Tool support* Requirements management involves the processing of large amounts of information about the requirements. Tools that may be used range from specialist requirements management systems to spreadsheets and simple database systems.

Requirements management needs automated support and the software tools for this should be chosen during the planning phase. You need tool support for:

1. *Requirements storage* The requirements should be maintained in a secure, managed data store that is accessible to everyone involved in the requirements engineering process.
2. *Change management* The process of change management (Figure 4.18) is simplified if active tool support is available.
3. *Traceability management* As discussed above, tool support for traceability allows related requirements to be discovered. Some tools are available which use natural language processing techniques to help discover possible relationships between requirements.

For small systems, it may not be necessary to use specialized requirements management tools. The requirements management process may be supported using the facilities available in word processors, spreadsheets, and PC databases. However, for larger systems, more specialized tool support is required. I have included links to information about requirements management tools in the book's web pages.

4.7.2 Requirements change management

Requirements change management (Figure 4.18) should be applied to all proposed changes to a system's requirements after the requirements document has been approved. Change management is essential because you need to decide if the benefits of implementing new requirements are justified by the costs of implementation. The advantage of

Requirements tracea

You need to keep track of the relationships between requirements, their sources, and the system design so that you can analyze the reasons for proposed changes and the impact that these changes are likely to have on other parts of the system. You need to be able to trace how a change ripples its way through the system. Why?

http://www.SoftwareEngineering-9.com/Web/Requirements/ReqTraceability.html

using a formal process for change management is that all change proposals are treated consistently and changes to the requirements document are made in a controlled way.

There are three principal stages to a change management process:

1. *Problem analysis and change specification* The process starts with an identified requirements problem or, sometimes, with a specific change proposal. During this stage, the problem or the change proposal is analyzed to check that it is valid. This analysis is fed back to the change requestor who may respond with a more specific requirements change proposal, or decide to withdraw the request.
2. *Change analysis and costing* The effect of the proposed change is assessed using traceability information and general knowledge of the system requirements. The cost of making the change is estimated both in terms of modifications to the requirements document and, if appropriate, to the system design and implementation. Once this analysis is completed, a decision is made whether or not to proceed with the requirements change.
3. *Change implementation* The requirements document and, where necessary, the system design and implementation, are modified. You should organize the requirements document so that you can make changes to it without extensive rewriting or reorganization. As with programs, changeability in documents is achieved by minimizing external references and making the document sections as modular as possible. Thus, individual sections can be changed and replaced without affecting other parts of the document.

If a new requirement has to be urgently implemented, there is always a temptation to change the system and then retrospectively modify the requirements document. You should try to avoid this as it almost inevitably leads to the requirements specification and the system implementation getting out of step. Once system changes have been made, it is easy to forget to include these changes in the requirements document or to add information to the requirements document that is inconsistent with the implementation.

Agile development processes, such as extreme programming, have been designed to cope with requirements that change during the development process. In these processes, when a user proposes a requirements change, this change does not go through a formal change management process. Rather, the user has to prioritize that change and, if it is high priority, decide what system features that were planned for the next iteration should be dropped.

REFERENCES

Beck, K. (1999). 'Embracing Change with Extreme Programming'. *IEEE Computer*, **32** (10), 70–8.

Crabtree, A. (2003). *Designing Collaborative Systems: A Practical Guide to Ethnography.* London: Springer-Verlag.

Davis, A. M. (1993). *Software Requirements: Objects, Functions and States.* Englewood Cliffs, NJ: Prentice Hall.

IEEE. (1998). 'IEEE Recommended Practice for Software Requirements Specifications'. In *IEEE Software Engineering Standards Collection.* Los Alamitos, Ca.: IEEE Computer Society Press.

Jacobson, I., Christerson, M., Jonsson, P. and Overgaard, G. (1993). *Object-Oriented Software Engineering.* Wokingham: Addison-Wesley.

Kotonya, G. and Sommerville, I. (1998). *Requirements Engineering: Processes and Techniques.* Chichester, UK: John Wiley and Sons.

Larman, C. (2002). *Applying UML and Patterns: An Introduction to Object-oriented Analysis and Design and the Unified Process.* Englewood Cliff, NJ: Prentice Hall.

Martin, D., Rodden, T., Rouncefield, M., Sommerville, I. and Viller, S. (2001). 'Finding Patterns in the Fieldwork'. *Proc. ECSCW'01.* Bonn: Kluwer. 39–58.

Martin, D., Rouncefield, M. and Sommerville, I. (2002). 'Applying patterns of interaction to work (re)design: E-government and planning'. *Proc. ACM CHI'2002,* ACM Press. 235–42.

Martin, D. and Sommerville, I. (2004). 'Patterns of interaction: Linking ethnomethodology and design'. *ACM Trans. on Computer-Human Interaction,* **11** (1), 59–89.

Robertson, S. and Robertson, J. (1999). *Mastering the Requirements Process.* Harlow, UK: Addison-Wesley.

Sommerville, I., Rodden, T., Sawyer, P., Bentley, R. and Twidale, M. (1993). 'Integrating ethnography into the requirements engineering process'. *Proc. RE'93,* San Diego CA.: IEEE Computer Society Press. 165–73.

Stevens, P. and Pooley, R. (2006). *Using UML: Software Engineering with Objects and Components, 2nd ed.* Harlow, UK: Addison Wesley.

Suchman, L. (1987). *Plans and Situated Actions.* Cambridge: Cambridge University Press.

Viller, S. and Sommerville, I. (1999). 'Coherence: An Approach to Representing Ethnographic Analyses in Systems Design'. *Human-Computer Interaction,* **14** (1 & 2), 9–41.

Viller, S. and Sommerville, I. (2000). 'Ethnographically informed analysis for software engineers'. *Int. J. of Human-Computer Studies,* **53** (1), 169–96.

5

System modeling

Objectives

The aim of this chapter is to introduce some types of system model that may be developed as part of the requirements engineering and system design processes. When you have read the chapter, you will:

- understand how graphical models can be used to represent software systems;
- understand why different types of model are required and the fundamental system modeling perspectives of context, interaction, structure, and behavior;
- have been introduced to some of the diagram types in the Unified Modeling Language (UML) and how these diagrams may be used in system modeling;
- be aware of the ideas underlying model-driven engineering, where a system is automatically generated from structural and behavioral models.

Contents

System modeling is the process of developing abstract models of a system, with each model presenting a different view or perspective of that system. System modeling has generally come to mean representing the system using some kind of graphical notation, which is now almost always based on notations in the Unified Modeling Language (UML). However, it is also possible to develop formal (mathematical) models of a system, usually as a detailed system specification. I cover graphical modeling using the UML in this chapter and formal modeling in Chapter 12.

Models are used during the requirements engineering process to help derive the requirements for a system, during the design process to describe the system to engineers implementing the system and after implementation to document the system's structure and operation. You may develop models of both the existing system and the system to be developed:

1. Models of the existing system are used during requirements engineering. They help clarify what the existing system does and can be used as a basis for discussing its strengths and weaknesses. These then lead to requirements for the new system.
2. Models of the new system are used during requirements engineering to help explain the proposed requirements to other system stakeholders. Engineers use these models to discuss design proposals and to document the system for implementation. In a model-driven engineering process, it is possible to generate a complete or partial system implementation from the system model.

The most important aspect of a system model is that it leaves out detail. A model is an abstraction of the system being studied rather than an alternative representation of that system. Ideally, a representation of a system should maintain all the information about the entity being represented. An abstraction deliberately simplifies and picks out the most salient characteristics. For example, in the very unlikely event of this book being serialized in a newspaper, the presentation there would be an abstraction of the book's key points. If it were translated from English into Italian, this would be an alternative representation. The translator's intention would be to maintain all the information as it is presented in English.

You may develop different models to represent the system from different perspectives. For example:

1. An external perspective, where you model the context or environment of the system.
2. An interaction perspective where you model the interactions between a system and its environment or between the components of a system.
3. A structural perspective, where you model the organization of a system or the structure of the data that is processed by the system.
4. A behavioral perspective, where you model the dynamic behavior of the system and how it responds to events.

These perspectives have much in common with Krutchen's 4 + 1 view of system architecture (Kruchten, 1995), where he suggests that you should document a system's architecture and organization from different perspectives. I discuss this 4 + 1 approach in Chapter 6.

In this chapter, I use diagrams defined in UML (Booch et al., 2005; Rumbaugh et al., 2004), which has become a standard modeling language for object-oriented modeling. The UML has many diagram types and so supports the creation of many different types of system model. However, a survey in 2007 (Erickson and Siau, 2007) showed that most users of the UML thought that five diagram types could represent the essentials of a system:

1. Activity diagrams, which show the activities involved in a process or in data processing.
2. Use case diagrams, which show the interactions between a system and its environment.
3. Sequence diagrams, which show interactions between actors and the system and between system components.
4. Class diagrams, which show the object classes in the system and the associations between these classes.
5. State diagrams, which show how the system reacts to internal and external events.

As I do not have space to discuss all of the UML diagram types here, I focus on how these five key types of diagram are used in system modeling.

When developing system models, you can often be flexible in the way that the graphical notation is used. You do not always need to stick rigidly to the details of a notation. The detail and rigor of a model depends on how you intend to use it. There are three ways in which graphical models are commonly used:

1. As a means of facilitating discussion about an existing or proposed system.
2. As a way of documenting an existing system.
3. As a detailed system description that can be used to generate a system implementation.

In the first case, the purpose of the model is to stimulate the discussion amongst the software engineers involved in developing the system. The models may be incomplete (so long as they cover the key points of the discussion) and they may use the modeling notation informally. This is how models are normally used in so-called 'agile modeling' (Ambler and Jeffries, 2002). When models are used as documentation, they do not have to be complete as you may only wish to develop models for some parts of a system. However, these models have to be correct—they should use the notation correctly and be an accurate description of the system.

The Unified Modeling Language

The Unified Modeling Language is a set of 13 different diagram types that may be used to model software systems. It emerged from work in the 1990s on object-oriented modeling where similar object-oriented notations were integrated to create the UML. A major revision (UML 2) was finalized in 2004. The UML is universally accepted as the standard approach for developing models of software systems. Variants have been proposed for more general system modeling.

http://www.SoftwareEngineering-9.com/Web/UML/

In the third case, where models are used as part of a model-based development process, the system models have to be both complete and correct. The reason for this is that they are used as a basis for generating the source code of the system. Therefore, you have to be very careful not to confuse similar symbols, such as stick and block arrowheads, that have different meanings.

5.1 Context models

At an early stage in the specification of a system, you should decide on the system boundaries. This involves working with system stakeholders to decide what functionality should be included in the system and what is provided by the system's environment. You may decide that automated support for some business processes should be implemented but others should be manual processes or supported by different systems. You should look at possible overlaps in functionality with existing systems and decide where new functionality should be implemented. These decisions should be made early in the process to limit the system costs and the time needed for understanding the system requirements and design.

In some cases, the boundary between a system and its environment is relatively clear. For example, where an automated system is replacing an existing manual or computerized system, the environment of the new system is usually the same as the existing system's environment. In other cases, there is more flexibility, and you decide what constitutes the boundary between the system and its environment during the requirements engineering process.

For example, say you are developing the specification for the patient information system for mental healthcare. This system is intended to manage information about patients attending mental health clinics and the treatments that have been prescribed. In developing the specification for this system, you have to decide whether the system should focus exclusively on collecting information about consultations (using other systems to collect personal information about patients) or whether it should also collect personal patient information. The advantage of relying on other systems for patient information is that you avoid duplicating data. The major disadvantage, however, is that using other systems may make it slower to access information. If these systems are unavailable, then the MHC-PMS cannot be used.

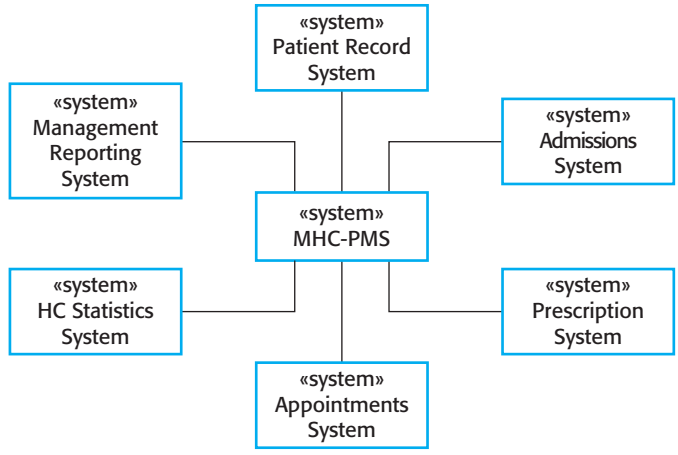


Figure 5.1 The context of the MHC-PMS

The definition of a system boundary is not a value-free judgment. Social and organizational concerns may mean that the position of a system boundary may be determined by non-technical factors. For example, a system boundary may be deliberately positioned so that the analysis process can all be carried out on one site; it may be chosen so that a particularly difficult manager need not be consulted; it may be positioned so that the system cost is increased and the system development division must therefore expand to design and implement the system.

Once some decisions on the boundaries of the system have been made, part of the analysis activity is the definition of that context and the dependencies that a system has on its environment. Normally, producing a simple architectural model is the first step in this activity.

Figure 5.1 is a simple context model that shows the patient information system and the other systems in its environment. From Figure 5.1, you can see that the MHC-PMS is connected to an appointments system and a more general patient record system with which it shares data. The system is also connected to systems for management reporting and hospital bed allocation and a statistics system that collects information for research. Finally, it makes use of a prescription system to generate prescriptions for patients' medication.

Context models normally show that the environment includes several other automated systems. However, they do not show the types of relationships between the systems in the environment and the system that is being specified. External systems might produce data for or consume data from the system. They might share data with the system, or they might be connected directly, through a network or not connected at all. They might be physically co-located or located in separate buildings. All of these relations may affect the requirements and design of the system being defined and must be taken into account.

Therefore, simple context models are used along with other models, such as business process models. These describe human and automated processes in which particular software systems are used.

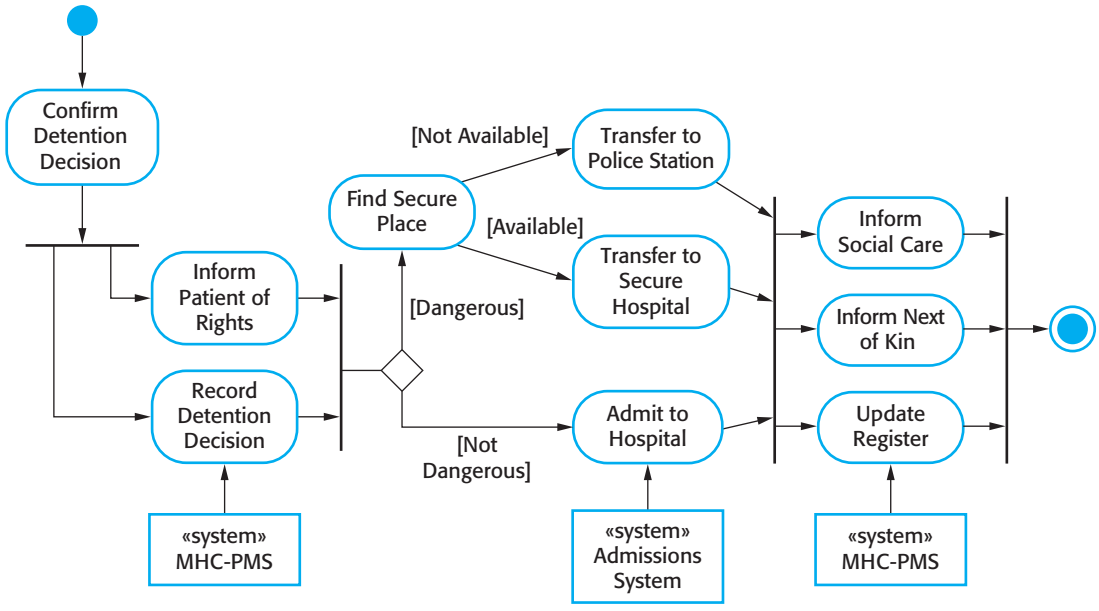


Figure 5.2 Process model of involuntary detention

Figure 5.2 is a model of an important system process that shows the processes in which the MHC-PMS is used. Sometimes, patients who are suffering from mental health problems may be a danger to others or to themselves. They may therefore have to be detained against their will in a hospital so that treatment can be administered. Such detention is subject to strict legal safeguards—for example, the decision to detain a patient must be regularly reviewed so that people are not held indefinitely without good reason. One of the functions of the MHC-PMS is to ensure that such safeguards are implemented.

Figure 5.2 is a UML activity diagram. Activity diagrams are intended to show the activities that make up a system process and the flow of control from one activity to another. The start of a process is indicated by a filled circle; the end by a filled circle inside another circle. Rectangles with round corners represent activities, that is, the specific sub-processes that must be carried out. You may include objects in activity charts. In Figure 5.2, I have shown the systems that are used to support different processes. I have indicated that these are separate systems using the UML stereotype feature.

In a UML activity diagram, arrows represent the flow of work from one activity to another. A solid bar is used to indicate activity coordination. When the flow from more than one activity leads to a solid bar then all of these activities must be complete before progress is possible. When the flow from a solid bar leads to a number of activities, these may be executed in parallel. Therefore, in Figure 5.2, the activities to inform social care and the patient's next of kin, and to update the detention register may be concurrent.

Arrows may be annotated with guards that indicate the condition when that flow is taken. In Figure 5.2, you can see guards showing the flows for patients who are

dangerous and not dangerous to society. Patients who are dangerous to society must be detained in a secure facility. However, patients who are suicidal and so are a danger to themselves may be detained in an appropriate ward in a hospital.

5.2 Interaction models

All systems involve interaction of some kind. This can be user interaction, which involves user inputs and outputs, interaction between the system being developed and other systems or interaction between the components of the system. Modeling user interaction is important as it helps to identify user requirements. Modeling system to system interaction highlights the communication problems that may arise. Modeling component interaction helps us understand if a proposed system structure is likely to deliver the required system performance and dependability.

In this section, I cover two related approaches to interaction modeling:

1. Use case modeling, which is mostly used to model interactions between a system and external actors (users or other systems).
2. Sequence diagrams, which are used to model interactions between system components, although external agents may also be included.

