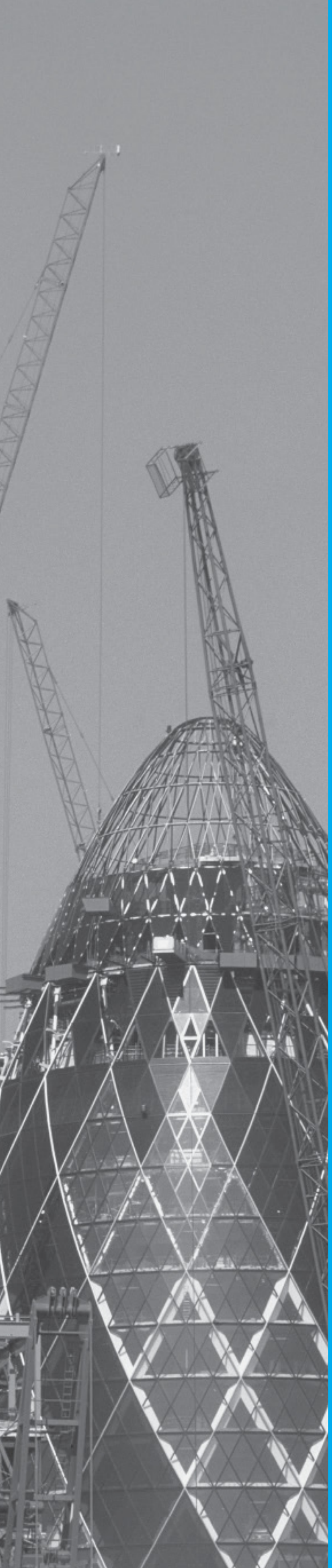


4 Requirements engineering

Objectives

The objective of this chapter is to introduce software requirements and to explain the processes involved in discovering and documenting these requirements. When you have read the chapter, you will:

- understand the concepts of user and system requirements and why these requirements should be written in different ways;
- understand the differences between functional and non-functional software requirements;
- understand the main requirements engineering activities of elicitation, analysis, and validation, and the relationships between these activities;
- understand why requirements management is necessary and how it supports other requirements engineering activities.

Contents

The requirements for a system are the descriptions of the services that a system should provide 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 (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. The user requirements may vary from broad statements of the system features required to detailed, precise descriptions of the system functionality.

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 kinds of requirement are needed to communicate information about a system to different types of reader. Figure 4.1 illustrates the distinction between user and system requirements. This example from the mental health care patient information system (Mentcare) 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

[†]Davis, A. M. 1993. Software Requirements: Objects, Functions and States. Englewood Cliffs, NJ: Prentice-Hall.

User requirements definition

1. The Mentcare system 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 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. 10mg, 20mg, etc.) separate reports shall be created for each dose unit.
1.5 Access to drug cost reports shall be restricted to authorized users as listed on a management access control list.

Figure 4.1 User and system requirements

general. The system requirements provide more specific information about the services and functions of the system that is to be implemented.

You need to write requirements at different levels of detail because different types of readers use them in different ways. Figure 4.2 shows the types of 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.

The different types of document readers shown in Figure 4.2 are examples of system stakeholders. As well as users, many other people have some kind of interest in the system. System stakeholders include anyone who is affected by the system in some way and so anyone who has a legitimate interest in it. Stakeholders range from end-users of a system through managers to external stakeholders such as regulators,

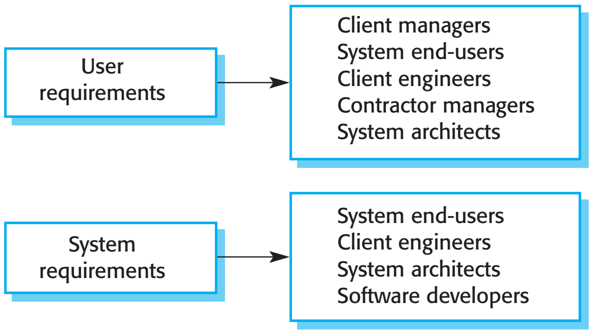


Figure 4.2 Readers of different types of requirements specification

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: (1) Does the system contribute to the overall objectives of the organization? (2) Can the system be implemented within schedule and budget using current technology? and (3) 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://software-engineering-book.com/web/feasibility-study/

who certify the acceptability of the system. For example, system stakeholders for the Mentcare system include:

1. Patients whose information is recorded in the system and relatives of these patients.
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. Health care 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.

Requirements engineering is usually presented as the first stage of the software engineering process. However, some understanding of the system requirements may have to be developed before a decision is made to go ahead with the procurement or development of a system. This early-stage RE establishes a high-level view of what the system might do and the benefits that it might provide. These may then be considered in a feasibility study, which tries to assess whether or not the system is technically and financially feasible. The results of that study help management decide whether or not to go ahead with the procurement or development of the system.

In this chapter, I present a "traditional" view of requirements rather than requirements in agile processes, which I discussed in Chapter 3. For the majority of large systems, it is still the case that there is a clearly identifiable requirements engineering phase before implementation of the system begins. The outcome is a requirements document, which may be part of the system development contract. Of course, subsequent changes are made to the requirements, and user requirements may be expanded into more detailed system requirements. Sometimes an agile approach of concurrently eliciting the requirements as the system is developed may be used to add detail and to refine the user requirements.

4.1 Functional and non-functional requirements

Software system requirements are often classified as functional 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 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 requirements 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 effectively.

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 should be written in natural language so that system users and managers can understand them. Functional system requirements expand the user requirements and are written for system developers. They should describe the system functions, their inputs and outputs, and exceptions 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. This means that these engineers may not know whether or not a domain requirement has been missed out or conflicts with other requirements.

http://software-engineering-book.com/web/domain-requirements/

requirements for the Mentcare 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 user requirements define specific functionality that should be included in the system. The requirements show that functional requirements may be written at different levels of detail (contrast requirements 1 and 3).

Functional requirements, as the name suggests, have traditionally focused on what the system should do. However, if an organization decides that an existing off-the-shelf system software product can meet its needs, then there is very little point in developing a detailed functional specification. In such cases, the focus should be on the development of information requirements that specify the information needed for people to do their work. Information requirements specify the information needed and how it is to be delivered and organized. Therefore, an information requirement for the Mentcare system might specify what information is to be included in the list of patients expected for appointments that day.

Imprecision in the requirements specification can lead to disputes between customers and software developers. 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 Mentcare system requirement in the above list 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, regardless of the clinic.

A medical staff member specifying a search requirement 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 so that it is easier to implement. Their search function may require the user to choose a clinic and then carry out the search of the patients who attended that clinic. This involves more user input and so takes longer to complete the search.

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

In practice, it is only possible to achieve requirements consistency and completeness for very small software systems. One reason is that it is easy to make mistakes and omissions when writing specifications for large, complex systems. Another reason is that large systems have many stakeholders, with different backgrounds and expectations. Stakeholders are likely to have different—and often inconsistent—needs. These inconsistencies may not be obvious when the requirements are originally specified, and the inconsistent requirements may only be discovered after deeper analysis or during system development.

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. These non-functional requirements usually specify or constrain characteristics of the system as a whole. They may relate to emergent system properties such as reliability, response time, and memory use. 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 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.

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

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 in an embedded system, you may have to organize the system to minimize communications between components.

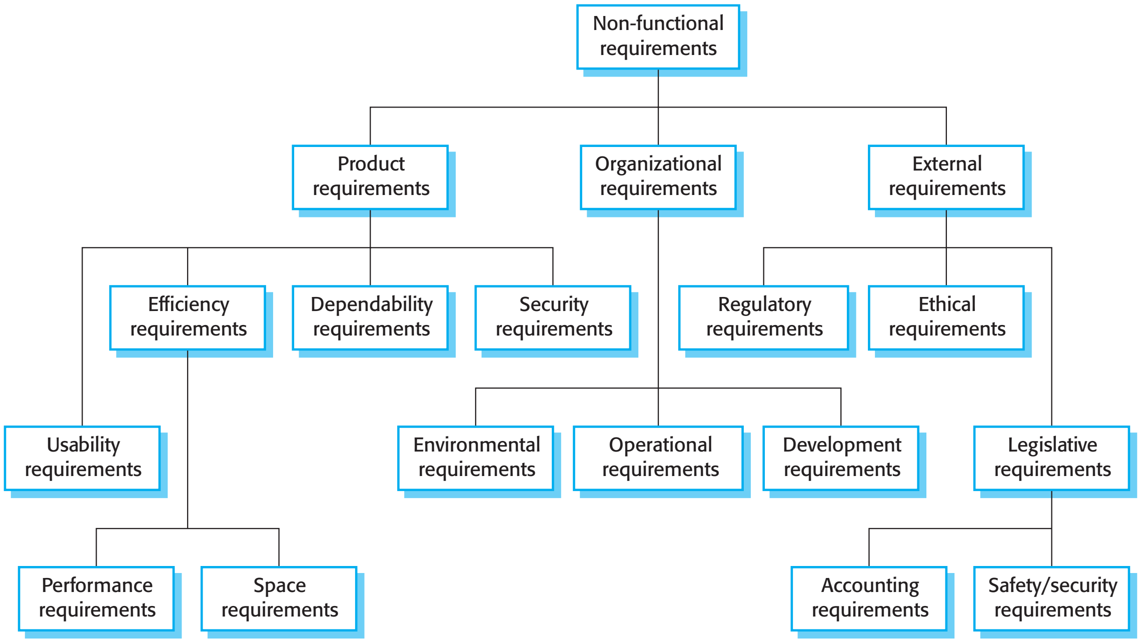


Figure 4.3 Types of non-functional requirements

2. An individual non-functional requirement, such as a security requirement, may generate several, related functional requirements that define new system services that are required if the non-functional requirement is to be implemented. In addition, it may also generate requirements that constrain existing requirements; for example, it may limit access to information in the system.