Use case models and sequence diagrams present interaction at different levels of detail and so may be used together. The details of the interactions involved in a high-level use case may be documented in a sequence diagram. The UML also includes communication diagrams that can be used to model interactions. I don't discuss these here as they are an alternative representation of sequence charts. In fact, some tools can generate a communication diagram from a sequence diagram.

5.2.1 Use case modeling

Use case modeling was originally developed by Jacobson et al. (1993) in the 1990s and was incorporated into the first release of the UML (Rumbaugh et al., 1999). As I have discussed in Chapter 4, use case modeling is widely used to support requirements elicitation. A use case can be taken as a simple scenario that describes what a user expects from a system.

Each use case represents a discrete task that involves external interaction with a system. In its simplest form, a use case is shown as an ellipse with the actors involved in the use case represented as stick figures. Figure 5.3 shows a use case from the MHC-PMS that represents the task of uploading data from the MHC-PMS to a more general patient record system. This more general system maintains summary data about a patient rather than the data about each consultation, which is recorded in the MHC-PMS.

Figure 5.3 Transfer-data use case

Notice that there are two actors in this use case: the operator who is transferring the data and the patient record system. The stick figure notation was originally developed to cover human interaction but it is also now used to represent other external systems and hardware. Formally, use case diagrams should use lines without arrows as arrows in the UML indicate the direction of flow of messages. Obviously, in a use case messages pass in both directions. However, the arrows in Figure 5.3 are used informally to indicate that the medical receptionist initiates the transaction and data is transferred to the patient record system.

Use case diagrams give a fairly simple overview of an interaction so you have to provide more detail to understand what is involved. This detail can either be a simple textual description, a structured description in a table, or a sequence diagram as discussed below. You chose the most appropriate format depending on the use case and the level of detail that you think is required in the model. I find a standard tabular format to be the most useful. Figure 5.4 shows a tabular description of the 'Transfer data' use case.

As I have discussed in Chapter 4, composite use case diagrams show a number of different use cases. Sometimes, it is possible to include all possible interactions with a system in a single composite use case diagram. However, this may be impossible because of the number of use cases. In such cases, you may develop several diagrams, each of which shows related use cases. For example, Figure 5.5 shows all of the use cases in the MHC-PMS in which the actor 'Medical Receptionist' is involved.

Figure 5.4 Tabular description of the 'Transfer data' use case

MHC-PMS: Transfer data	
Actors	Medical receptionist, patient records system (PRS)
Description	A receptionist may transfer data from the MHC-PMS to a general patient record database that is maintained by a health authority. The information transferred may either be updated personal information (address, phone number, etc.) or a summary of the patient's diagnosis and treatment.
Data	Patient's personal information, treatment summary
Stimulus	User command issued by medical receptionist
Response	Confirmation that PRS has been updated
Comments	The receptionist must have appropriate security permissions to access the patient information and the PRS.

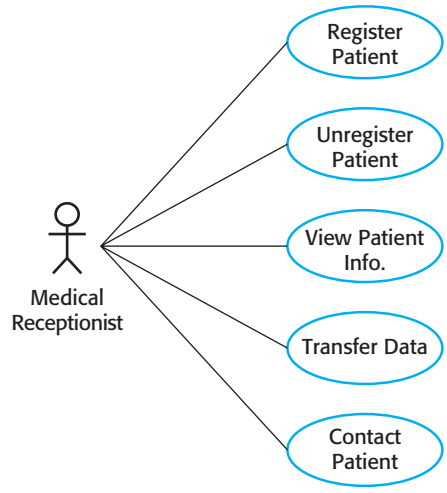


Figure 5.5 Use cases involving the role 'medical receptionist'

5.2.2 Sequence diagrams

Sequence diagrams in the UML are primarily used to model the interactions between the actors and the objects in a system and the interactions between the objects themselves. The UML has a rich syntax for sequence diagrams, which allows many different kinds of interaction to be modeled. I don't have space to cover all possibilities here so I focus on the basics of this diagram type.

As the name implies, a sequence diagram shows the sequence of interactions that take place during a particular use case or use case instance. Figure 5.6 is an example of a sequence diagram that illustrates the basics of the notation. This diagram models the interactions involved in the View patient information use case, where a medical receptionist can see some patient information.

The objects and actors involved are listed along the top of the diagram, with a dotted line drawn vertically from these. Interactions between objects are indicated by annotated arrows. The rectangle on the dotted lines indicates the lifeline of the object concerned (i.e., the time that object instance is involved in the computation). You read the sequence of interactions from top to bottom. The annotations on the arrows indicate the calls to the objects, their parameters, and the return values. In this example, I also show the notation used to denote alternatives. A box named alt is used with the conditions indicated in square brackets.

You can read Figure 5.6 as follows:

1. The medical receptionist triggers the ViewInfo method in an instance P of the PatientInfo object class, supplying the patient's identifier, PID. P is a user interface object, which is displayed as a form showing patient information.
2. The instance P calls the database to return the information required, supplying the receptionist's identifier to allow security checking (at this stage, we do not care where this UID comes from).

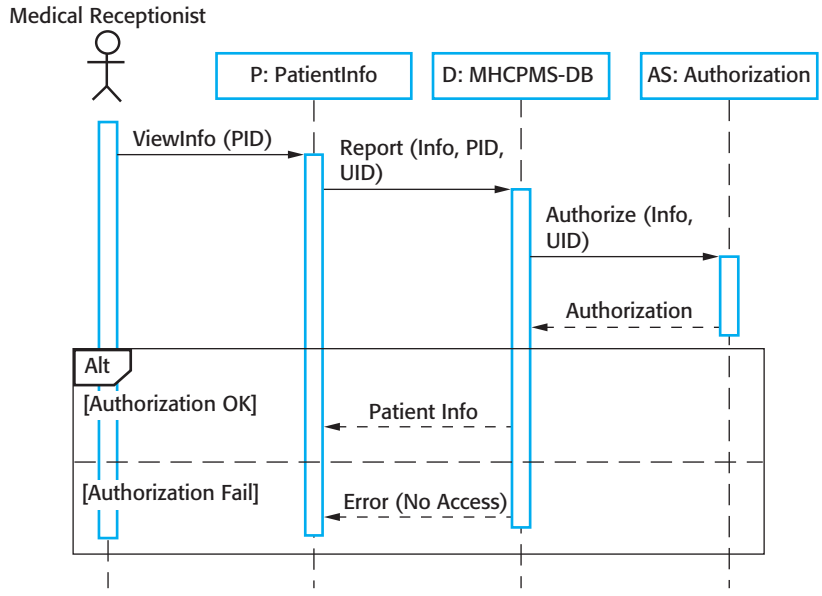


Figure 5.6 Sequence diagram for View patient information

3. The database checks with an authorization system that the user is authorized for this action.
4. If authorized, the patient information is returned and a form on the user's screen is filled in. If authorization fails, then an error message is returned.

Figure 5.7 is a second example of a sequence diagram from the same system that illustrates two additional features. These are the direct communication between the actors in the system and the creation of objects as part of a sequence of operations. In this example, an object of type Summary is created to hold the summary data that is to be uploaded to the PRS (patient record system). You can read this diagram as follows:

1. The receptionist logs on to the PRS.
2. There are two options available. These allow the direct transfer of updated patient information to the PRS and the transfer of summary health data from the MHC-PMS to the PRS.
3. In each case, the receptionist's permissions are checked using the authorization system.
4. Personal information may be transferred directly from the user interface object to the PRS. Alternatively, a summary record may be created from the database and that record is then transferred.
5. On completion of the transfer, the PRS issues a status message and the user logs off.

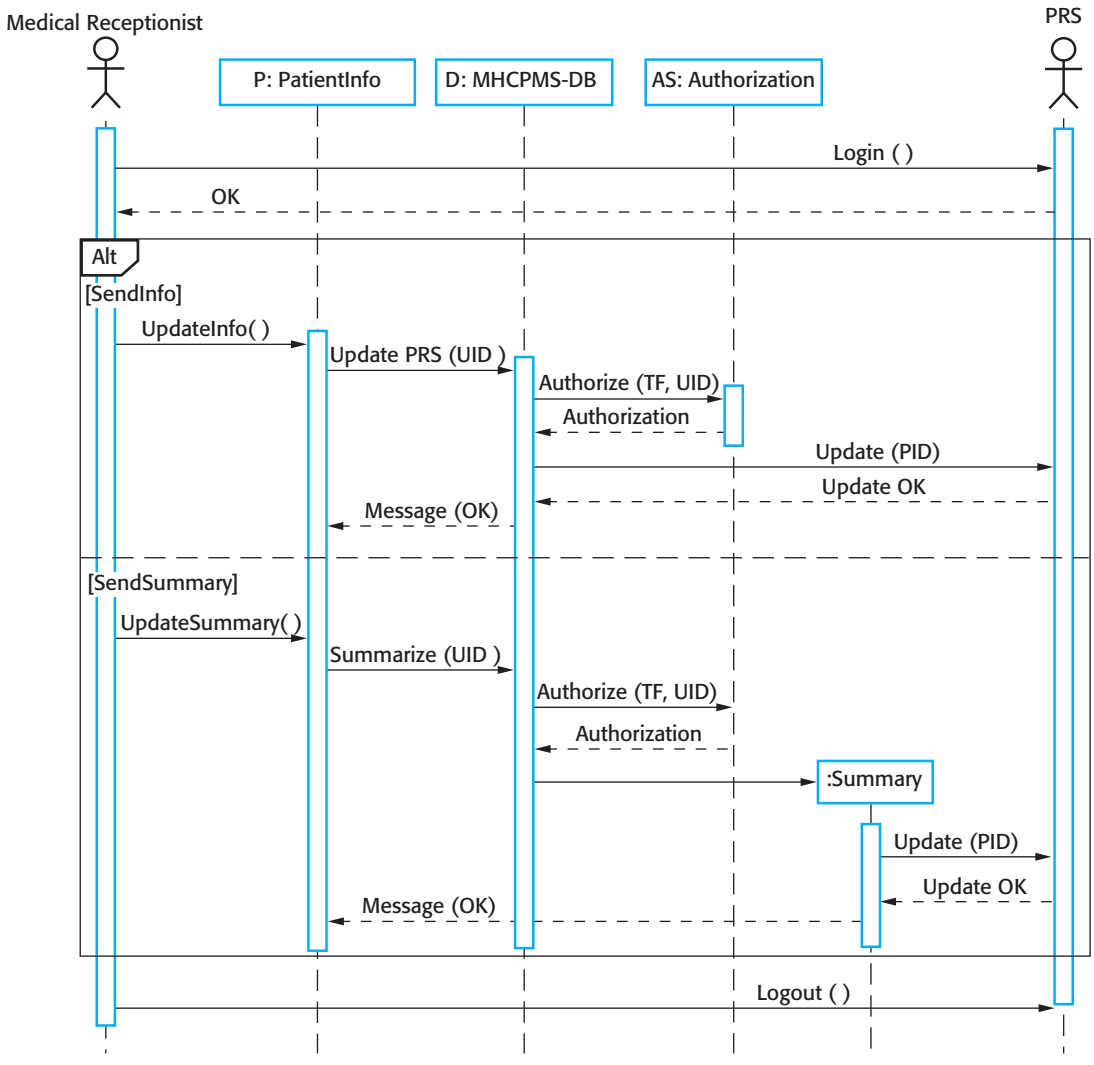


Figure 5.7 Sequence diagram for transfer data

Unless you are using sequence diagrams for code generation or detailed documentation, you don't have to include every interaction in these diagrams. If you develop system models early in the development process to support requirements engineering and high-level design, there will be many interactions which depend on implementation decisions. For example, in Figure 5.7 the decision on how to get the user's identifier to check authorization is one that can be delayed. In an implementation, this might involve interacting with a User object but this is not important at this stage and so need not be included in the sequence diagram.

Object-oriented requirements analysis

In object-oriented requirements analysis, you model real-world entities using object classes. You may create different types of object model, showing how object classes are related to each other, how objects are aggregated to form other objects, how objects interact with other objects, and so on. These each present unique information about the system that is being specified.

http://www.SoftwareEngineering-9.com/Web/OORA/

5.3 Structural models

Structural models of software display the organization of a system in terms of the components that make up that system and their relationships. Structural models may be static models, which show the structure of the system design or dynamic models, which show the organization of the system when it is executing. These are not the same things—the dynamic organization of a system as a set of interacting threads may be very different from a static model of the system components.

You create structural models of a system when you are discussing and designing the system architecture. Architectural design is a particularly important topic in software engineering and UML component, package, and deployment diagrams may all be used when presenting architectural models. I cover different aspects of software architecture and architectural modeling in Chapters 6, 18, and 19. In this section, I focus on the use of class diagrams for modeling the static structure of the object classes in a software system.

5.3.1 Class diagrams

Class diagrams are used when developing an object-oriented system model to show the classes in a system and the associations between these classes. Loosely, an object class can be thought of as a general definition of one kind of system object. An association is a link between classes that indicates that there is a relationship between these classes. Consequently, each class may have to have some knowledge of its associated class.

When you are developing models during the early stages of the software engineering process, objects represent something in the real world, such as a patient, a prescription, a doctor, etc. As an implementation is developed, you usually need to define additional implementation objects that are used to provide the required system functionality. Here, I focus on the modeling of real-world objects as part of the requirements or early software design processes.

Class diagrams in the UML can be expressed at different levels of detail. When you are developing a model, the first stage is usually to look at the world, identify the essential objects, and represent these as classes. The simplest way of writing these is to write the class name in a box. You can also simply note the existence of an association

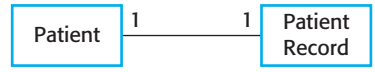


Figure 5.8 UML classes and association

by drawing a line between classes. For example, Figure 5.8 is a simple class diagram showing two classes: Patient and Patient Record with an association between them.

In Figure 5.8, I illustrate a further feature of class diagrams—the ability to show how many objects are involved in the association. In this example, each end of the association is annotated with a 1, meaning that there is a 1:1 relationship between objects of these classes. That is, each patient has exactly one record and each record maintains information about exactly one patient. As you can see from later examples, other multiplicities are possible. You can define that an exact number of objects are involved or, by using a *, as shown in Figure 5.9, that there are an indefinite number of objects involved in the association.

Figure 5.9 develops this type of class diagram to show that objects of class Patient are also involved in relationships with a number of other classes. In this example, I show that you can name associations to give the reader an indication of the type of relationship that exists. The UML also allows the role of the objects participating in the association to be specified.

At this level of detail, class diagrams look like semantic data models. Semantic data models are used in database design. They show the data entities, their associated attributes, and the relations between these entities. This approach to modeling was first proposed in the mid-1970s by Chen (1976); several variants have been developed since then (Codd, 1979; Hammer and McLeod, 1981; Hull and King, 1987), all with the same basic form.

The UML does not include a specific notation for this database modeling as it assumes an object-oriented development process and models data using objects and their relationships. However, you can use the UML to represent a semantic data model. You can think of entities in a semantic data model as simplified object classes

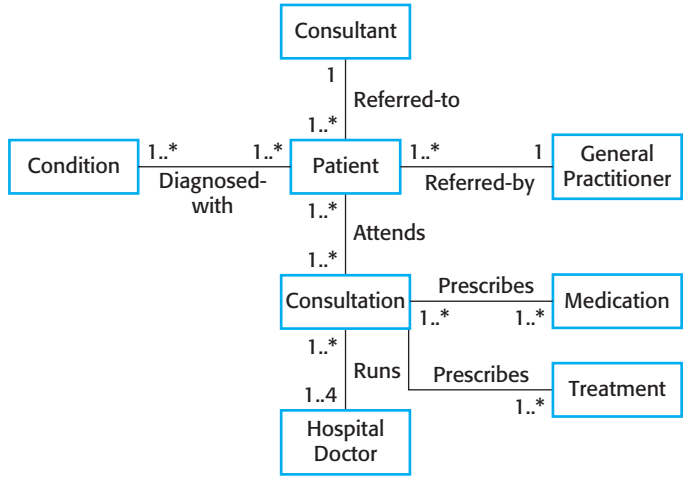


Figure 5.9 Classes and associations in the MHC-PMS

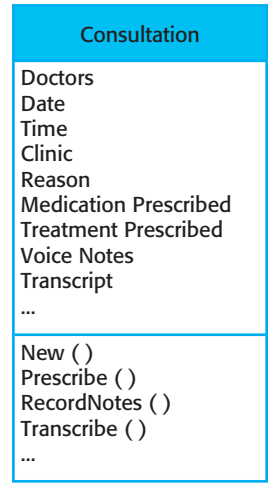


Figure 5.10 The consultation class

(they have no operations), attributes as object class attributes and relations as named associations between object classes.

When showing the associations between classes, it is convenient to represent these classes in the simplest possible way. To define them in more detail, you add information about their attributes (the characteristics of an object) and operations (the things that you can request from an object). For example, a Patient object will have the attribute Address and you may include an operation called ChangeAddress, which is called when a patient indicates that they have moved from one address to another. In the UML, you show attributes and operations by extending the simple rectangle that represents a class. This is illustrated in Figure 5.10 where:

1. The name of the object class is in the top section.
2. The class attributes are in the middle section. This must include the attribute names and, optionally, their types.
3. The operations (called methods in Java and other OO programming languages) associated with the object class are in the lower section of the rectangle.

Figure 5.10 shows possible attributes and operations on the class Consultation. In this example, I assume that doctors record voice notes that are transcribed later to record details of the consultation. To prescribe medication, the doctor involved must use the Prescribe method to generate an electronic prescription.

5.3.2 Generalization

Generalization is an everyday technique that we use to manage complexity. Rather than learn the detailed characteristics of every entity that we experience, we place these entities in more general classes (animals, cars, houses, etc.) and learn the characteristics of

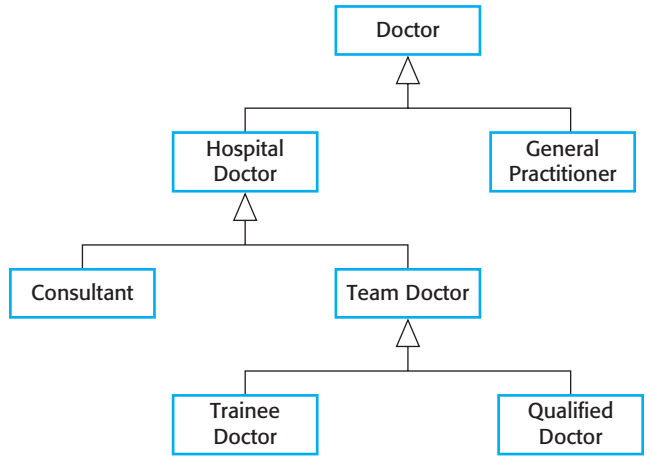


Figure 5.11 A generalization hierarchy

these classes. This allows us to infer that different members of these classes have some common characteristics (e.g., squirrels and rats are rodents). We can make general statements that apply to all class members (e.g., all rodents have teeth for gnawing).

In modeling systems, it is often useful to examine the classes in a system to see if there is scope for generalization. This means that common information will be maintained in one place only. This is good design practice as it means that, if changes are proposed, then you do not have to look at all classes in the system to see if they are affected by the change. In object-oriented languages, such as Java, generalization is implemented using the class inheritance mechanisms built into the language.

The UML has a specific type of association to denote generalization, as illustrated in Figure 5.11. The generalization is shown as an arrowhead pointing up to the more general class. This shows that general practitioners and hospital doctors can be generalized as doctors and that there are three types of Hospital Doctor—those that have just graduated from medical school and have to be supervised (Trainee Doctor); those that can work unsupervised as part of a consultant's team (Registered Doctor); and consultants, who are senior doctors with full decision-making responsibilities.

In a generalization, the attributes and operations associated with higher-level classes are also associated with the lower-level classes. In essence, the lower-level classes are subclasses inherit the attributes and operations from their superclasses. These lower-level classes then add more specific attributes and operations. For example, all doctors have a name and phone number; all hospital doctors have a staff number and a department but general practitioners don't have these attributes as they work independently. They do however, have a practice name and address. This is illustrated in Figure 5.12, which shows part of the generalization hierarchy that I have extended with class attributes. The operations associated with the class Doctor are intended to register and de-register that doctor with the MHC-PMS.

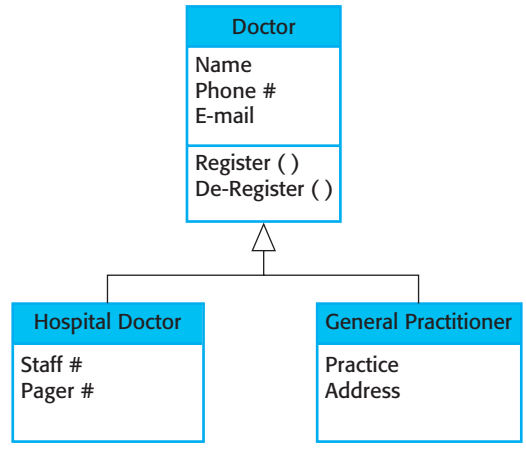


Figure 5.12 A generalization hierarchy with added detail

5.3.3 Aggregation

Objects in the real world are often composed of different parts. For example, a study pack for a course may be composed of a book, PowerPoint slides, quizzes, and recommendations for further reading. Sometimes in a system model, you need to illustrate this. The UML provides a special type of association between classes called aggregation that means that one object (the whole) is composed of other objects (the parts). To show this, we use a diamond shape next to the class that represents the whole. This is shown in Figure 5.13, which shows that a patient record is a composition of Patient and an indefinite number of Consultations.

5.4 Behavioral models

Behavioral models are models of the dynamic behavior of the system as it is executing. They show what happens or what is supposed to happen when a system responds to a stimulus from its environment. You can think of these stimuli as being of two types:

1. *Data* Some data arrives that has to be processed by the system.
2. *Events* Some event happens that triggers system processing. Events may have associated data but this is not always the case.

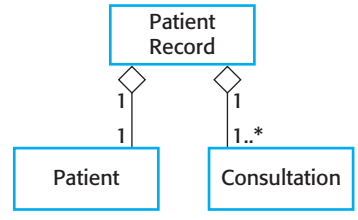


Figure 5.13 The aggregation association

Data-flow diagrams

Data-flow diagrams (DFDs) are system models that show a functional perspective where each transformation represents a single function or process. DFDs are used to show how data flows through a sequence of processing steps. For example, a processing step could be the filtering of duplicate records in a customer database. The data is transformed at each step before moving on to the next stage. These processing steps or transformations represent software processes or functions where data-flow diagrams are used to document a software design.

http://www.SoftwareEngineering-9.com/Web/DFDs

Many business systems are data processing systems that are primarily driven by data. They are controlled by the data input to the system with relatively little external event processing. Their processing involves a sequence of actions on that data and the generation of an output. For example, a phone billing system will accept information about calls made by a customer, calculate the costs of these calls, and generate a bill to be sent to that customer. By contrast, real-time systems are often event driven with minimal data processing. For example, a landline phone switching system responds to events such as 'receiver off hook' by generating a dial tone, or the pressing of keys on a handset by capturing the phone number, etc.

5.4.1 Data-driven modeling

Data-driven models show the sequence of actions involved in processing input data and generating an associated output. They are particularly useful during the analysis of requirements as they can be used to show end-to-end processing in a system. That is, they show the entire sequence of actions that take place from an input being processed to the corresponding output, which is the system's response.

Data-driven models were amongst the first graphical software models. In the 1970s, structured methods such as DeMarco's Structured Analysis (DeMarco, 1978) introduced data-flow diagrams (DFDs) as a way of illustrating the processing steps in a system. Data-flow models are useful because tracking and documenting how the data associated with a particular process moves through the system helps analysts and designers understand what is going on. Data-flow diagrams are simple and intuitive and it is usually possible to explain them to potential system users who can then participate in validating the model.

The UML does not support data-flow diagrams as they were originally proposed and used for modeling data processing. The reason for this is that DFDs focus on system functions and do not recognize system objects. However, because data-driven systems are so common in business, UML 2.0 introduced activity diagrams, which are similar to data-flow diagrams. For example, Figure 5.14 shows the chain of processing involved in the insulin pump software. In this diagram, you can see the processing steps (represented as activities) and the data flowing between these steps (represented as objects).

An alternative way of showing the sequence of processing in a system is to use UML sequence diagrams. You have seen how these can be used to model interaction but, if

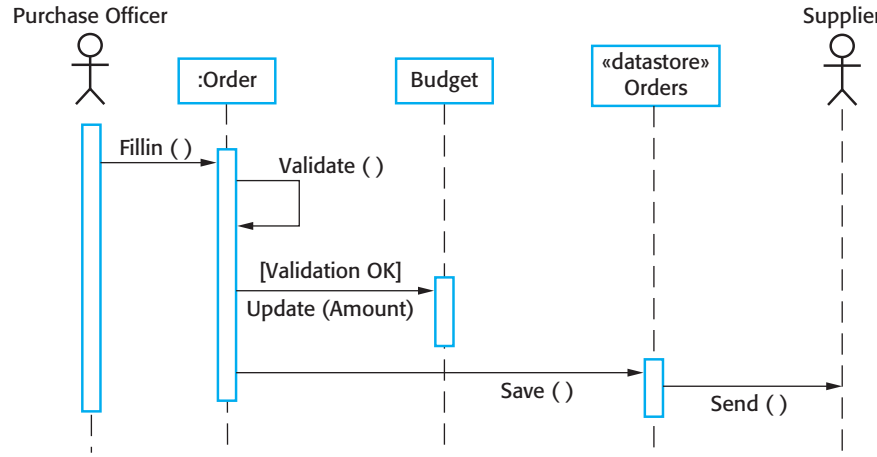


Figure 5.14 An activity model of the insulin pump's operation

you draw these so that messages are only sent from left to right, then they show the sequential data processing in the system. Figure 5.15 illustrates this, using a sequence model of the processing of an order and sending it to a supplier. Sequence models highlight objects in a system, whereas data-flow diagrams highlight the functions. The equivalent data-flow diagram for order processing is shown on the book's web pages.

5.4.2 Event-driven modeling

Event-driven modeling shows how a system responds to external and internal events. It is based on the assumption that a system has a finite number of states and that events (stimuli) may cause a transition from one state to another. For example, a system controlling a valve may move from a state 'Valve open' to a state 'Valve closed' when an operator command (the stimulus) is received. This view of a system is particularly appropriate for real-time systems. Event-based modeling was introduced in real-time design methods such as those proposed by Ward and Mellor (1985) and Harel (1987, 1988).

The UML supports event-based modeling using state diagrams, which were based on Statecharts (Harel, 1987, 1988). State diagrams show system states and events that cause

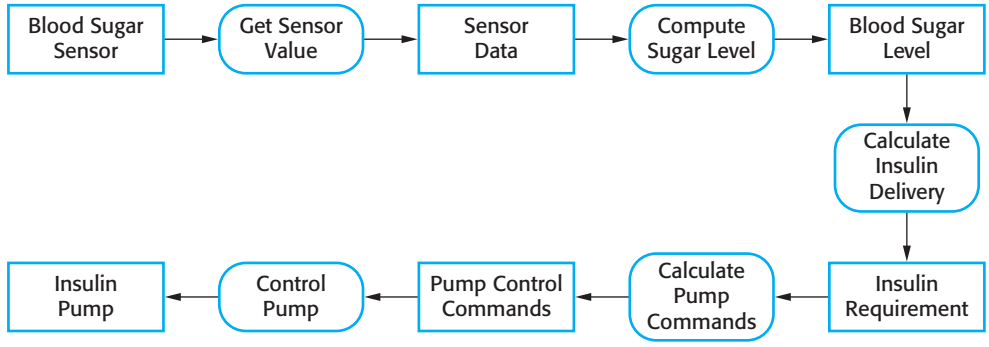


Figure 5.15 Order processing

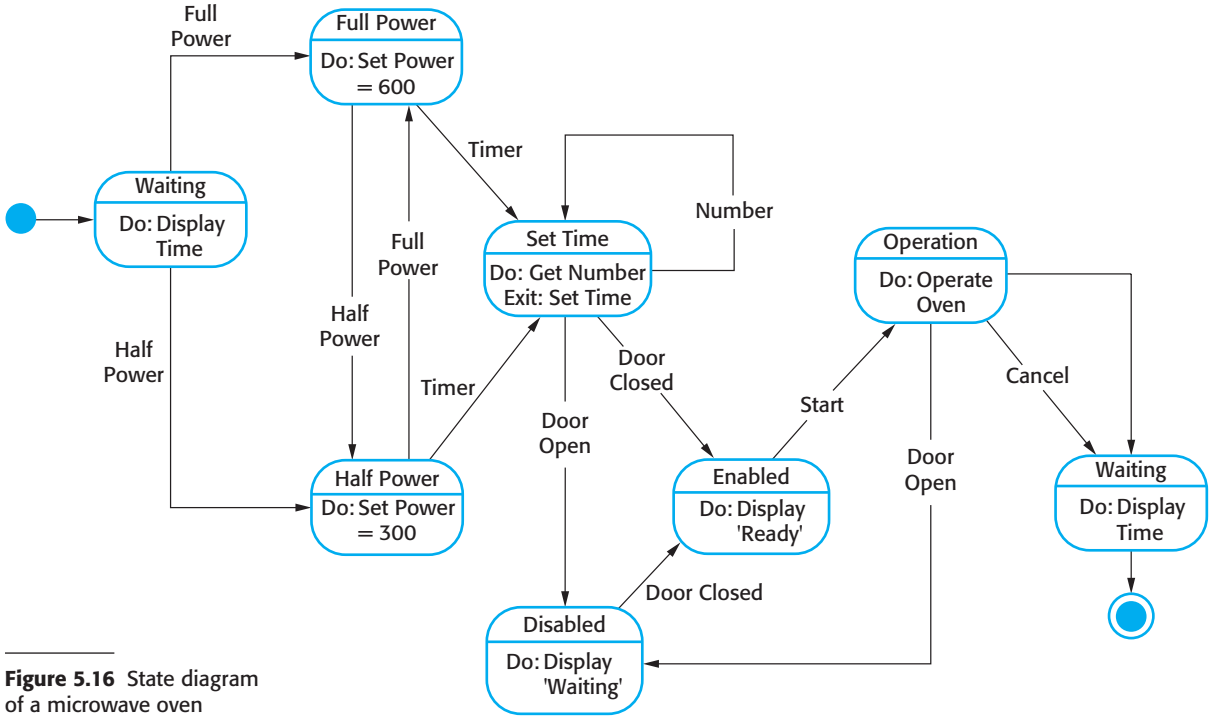


Figure 5.16 State diagram of a microwave oven

transitions from one state to another. They do not show the flow of data within the system but may include additional information on the computations carried out in each state.

I use an example of control software for a very simple microwave oven to illustrate event-driven modeling. Real microwave ovens are actually much more complex than this system but the simplified system is easier to understand. This simple microwave has a switch to select full or half power, a numeric keypad to input the cooking time, a start/stop button, and an alphanumeric display.

I have assumed that the sequence of actions in using the microwave is:

1. Select the power level (either half power or full power).
2. Input the cooking time using a numeric keypad.
3. Press Start and the food is cooked for the given time.

For safety reasons, the oven should not operate when the door is open and, on completion of cooking, a buzzer is sounded. The oven has a very simple alphanumeric display that is used to display various alerts and warning messages.

In UML state diagrams, rounded rectangles represent system states. They may include a brief description (following 'do') of the actions taken in that state. The labeled arrows represent stimuli that force a transition from one state to another. You can indicate start and end states using filled circles, as in activity diagrams.

From Figure 5.16, you can see that the system starts in a waiting state and responds initially to either the full-power or the half-power button. Users can change

State	Description
Waiting	The oven is waiting for input. The display shows the current time.
Half power	The oven power is set to 300 watts. The display shows 'Half power'.
Full power	The oven power is set to 600 watts. The display shows 'Full power'.
Set time	The cooking time is set to the user's input value. The display shows the cooking time selected and is updated as the time is set.
Disabled	Oven operation is disabled for safety. Interior oven light is on. Display shows 'Not ready'.
Enabled	Oven operation is enabled. Interior oven light is off. Display shows 'Ready to cook'.
Operation	Oven in operation. Interior oven light is on. Display shows the timer countdown. On completion of cooking, the buzzer is sounded for five seconds. Oven light is on. Display shows 'Cooking complete' while buzzer is sounding.
Stimulus	**Description**
Half power	The user has pressed the half-power button.
Full power	The user has pressed the full-power button.
Timer	The user has pressed one of the timer buttons.
Number	The user has pressed a numeric key.
Door open	The oven door switch is not closed.
Door closed	The oven door switch is closed.
Start	The user has pressed the Start button.
Cancel	The user has pressed the Cancel button.

Figure 5.17 States and stimuli for the microwave oven

their mind after selecting one of these and press the other button. The time is set and, if the door is closed, the Start button is enabled. Pushing this button starts the oven operation and cooking takes place for the specified time. This is the end of the cooking cycle and the system returns to the waiting state.

The UML notation lets you indicate the activity that takes place in a state. In a detailed system specification you have to provide more detail about both the stimuli and the system states. I illustrate this in Figure 5.17, which shows a tabular description of each state and how the stimuli that force state transitions are generated.

The problem with state-based modeling is that the number of possible states increases rapidly. For large system models, therefore, you need to hide detail in the

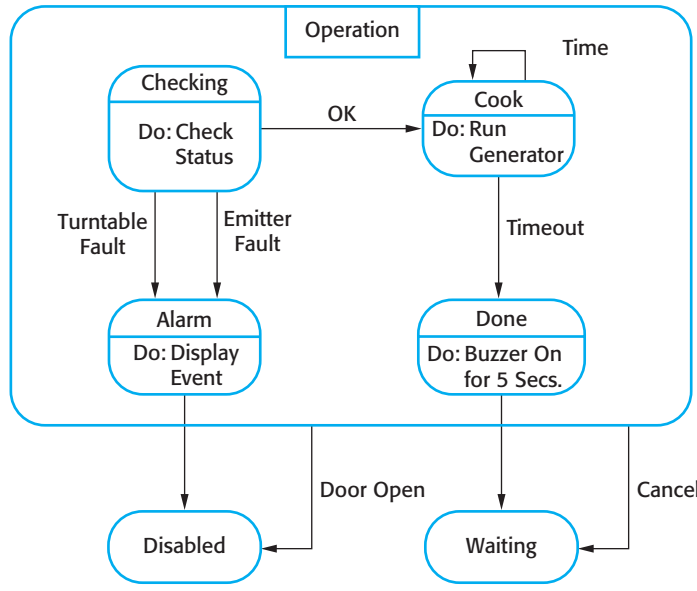


Figure 5.18 Microwave oven operation

models. One way to do this is by using the notion of a superstate that encapsulates a number of separate states. This superstate looks like a single state on a high-level model but is then expanded to show more detail on a separate diagram. To illustrate this concept, consider the Operation state in Figure 5.15. This is a superstate that can be expanded, as illustrated in Figure 5.18.

The Operation state includes a number of sub-states. It shows that operation starts with a status check and that if any problems are discovered an alarm is indicated and operation is disabled. Cooking involves running the microwave generator for the specified time; on completion, a buzzer is sounded. If the door is opened during operation, the system moves to the disabled state, as shown in Figure 5.15.

5.5 Model-driven engineering

Model-driven engineering (MDE) is an approach to software development where models rather than programs are the principal outputs of the development process (Kent, 2002; Schmidt, 2006). The programs that execute on a hardware/software platform are then generated automatically from the models. Proponents of MDE argue that this raises the level of abstraction in software engineering so that engineers no longer have to be concerned with programming language details or the specifics of execution platforms.

Model-driven engineering has its roots in model-driven architecture (MDA) which was proposed by the Object Management Group (OMG) in 2001 as a new software development paradigm. Model-driven engineering and model-driven architecture are often seen as the same thing. However, I think that MDE has a wider scope than

MDA. As I discuss later in this section, MDA focuses on the design and implementation stages of software development whereas MDE is concerned with all aspects of the software engineering process. Therefore, topics such as model-based requirements engineering, software processes for model-based development, and model-based testing are part of MDE but not, currently, part of MDA.

Although MDA has been in use since 2001, model-based engineering is still at an early stage of development and it is unclear whether or not it will have a significant effect on software engineering practice. The main arguments for and against MDE are:

1. *For MDE* Model-based engineering allows engineers to think about systems at a high level of abstraction, without concern for the details of their implementation. This reduces the likelihood of errors, speeds up the design and implementation process, and allows for the creation of reusable, platform-independent application models. By using powerful tools, system implementations can be generated for different platforms from the same model. Therefore, to adapt the system to some new platform technology, it is only necessary to write a translator for that platform. When this is available, all platform-independent models can be rapidly rehosted on the new platform.