Nonfunctional requirements arise through user needs because of budget constraints, organizational policies, the need for interoperability with other software or 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 external sources:

1. *Product requirements* These requirements specify or constrain the runtime behavior of the software. Examples include performance requirements for 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 organizations. Examples include operational process requirements that define how the system will be used; development process requirements that specify the

PRODUCT REQUIREMENT
The Mentcare system shall be available to all clinics during normal working hours (Mon–Fri, 08:30–17:30). Downtime within normal working hours shall not exceed 5 seconds in any one day.

ORGANIZATIONAL REQUIREMENT
Users of the Mentcare system shall identify 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 possible non-functional requirements for the Mentcare system

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 nuclear safety authority; 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.

Figure 4.4 shows examples of product, organizational, and external requirements that could be included in the Mentcare system specification. The product requirement is an availability requirement that defines when the system has to be available and the allowed downtime each day. It says nothing about the functionality of the Mentcare system 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 health care 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 stakeholders propose 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.

Property	Measure
Speed	Processed transactions/second User/event response time Screen refresh time
Size	Megabytes/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

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 following description 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 two 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 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 simple 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 reliability (for example) 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 identification requirement in Figure 4.4 requires a card reader to be installed with each computer that connects to the system. However, there may be another requirement that requests mobile access to the system from doctors' or nurses' tablets or smartphones. These are not normally equipped with card readers so, in these circumstances, some alternative identification method may have to be supported.

It is difficult 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, ideally, 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.

Non-functional requirements such as reliability, safety, and confidentiality requirements are particularly important for critical systems. I cover these dependability requirements in Part 2, which describes ways of specifying reliability, safety, and security requirements.

4.2 Requirements engineering processes

As I discussed in Chapter 2, requirements engineering involves three key activities. These are discovering requirements by interacting with stakeholders (elicitation and analysis); converting these requirements into a 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 2.4. However, in practice, requirements engineering is an iterative process in which the activities are interleaved.

Figure 4.6 shows this interleaving. The activities are organized as an iterative process around a spiral. The output of the RE process is a system requirements document. The amount of time and effort devoted to each activity in an iteration depends on the stage of the overall process, the type of system being developed, and the budget that is available.

Early in the process, most effort will be spent on understanding high-level business and 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 non-functional requirements and more 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 that 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.

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; and modifications are made to the system's hardware, software, and organizational environment. Changes have to be managed to understand the impact on other requirements and the cost and system implications of making the change. I discuss this process of requirements management in Section 4.6.

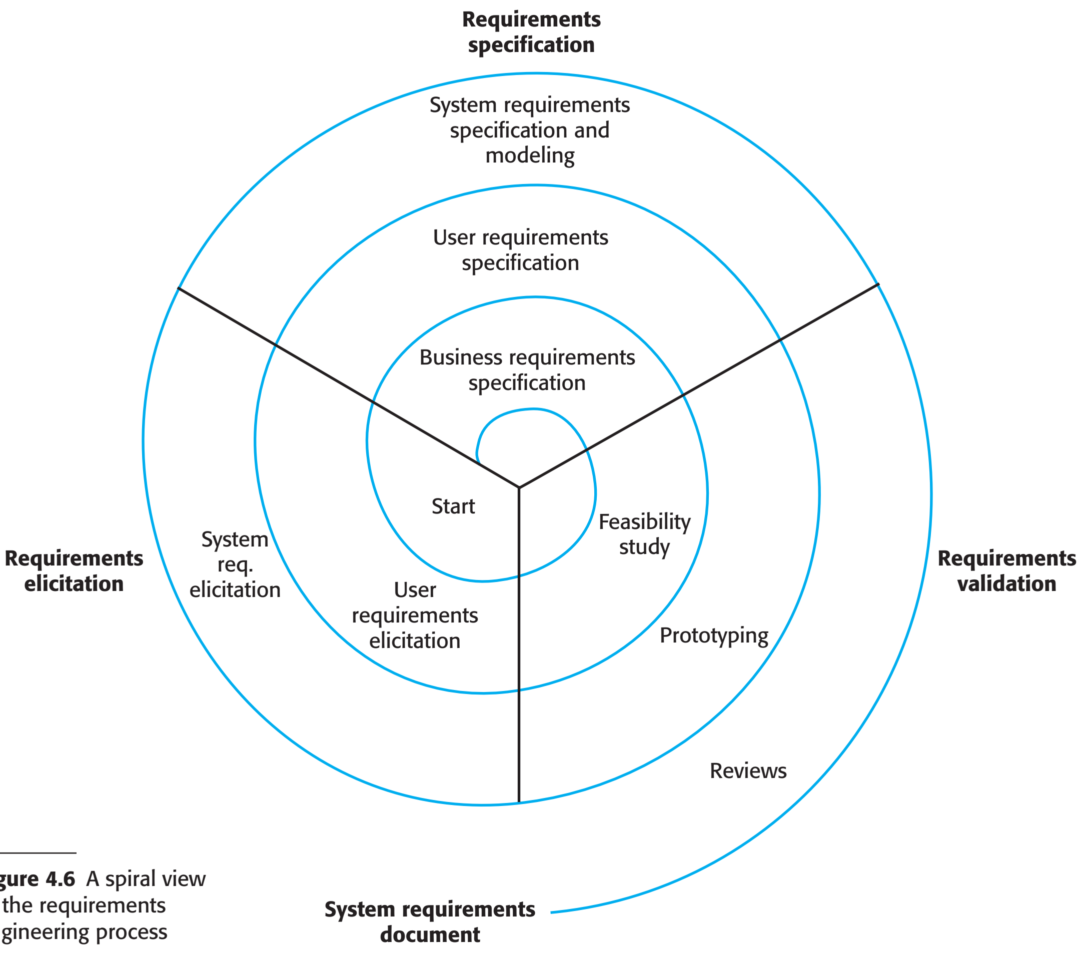

Requirements specification
System requirements specification and modeling
User requirements specification
Business requirements specification
Start
Feasibility study
Requirements elicitation
System req. elicitation
User requirements elicitation
Prototyping
Reviews
Requirements validation
System requirements document

Figure 4.6 A spiral view of the requirements engineering process

4.3 Requirements elicitation

The aims of the requirements elicitation process are to understand the work that stakeholders do and how they might use a new system to help support that work. During requirements elicitation, software engineers work with stakeholders to find out about the application domain, work activities, the services and system features that stakeholders want, the required performance of the system, hardware constraints, and so on.

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.

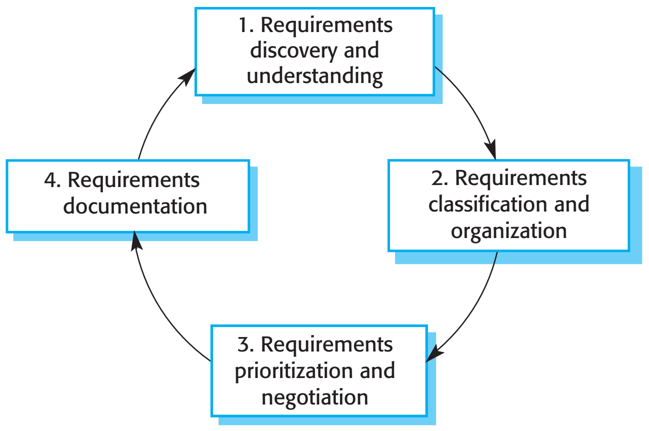

1. Requirements discovery and understanding
2. Requirements classification and organization
3. Requirements prioritization and negotiation
4. Requirements documentation

Figure 4.7 The requirements elicitation and analysis process

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, with diverse requirements, may express their requirements 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.

A process model of the elicitation and analysis process is shown in Figure 4.7. 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, and the standards used.

The process activities are:

1. *Requirements discovery and understanding* 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.

2. *Requirements classification and organization* This activity takes the unstructured collection of requirements, groups related requirements and organizes them into coherent clusters.

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

Viewpoints

A viewpoint is a 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, or others. They help identify the people who can provide information about their requirements and structure the requirements for analysis.

http://www.software-engineering-book.com/web/viewpoints/

through negotiation. Usually, stakeholders have to meet to resolve differences and agree on compromise requirements.

4. *Requirements documentation* The requirements are documented and input into the next round of the spiral. An early draft of the software requirements documents may be produced at this stage, or the requirements may simply be maintained informally on whiteboards, wikis, or other shared spaces.

Figure 4.7 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 has been produced.

To simplify the analysis of requirements, it is helpful to organize and group the stakeholder information. One way of doing so is to consider each stakeholder group to be a viewpoint and to collect all requirements from that group into the viewpoint. You may also include viewpoints to represent domain requirements and constraints from other systems. Alternatively, you can use a model of the system architecture to identify subsystems and to associate requirements with each subsystem.

Inevitably, different stakeholders have different views on the importance and priority of requirements, and sometimes these views are conflicting. If some stakeholders feel that their views have not been properly considered, then they may deliberately attempt to undermine the RE process. Therefore, it is important that you organize regular stakeholder meetings. Stakeholders should have the opportunity to express their concerns and agree on requirements compromises.

At the requirements documentation stage, it is important that you use simple language and diagrams to describe the requirements. This makes it possible for stakeholders to understand and comment on these requirements. To make information sharing easier, it is best to use a shared document (e.g., on Google Docs or Office 365) or a wiki that is accessible to all interested stakeholders.

4.3.1 Requirements elicitation techniques

Requirements elicitation involves meeting with stakeholders of different kinds to discover information about the proposed system. You may supplement this information with knowledge of existing systems and their usage and information from documents of various kinds. You need to spend time understanding how people work, what they produce, how they use other systems, and how they may need to change to accommodate a new system.

There are two fundamental approaches to requirements elicitation:

1. Interviewing, where you talk to people about what they do.
2. Observation or ethnography, where you watch people doing their job to see what artifacts they use, how they use them, and so on.

You should use a mix of interviewing and observation to collect information and, from that, you derive the requirements, which are then the basis for further discussions.

4.3.1.1 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 predefined set of questions.
2. Open interviews, in which there is no predefined agenda. The requirements engineering team explores a range of issues with system stakeholders and hence develops 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 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, and so they are usually happy to get involved in interviews. However, unless you have a system prototype to demonstrate, you should not expect stakeholders to suggest specific and detailed requirements. Everyone finds it difficult to visualize what a system might be like. You need to analyze the information collected and to generate the requirements from this.

Eliciting domain knowledge through interviews can be difficult, for two reasons:

1. All application specialists use jargon specific to their area of work. It is impossible for them to discuss domain requirements without using this terminology. They normally use words in a precise and subtle way that requirements engineers may 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 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.

To be an effective interviewer, you should bear two things in mind:

1. You should be open-minded, avoid preconceived ideas about the requirements, and willing to listen to stakeholders. If the stakeholder comes up with surprising requirements, then you should be willing to change your mind about the system.
2. You should prompt the interviewee to get discussions going by using a springboard question or 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 is used along with other information about the system from documentation describing business processes or existing systems, user observations, and developer experience. 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.

4.3.1.2 Ethnography

Software systems do not exist in isolation. They are used in a social and organizational environment, and software system requirements may be generated or constrained by that environment. One reason why many software systems are delivered but never used is that their requirements do not take proper account of how social and organizational factors affect the practical operation of the system. It is therefore very important that, during the requirements engineering process, you try to understand the social and organizational issues that affect the use of the system.

Ethnography is an observational technique that can be used to understand operational processes and help derive requirements for software to support 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 are 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 that are not obvious to individuals, may only become clear when noticed by an unbiased observer. For example, a workgroup 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 (Suchman 1983) pioneered the use of ethnography to study office work. She found that 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 (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 integrating ethnography into the software engineering process by linking it with requirements engineering methods (Viller and Sommerville 2000) and documenting patterns of interaction in cooperative systems (Martin and Sommerville 2004).

Ethnography is particularly effective for discovering two types of requirements:

1. Requirements derived from the way in which people actually work, rather than the way in which business process definitions say they ought to work. In practice, people never follow formal processes. 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. The conflict alert system is sensitive and issues audible warnings even when planes are far apart. Controllers may find these distracting and prefer to use other strategies to ensure that planes are not on conflicting flight paths.

2. Requirements derived from cooperation and awareness of other people's activities. For example, air traffic controllers (ATCs) may use an awareness of other controlles' work to predict the number of aircraft 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 the development of a system prototype (Figure 4.8). 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).

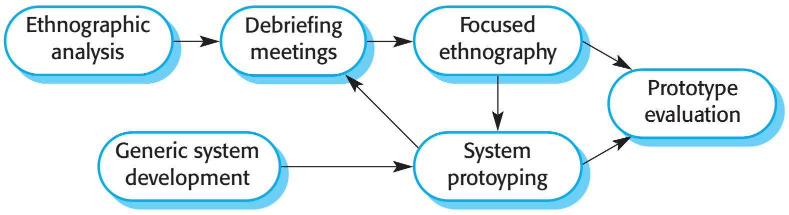


Figure 4.8 Ethnography and prototyping for requirements analysis

Ethnography is helpful to understand existing systems, but this understanding does not always help with innovation. Innovation is particularly relevant for new product development. Commentators have suggested that Nokia used ethnography to discover how people used their phones and developed new phone models on that basis; Apple, on the other hand, ignored current use and revolutionized the mobile phone industry with the introduction of the iPhone.

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 effective for discovering broader organizational or domain requirements or for suggestion innovations. You therefore have to use ethnography as one of a number of techniques for requirements elicitation.

4.3.2 Stories and scenarios

People find it easier to relate to real-life examples than abstract descriptions. They are not good at telling you the system requirements. However, they may be able to describe how they handle particular situations or imagine things that they might do in a new way of working. Stories and scenarios are ways of capturing this kind of information. You can then use these when interviewing groups of stakeholders to discuss the system with other stakeholders and to develop more specific system requirements.

Stories and scenarios are essentially the same thing. They are a description of how the system can be used for some particular task. They describe what people do, what information they use and produce, and what systems they may use in this process. The difference is in the ways that descriptions are structured and in the level of detail presented. Stories are written as narrative text and present a high-level description of system use; scenarios are usually structured with specific information collected such as inputs and outputs. I find stories to be effective in setting out the "big picture." Parts of stories can then be developed in more detail and represented as scenarios.

Figure 4.9 is an example of a story that I developed to understand the requirements for the iLearn digital learning environment that I introduced in Chapter 1. This story describes a situation in a primary (elementary) school where the teacher is using the environment to support student projects on the fishing industry. You can see this is a very high-level description. Its purpose is to facilitate discussion of how the iLearn system might be used and to act as a starting point for eliciting the requirements for that system.

Photo sharing in the classroom

Jack is a primary school teacher in Ullapool (a village in northern Scotland). He has decided that a class project should be focused on the fishing industry in the area, looking at the history, development, and economic impact of fishing. As part of this project, pupils are asked to gather and share reminiscences from relatives, use newspaper archives, and collect old photographs related to fishing and fishing communities in the area. Pupils use an iLearn wiki to gather together fishing stories and SCRAN (a history resources site) to access newspaper archives and photographs. However, Jack also needs a photo-sharing site because he wants pupils to take and comment on each other's photos and to upload scans of old photographs that they may have in their families.

Jack sends an email to a primary school teachers' group, which he is a member of, to see if anyone can recommend an appropriate system. Two teachers reply, and both suggest that he use KidsTakePics, a photo-sharing site that allows teachers to check and moderate content. As KidsTakePics is not integrated with the iLearn authentication service, he sets up a teacher and a class account. He uses the iLearn setup service to add KidsTakePics to the services seen by the pupils in his class so that when they log in, they can immediately use the system to upload photos from their mobile devices and class computers.

Figure 4.9 A user story for the iLearn system

The advantage of stories is that everyone can easily relate to them. We found this approach to be particularly useful to get information from a wider community than we could realistically interview. We made the stories available on a wiki and invited teachers and students from across the country to comment on them.

These high-level stories do not go into detail about a system, but they can be developed into more specific scenarios. Scenarios are descriptions of example user interaction sessions. I think that it is best to present scenarios in a structured way rather than as narrative text. User stories used in agile methods such as Extreme Programming, are actually narrative scenarios rather than general stories to help elicit requirements.