2. *Against MDE* As I discussed earlier in this chapter, models are a good way of facilitating discussions about a software design. However, it does not always follow that the abstractions that are supported by the model are the right abstractions for implementation. So, you may create informal design models but then go on to implement the system using an off-the-shelf, configurable package. Furthermore, the arguments for platform independence are only valid for large long-lifetime systems where the platforms become obsolete during a system's lifetime. However, for this class of systems, we know that implementation is not the major problem—requirements engineering, security and dependability, integration with legacy systems, and testing are more significant.

There have been significant MDE success stories reported by the OMG on their Web pages (www.omg.org/mda/products_success.htm) and the approach is used within large companies such as IBM and Siemens. The techniques have been used successfully in the development of large, long-lifetime software systems such as air traffic management systems. Nevertheless, at the time of writing, model-driven approaches are not widely used for software engineering. Like formal methods of software engineering, which I discuss in Chapter 12, I believe that MDE is an important development. However, as is also the case with formal methods, it is not clear whether the costs and risks of model-driven approaches outweigh the possible benefits.

5.5.1 Model-driven architecture

Model-driven architecture (Kleppe, et al., 2003; Mellor et al., 2004; Stahl and Voelter, 2006) is a model-focused approach to software design and implementation that uses a sub-set of UML models to describe a system. Here, models at different

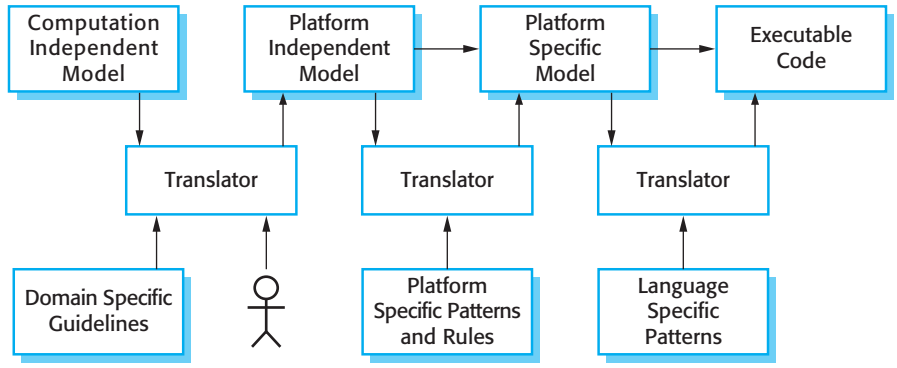


Figure 5.19 MDA transformations

levels of abstraction are created. From a high-level platform independent model it is possible, in principle, to generate a working program without manual intervention.

The MDA method recommends that three types of abstract system model should be produced:

1. A computation independent model (CIM) that models the important domain abstractions used in the system. CIMs are sometimes called domain models. You may develop several different CIMs, reflecting different views of the system. For example, there may be a security CIM in which you identify important security abstractions such as an asset and a role and a patient record CIM, in which you describe abstractions such as patients, consultations, etc.
2. A platform independent model (PIM) that models the operation of the system without reference to its implementation. The PIM is usually described using UML models that show the static system structure and how it responds to external and internal events.
3. Platform specific models (PSM) which are transformations of the platform-independent model with a separate PSM for each application platform. In principle, there may be layers of PSM, with each layer adding some platform-specific detail. So, the first-level PSM could be middleware-specific but database independent. When a specific database has been chosen, a database-specific PSM can then be generated.

As I have said, transformations between these models may be defined and applied automatically by software tools. This is illustrated in Figure 5.19, which also shows a final level of automatic transformation. A transformation is applied to the PSM to generate executable code that runs on the designated software platform.

At the time of writing, automatic CIM to PIM translation is still at the research prototype stage. It is unlikely that completely automated translation tools will be available in the near future. Human intervention, indicated by a stick figure in Figure 5.19, will be needed for the foreseeable future. CIMs are related and part of the translation

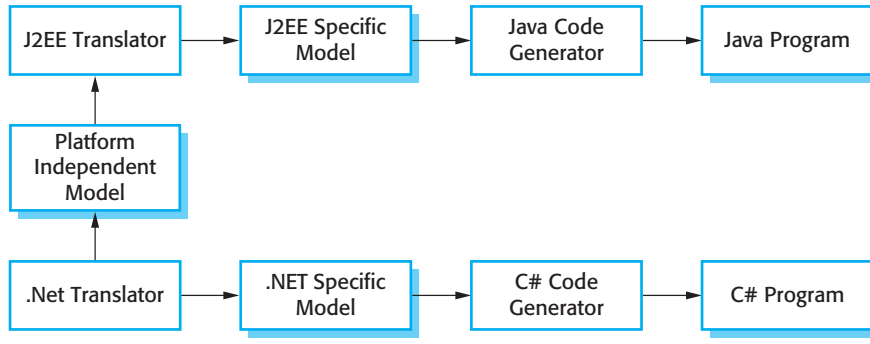


Figure 5.20 Multiple platform-specific models

process may involve linking concepts in different CIMs. For example, the concept of a role in a security CIM may be mapped onto the concept of a staff member in a hospital CIM. Mellor and Balcer (2002) give the name 'bridges' to the information that supports mapping from one CIM to another.

The translation of PIMs to PSMs is more mature and several commercial tools are available that provide translators from PIMs to common platforms such as Java and J2EE. These rely on an extensive library of platform-specific rules and patterns to convert the PIM to the PSM. There may be several PSMs for each PIM in the system. If a software system is intended to run on different platforms (e.g., J2EE and .NET), then it is only necessary to maintain the PIM. The PSMs for each platform are automatically generated. This is illustrated in Figure 5.20.

Although MDA-support tools include platform-specific translators, it is often the case that these will only offer partial support for the translation from PIMs to PSMs. In the vast majority of cases, the execution environment for a system is more than the standard execution platform (e.g., J2EE, .NET, etc.). It also includes other application systems, application libraries that are specific to a company, and user interface libraries. As these vary significantly from one company to another, standard tool support is not available. Therefore, when MDA is introduced, special purpose translators may have to be created that take the characteristics of the local environment into account. In some cases (e.g., for user interface generation), completely automated PIM to PSM translation may be impossible.

There is an uneasy relationship between agile methods and model-driven architecture. The notion of extensive up-front modeling contradicts the fundamental ideas in the agile manifesto and I suspect that few agile developers feel comfortable with model-driven engineering. The developers of MDA claim that it is intended to support an iterative approach to development and so can be used within agile methods (Mellor, et al., 2004). If transformations can be completely automated and a complete program generated from a PIM, then, in principle, MDA could be used in an agile development process as no separate coding would be required. However, as far as I am aware, there are no MDA tools that support practices such as regression testing and test-driven development.

5.5.2 Executable UML

The fundamental notion behind model-driven engineering is that completely automated transformation of models to code should be possible. To achieve this, you have to be able to construct graphical models whose semantics are well defined. You also need a way of adding information to graphical models about the ways in which the operations defined in the model are implemented. This is possible using a subset of UML 2, called Executable UML or xUML (Mellor and Balcer, 2002). I don't have space here to describe the details of xUML, so I simply present a short overview of its main features.

UML was designed as a language for supporting and documenting software design, not as a programming language. The designers of UML were not concerned with semantic details of the language but with its expressiveness. They introduced useful notions such as use case diagrams that help with the design but which are too informal to support execution. To create an executable sub-set of UML, the number of model types has therefore been dramatically reduced to three key model types:

1. Domain models identify the principal concerns in the system. These are defined using UML class diagrams that include objects, attributes, and associations.
2. Class models, in which classes are defined, along with their attributes and operations.
3. State models, in which a state diagram is associated with each class and is used to describe the lifecycle of the class.

The dynamic behavior of the system may be specified declaratively using the object constraint language (OCL) or may be expressed using UML's action language. The action language is like a very high-level programming language where you can refer to objects and their attributes and specify actions to be carried out.

KEY POINTS

- A model is an abstract view of a system that ignores some system details. Complementary system models can be developed to show the system's context, interactions, structure, and behavior.
- Context models show how a system that is being modeled is positioned in an environment with other systems and processes. They help define the boundaries of the system to be developed.
- Use case diagrams and sequence diagrams are used to describe the interactions between user the system being designed and users/other systems. Use cases describe interactions between a system and external actors; sequence diagrams add more information to these by showing interactions between system objects.

- Structural models show the organization and architecture of a system. Class diagrams are used to define the static structure of classes in a system and their associations.
- Behavioral models are used to describe the dynamic behavior of an executing system. This can be modeled from the perspective of the data processed by the system or by the events that stimulate responses from a system.
- Activity diagrams may be used to model the processing of data, where each activity represents one process step.
- State diagrams are used to model a system's behavior in response to internal or external events.
- Model-driven engineering is an approach to software development in which a system is represented as a set of models that can be automatically transformed to executable code.

FURTHER READING

Requirements Analysis and System Design. This book focuses on information systems analysis and discusses how different UML models can be used in the analysis process. (L. Maciaszek, Addison-Wesley, 2001.)

MDA Distilled: Principles of Model-driven Architecture .This is a concise and accessible introduction to the MDA method. It is written by enthusiasts so the book says very little about possible problems with this approach. (S. J. Mellor, K. Scott and D. Weise, Addison-Wesley, 2004.)

Using UML: Software Engineering with Objects and Components, 2nd ed. A short, readable introduction to the use of the UML in system specification and design. This book is excellent for learning and understanding the UML, although it is not a full description of the notation. (P. Stevens with R. Pooley, Addison-Wesley, 2006.)

EXERCISES

5.1. Explain why it is important to model the context of a system that is being developed. Give two examples of possible errors that could arise if software engineers do not understand the system context.

5.2. How might you use a model of a system that already exists? Explain why it is not always necessary for such a system model to be complete and correct. Would the same be true if you were developing a model of a new system?

6

Architectural design

Objectives

The objective of this chapter is to introduce the concepts of software architecture and architectural design. When you have read the chapter, you will:

- understand why the architectural design of software is important;
- understand the decisions that have to be made about the system architecture during the architectural design process;
- have been introduced to the idea of architectural patterns, well-tried ways of organizing system architectures, which can be reused in system designs;
- know the architectural patterns that are often used in different types of application system, including transaction processing systems and language processing systems.

Contents

Architectural design is concerned with understanding how a system should be organized and designing the overall structure of that system. In the model of the software development process, as shown in Chapter 2, architectural design is the first stage in the software design process. It is the critical link between design and requirements engineering, as it identifies the main structural components in a system and the relationships between them. The output of the architectural design process is an architectural model that describes how the system is organized as a set of communicating components.

In agile processes, it is generally accepted that an early stage of the development process should be concerned with establishing an overall system architecture. Incremental development of architectures is not usually successful. While refactoring components in response to changes is usually relatively easy, refactoring a system architecture is likely to be expensive.

To help you understand what I mean by system architecture, consider Figure 6.1. This shows an abstract model of the architecture for a packing robot system that shows the components that have to be developed. This robotic system can pack different kinds of object. It uses a vision component to pick out objects on a conveyor, identify the type of object, and select the right kind of packaging. The system then moves objects from the delivery conveyor to be packaged. It places packaged objects on another conveyor. The architectural model shows these components and the links between them.

In practice, there is a significant overlap between the processes of requirements engineering and architectural design. Ideally, a system specification should not include any design information. This is unrealistic except for very small systems. Architectural decomposition is usually necessary to structure and organize the specification. Therefore, as part of the requirements engineering process, you might propose an abstract system architecture where you associate groups of system functions or features with large-scale components or sub-systems. You can then use this decomposition to discuss the requirements and features of the system with stakeholders.

You can design software architectures at two levels of abstraction, which I call *architecture in the small* and *architecture in the large*:

1. Architecture in the small is concerned with the architecture of individual programs. At this level, we are concerned with the way that an individual program is decomposed into components. This chapter is mostly concerned with program architectures.

2. Architecture in the large is concerned with the architecture of complex enterprise systems that include other systems, programs, and program components. These enterprise systems are distributed over different computers, which may be owned and managed by different companies. I cover architecture in the large in Chapters 18 and 19, where I discuss distributed systems architectures.

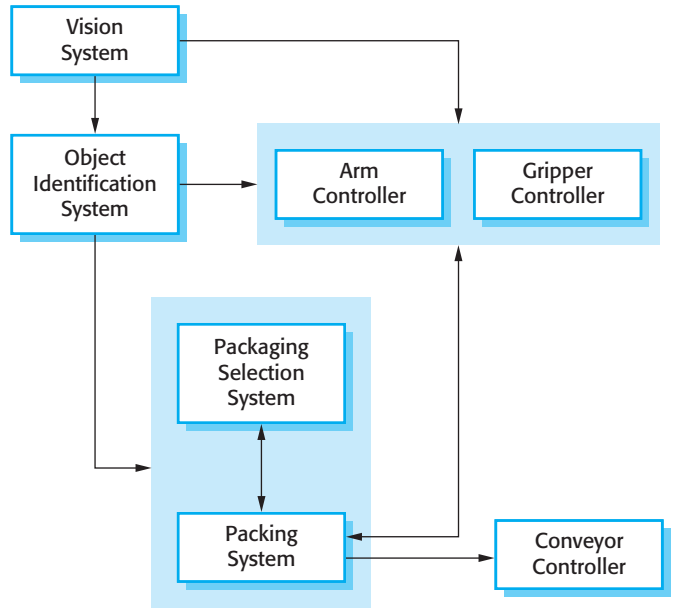


Figure 6.1 The architecture of a packing robot control system

Software architecture is important because it affects the performance, robustness, distributability, and maintainability of a system (Bosch, 2000). As Bosch discusses, individual components implement the functional system requirements. The non-functional requirements depend on the system architecture—the way in which these components are organized and communicate. In many systems, non-functional requirements are also influenced by individual components, but there is no doubt that the architecture of the system is the dominant influence.

Bass et al. (2003) discuss three advantages of explicitly designing and documenting software architecture:

1. *Stakeholder communication* The architecture is a high-level presentation of the system that may be used as a focus for discussion by a range of different stakeholders.
2. *System analysis* Making the system architecture explicit at an early stage in the system development requires some analysis. Architectural design decisions have a profound effect on whether or not the system can meet critical requirements such as performance, reliability, and maintainability.
3. *Large-scale reuse* A model of a system architecture is a compact, manageable description of how a system is organized and how the components interoperate. The system architecture is often the same for systems with similar requirements and so can support large-scale software reuse. As I explain in Chapter 16, it may be possible to develop product-line architectures where the same architecture is reused across a range of related systems.

Hofmeister et al. (2000) propose that a software architecture can serve firstly as a design plan for the negotiation of system requirements, and secondly as a means of structuring discussions with clients, developers, and managers. They also suggest that it is an essential tool for complexity management. It hides details and allows the designers to focus on the key system abstractions.

System architectures are often modeled using simple block diagrams, as in Figure 6.1. Each box in the diagram represents a component. Boxes within boxes indicate that the component has been decomposed to sub-components. Arrows mean that data and or control signals are passed from component to component in the direction of the arrows. You can see many examples of this type of architectural model in Booch's software architecture catalog (Booch, 2009).

Block diagrams present a high-level picture of the system structure, which people from different disciplines, who are involved in the system development process, can readily understand. However, in spite of their widespread use, Bass et al. (2003) dislike informal block diagrams for describing an architecture. They claim that these informal diagrams are poor architectural representations, as they show neither the type of the relationships among system components nor the components' externally visible properties.

The apparent contradictions between practice and architectural theory arise because there are two ways in which an architectural model of a program is used:

1. *As a way of facilitating discussion about the system design* A high-level architectural view of a system is useful for communication with system stakeholders and project planning because it is not cluttered with detail. Stakeholders can relate to it and understand an abstract view of the system. They can then discuss the system as a whole without being confused by detail. The architectural model identifies the key components that are to be developed so managers can start assigning people to plan the development of these systems.

2. *As a way of documenting an architecture that has been designed* The aim here is to produce a complete system model that shows the different components in a system, their interfaces, and their connections. The argument for this is that such a detailed architectural description makes it easier to understand and evolve the system.

Block diagrams are an appropriate way of describing the system architecture during the design process, as they are a good way of supporting communications between the people involved in the process. In many projects, these are often the only architectural documentation that exists. However, if the architecture of a system is to be thoroughly documented then it is better to use a notation with well-defined semantics for architectural description. However, as I discuss in Section 6.2, some people think that detailed documentation is neither useful, nor really worth the cost of its development.

6.1 Architectural design decisions

Architectural design is a creative process where you design a system organization that will satisfy the functional and non-functional requirements of a system. Because it is a creative process, the activities within the process depend on the type of system being developed, the background and experience of the system architect, and the specific requirements for the system. It is therefore useful to think of architectural design as a series of decisions to be made rather than a sequence of activities.

During the architectural design process, system architects have to make a number of structural decisions that profoundly affect the system and its development process. Based on their knowledge and experience, they have to consider the following fundamental questions about the system:

1. Is there a generic application architecture that can act as a template for the system that is being designed?
2. How will the system be distributed across a number of cores or processors?
3. What architectural patterns or styles might be used?
4. What will be the fundamental approach used to structure the system?
5. How will the structural components in the system be decomposed into subcomponents?
6. What strategy will be used to control the operation of the components in the system?
7. What architectural organization is best for delivering the non-functional requirements of the system?
8. How will the architectural design be evaluated?
9. How should the architecture of the system be documented?

Although each software system is unique, systems in the same application domain often have similar architectures that reflect the fundamental concepts of the domain. For example, application product lines are applications that are built around a core architecture with variants that satisfy specific customer requirements. When designing a system architecture, you have to decide what your system and broader application classes have in common, and decide how much knowledge from these application architectures you can reuse. I discuss generic application architectures in Section 6.4 and application product lines in Chapter 16.

For embedded systems and systems designed for personal computers, there is usually only a single processor and you will not have to design a distributed architecture for the system. However, most large systems are now distributed systems in which the system software is distributed across many different computers. The choice of distribution architecture is a key decision that affects the performance and

reliability of the system. This is a major topic in its own right and I cover it separately in Chapter 18.

The architecture of a software system may be based on a particular architectural pattern or style. An architectural pattern is a description of a system organization (Garlan and Shaw, 1993), such as a client–server organization or a layered architecture. Architectural patterns capture the essence of an architecture that has been used in different software systems. You should be aware of common patterns, where they can be used, and their strengths and weaknesses when making decisions about the architecture of a system. I discuss a number of frequently used patterns in Section 6.3.

Garlan and Shaw's notion of an architectural style (style and pattern have come to mean the same thing) covers questions 4 to 6 in the previous list. You have to choose the most appropriate structure, such as client–server or layered structuring, that will enable you to meet the system requirements. To decompose structural system units, you decide on the strategy for decomposing components into sub-components. The approaches that you can use allow different types of architecture to be implemented. Finally, in the control modeling process, you make decisions about how the execution of components is controlled. You develop a general model of the control relationships between the various parts of the system.

Because of the close relationship between non-functional requirements and software architecture, the particular architectural style and structure that you choose for a system should depend on the non-functional system requirements:

1. *Performance* If performance is a critical requirement, the architecture should be designed to localize critical operations within a small number of components, with these components all deployed on the same computer rather than distributed across the network. This may mean using a few relatively large components rather than small, fine-grain components, which reduces the number of component communications. You may also consider run-time system organizations that allow the system to be replicated and executed on different processors.
2. *Security* If security is a critical requirement, a layered structure for the architecture should be used, with the most critical assets protected in the innermost layers, with a high level of security validation applied to these layers.
3. *Safety* If safety is a critical requirement, the architecture should be designed so that safety-related operations are all located in either a single component or in a small number of components. This reduces the costs and problems of safety validation and makes it possible to provide related protection systems that can safely shut down the system in the event of failure.
4. *Availability* If availability is a critical requirement, the architecture should be designed to include redundant components so that it is possible to replace and update components without stopping the system. I describe two fault-tolerant system architectures for high-availability systems in Chapter 13.
5. *Maintainability* If maintainability is a critical requirement, the system architecture should be designed using fine-grain, self-contained components that may

readily be changed. Producers of data should be separated from consumers and shared data structures should be avoided.

Obviously there is potential conflict between some of these architectures. For example, using large components improves performance and using small, fine-grain components improves maintainability. If both performance and maintainability are important system requirements, then some compromise must be found. This can sometimes be achieved by using different architectural patterns or styles for different parts of the system.

Evaluating an architectural design is difficult because the true test of an architecture is how well the system meets its functional and non-functional requirements when it is in use. However, you can do some evaluation by comparing your design against reference architectures or generic architectural patterns. Bosch's (2000) description of the non-functional characteristics of architectural patterns can also be used to help with architectural evaluation.

6.2 Architectural views

I explained in the introduction to this chapter that architectural models of a software system can be used to focus discussion about the software requirements or design. Alternatively, they may be used to document a design so that it can be used as a basis for more detailed design and implementation, and for the future evolution of the system. In this section, I discuss two issues that are relevant to both of these:

1. What views or perspectives are useful when designing and documenting a system's architecture?
2. What notations should be used for describing architectural models?

It is impossible to represent all relevant information about a system's architecture in a single architectural model, as each model only shows one view or perspective of the system. It might show how a system is decomposed into modules, how the run-time processes interact, or the different ways in which system components are distributed across a network. All of these are useful at different times so, for both design and documentation, you usually need to present multiple views of the software architecture.

There are different opinions as to what views are required. Krutchen (1995), in his well-known 4+1 view model of software architecture, suggests that there should be four fundamental architectural views, which are related using use cases or scenarios. The views that he suggests are:

1. A logical view, which shows the key abstractions in the system as objects or object classes. It should be possible to relate the system requirements to entities in this logical view.

2. A process view, which shows how, at run-time, the system is composed of interacting processes. This view is useful for making judgments about non-functional system characteristics such as performance and availability.
3. A development view, which shows how the software is decomposed for development, that is, it shows the breakdown of the software into components that are implemented by a single developer or development team. This view is useful for software managers and programmers.
4. A physical view, which shows the system hardware and how software components are distributed across the processors in the system. This view is useful for systems engineers planning a system deployment.

Hofmeister et al. (2000) suggest the use of similar views but add to this the notion of a conceptual view. This view is an abstract view of the system that can be the basis for decomposing high-level requirements into more detailed specifications, help engineers make decisions about components that can be reused, and represent a product line (discussed in Chapter 16) rather than a single system. Figure 6.1, which describes the architecture of a packing robot, is an example of a conceptual system view.

In practice, conceptual views are almost always developed during the design process and are used to support architectural decision making. They are a way of communicating the essence of a system to different stakeholders. During the design process, some of the other views may also be developed when different aspects of the system are discussed, but there is no need for a complete description from all perspectives. It may also be possible to associate architectural patterns, discussed in the next section, with the different views of a system.

There are differing views about whether or not software architects should use the UML for architectural description (Clements, et al., 2002). A survey in 2006 (Lange et al., 2006) showed that, when the UML was used, it was mostly applied in a loose and informal way. The authors of that paper argued that this was a bad thing. I disagree with this view. The UML was designed for describing object-oriented systems and, at the architectural design stage, you often want to describe systems at a higher level of abstraction. Object classes are too close to the implementation to be useful for architectural description.

I don't find the UML to be useful during the design process itself and prefer informal notations that are quicker to write and which can be easily drawn on a whiteboard. The UML is of most value when you are documenting an architecture in detail or using model-driven development, as discussed in Chapter 5.

A number of researchers have proposed the use of more specialized architectural description languages (ADLs) (Bass et al., 2003) to describe system architectures. The basic elements of ADLs are components and connectors, and they include rules and guidelines for well-formed architectures. However, because of their specialized nature, domain and application specialists find it hard to understand and use ADLs. This makes it difficult to assess their usefulness for practical software engineering. ADLs designed for a particular domain (e.g., automobile systems) may be used as a

Name	MVC (Model-View-Controller)
Description	Separates presentation and interaction from the system data. The system is structured into three logical components that interact with each other. The Model component manages the system data and associated operations on that data. The View component defines and manages how the data is presented to the user. The Controller component manages user interaction (e.g., key presses, mouse clicks, etc.) and passes these interactions to the View and the Model. See Figure 6.3.
Example	Figure 6.4 shows the architecture of a web-based application system organized using the MVC pattern.
When used	Used when there are multiple ways to view and interact with data. Also used when the future requirements for interaction and presentation of data are unknown.
Advantages	Allows the data to change independently of its representation and vice versa. Supports presentation of the same data in different ways with changes made in one representation shown in all of them.
Disadvantages	Can involve additional code and code complexity when the data model and interactions are simple.

Figure 6.2 The model-view-controller (MVC) pattern

basis for model-driven development. However, I believe that informal models and notations, such as the UML, will remain the most commonly used ways of documenting system architectures.

Users of agile methods claim that detailed design documentation is mostly unused. It is, therefore, a waste of time and money to develop it. I largely agree with this view and I think that, for most systems, it is not worth developing a detailed architectural description from these four perspectives. You should develop the views that are useful for communication and not worry about whether or not your architectural documentation is complete. However, an exception to this is when you are developing critical systems, when you need to make a detailed dependability analysis of the system. You may need to convince external regulators that your system conforms to their regulations and so complete architectural documentation may be required.

6.3 Architectural patterns

The idea of patterns as a way of presenting, sharing, and reusing knowledge about software systems is now widely used. The trigger for this was the publication of a book on object-oriented design patterns (Gamma et al., 1995), which prompted the development of other types of pattern, such as patterns for organizational design (Coplien and Harrison, 2004), usability patterns (Usability Group, 1998), interaction (Martin and Sommerville, 2004), configuration management (Berczuk and

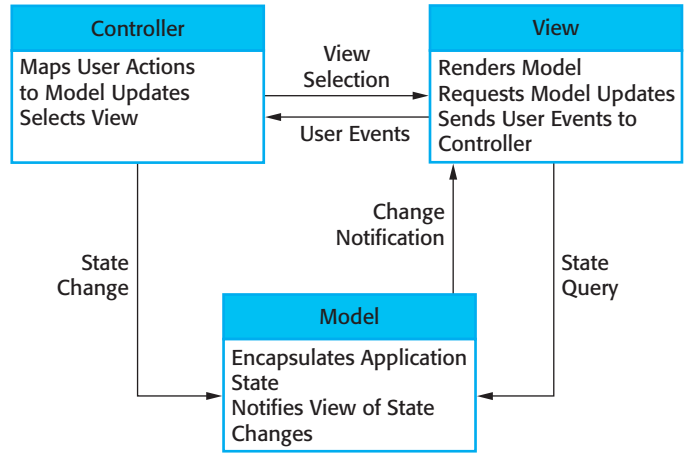


Figure 6.3 The organization of the MVC

Appleton, 2002), and so on. Architectural patterns were proposed in the 1990s under the name 'architectural styles' (Shaw and Garlan, 1996), with a five-volume series of handbooks on pattern-oriented software architecture published between 1996 and 2007 (Buschmann et al., 1996; Buschmann et al., 2007a; Buschmann et al., 2007b; Kircher and Jain, 2004; Schmidt et al., 2000).

In this section, I introduce architectural patterns and briefly describe a selection of architectural patterns that are commonly used in different types of systems. For more information about patterns and their use, you should refer to published pattern handbooks.

You can think of an architectural pattern as a stylized, abstract description of good practice, which has been tried and tested in different systems and environments. So, an architectural pattern should describe a system organization that has been successful in previous systems. It should include information of when it is and is not appropriate to use that pattern, and the pattern's strengths and weaknesses.

For example, Figure 6.2 describes the well-known Model-View-Controller pattern. This pattern is the basis of interaction management in many web-based systems. The stylized pattern description includes the pattern name, a brief description (with an associated graphical model), and an example of the type of system where the pattern is used (again, perhaps with a graphical model). You should also include information about when the pattern should be used and its advantages and disadvantages. Graphical models of the architecture associated with the MVC pattern are shown in Figures 6.3 and 6.4. These present the architecture from different views—Figure 6.3 is a conceptual view and Figure 6.4 shows a possible run-time architecture when this pattern is used for interaction management in a web-based system.

In a short section of a general chapter, it is impossible to describe all of the generic patterns that can be used in software development. Rather, I present some selected examples of patterns that are widely used and which capture good architectural design principles. I have included some further examples of generic architectural patterns on the book's web pages.

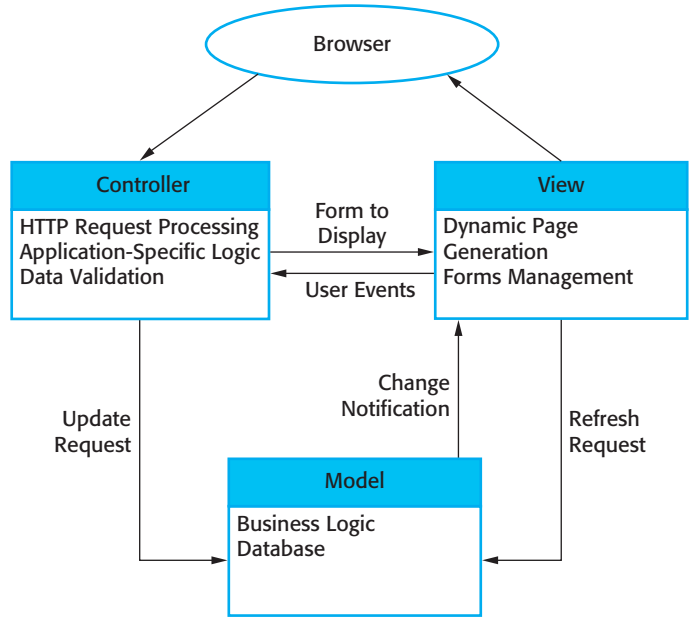


Figure 6.4 Web application architecture using the MVC pattern

6.3.1 Layered architecture

The notions of separation and independence are fundamental to architectural design because they allow changes to be localized. The MVC pattern, shown in Figure 6.2, separates elements of a system, allowing them to change independently. For example, adding a new view or changing an existing view can be done without any changes to the underlying data in the model. The layered architecture pattern is another way of achieving separation and independence. This pattern is shown in Figure 6.5. Here, the system functionality is organized into separate layers, and each layer only relies on the facilities and services offered by the layer immediately beneath it.

This layered approach supports the incremental development of systems. As a layer is developed, some of the services provided by that layer may be made available to users. The architecture is also changeable and portable. So long as its interface is unchanged, a layer can be replaced by another, equivalent layer. Furthermore, when layer interfaces change or new facilities are added to a layer, only the adjacent layer is affected. As layered systems localize machine dependencies in inner layers, this makes it easier to provide multi-platform implementations of an application system. Only the inner, machine-dependent layers need be re-implemented to take account of the facilities of a different operating system or database.

Figure 6.6 is an example of a layered architecture with four layers. The lowest layer includes system support software—typically database and operating system support. The next layer is the application layer that includes the components concerned with the application functionality and utility components that are used by other application components. The third layer is concerned with user interface

Name	Layered architecture
Description	Organizes the system into layers with related functionality associated with each layer. A layer provides services to the layer above it so the lowest-level layers represent core services that are likely to be used throughout the system. See Figure 6.6.
Example	A layered model of a system for sharing copyright documents held in different libraries, as shown in Figure 6.7.
When used	Used when building new facilities on top of existing systems; when the development is spread across several teams with each team responsibility for a layer of functionality; when there is a requirement for multi-level security.
Advantages	Allows replacement of entire layers so long as the interface is maintained. Redundant facilities (e.g., authentication) can be provided in each layer to increase the dependability of the system.
Disadvantages	In practice, providing a clean separation between layers is often difficult and a high-level layer may have to interact directly with lower-level layers rather than through the layer immediately below it. Performance can be a problem because of multiple levels of interpretation of a service request as it is processed at each layer.

Figure 6.5 The layered architecture pattern

management and providing user authentication and authorization, with the top layer providing user interface facilities. Of course, the number of layers is arbitrary. Any of the layers in Figure 6.6 could be split into two or more layers.

Figure 6.7 is an example of how this layered architecture pattern can be applied to a library system called LIBSYS, which allows controlled electronic access to copyright material from a group of university libraries. This has a five-layer architecture, with the bottom layer being the individual databases in each library.

You can see another example of the layered architecture pattern in Figure 6.17 (found in Section 6.4). This shows the organization of the system for mental health-care (MHC-PMS) that I have discussed in earlier chapters.

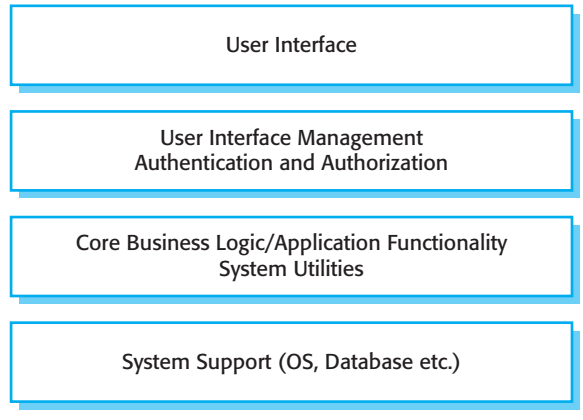


Figure 6.6 A generic layered architecture

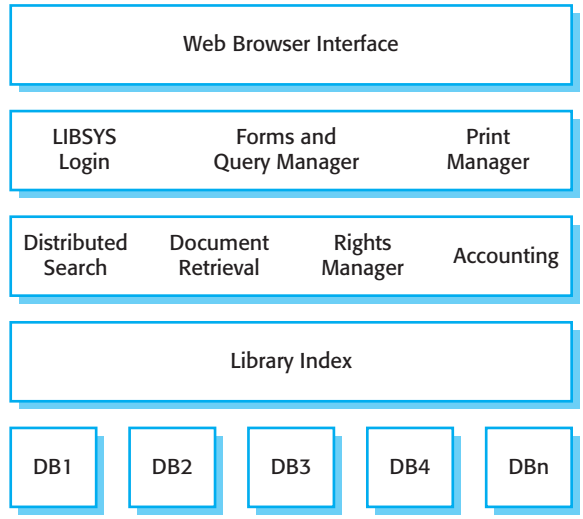


Figure 6.7 The architecture of the LIBSYS system

6.3.2 Repository architecture

The layered architecture and MVC patterns are examples of patterns where the view presented is the conceptual organization of a system. My next example, the Repository pattern (Figure 6.8), describes how a set of interacting components can share data.

Figure 6.8 The repository pattern

The majority of systems that use large amounts of data are organized around a shared database or repository. This model is therefore suited to applications in which

Name	Repository
Description	All data in a system is managed in a central repository that is accessible to all system components. Components do not interact directly, only through the repository.
Example	Figure 6.9 is an example of an IDE where the components use a repository of system design information. Each software tool generates information which is then available for use by other tools.
When used	You should use this pattern when you have a system in which large volumes of information are generated that has to be stored for a long time. You may also use it in data-driven systems where the inclusion of data in the repository triggers an action or tool.
Advantages	Components can be independent—they do not need to know of the existence of other components. Changes made by one component can be propagated to all components. All data can be managed consistently (e.g., backups done at the same time) as it is all in one place.
Disadvantages	The repository is a single point of failure so problems in the repository affect the whole system. May be inefficiencies in organizing all communication through the repository. Distributing the repository across several computers may be difficult.

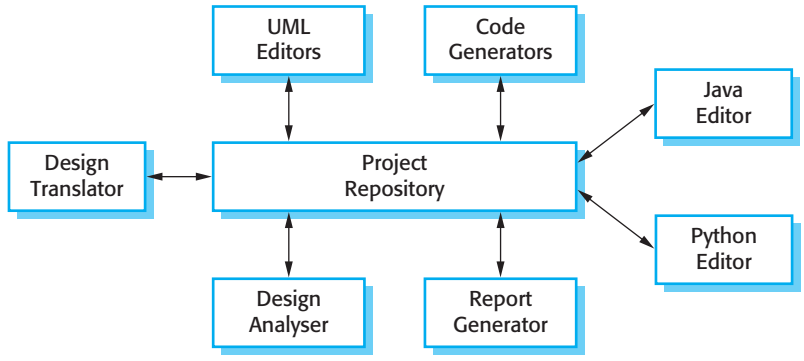


Figure 6.9 A repository architecture for an IDE

data is generated by one component and used by another. Examples of this type of system include command and control systems, management information systems, CAD systems, and interactive development environments for software.

Figure 6.9 is an illustration of a situation in which a repository might be used. This diagram shows an IDE that includes different tools to support model-driven development. The repository in this case might be a version-controlled environment (as discussed in Chapter 25) that keeps track of changes to software and allows roll-back to earlier versions.