A scenario starts with an outline of the interaction. During the elicitation process, details are added 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 expect 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 resulting problems can be 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 ends.

As an example of a scenario, Figure 4.10 describes what happens when a student uploads photos to the KidsTakePics system, as explained in Figure 4.9. The key difference between this system and other systems is that a teacher moderates the uploaded photos to check that they are suitable for sharing.

You can see this is a much more detailed description than the story in Figure 4.9, and so it can be used to propose requirements for the iLearn system. Like stories, scenarios can be used to facilitate discussions with stakeholders who sometimes may have different ways of achieving the same result.

Uploading photos to KidsTakePics

Initial assumption: A user or a group of users have one or more digital photographs to be uploaded to the picture-sharing site. These photos are saved on either a tablet or a laptop computer. They have successfully logged on to KidsTakePics.

Normal: The user chooses to upload photos and is prompted to select the photos to be uploaded on the computer and to select the project name under which the photos will be stored. Users should also be given the option of inputting keywords that should be associated with each uploaded photo. Uploaded photos are named by creating a conjunction of the user name with the filename of the photo on the local computer.

On completion of the upload, the system automatically sends an email to the project moderator, asking them to check new content, and generates an on-screen message to the user that this checking has been done.

What can go wrong: No moderator is associated with the selected project. An email is automatically generated to the school administrator asking them to nominate a project moderator. Users should be informed of a possible delay in making their photos visible.

Photos with the same name have already been uploaded by the same user. The user should be asked if he or she wishes to re-upload the photos with the same name, rename the photos, or cancel the upload. If users choose to re-upload the photos, the originals are overwritten. If they choose to rename the photos, a new name is automatically generated by adding a number to the existing filename.

Other activities: The moderator may be logged on to the system and may approve photos as they are uploaded.

System state on completion: User is logged on. The selected photos have been uploaded and assigned a status "awaiting moderation." Photos are visible to the moderator and to the user who uploaded them.

Figure 4.10 Scenario for uploading photos in KidsTakePics

4.4 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 almost impossible to achieve. Stakeholders interpret the requirements in different ways, and there are often inherent conflicts and inconsistencies in the requirements.

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, or mathematical system models can also be used. Figure 4.11 summarizes possible notations for writing system 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.

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.
Graphical notations	Graphical models, supplemented by text annotations, are used to define the functional requirements for the system. UML (unified modeling language) 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 they are reluctant to accept it as a system contract. (I discuss this approach, in Chapter 10, which covers system dependability.)

Figure 4.11 Notations for writing system requirements

System requirements are expanded versions of the user requirements that software engineers use as the starting point for the system design. They add detail and explain how the system should provide the user requirements. 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 only 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 neither possible nor desirable 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 the different subsystems that make up the system. We did this when we were defining the requirements for the iLearn system, where we proposed the architecture shown in Figure 1.8.

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 11, may be necessary. An external regulator who needs to certify that the system is safe may specify that an architectural design that has already been certified should be used.

4.4.1 Natural language specification

Natural language has been used to write requirements for software since the 1950s. It is expressive, intuitive, and universal. It is also potentially vague and ambiguous, and its interpretation depends on the background of the reader. As a result, there

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 4.12 Example requirements for the insulin pump software system

have been many proposals for alternative ways to write requirements. However, none of these proposals has 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 these 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. I suggest that, wherever possible, you should write the requirement in one or two sentences of natural language.
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 such as "architecture" and "module" to be misunderstood. Wherever possible, you should 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 and who proposed the requirement (the requirement source), so that you know whom to consult if the requirement has to be changed. Requirements rationale is particularly useful when requirements are changed, as it may help decide what changes would be undesirable.

Figure 4.12 illustrates how these guidelines may be used. It includes two requirements for the embedded software for the automated insulin pump, introduced in Chapter 1. Other requirements for this embedded system are defined in the insulin pump requirements document, which can be downloaded from the book's web pages.

4.4.2 Structured specifications

Structured natural language is a way of writing system requirements where requirements are written in a standard way rather than as free-form text. This approach maintains most of the expressiveness and understandability of natural language but

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://software-engineering-book.com/web/natural-language/

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 2013), 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, and supporting materials. This is similar to the approach used in the example of a structured specification shown in Figure 4.13.

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 4.13.

When a standard format 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 the origin of these inputs.
3. A description of its outputs and the destination of these outputs.
4. Information about the information needed for the computation or other entities in the system that are required (the "requires" part).
5. A description of the action to be taken.
6. If a functional approach is used, a precondition setting out what must be true before the function is called, and a postcondition 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

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. (see Figure 4.14)
Requires	Two previous readings so that the rate of change of sugar level can be computed.
Precondition	The insulin reservoir contains at least the maximum allowed single dose of insulin.
Postcondition	r0 is replaced by r1 then r1 is replaced by r2.
Side effects	None.

Figure 4.13 The structured specification of a requirement for an insulin pump

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 4.14 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 4.14 The tabular specification of computation in an insulin pump

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 & ((r2 − r1) ≥ (r1 − r0))	CompDose = round ((r2 − r1)/4)
	If rounded result = 0 then
	CompDose = MinimumDose

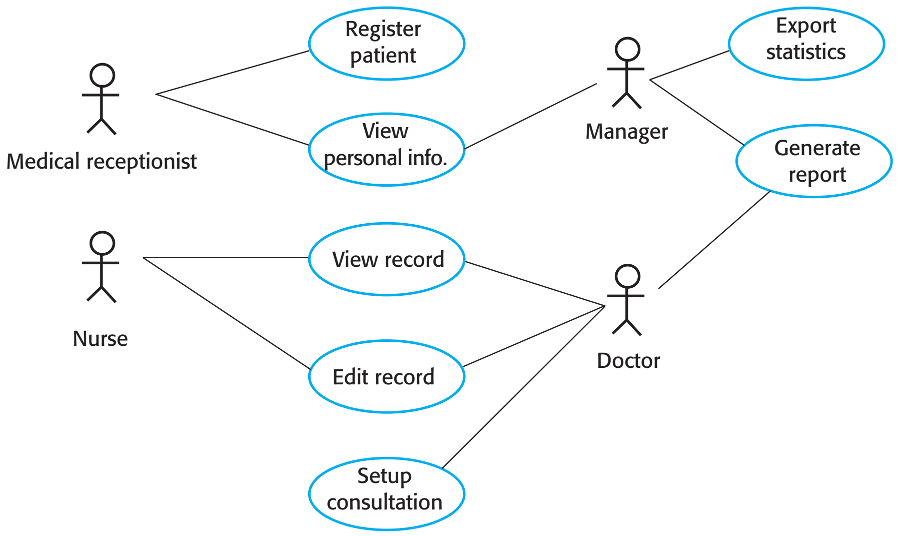


Figure 4.15 Use cases for the Mentcare system

4.4.3 Use cases

Use cases are a way of describing interactions between users and a system using a graphical model and structured text. They were first introduced in the Objectory method (Jacobsen et al. 1993) and have now become a fundamental feature of the Unified Modeling Language (UML). In their simplest form, a use case identifies the actors involved in an interaction and names the type of interaction. You then add additional information describing the interaction with the system. The additional information may be a textual description or one or more graphical models such as the UML sequence or state charts (see Chapter 5).

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 Mentcare system.

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 patient record at the same time. One doctor initiates the consultation by choosing the people involved from a dropdown 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 call for voice communication can be separately arranged.*

The UML is a standard for object-oriented modeling, so use cases and use case-based elicitation are used in the requirements engineering process. However, my experience with use cases is that they are too fine-grained to be useful in discussing requirements. Stakeholders don't understand the term *use case*; they don't find the graphical model to be useful, and they are often not interested in a detailed description of each and every system interaction. Consequently, I find use cases to be more helpful in systems design than in requirements engineering. I discuss use cases further in Chapter 5, which shows how they are used alongside other system models to document a system design.

Some people think that each use case is a single, low-level interaction scenario. Others, such as Stevens and Pooley (Stevens and Pooley 2006), suggest that each use case includes a set of related, low-level scenarios. Each of these scenarios is a single thread through the use case. Therefore, there would be a scenario for the normal interaction plus scenarios for each possible exception. In practice, you can use them in either way.

4.4.4 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 may 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 described in an introductory chapter in the system requirements specification.

Requirements documents are essential when systems are outsourced for development, when different teams develop different parts of the system, and when a detailed analysis of the requirements is mandatory. In other circumstances, such as software product or business system development, a detailed requirements document may not be needed.

Agile 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, agile approaches often collect user requirements incrementally and write these on cards or whiteboards as short user stories. The user then prioritizes these stories 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 4.16 shows possible users of the document and how they use it.

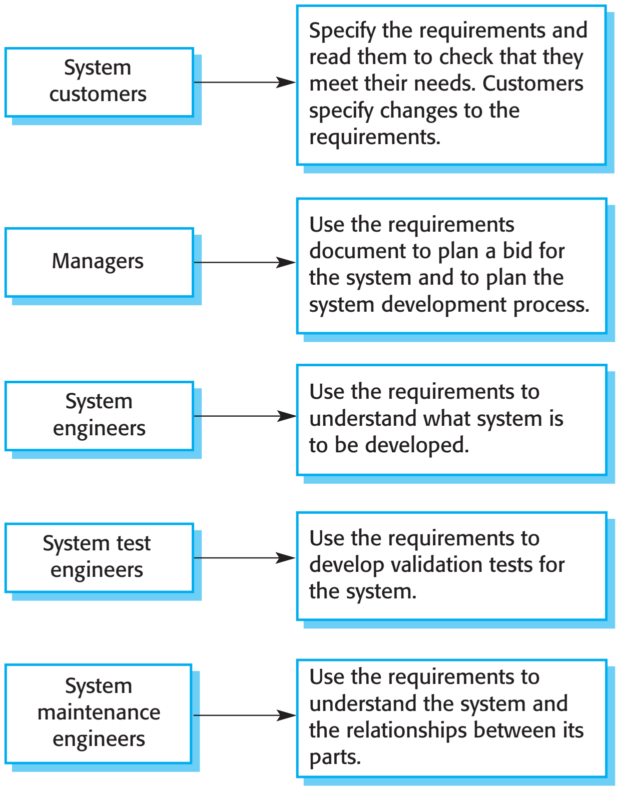

System customers
Specify the requirements and read them to check that they meet their needs. Customers specify changes to the requirements.
Managers
Use the requirements document to plan a bid for the system and to plan the system development process.
System engineers
Use the requirements to understand what system is to be developed.
System test engineers
Use the requirements to develop validation tests for the system.
System maintenance engineers
Use the requirements to understand the system and the relationships between its parts.

Figure 4.16 Users of a requirements document

The diversity of possible users means that the requirements document has to be a compromise. It has to describe the requirements for customers, define the requirements in precise detail for developers and testers, as well as include information about future system evolution. Information on anticipated changes helps system designers to avoid restrictive design decisions and maintenance engineers 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 detailed requirements because safety and security have to be analyzed in detail to find possible requirements errors. 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 less detailed. Details can be added to the requirements and ambiguities resolved during development of the system.

Figure 4.17 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 one that can be adapted to specific uses. In this case, the standard has been extended 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.

Chapter	Description
Preface	This defines 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 describes 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 defines 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 nonfunctional 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 presents 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 describes the functional and nonfunctional requirements in more detail. If necessary, further detail may also be added to the nonfunctional requirements. Interfaces to other systems may be defined.
System models	This chapter includes graphical system models showing the relationships between the system components and the system and its environment. Examples of possible models are object models, data-flow models, or semantic data models.
System evolution	This describes 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 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 4.17 The structure of a requirements document

Naturally, the information included in a requirements document depends on the type of software being developed and the approach to development that is to be used. A requirements document with a structure like that shown in Figure 4.17 might be produced for a complex engineering system that includes hardware and software developed by different companies. The requirements document is likely to be long and detailed. It is therefore important that a comprehensive table of contents and document index be included so that readers can easily find the information they need.

By contrast, the requirements document for an in-house software product will leave out many of detailed chapters suggested above. The focus will be on defining the user requirements and high-level, nonfunctional system requirements. The system designers and programmers use their judgment to decide how to meet the outline user requirements for the system.

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://software-engineering-book.com/web/requirements-standard/

4.5 Requirements validation

Requirements validation is the process of checking that requirements define the system that the customer really wants. It overlaps with elicitation and analysis, as it is concerned with finding problems with the requirements. Requirements validation is critically 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. A change to the requirements usually means that the system design and implementation must also be changed. Furthermore, the system must then be retested.

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* These check that the requirements reflect the real needs of system users. Because of changing circumstances, the user requirements may have changed since they were originally elicited.

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* By using knowledge of existing technologies, the requirements should be checked to ensure that they can be implemented within the proposed budget for the system. 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.

Requirements reviews

A requirements review is a process in which 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://software-engineering-book.com/web/requirements-reviews/

A number of requirements validation techniques 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* This involves developing an executable model of a system and using this with end-users and customers to see if it meets their needs and expectations. Stakeholders experiment with the system and feed back requirements changes to the development team.
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 test-driven development.

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. Inevitably, further requirements changes will be needed to correct omissions and misunderstandings after agreement has been reached on the requirements document.

4.6 Requirements change

The requirements for large software systems are always changing. One reason for the frequent changes is that these systems are often developed to address "wicked" problems—problems that cannot be completely defined (Rittel and Webber 1973). Because the problem cannot be fully defined, the software requirements are bound to

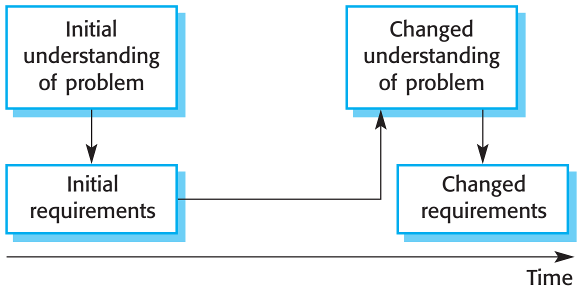

Initial understanding of problem
Changed understanding of problem
Initial requirements
Changed requirements
Time

Figure 4.18
Requirements evolution

be incomplete. During the software development process, the stakeholders' understanding of the problem is constantly changing (Figure 4.18). The system requirements must then evolve to reflect this changed problem understanding.

Once a system has been installed and is regularly used, new requirements inevitably emerge. This is partly a consequence of errors and omissions in the original requirements that have to be corrected. However, most changes to system requirements arise because of changes to the business environment of the system:

1. The business and technical environment of the system always changes after installation. New hardware may be introduced and existing hardware updated. 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 require system compliance.
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 stakeholder community, with stakeholders having different requirements. Their priorities may be conflicting or contradictory. The final system requirements are inevitably a compromise, and some stakeholders have to be given priority. With experience, it is often discovered that the balance of support given to different stakeholders has to be changed and the requirements re-prioritized.

As requirements are evolving, 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 therefore need a formal process for making change proposals and linking these to system requirements. This process of "requirements management" should start as soon as a draft version of the requirements document is available.

Agile development processes 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

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://software-engineering-book.com/web/changing-requirements/

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 for the change to be implemented.

The problem with this approach is that users are not necessarily the best people to decide on whether or not a requirements change is cost-effective. In systems with multiple stakeholders, changes will benefit some stakeholders and not others. It is often better for an independent authority, who can balance the needs of all stakeholders, to decide on the changes that should be accepted.

4.6.1 Requirements management planning

Requirements management planning is concerned with establishing how a set of evolving requirements will be managed. During the planning stage, you have to decide on a number of issues:

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 shared 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.

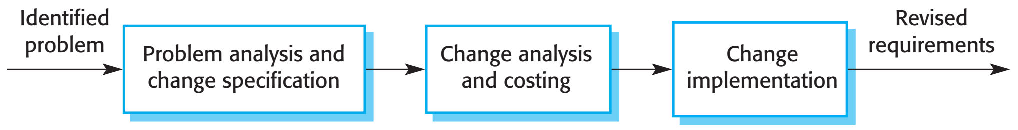

Identified problem
Problem analysis and change specification
Change analysis and costing
Change implementation
Revised requirements

Figure 4.19 Requirements change management

2. *Change management* The process of change management (Figure 4.19) is simplified if active tool support is available. Tools can keep track of suggested changes and responses to these suggestions.

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, you do not need to use specialized requirements management tools. Requirements management can be supported using shared web documents, spreadsheets, and databases. However, for larger systems, more specialized tool support, using systems such as DOORS (IBM 2013), makes it much easier to keep track of a large number of changing requirements.