Organizing tools around a repository is an efficient way to share large amounts of data. There is no need to transmit data explicitly from one component to another. However, components must operate around an agreed repository data model. Inevitably, this is a compromise between the specific needs of each tool and it may be difficult or impossible to integrate new components if their data models do not fit the agreed schema. In practice, it may be difficult to distribute the repository over a number of machines. Although it is possible to distribute a logically centralized repository, there may be problems with data redundancy and inconsistency.

In the example shown in Figure 6.9, the repository is passive and control is the responsibility of the components using the repository. An alternative approach, which has been derived for AI systems, uses a 'blackboard' model that triggers components when particular data become available. This is appropriate when the form of the repository data is less well structured. Decisions about which tool to activate can only be made when the data has been analyzed. This model is introduced by Nii (1986). Bosch (2000) includes a good discussion of how this style relates to system quality attributes.

6.3.3 Client–server architecture

The repository pattern is concerned with the static structure of a system and does not show its run-time organization. My next example illustrates a very commonly used run-time organization for distributed systems. The Client–server pattern is described in Figure 6.10.

Name	Client–server
Description	In a client–server architecture, the functionality of the system is organized into services, with each service delivered from a separate server. Clients are users of these services and access servers to make use of them.
Example	Figure 6.11 is an example of a film and video/DVD library organized as a client–server system.
When used	Used when data in a shared database has to be accessed from a range of locations. Because servers can be replicated, may also be used when the load on a system is variable.
Advantages	The principal advantage of this model is that servers can be distributed across a network. General functionality (e.g., a printing service) can be available to all clients and does not need to be implemented by all services.
Disadvantages	Each service is a single point of failure so susceptible to denial of service attacks or server failure. Performance may be unpredictable because it depends on the network as well as the system. May be management problems if servers are owned by different organizations.

Figure 6.10 The client–server pattern

A system that follows the client–server pattern is organized as a set of services and associated servers, and clients that access and use the services. The major components of this model are:

1. A set of servers that offer services to other components. Examples of servers include print servers that offer printing services, file servers that offer file management services, and a compile server, which offers programming language compilation services.
2. A set of clients that call on the services offered by servers. There will normally be several instances of a client program executing concurrently on different computers.
3. A network that allows the clients to access these services. Most client–server systems are implemented as distributed systems, connected using Internet protocols.

Client–server architectures are usually thought of as distributed systems architectures but the logical model of independent services running on separate servers can be implemented on a single computer. Again, an important benefit is separation and independence. Services and servers can be changed without affecting other parts of the system.

Clients may have to know the names of the available servers and the services that they provide. However, servers do not need to know the identity of clients or how many clients are accessing their services. Clients access the services provided by a server through remote procedure calls using a request-reply protocol such as the http

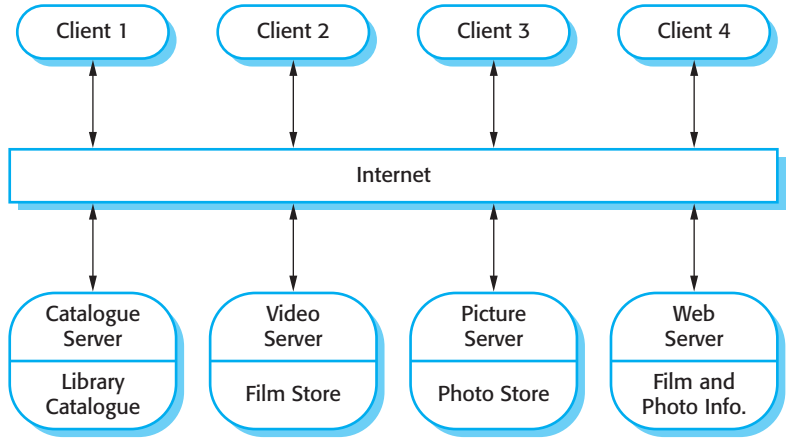


Figure 6.11 A client–server architecture for a film library

protocol used in the WWW. Essentially, a client makes a request to a server and waits until it receives a reply.

Figure 6.11 is an example of a system that is based on the client–server model. This is a multi-user, web-based system for providing a film and photograph library. In this system, several servers manage and display the different types of media. Video frames need to be transmitted quickly and in synchrony but at relatively low resolution. They may be compressed in a store, so the video server can handle video compression and decompression in different formats. Still pictures, however, must be maintained at a high resolution, so it is appropriate to maintain them on a separate server.

The catalog must be able to deal with a variety of queries and provide links into the web information system that includes data about the film and video clips, and an e-commerce system that supports the sale of photographs, film, and video clips. The

Figure 6.12 The pipe and filter pattern

Name	Pipe and filter
Description	The processing of the data in a system is organized so that each processing component (filter) is discrete and carries out one type of data transformation. The data flows (as in a pipe) from one component to another for processing.
Example	Figure 6.13 is an example of a pipe and filter system used for processing invoices.
When used	Commonly used in data processing applications (both batch- and transaction-based) where inputs are processed in separate stages to generate related outputs.
Advantages	Easy to understand and supports transformation reuse. Workflow style matches the structure of many business processes. Evolution by adding transformations is straightforward. Can be implemented as either a sequential or concurrent system.
Disadvantages	The format for data transfer has to be agreed upon between communicating transformations. Each transformation must parse its input and unparse its output to the agreed form. This increases system overhead and may mean that it is impossible to reuse functional transformations that use incompatible data structures.

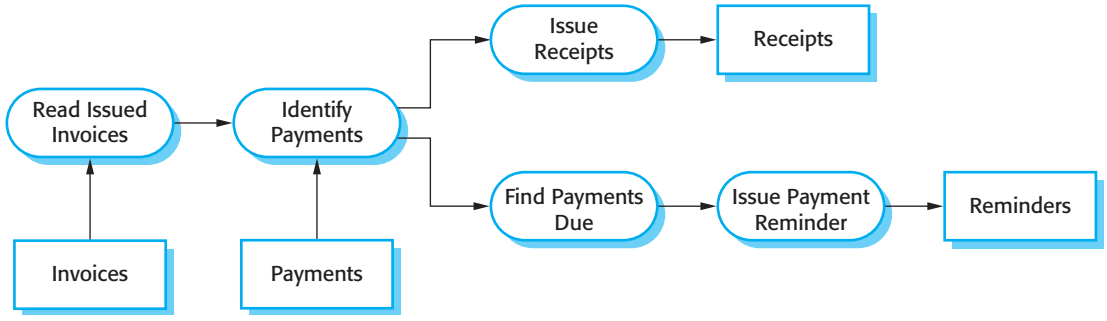


Figure 6.13 An example of the pipe and filter architecture

client program is simply an integrated user interface, constructed using a web browser, to access these services.

The most important advantage of the client–server model is that it is a distributed architecture. Effective use can be made of networked systems with many distributed processors. It is easy to add a new server and integrate it with the rest of the system or to upgrade servers transparently without affecting other parts of the system. I discuss distributed architectures, including client–server architectures and distributed object architectures, in Chapter 18.

6.3.4 Pipe and filter architecture

My final example of an architectural pattern is the pipe and filter pattern. This is a model of the run-time organization of a system where functional transformations process their inputs and produce outputs. Data flows from one to another and is transformed as it moves through the sequence. Each processing step is implemented as a transform. Input data flows through these transforms until converted to output. The transformations may execute sequentially or in parallel. The data can be processed by each transform item by item or in a single batch.

The name 'pipe and filter' comes from the original Unix system where it was possible to link processes using 'pipes'. These passed a text stream from one process to another. Systems that conform to this model can be implemented by combining Unix commands, using pipes and the control facilities of the Unix shell. The term 'filter' is used because a transformation 'filters out' the data it can process from its input data stream.

Variants of this pattern have been in use since computers were first used for automatic data processing. When transformations are sequential with data processed in batches, this pipe and filter architectural model becomes a batch sequential model, a common architecture for data processing systems (e.g., a billing system). The architecture of an embedded system may also be organized as a process pipeline, with each process executing concurrently. I discuss the use of this pattern in embedded systems in Chapter 20.

An example of this type of system architecture, used in a batch processing application, is shown in Figure 6.13. An organization has issued invoices to customers. Once a week, payments that have been made are reconciled with the invoices. For

Architectural patterns for control

There are specific architectural patterns that reflect commonly used ways of organizing control in a system. These include centralized control, based on one component calling other components, and event-based control, where the system reacts to external events.

http://www.SoftwareEngineering-9.com/Web/Architecture/ArchPatterns/

those invoices that have been paid, a receipt is issued. For those invoices that have not been paid within the allowed payment time, a reminder is issued.

Interactive systems are difficult to write using the pipe and filter model because of the need for a stream of data to be processed. Although simple textual input and output can be modeled in this way, graphical user interfaces have more complex I/O formats and a control strategy that is based on events such as mouse clicks or menu selections. It is difficult to translate this into a form compatible with the pipelining model.

6.4 Application architectures

Application systems are intended to meet a business or organizational need. All businesses have much in common—they need to hire people, issue invoices, keep accounts, and so on. Businesses operating in the same sector use common sector-specific applications. Therefore, as well as general business functions, all phone companies need systems to connect calls, manage their network, issue bills to customers, etc. Consequently, the application systems used by these businesses also have much in common.

These commonalities have led to the development of software architectures that describe the structure and organization of particular types of software systems. Application architectures encapsulate the principal characteristics of a class of systems. For example, in real-time systems, there might be generic architectural models of different system types, such as data collection systems or monitoring systems. Although instances of these systems differ in detail, the common architectural structure can be reused when developing new systems of the same type.

The application architecture may be re-implemented when developing new systems but, for many business systems, application reuse is possible without re-implementation. We see this in the growth of Enterprise Resource Planning (ERP) systems from companies such as SAP and Oracle, and vertical software packages (COTS) for specialized applications in different areas of business. In these systems, a generic system is configured and adapted to create a specific business application.

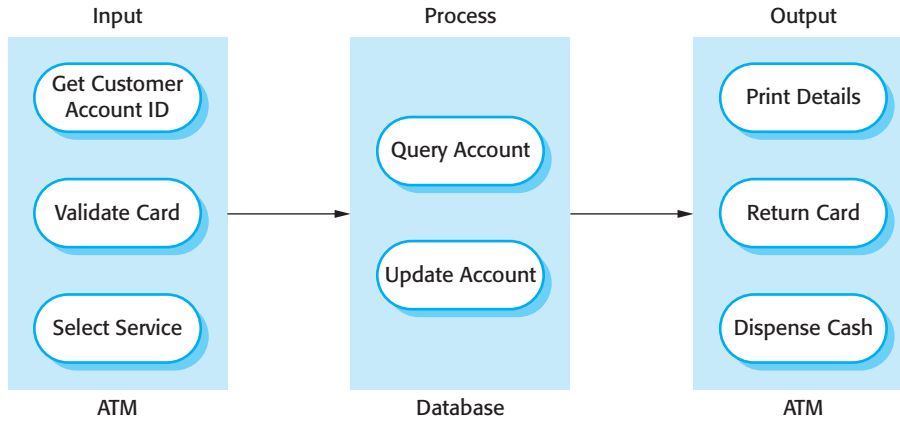


Figure 6.15 The software architecture of an ATM system

has ensured that the transaction is properly completed, it signals to the application that processing has finished.

Transaction processing systems may be organized as a 'pipe and filter' architecture with system components responsible for input, processing, and output. For example, consider a banking system that allows customers to query their accounts and withdraw cash from an ATM. The system is composed of two cooperating software components—the ATM software and the account processing software in the bank's database server. The input and output components are implemented as software in the ATM and the processing component is part of the bank's database server. Figure 6.15 shows the architecture of this system, illustrating the functions of the input, process, and output components.

6.4.2 Information systems

All systems that involve interaction with a shared database can be considered to be transaction-based information systems. An information system allows controlled access to a large base of information, such as a library catalog, a flight timetable, or the records of patients in a hospital. Increasingly, information systems are web-based systems that are accessed through a web browser.

Figure 6.16 a very general model of an information system. The system is modeled using a layered approach (discussed in Section 6.3) where the top layer supports the user interface and the bottom layer is the system database. The user communications layer handles all input and output from the user interface, and the information retrieval layer includes application-specific logic for accessing and updating the database. As we shall see later, the layers in this model can map directly onto servers in an Internet-based system.

As an example of an instantiation of this layered model, Figure 6.17 shows the architecture of the MHC-PMS. Recall that this system maintains and manages details of patients who are consulting specialist doctors about mental health problems. I have

Software design and implementation is the stage in the software engineering process at which an executable software system is developed. For some simple systems, software design and implementation is software engineering, and all other activities are merged with this process. However, for large systems, software design and implementation is only one of a set of processes (requirements engineering, verification and validation, etc.) involved in software engineering.

Software design and implementation activities are invariably interleaved. Software design is a creative activity in which you identify software components and their relationships, based on a customer's requirements. Implementation is the process of realizing the design as a program. Sometimes, there is a separate design stage and this design is modeled and documented. At other times, a design is in the programmer's head or roughly sketched on a whiteboard or sheets of paper. Design is about how to solve a problem, so there is always a design process. However, it isn't always necessary or appropriate to describe the design in detail using the UML or other design description language.

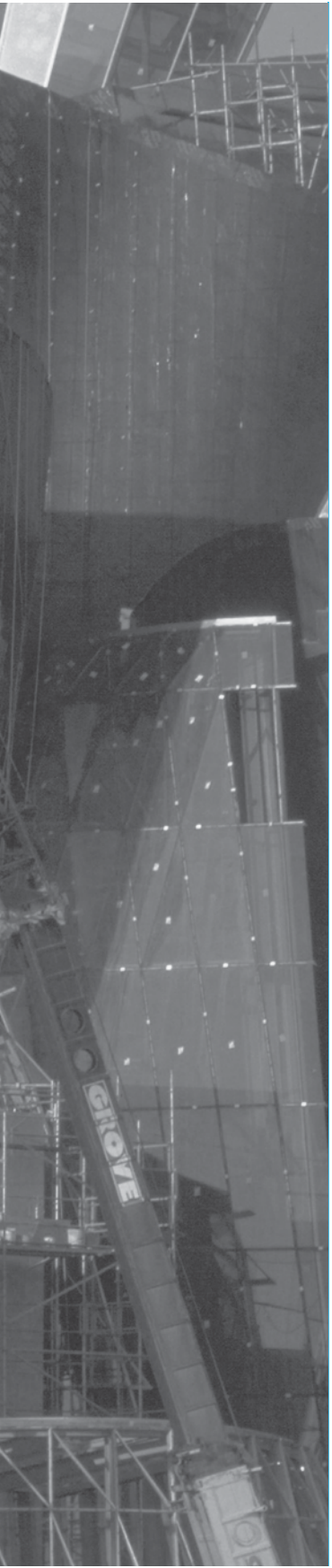
Design and implementation are closely linked and you should normally take implementation issues into account when developing a design. For example, using the UML to document a design may be the right thing to do if you are programming in an object-oriented language such as Java or C#. It is less useful, I think, if you are developing in a dynamically typed language like Python and makes no sense at all if you are implementing your system by configuring an off-the-shelf package. As I discussed in Chapter 3, agile methods usually work from informal sketches of the design and leave many design decisions to programmers.

One of the most important implementation decisions that has to be made at an early stage of a software project is whether or not you should buy or build the application software. In a wide range of domains, it is now possible to buy off-the-shelf systems (COTS) that can be adapted and tailored to the users' requirements. For example, if you want to implement a medical records system, you can buy a package that is already used in hospitals. It can be cheaper and faster to use this approach rather than developing a system in a conventional programming language.

When you develop an application in this way, the design process becomes concerned with how to use the configuration features of that system to deliver the system requirements. You don't usually develop design models of the system, such as models of the system objects and their interactions. I discuss this COTS-based approach to development in Chapter 16.

I assume that most readers of this book will have had experience of program design and implementation. This is something that you acquire as you learn to program and master the elements of a programming language like Java or Python. You will have probably learned about good programming practice in the programming languages that you have studied, as well as how to debug programs that you have developed. Therefore, I don't cover programming topics here. Instead, this chapter has two aims:

1. To show how system modeling and architectural design (covered in Chapters 5 and 6) are put into practice in developing an object-oriented software design.



8

Software testing

Objectives

The objective of this chapter is to introduce software testing and software testing processes. When you have read the chapter, you will:

- understand the stages of testing from testing, during development to acceptance testing by system customers;
- have been introduced to techniques that help you choose test cases that are geared to discovering program defects;
- understand test-first development, where you design tests before writing code and run these tests automatically;
- know the important differences between component, system, and release testing and be aware of user testing processes and techniques.

Contents

Testing is intended to show that a program does what it is intended to do and to discover program defects before it is put into use. When you test software, you execute a program using artificial data. You check the results of the test run for errors, anomalies, or information about the program's non-functional attributes.

The testing process has two distinct goals:

1. To demonstrate to the developer and the customer that the software meets its requirements. For custom software, this means that there should be at least one test for every requirement in the requirements document. For generic software products, it means that there should be tests for all of the system features, plus combinations of these features, that will be incorporated in the product release.
2. To discover situations in which the behavior of the software is incorrect, undesirable, or does not conform to its specification. These are a consequence of software defects. Defect testing is concerned with rooting out undesirable system behavior such as system crashes, unwanted interactions with other systems, incorrect computations, and data corruption.

The first goal leads to validation testing, where you expect the system to perform correctly using a given set of test cases that reflect the system's expected use. The second goal leads to defect testing, where the test cases are designed to expose defects. The test cases in defect testing can be deliberately obscure and need not reflect how the system is normally used. Of course, there is no definite boundary between these two approaches to testing. During validation testing, you will find defects in the system; during defect testing, some of the tests will show that the program meets its requirements.

The diagram shown in Figure 8.1 may help to explain the differences between validation testing and defect testing. Think of the system being tested as a black box. The system accepts inputs from some input set I and generates outputs in an output set O. Some of the outputs will be erroneous. These are the outputs in set O_e that are generated by the system in response to inputs in the set I_e. The priority in defect testing is to find those inputs in the set I_e because these reveal problems with the system. Validation testing involves testing with correct inputs that are outside I_e. These stimulate the system to generate the expected correct outputs.