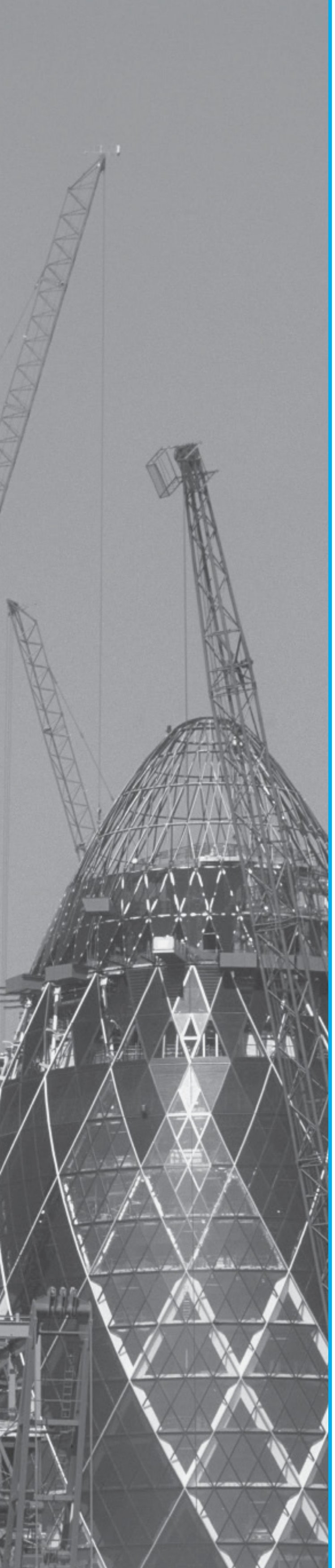
4.6.2 Requirements change management

Requirements change management (Figure 4.19) 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 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 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 as to whether or not to proceed with the requirements change.



5 System modeling

Objectives

The aim of this chapter is to introduce system models that may be developed as part of 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 and why several types of model are needed to fully represent a system;
- understand the fundamental system modeling perspectives of context, interaction, structure, and behavior;
- understand the principal diagram types in the Unified Modeling Language (UML) and how these diagrams may be used in system modeling;
- have been introduced to model-driven engineering, where an executable 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 now usually means representing a system using some kind of graphical notation based on diagram types 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 here, and formal modeling is briefly discussed in Chapter 10.

Models are used during the requirements engineering process to help derive the detailed 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 they can be used to focus a stakeholder discussion on its strengths and weaknesses.
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. If you use a model-driven engineering process (Brambilla, Cabot, and Wimmer 2012), you can generate a complete or partial system implementation from system models.

It is important to understand that a system model is not a complete representation of system. It purposely leaves out detail to make it easier to understand. A model is an abstraction of the system being studied rather than an alternative representation of that system. A representation of a system should maintain all the information about the entity being represented. An abstraction deliberately simplifies a system design and picks out the most salient characteristics. For example, the PowerPoint slides that accompany this book are an abstraction of the book's key points. However, if the book 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 processed by the system.
4. A behavioral perspective, where you model the dynamic behavior of the system and how it responds to events.

The Unified Modeling Language

The Unified Modeling Language (UML) 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, such as SysML, have been proposed for more general system modeling.

http://software-engineering-book.com/web/uml/

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 depend on how you intend to use it. There are three ways in which graphical models are commonly used:

1. As a way to stimulate and focus discussion about an existing or proposed system. The purpose of the model is to stimulate and focus discussion among the software engineers involved in developing the system. The models may be incomplete (as 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 agile modeling (Ambler and Jeffries 2002).

2. As a way of documenting an existing system. When models are used as documentation, they do not have to be complete, as you may only need to use models to document 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.

3. As a detailed system description that can be used to generate a system implementation. Where models are used as part of a model-based development process, the system models have to be both complete and correct. They are used as a basis for generating the source code of the system, and you therefore have to be very careful not to confuse similar symbols, such as stick and block arrowheads, that may have different meanings.

In this chapter, I use diagrams defined in the Unified Modeling Language (UML) (Rumbaugh, Jacobson, and Booch 2004; Booch, Rumbaugh, and Jacobson 2005), which has become a standard language for object-oriented modeling. The UML has 13 diagram types and so supports the creation of many different types of system model. However, a survey (Erickson and Siau 2007) showed that most users of the UML thought that five diagram types could represent the essentials of a system. I therefore concentrate on these five UML diagram types here:

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.

5.1 Context models

At an early stage in the specification of a system, you should decide on the system boundaries, that is, on what is and is not part of the system being developed. This involves working with system stakeholders to decide what functionality should be included in the system and what processing and operations should be carried out in the system's operational environment. You may decide that automated support for some business processes should be implemented in the software being developed but that other processes should be manual 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 Mentcare patient information system. 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, and if these systems are unavailable, then it may be impossible to use the Mentcare system.

In some situations, the user base for a system is very diverse, and users have a wide range of different system requirements. You may decide not to define

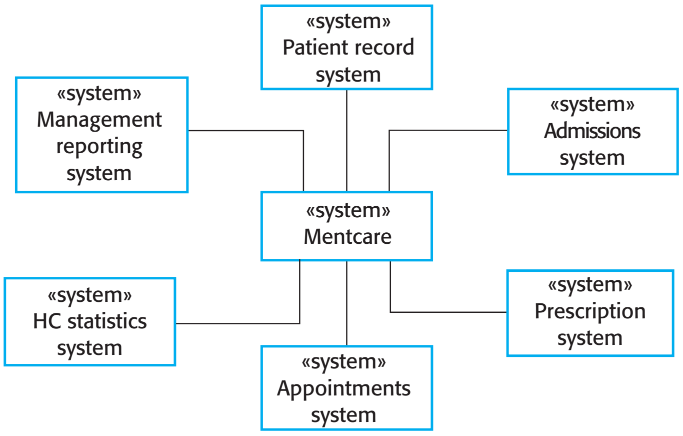

«system» Patient record system
«system» Management reporting system
«system» Admissions system
«system» Mentcare
«system» HC statistics system
«system» Prescription system
«system» Appointments system

Figure 5.1 The context of the Mentcare system

boundaries explicitly but instead to develop a configurable system that can be adapted to the needs of different users. This was the approach that we adopted in the iLearn systems, introduced in Chapter 1. There, users range from very young children who can't read through to young adults, their teachers, and school administrators. Because these groups need different system boundaries, we specified a configuration system that would allow the boundaries to be specified when the system was deployed.

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 nontechnical factors. For example, a system boundary may be deliberately positioned so that the complete analysis process can be carried out on one site; it may be chosen so that a particularly difficult manager need not be consulted; and 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 context model that shows the Mentcare system and the other systems in its environment. You can see that the Mentcare system 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 admissions, 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

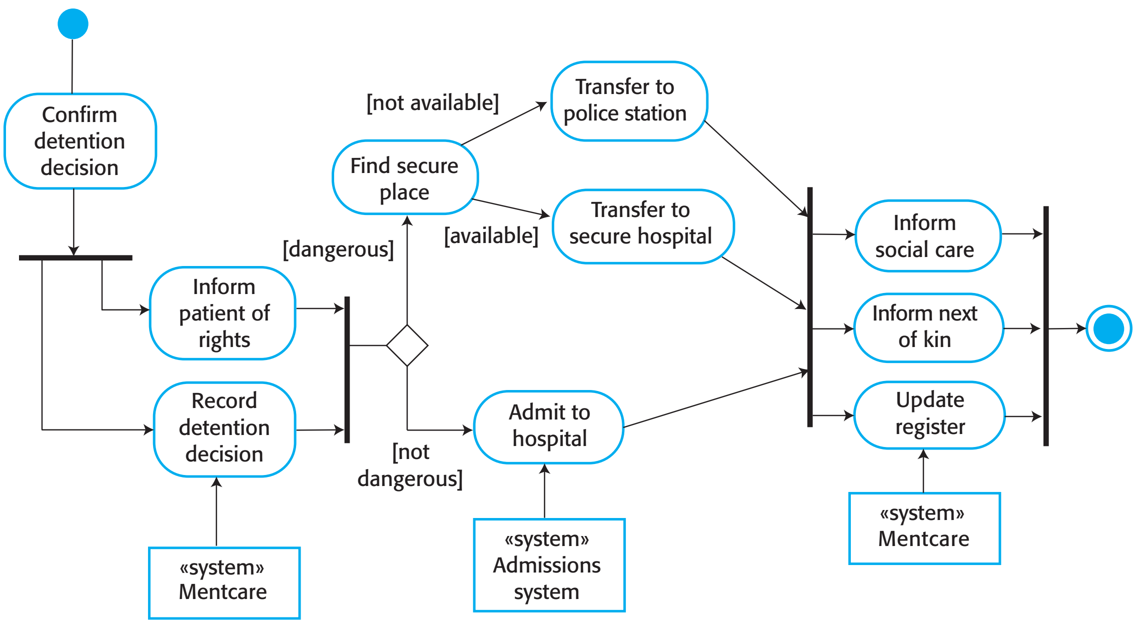


Figure 5.2 A process model of involuntary detention

these relations may affect the requirements and design of the system being defined and so 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.

UML activity diagrams may be used to show the business processes in which systems are used. Figure 5.2 is a UML activity diagram that shows where the Mentcare system is used in an important mental health care process—involuntary detention.

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 critical function of the Mentcare system is to ensure that such safeguards are implemented and that the rights of patients are respected.

UML activity diagrams show the activities in a 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 subprocesses that must be carried out. You may include objects in activity charts. Figure 5.2 shows the systems that are used to support different subprocesses within the involuntary detection process. I have shown that these are separate systems by using the UML stereotype feature where the type of entity in the box between chevrons is shown.

Arrows represent the flow of work from one activity to another, and a solid bar indicates activity coordination. When the flow from more than one activity leads to a

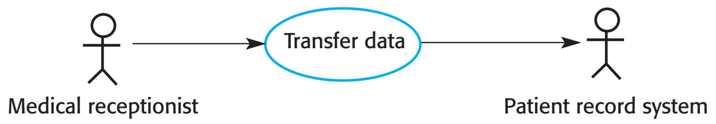


Figure 5.3 Transfer-data use case

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, as well as to update the detention register, may be concurrent.

Arrows may be annotated with guards (in square brackets) that specify when that flow is followed. 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 are a danger to themselves may be admitted to an appropriate ward in a hospital, where they can be kept under close supervision.

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 software being developed and other systems in its environment; or interaction between the components of a software system. User interaction modeling 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.

This section discusses two related approaches to interaction modeling:

1. Use case modeling, which is mostly used to model interactions between a system and external agents (human 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 interactions at different levels of detail and so may be used together. For example, 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 describe this diagram type because communication diagrams are simply an alternative representation of sequence diagrams.

5.2.1 Use case modeling

Use case modeling was originally developed by Ivar Jacobsen in the 1990s (Jacobsen et al. 1993), and a UML diagram type to support use case modeling is part of the

Mentcare system: Transfer data	
Actors	Medical receptionist, Patient records system (PRS)
Description	A receptionist may transfer data from the Mentcare system 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 5.4 Tabular description of the Transfer-data use case

UML. A use case can be taken as a simple description of what a user expects from a system in that interaction. I have discussed use cases for requirements elicitation in Chapter 4. As I said in Chapter 4, I find use case models to be more useful in the early stages of system design rather than in requirements engineering.

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 Mentcare system that represents the task of uploading data from the Mentcare system to a more general patient record system. This more general system maintains summary data about a patient rather than data about each consultation, which is recorded in the Mentcare system.

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 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 simple overview of an interaction, and you need to add more detail for complete interaction description. This detail can either be a simple textual description, a structured description in a table, or a sequence diagram. You choose 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.

Composite use case diagrams show a number of different use cases. Sometimes it is possible to include all possible interactions within 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 Mentcare system

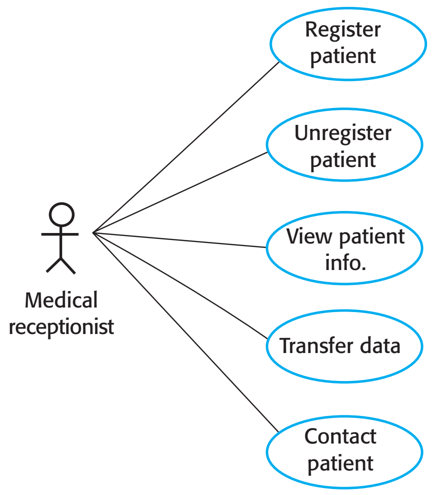


Figure 5.5 Use cases involving the role "Medical receptionist"

in which the actor "Medical Receptionist" is involved. Each of these should be accompanied by a more detailed description.

The UML includes a number of constructs for sharing all or part of a use case in other use case diagrams. While these constructs can sometimes be helpful for system designers, my experience is that many people, especially end-users, find them difficult to understand. For this reason, these constructs are not described here.

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. As space does not allow covering all possibilities here, the focus will be 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. Annotated arrows indicate interactions between objects. 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. This example also shows the notation used to denote alternatives. A box named alt is used with the

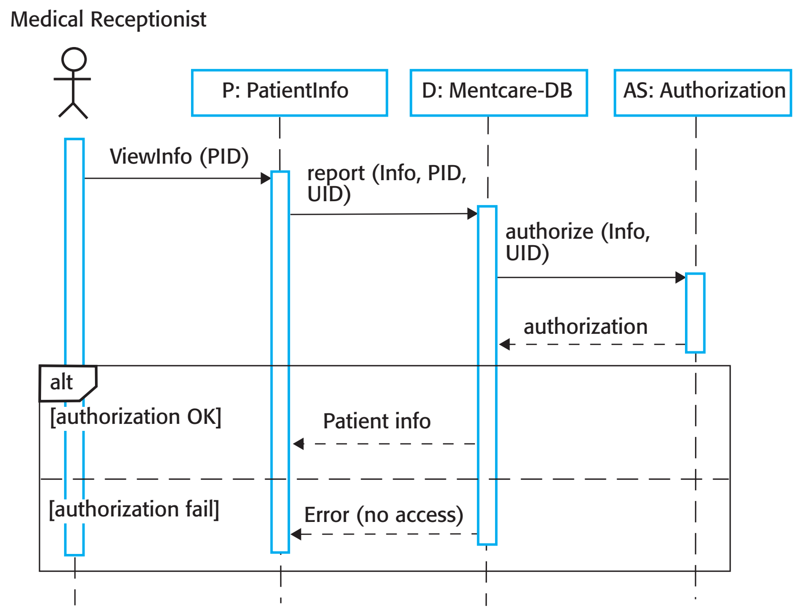


Figure 5.6 Sequence diagram for View patient information

conditions indicated in square brackets, with alternative interaction options separated by a dotted line.

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 to identify the required information. 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, it is not important where the receptionist's UID comes from.)
3. The database checks with an authorization system that the receptionist is authorized for this action.
4. If authorized, the patient information is returned and is displayed on a form on the user's screen. If authorization fails, then an error message is returned. The box denoted by "alt" in the top-left corner is a choice box indicating that one of the contained interactions will be executed. The condition that selects the choice is shown in square brackets.

Figure 5.7 is a further 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

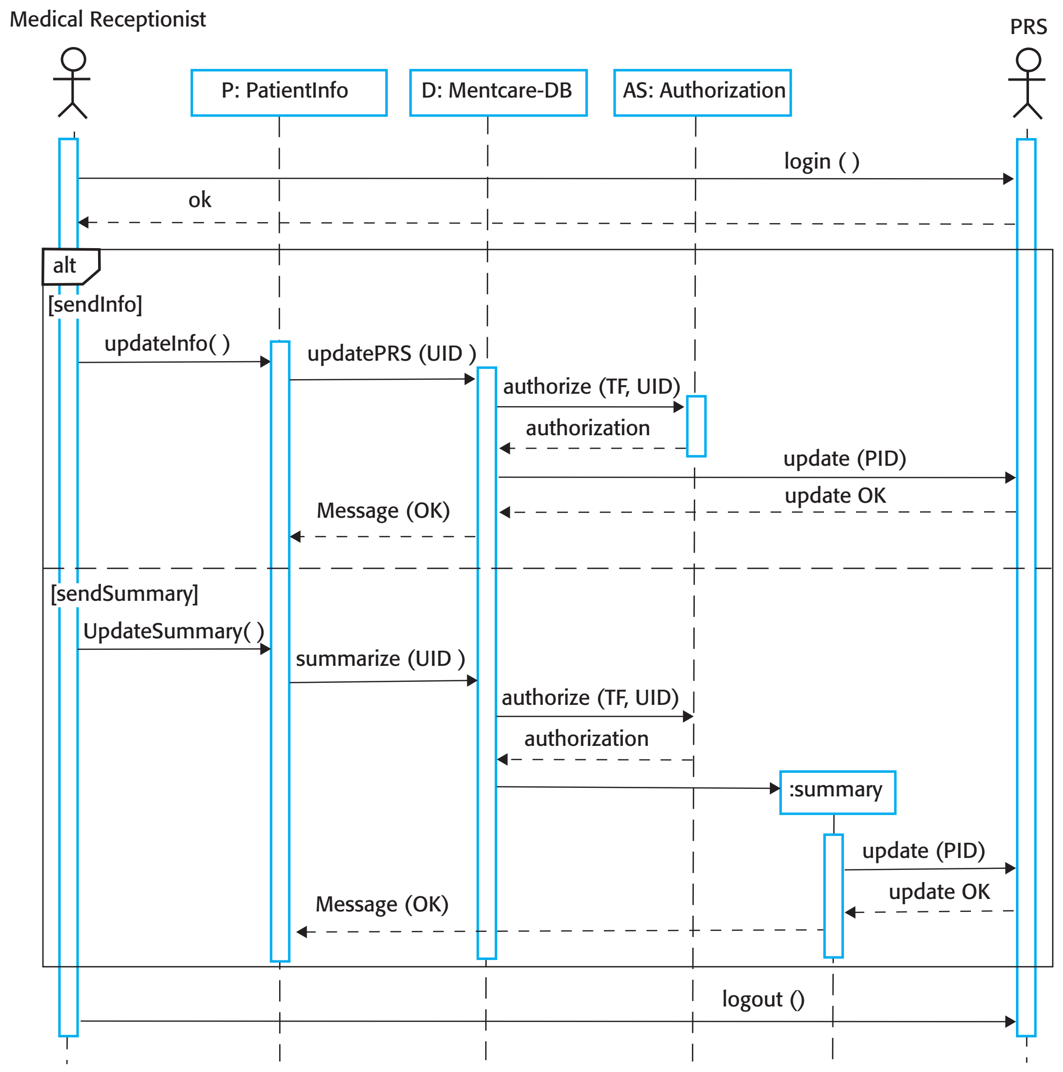


Figure 5.7 Sequence diagram for Transfer Data

to be uploaded to a national PRS (patient records system). You can read this diagram as follows:

1. The receptionist logs on to the PRS.
2. Two options are available (as shown in the "alt" box). These allow the direct transfer of updated patient information from the Mentcare database to the PRS and the transfer of summary health data from the Mentcare database 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.