Testing cannot demonstrate that the software is free of defects or that it will behave as specified in every circumstance. It is always possible that a test that you have overlooked could discover further problems with the system. As Edsger Dijkstra, an early contributor to the development of software engineering, eloquently stated (Dijkstra et al., 1972):

Testing can only show the presence of errors, not their absence

Testing is part of a broader process of software verification and validation (V & V). Verification and validation are not the same thing, although they are often confused.

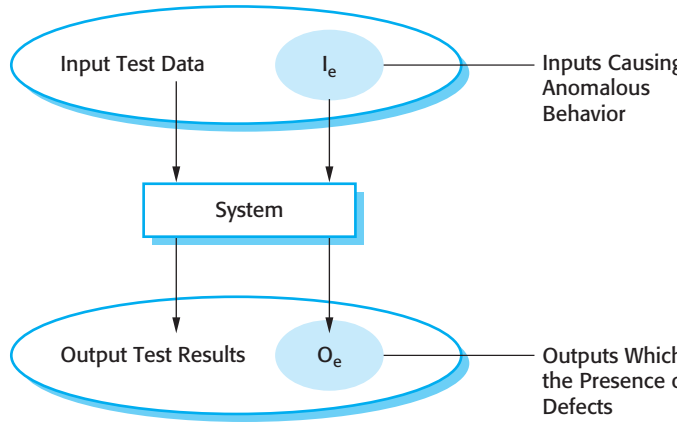


Figure 8.1 An input-output model of program testing

Barry Boehm, a pioneer of software engineering, succinctly expressed the difference between them (Boehm, 1979):

- 'Validation: Are we building the right product?'
- 'Verification: Are we building the product right?'

Verification and validation processes are concerned with checking that software being developed meets its specification and delivers the functionality expected by the people paying for the software. These checking processes start as soon as requirements become available and continue through all stages of the development process.

The aim of verification is to check that the software meets its stated functional and non-functional requirements. Validation, however, is a more general process. The aim of validation is to ensure that the software meets the customer's expectations. It goes beyond simply checking conformance with the specification to demonstrating that the software does what the customer expects it to do. Validation is essential because, as I discussed in Chapter 4, requirements specifications do not always reflect the real wishes or needs of system customers and users.

The ultimate goal of verification and validation processes is to establish confidence that the software system is 'fit for purpose'. This means that the system must be good enough for its intended use. The level of required confidence depends on the system's purpose, the expectations of the system users, and the current marketing environment for the system:

1. *Software purpose* The more critical the software, the more important that it is reliable. For example, the level of confidence required for software used to control a safety-critical system is much higher than that required for a prototype that has been developed to demonstrate new product ideas.
2. *User expectations* Because of their experiences with buggy, unreliable software, many users have low expectations of software quality. They are not surprised when their software fails. When a new system is installed, users may tolerate

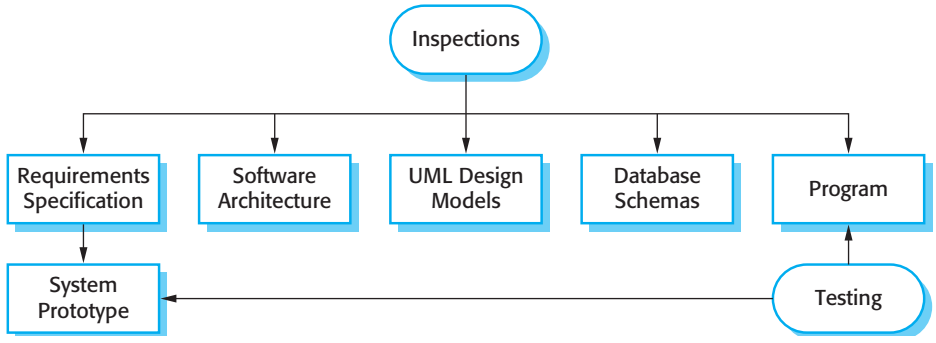


Figure 8.2 Inspections and testing

failures because the benefits of use outweigh the costs of failure recovery. In these situations, you may not need to devote as much time to testing the software. However, as software matures, users expect it to become more reliable so more thorough testing of later versions may be required.

3. *Marketing environment* When a system is marketed, the sellers of the system must take into account competing products, the price that customers are willing to pay for a system, and the required schedule for delivering that system. In a competitive environment, a software company may decide to release a program before it has been fully tested and debugged because they want to be the first into the market. If a software product is very cheap, users may be willing to tolerate a lower level of reliability.

As well as software testing, the verification and validation process may involve software inspections and reviews. Inspections and reviews analyze and check the system requirements, design models, the program source code, and even proposed system tests. These are so-called 'static' V & V techniques in which you don't need to execute the software to verify it. Figure 8.2 shows that software inspections and testing support V & V at different stages in the software process. The arrows indicate the stages in the process where the techniques may be used.

Inspections mostly focus on the source code of a system but any readable representation of the software, such as its requirements or a design model, can be inspected. When you inspect a system, you use knowledge of the system, its application domain, and the programming or modeling language to discover errors.

There are three advantages of software inspection over testing:

1. During testing, errors can mask (hide) other errors. When an error leads to unexpected outputs, you can never be sure if later output anomalies are due to a new error or are side effects of the original error. Because inspection is a static process, you don't have to be concerned with interactions between errors. Consequently, a single inspection session can discover many errors in a system.

Test planning

Test planning is concerned with scheduling and resourcing all of the activities in the testing process. It involves defining the testing process, taking into account the people and the time available. Usually, a test plan will be created, which defines what is to be tested, the predicted testing schedule, and how tests will be recorded. For critical systems, the test plan may also include details of the tests to be run on the software.

http://www.SoftwareEngineering-9.com/Web/Testing/Planning.html

2. Incomplete versions of a system can be inspected without additional costs. If a program is incomplete, then you need to develop specialized test harnesses to test the parts that are available. This obviously adds to the system development costs.

3. As well as searching for program defects, an inspection can also consider broader quality attributes of a program, such as compliance with standards, portability, and maintainability. You can look for inefficiencies, inappropriate algorithms, and poor programming style that could make the system difficult to maintain and update.

Program inspections are an old idea and there have been several studies and experiments that have demonstrated that inspections are more effective for defect discovery than program testing. Fagan (1986) reported that more than 60% of the errors in a program can be detected using informal program inspections. In the Cleanroom process (Prowell et al., 1999), it is claimed that more than 90% of defects can be discovered in program inspections.

However, inspections cannot replace software testing. Inspections are not good for discovering defects that arise because of unexpected interactions between different parts of a program, timing problems, or problems with system performance. Furthermore, especially in small companies or development groups, it can be difficult and expensive to put together a separate inspection team as all potential members of the team may also be software developers. I discuss reviews and inspections in more detail in Chapter 24 (Quality Management). Automated static analysis, where the source text of a program is automatically analyzed to discover anomalies, is explained in Chapter 15. In this chapter, I focus on testing and testing processes.

Figure 8.3 is an abstract model of the 'traditional' testing process, as used in plan-driven development. Test cases are specifications of the inputs to the test and the expected output from the system (the test results), plus a statement of what is being tested. Test data are the inputs that have been devised to test a system. Test data can sometimes be generated automatically, but automatic test case generation is impossible, as people who understand what the system is supposed to do must be involved to specify the expected test results. However, test execution can be automated. The expected results are automatically compared with the predicted results so there is no need for a person to look for errors and anomalies in the test run.

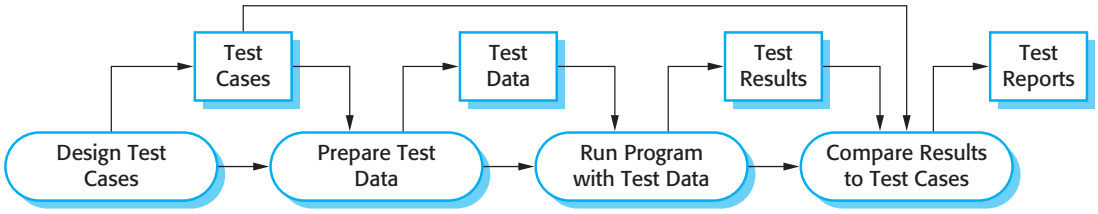


Figure 8.3 A model of the software testing process

Typically, a commercial software system has to go through three stages of testing:

1. Development testing, where the system is tested during development to discover bugs and defects. System designers and programmers are likely to be involved in the testing process.
2. Release testing, where a separate testing team tests a complete version of the system before it is released to users. The aim of release testing is to check that the system meets the requirements of system stakeholders.
3. User testing, where users or potential users of a system test the system in their own environment. For software products, the 'user' may be an internal marketing group who decide if the software can be marketed, released, and sold. Acceptance testing is one type of user testing where the customer formally tests a system to decide if it should be accepted from the system supplier or if further development is required.

In practice, the testing process usually involves a mixture of manual and automated testing. In manual testing, a tester runs the program with some test data and compares the results to their expectations. They note and report discrepancies to the program developers. In automated testing, the tests are encoded in a program that is run each time the system under development is to be tested. This is usually faster than manual testing, especially when it involves regression testing—re-running previous tests to check that changes to the program have not introduced new bugs.

The use of automated testing has increased considerably over the past few years. However, testing can never be completely automated as automated tests can only check that a program does what it is supposed to do. It is practically impossible to use automated testing to test systems that depend on how things look (e.g., a graphical user interface), or to test that a program does not have unwanted side effects.

8.1 Development testing

Development testing includes all testing activities that are carried out by the team developing the system. The tester of the software is usually the programmer who developed that software, although this is not always the case. Some development processes use programmer/tester pairs (Cusamano and Selby, 1998) where each

Debugging

Debugging is the process of fixing errors and problems that have been discovered by testing. Using information from the program tests, debuggers use their knowledge of the programming language and the intended outcome of the test to locate and repair the program error. This process is often supported by interactive debugging tools that provide extra information about program execution.

http://www.SoftwareEngineering-9.com/Web/Testing/Debugging.html

programmer has an associated tester who develops tests and assists with the testing process. For critical systems, a more formal process may be used, with a separate testing group within the development team. They are responsible for developing tests and maintaining detailed records of test results.

During development, testing may be carried out at three levels of granularity:

1. Unit testing, where individual program units or object classes are tested. Unit testing should focus on testing the functionality of objects or methods.
2. Component testing, where several individual units are integrated to create composite components. Component testing should focus on testing component interfaces.
3. System testing, where some or all of the components in a system are integrated and the system is tested as a whole. System testing should focus on testing component interactions.

Development testing is primarily a defect testing process, where the aim of testing is to discover bugs in the software. It is therefore usually interleaved with debugging—the process of locating problems with the code and changing the program to fix these problems.

8.1.1 Unit testing

Unit testing is the process of testing program components, such as methods or object classes. Individual functions or methods are the simplest type of component. Your tests should be calls to these routines with different input parameters. You can use the approaches to test case design discussed in Section 8.1.2, to design the function or method tests.

When you are testing object classes, you should design your tests to provide coverage of all of the features of the object. This means that you should:

- test all operations associated with the object;
- set and check the value of all attributes associated with the object;
- put the object into all possible states. This means that you should simulate all events that cause a state change.

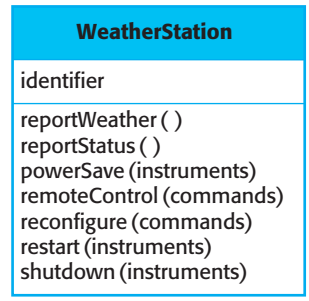


Figure 8.4 The weather station object interface

Consider, for example, the weather station object from the example that I discussed in Chapter 7. The interface of this object is shown in Figure 8.4. It has a single attribute, which is its identifier. This is a constant that is set when the weather station is installed. You therefore only need a test that checks if it has been properly set up. You need to define test cases for all of the methods associated with the object such as reportWeather, reportStatus, etc. Ideally, you should test methods in isolation but, in some cases, some test sequences are necessary. For example, to test the method that shuts down the weather station instruments (shutdown), you need to have executed the restart method.

Generalization or inheritance makes object class testing more complicated. You can't simply test an operation in the class where it is defined and assume that it will work as expected in the subclasses that inherit the operation. The operation that is inherited may make assumptions about other operations and attributes. These may not be valid in some subclasses that inherit the operation. You therefore have to test the inherited operation in all of the contexts where it is used.

To test the states of the weather station, you use a state model, such as the one shown in Figure 7.8 in the previous chapter. Using this model, you can identify sequences of state transitions that have to be tested and define event sequences to force these transitions. In principle, you should test every possible state transition sequence, although in practice this may be too expensive. Examples of state sequences that should be tested in the weather station include:

Shutdown → Running → Shutdown
Configuring → Running → Testing → Transmitting → Running
Running → Collecting → Running → Summarizing → Transmitting → Running

Whenever possible, you should automate unit testing. In automated unit testing, you make use of a test automation framework (such as JUnit) to write and run your program tests. Unit testing frameworks provide generic test classes that you extend to create specific test cases. They can then run all of the tests that you have implemented and report, often through some GUI, on the success or failure of the tests. An entire test suite can often be run in a few seconds so it is possible to execute all the tests every time you make a change to the program.

An automated test has three parts:

1. A setup part, where you initialize the system with the test case, namely the inputs and expected outputs.

2. A call part, where you call the object or method to be tested.
3. An assertion part where you compare the result of the call with the expected result. If the assertion evaluates to true, the test has been successful; if false, then it has failed.

Sometimes the object that you are testing has dependencies on other objects that may not have been written or which slow down the testing process if they are used. For example, if your object calls a database, this may involve a slow setup process before it can be used. In these cases, you may decide to use mock objects. Mock objects are objects with the same interface as the external objects being used that simulate its functionality. Therefore, a mock object simulating a database may have only a few data items that are organized in an array. They can therefore be accessed quickly, without the overheads of calling a database and accessing disks. Similarly, mock objects can be used to simulate abnormal operation or rare events. For example, if your system is intended to take action at certain times of day, your mock object can simply return those times, irrespective of the actual clock time.

8.1.2 Choosing unit test cases

Testing is expensive and time consuming, so it is important that you choose effective unit test cases. Effectiveness, in this case, means two things:

1. The test cases should show that, when used as expected, the component that you are testing does what it is supposed to do.
2. If there are defects in the component, these should be revealed by test cases.

You should therefore write two kinds of test case. The first of these should reflect normal operation of a program and should show that the component works. For example, if you are testing a component that creates and initializes a new patient record, then your test case should show that the record exists in a database and that its fields have been set as specified. The other kind of test case should be based on testing experience of where common problems arise. It should use abnormal inputs to check that these are properly processed and do not crash the component.

I discuss two possible strategies here that can be effective in helping you choose test cases. These are:

1. Partition testing, where you identify groups of inputs that have common characteristics and should be processed in the same way. You should choose tests from within each of these groups.
2. Guideline-based testing, where you use testing guidelines to choose test cases. These guidelines reflect previous experience of the kinds of errors that programmers often make when developing components.

Path testing

Path testing is a testing strategy that aims to exercise every independent execution path through a component or program. If every independent path is executed, then all statements in the component must have been executed at least once. All conditional statements are tested for both true and false cases. In an object-oriented development process, path testing may be used when testing the methods associated with objects.

http://www.SoftwareEngineering-9.com/Web/Testing/PathTest.html

Whittaker's book (2002) includes many examples of guidelines that can be used in test case design. Some of the most general guidelines that he suggests are:

- Choose inputs that force the system to generate all error messages;
- Design inputs that cause input buffers to overflow;
- Repeat the same input or series of inputs numerous times;
- Force invalid outputs to be generated;
- Force computation results to be too large or too small.

As you gain experience with testing, you can develop your own guidelines about how to choose effective test cases. I give more examples of testing guidelines in the next section of this chapter.

8.1.3 Component testing

Software components are often composite components that are made up of several interacting objects. For example, in the weather station system, the reconfiguration component includes objects that deal with each aspect of the reconfiguration. You access the functionality of these objects through the defined component interface. Testing composite components should therefore focus on showing that the component interface behaves according to its specification. You can assume that unit tests on the individual objects within the component have been completed.

Figure 8.7 illustrates the idea of component interface testing. Assume that components A, B, and C have been integrated to create a larger component or subsystem. The test cases are not applied to the individual components but rather to the interface of the composite component created by combining these components. Interface errors in the composite component may not be detectable by testing the individual objects because these errors result from interactions between the objects in the component.

There are different types of interface between program components and, consequently, different types of interface error that can occur:

1. *Parameter interfaces* These are interfaces in which data or sometimes function references are passed from one component to another. Methods in an object have a parameter interface.

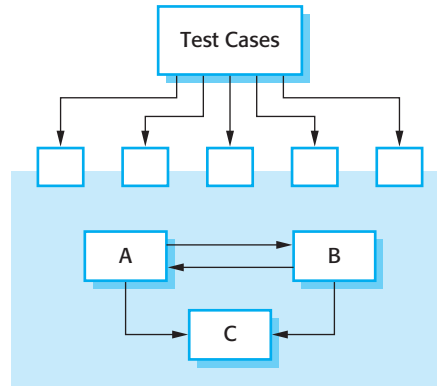


Figure 8.7 Interface testing

2. *Shared memory interfaces* These are interfaces in which a block of memory is shared between components. Data is placed in the memory by one subsystem and retrieved from there by other sub-systems. This type of interface is often used in embedded systems, where sensors create data that is retrieved and processed by other system components.

3. *Procedural interfaces* These are interfaces in which one component encapsulates a set of procedures that can be called by other components. Objects and reusable components have this form of interface.

4. *Message passing interfaces* These are interfaces in which one component requests a service from another component by passing a message to it. A return message includes the results of executing the service. Some object-oriented systems have this form of interface, as do client–server systems.

Interface errors are one of the most common forms of error in complex systems (Lutz, 1993). These errors fall into three classes:

- *Interface misuse* A calling component calls some other component and makes an error in the use of its interface. This type of error is common with parameter interfaces, where parameters may be of the wrong type or be passed in the wrong order, or the wrong number of parameters may be passed.
- *Interface misunderstanding* A calling component misunderstands the specification of the interface of the called component and makes assumptions about its behavior. The called component does not behave as expected which then causes unexpected behavior in the calling component. For example, a binary search method may be called with a parameter that is an unordered array. The search would then fail.
- *Timing errors* These occur in real-time systems that use a shared memory or a message-passing interface. The producer of data and the consumer of data may

Incremental integration and testing

System testing involves integrating different components then testing the integrated system that you have created. You should always use an incremental approach to integration and testing (i.e., you should integrate a component, test the system, integrate another component, test again, and so on). This means that if problems occur, it is probably due to interactions with the most recently integrated component.

Incremental integration and testing is fundamental to agile methods such as XP, where regression tests (see Section 8.2) are run every time a new increment is integrated.

http://www.SoftwareEngineering-9.com/Web/Testing/Integration.html

8.1.4 System testing

System testing during development involves integrating components to create a version of the system and then testing the integrated system. System testing checks that components are compatible, interact correctly and transfer the right data at the right time across their interfaces. It obviously overlaps with component testing but there are two important differences:

1. During system testing, reusable components that have been separately developed and off-the-shelf systems may be integrated with newly developed components. The complete system is then tested.
2. Components developed by different team members or groups may be integrated at this stage. System testing is a collective rather than an individual process. In some companies, system testing may involve a separate testing team with no involvement from designers and programmers.

When you integrate components to create a system, you get emergent behavior. This means that some elements of system functionality only become obvious when you put the components together. This may be planned emergent behavior, which has to be tested. For example, you may integrate an authentication component with a component that updates information. You then have a system feature that restricts information updating to authorized users. Sometimes, however, the emergent behavior is unplanned and unwanted. You have to develop tests that check that the system is only doing what it is supposed to do.

Therefore system testing should focus on testing the interactions between the components and objects that make up a system. You may also test reusable components or systems to check that they work as expected when they are integrated with new components. This interaction testing should discover those component bugs that are only revealed when a component is used by other components in the system. Interaction testing also helps find misunderstandings, made by component developers, about other components in the system.

Because of its focus on interactions, use case–based testing is an effective approach to system testing. Typically, each use case is implemented by several components or objects in the system. Testing the use case forces these interactions to

Software development does not stop when a system is delivered but continues throughout the lifetime of the system. After a system has been deployed, it inevitably has to change if it is to remain useful. Business changes and changes to user expectations generate new requirements for the existing software. Parts of the software may have to be modified to correct errors that are found in operation, to adapt it for changes to its hardware and software platform, and to improve its performance or other non-functional characteristics.

Software evolution is important because organizations have invested large amounts of money in their software and are now completely dependent on these systems. Their systems are critical business assets and they have to invest in system change to maintain the value of these assets. Consequently, most large companies spend more on maintaining existing systems than on new systems development. Based on an informal industry poll, Erlikh (2000) suggests that 85–90% of organizational software costs are evolution costs. Other surveys suggest that about two-thirds of software costs are evolution costs. For sure, the costs of software change are a large part of the IT budget for all companies.

Software evolution may be triggered by changing business requirements, by reports of software defects, or by changes to other systems in a software system's environment. Hopkins and Jenkins (2008) have coined the term 'brownfield software development' to describe situations in which software systems have to be developed and managed in an environment where they are dependent on many other software systems.

Therefore, the evolution of a system can rarely be considered in isolation. Changes to the environment lead to system change that may then trigger further environmental changes. Of course, the fact that systems have to evolve in a 'systems-rich' environment often increases the difficulties and costs of evolution. As well as understanding and analyzing an impact of a proposed change on the system itself, you may also have to assess how this may affect other systems in the operational environment.

Useful software systems often have a very long lifetime. For example, large military or infrastructure systems, such as air traffic control systems, may have a lifetime of 30 years or more. Business systems are often more than 10 years old. Software cost a lot of money so a company has to use a software system for many years to get a return on its investment. Obviously, the requirements of the installed systems change as the business and its environment change. Therefore, new releases of the systems, incorporating changes, and updates, are usually created at regular intervals.

You should, therefore, think of software engineering as a spiral process with requirements, design, implementation, and testing going on throughout the lifetime of the system (Figure 9.1). You start by creating release 1 of the system. Once delivered, changes are proposed and the development of release 2 starts almost immediately. In fact, the need for evolution may become obvious even before the system is deployed so that later releases of the software may be under development before the current version has been released.

This model of software evolution implies that a single organization is responsible for both the initial software development and the evolution of the software. Most

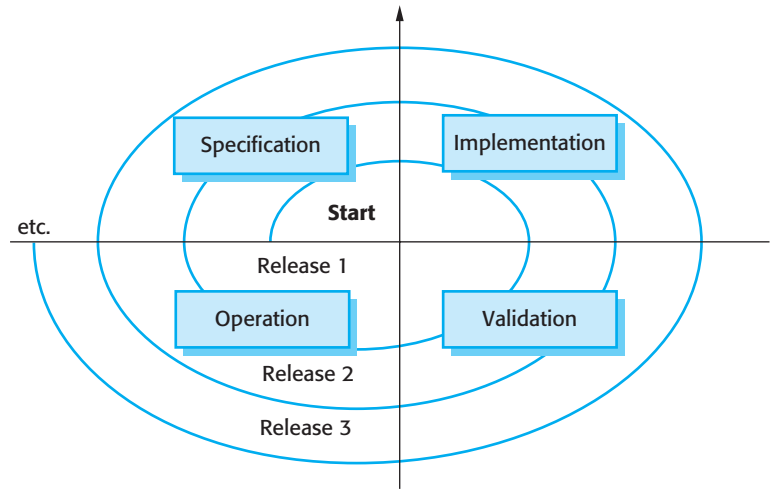


Figure 9.1 A spiral model of development and evolution

packaged software products are developed using this approach. For custom software, a different approach is commonly used. A software company develops software for a customer and the customer's own development staff then take over the system. They are responsible for software evolution. Alternatively, the software customer might issue a separate contract to a different company for system support and evolution.

In this case, there are likely to be discontinuities in the spiral process. Requirements and design documents may not be passed from one company to another. Companies may merge or reorganize and inherit software from other companies, and then find that this has to be changed. When the transition from development to evolution is not seamless, the process of changing the software after delivery is often called 'software maintenance'. As I discuss later in this chapter, maintenance involves extra process activities, such as program understanding, in addition to the normal activities of software development.

Rajlich and Bennett (2000) proposed an alternative view of the software evolution life cycle, as shown in Figure 9.2. In this model, they distinguish between evolution and servicing. Evolution is the phase in which significant changes to the software architecture and functionality may be made. During servicing, the only changes that are made are relatively small, essential changes.

During evolution, the software is used successfully and there is a constant stream of proposed requirements changes. However, as the software is modified, its structure tends to degrade and changes become more and more expensive. This often happens after a few years of use when other environmental changes, such as hardware and operating systems, are also often required. At some stage in the life cycle, the software reaches a transition point where significant changes, implementing new requirements, become less and less cost effective.

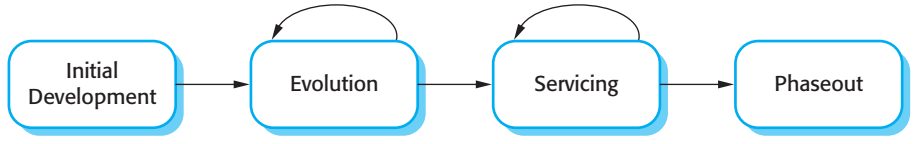


Figure 9.2 Evolution and servicing

1. *Fault repairs* Coding errors are usually relatively cheap to correct; design errors are more expensive as they may involve rewriting several program components. Requirements errors are the most expensive to repair because of the extensive system redesign which may be necessary.
2. *Environmental adaptation* This type of maintenance is required when some aspect of the system's environment such as the hardware, the platform operating system, or other support software changes. The application system must be modified to adapt it to cope with these environmental changes.
3. *Functionality addition* This type of maintenance is necessary when the system requirements change in response to organizational or business change. The scale of the changes required to the software is often much greater than for the other types of maintenance.

In practice, there is not a clear-cut distinction between these types of maintenance. When you adapt the system to a new environment, you may add functionality to take advantage of new environmental features. Software faults are often exposed because users use the system in unanticipated ways. Changing the system to accommodate their way of working is the best way to fix these faults.

These types of maintenance are generally recognized but different people sometimes give them different names. 'Corrective maintenance' is universally used to refer to maintenance for fault repair. However, 'adaptive maintenance' sometimes means adapting to a new environment and sometimes means adapting the software to new requirements. 'Perfective maintenance' sometimes means perfecting the software by implementing new requirements; in other cases it means maintaining the functionality of the system but improving its structure and its performance. Because of this naming uncertainty, I have avoided the use of all of these terms in this chapter.

There have been several studies of software maintenance which have looked at the relationships between maintenance and development and between different maintenance activities (Krogstie et al., 2005; Lientz and Swanson, 1980; Nosek and Palvia, 1990; Sousa, 1998). Because of differences in terminology, the details of these studies cannot be compared. In spite of changes in technology and different application domains, it seems that there has been remarkably little change in the distribution of evolution effort since the 1980s.

The surveys broadly agree that software maintenance takes up a higher proportion of IT budgets than new development (roughly two-thirds maintenance, one-third development). They also agree that more of the maintenance budget is spent on implementing new requirements than on fixing bugs. Figure 9.8 shows an approximate distribution of maintenance costs. The specific percentages will obviously vary from one organization to another but, universally, repairing system faults is not the most expensive maintenance activity. Evolving the system to cope with new environments and new or changed requirements consumes most maintenance effort.

The relative costs of maintenance and new development vary from one application domain to another. Guimaraes (1983) found that the maintenance costs for business application systems are broadly comparable with system development costs.

Software project management is an essential part of software engineering. Projects need to be managed because professional software engineering is always subject to organizational budget and schedule constraints. The project manager's job is to ensure that the software project meets and overcomes these constraints as well as delivering high-quality software. Good management cannot guarantee project success. However, bad management usually results in project failure: the software may be delivered late, cost more than originally estimated, or fail to meet the expectations of customers.

The success criteria for project management obviously vary from project to project but, for most projects, important goals are:

1. Deliver the software to the customer at the agreed time.
2. Keep overall costs within budget.
3. Deliver software that meets the customer's expectations.
4. Maintain a happy and well-functioning development team.

These goals are not unique to software engineering but are the goals of all engineering projects. However, software engineering is different from other types of engineering in a number of ways that make software management particularly challenging. Some of these differences are:

1. *The product is intangible* A manager of a shipbuilding or a civil engineering project can see the product being developed. If a schedule slips, the effect on the product is visible—parts of the structure are obviously unfinished. Software is intangible. It cannot be seen or touched. Software project managers cannot see progress by simply looking at the artifact that is being constructed. Rather, they rely on others to produce evidence that they can use to review the progress of the work.
2. *Large software projects are often 'one-off' projects* Large software projects are usually different in some ways from previous projects. Therefore, even managers who have a large body of previous experience may find it difficult to anticipate problems. Furthermore, rapid technological changes in computers and communications can make a manager's experience obsolete. Lessons learned from previous projects may not be transferable to new projects.
3. *Software processes are variable and organization-specific* The engineering process for some types of system, such as bridges and buildings, is well understood. However, software processes vary quite significantly from one organization to another. Although there has been significant progress in process standardization and improvement, we still cannot reliably predict when a particular software process is likely to lead to development problems. This is especially true when the software project is part of a wider systems engineering project.

Because of these issues, it is not surprising that some software projects are late, over budget, and behind schedule. Software systems are often new and technically

innovative. Engineering projects (such as new transport systems) that are innovative often also have schedule problems. Given the difficulties involved, it is perhaps remarkable that so many software projects are delivered on time and to budget!

It is impossible to write a standard job description for a software project manager. The job varies tremendously depending on the organization and the software product being developed. However, most managers take responsibility at some stage for some or all of the following activities:

1. *Project planning* Project managers are responsible for planning, estimating and scheduling project development, and assigning people to tasks. They supervise the work to ensure that it is carried out to the required standards and monitor progress to check that the development is on time and within budget.
2. *Reporting* Project managers are usually responsible for reporting on the progress of a project to customers and to the managers of the company developing the software. They have to be able to communicate at a range of levels, from detailed technical information to management summaries. They have to write concise, coherent documents that abstract critical information from detailed project reports. They must be able to present this information during progress reviews.
3. *Risk management* Project managers have to assess the risks that may affect a project, monitor these risks, and take action when problems arise.
4. *People management* Project managers are responsible for managing a team of people. They have to choose people for their team and establish ways of working that lead to effective team performance.
5. *Proposal writing* The first stage in a software project may involve writing a proposal to win a contract to carry out an item of work. The proposal describes the objectives of the project and how it will be carried out. It usually includes cost and schedule estimates and justifies why the project contract should be awarded to a particular organization or team. Proposal writing is a critical task as the survival of many software companies depends on having enough proposals accepted and contracts awarded. There can be no set guidelines for this task; proposal writing is a skill that you acquire through practice and experience.

In this chapter, I focus on risk management and people management. Project planning is an important topic in its own right, which I discuss in Chapter 23.

22.1 Risk management

Risk management is one of the most important jobs for a project manager. Risk management involves anticipating risks that might affect the project schedule or the quality of the software being developed, and then taking action to avoid these risks

(Hall, 1998; Ould, 1999). You can think of a risk as something that you'd prefer not to have happen. Risks may threaten the project, the software that is being developed, or the organization. There are, therefore, three related categories of risk:

1. *Project risks* Risks that affect the project schedule or resources. An example of a project risk is the loss of an experienced designer. Finding a replacement designer with appropriate skills and experience may take a long time and, consequently, the software design will take longer to complete.
2. *Product risks* Risks that affect the quality or performance of the software being developed. An example of a product risk is the failure of a purchased component to perform as expected. This may affect the overall performance of the system so that it is slower than expected.
3. *Business risks* Risks that affect the organization developing or procuring the software. For example, a competitor introducing a new product is a business risk. The introduction of a competitive product may mean that the assumptions made about sales of existing software products may be unduly optimistic.

Of course, these risk types overlap. If an experienced programmer leaves a project this can be a project risk because, even if they are immediately replaced, the schedule will be affected. It inevitably takes time for a new project member to understand the work that has been done, so they cannot be immediately productive. Consequently, the delivery of the system may be delayed. The loss of a team member can also be a product risk because a replacement may not be as experienced and so could make programming errors. Finally, it can be a business risk because that programmer's experience may be crucial in winning new contracts.

You should record the results of the risk analysis in the project plan along with a consequence analysis, which sets out the consequences of the risk for the project, product, and business. Effective risk management makes it easier to cope with problems and to ensure that these do not lead to unacceptable budget or schedule slippage.

The specific risks that may affect a project depend on the project and the organizational environment in which the software is being developed. However, there are also common risks that are not related to the type of software being developed and these can occur in any project. Some of these common risks are shown in Figure 22.1.

Risk management is particularly important for software projects because of the inherent uncertainties that most projects face. These stem from loosely defined requirements, requirements changes due to changes in customer needs, difficulties in estimating the time and resources required for software development, and differences in individual skills. You have to anticipate risks; understand the impact of these risks on the project, the product, and the business; and take steps to avoid these risks. You may need to draw up contingency plans so that, if the risks do occur, you can take immediate recovery action.

Risk	Affects	Description
Staff turnover	Project	Experienced staff will leave the project before it is finished.
Management change	Project	There will be a change of organizational management with different priorities.
Hardware unavailability	Project	Hardware that is essential for the project will not be delivered on schedule.
Requirements change	Project and product	There will be a larger number of changes to the requirements than anticipated.
Specification delays	Project and product	Specifications of essential interfaces are not available on schedule.
Size underestimate	Project and product	The size of the system has been underestimated.
CASE tool underperformance	Product	CASE tools, which support the project, do not perform as anticipated.
Technology change	Business	The underlying technology on which the system is built is superseded by new technology.
Product competition	Business	A competitive product is marketed before the system is completed.

Figure 22.1 Examples of common project, product, and business risks

An outline of the process of risk management is illustrated in Figure 22.2. It involves several stages:

1. *Risk identification* You should identify possible project, product, and business risks.
2. *Risk analysis* You should assess the likelihood and consequences of these risks.
3. *Risk planning* You should make plans to address the risk, either by avoiding it or minimizing its effects on the project.
4. *Risk monitoring* You should regularly assess the risk and your plans for risk mitigation and revise these when you learn more about the risk.

You should document the outcomes of the risk management process in a risk management plan. This should include a discussion of the risks faced by the project, an analysis of these risks, and information on how you propose to manage the risk if it seems likely to be a problem.

The risk management process is an iterative process that continues throughout the project. Once you have drawn up an initial risk management plan, you monitor the situation to detect emerging risks. As more information about the risks becomes available, you have to reanalyze the risks and decide if the risk priority has changed. You may then have to change your plans for risk avoidance and contingency management.

Risk	Strategy
Organizational financial problems	Prepare a briefing document for senior management showing how the project is making a very important contribution to the goals of the business and presenting reasons why cuts to the project budget would not be cost-effective.
Recruitment problems	Alert customer to potential difficulties and the possibility of delays; investigate buying-in components.
Staff illness	Reorganize team so that there is more overlap of work and people therefore understand each other's jobs.
Defective components	Replace potentially defective components with bought-in components of known reliability.
Requirements changes	Derive traceability information to assess requirements change impact; maximize information hiding in the design.
Organizational restructuring	Prepare a briefing document for senior management showing how the project is making a very important contribution to the goals of the business.
Database performance	Investigate the possibility of buying a higher-performance database.
Underestimated development time	Investigate buying-in components; investigate use of a program generator.

Figure 22.5 Strategies to help manage risk

Again, there is no simple process that can be followed for contingency planning. It relies on the judgment and experience of the project manager.

Figure 22.5 shows possible risk management strategies that have been identified for the key risks (i.e., those that are serious or intolerable) shown in Figure 22.4. These strategies fall into three categories:

1. *Avoidance strategies* Following these strategies means that the probability that the risk will arise will be reduced. An example of a risk avoidance strategy is the strategy for dealing with defective components shown in Figure 22.5.
2. *Minimization strategies* Following these strategies means that the impact of the risk will be reduced. An example of a risk minimization strategy is the strategy for staff illness shown in Figure 22.5.
3. *Contingency plans* Following these strategies means that you are prepared for the worst and have a strategy in place to deal with it. An example of a contingency strategy is the strategy for organizational financial problems that I have shown in Figure 22.5.

You can see a clear analogy here with the strategies used in critical systems to ensure reliability, security, and safety, where you must avoid, tolerate, or recover from failures. Obviously, it is best to use a strategy that avoids the risk. If this is not possible, you should use a strategy that reduces the chances that the risk will have serious effects. Finally, you should have strategies in place to cope with the risk if it arises. These should reduce the overall impact of a risk on the project or product.