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 that depend on implementation decisions. For example, in Figure 5.7 the decision on how to get the user identifier to check authorization is one that can be delayed. In an implementation, this might involve interacting with a User object. As this is not important at this stage, you do not need to include it in the sequence diagram.

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 organization 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. These can be models of the overall system architecture or more detailed models of the objects in the system and their relationships.

In this section, I focus on the use of class diagrams for modeling the static structure of the object classes in a software system. Architectural design is an important topic in software engineering, and UML component, package, and deployment diagrams may all be used when presenting architectural models. I cover architectural modeling in Chapters 6 and 17.

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 indicating that some relationship exists 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

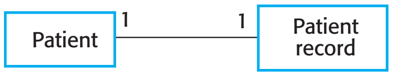


Figure 5.8 UML Classes and association

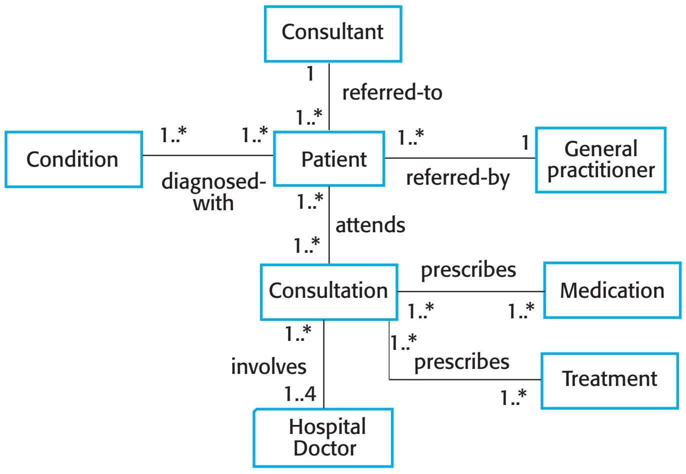


Figure 5.9 Classes and associations in the Mentcare system

prescription, or a doctor. As an implementation is developed, you define implementation objects to represent data that is manipulated by the system. In this section, the focus is on the modeling of real-world objects as part of the requirements or early software design processes. A similar approach is used for data structure modeling.

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 diagrams is to write the class name in a box. You can also note the existence of an 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. At this stage, you do not need to say what the association is.

Figure 5.9 develops the simple class diagram in Figure 5.8 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.

Figures 5.8 and 5.9, shows an important feature of class diagrams—the ability to show how many objects are involved in the association. In Figure 5.8 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 Figure 5.9, other multiplicities are possible. You can define that an exact number of objects are involved (e.g., 1..4) or, by using a *, indicate that there are an indefinite number of objects involved in the association. For example, the (1..*) multiplicity in Figure 5.9 on the relationship between Patient and Condition shows that a patient may suffer from several conditions and that the same condition may be associated with several patients.

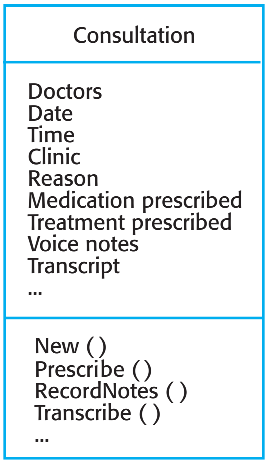


Figure 5.10 A Consultation class

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 (Hull and King 1987). The UML does not include a diagram type for database modeling, as it 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 (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 best to represent these classes in the simplest possible way, without attributes or operations. To define objects in more detail, you add information about their attributes (the object's characteristics) and operations (the object's functions). For example, a Patient object has the attribute Address, and you may include an operation called ChangeAddress, which is called when a patient indicates that he or she has moved from one address to another.

In the UML, you show attributes and operations by extending the simple rectangle that represents a class. I illustrate this in Figure 5.10 that shows an object representing a consultation between doctor and patient:

1. The name of the object class is in the top section.
2. The class attributes are in the middle section. This includes the attribute names and, optionally, their types. I don't show the types in Figure 5.10.
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. I show some but not all operations in Figure 5.10.

In the example shown in Figure 5.10, it is assumed 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.

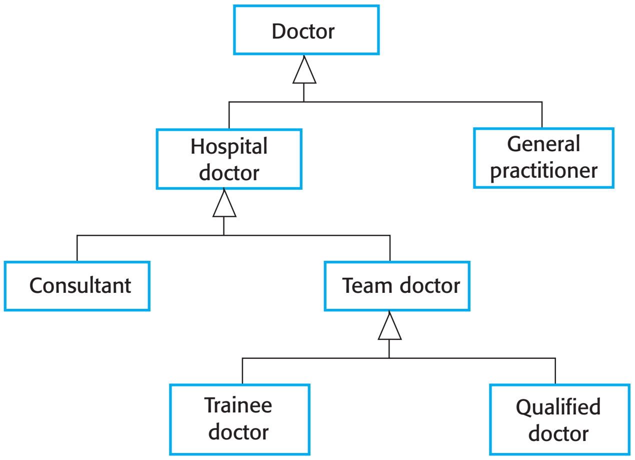


Figure 5.11 A generalization hierarchy

5.3.2 Generalization

Generalization is an everyday technique that we use to manage complexity. Rather than learn the detailed characteristics of everything that we experience, we learn about general classes (animals, cars, houses, etc.) and learn the characteristics of these classes. We then reuse knowledge by classifying things and focus on the differences between them and their class. For example, squirrels and rats are members of the class "rodents," and so share the characteristics of rodents. General statements apply to all class members; for example, all rodents have teeth for gnawing.

When you are modeling systems, it is often useful to examine the classes in a system to see if there is scope for generalization and class creation. 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. You can make the changes at the most general level. 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 indicates that general practitioners and hospital doctors can be generalized as doctors and that there are three types of Hospital Doctor: those who have just graduated from medical school and have to be supervised (Trainee Doctor); those who 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. The lower-level classes are subclasses that inherit the attributes and operations from their superclasses. These lower-level classes then add more specific attributes and operations.

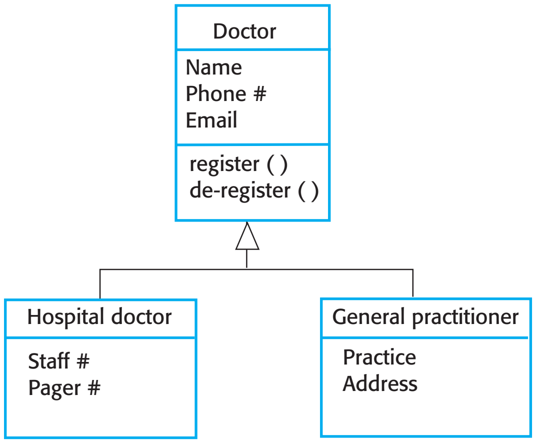


Figure 5.12 A generalization hierarchy with added detail

For example, all doctors have a name and phone number, and all hospital doctors have a staff number and carry a pager. General practitioners don't have these attributes, as they work independently, but they have an individual practice name and address. Figure 5.12 shows part of the generalization hierarchy, which I have extended with class attributes, for the class Doctor. The operations associated with the class Doctor are intended to register and de-register that doctor with the Mentcare system.

5.3.3 Aggregation

Objects in the real world are often made up 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, which means that one object (the whole) is composed of other objects (the parts). To define aggregation, a diamond shape is added to the link next to the class that represents the whole.

Figure 5.13 shows that a patient record is an aggregate of Patient and an indefinite number of Consultations. That is, the record maintains personal patient information as well as an individual record for each consultation with a doctor.

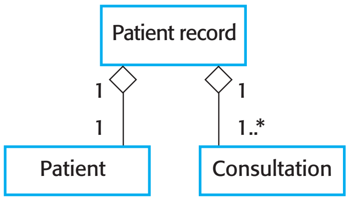


Figure 5.13 The aggregation association

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. Activity diagrams in the UML may be used to represent DFDs.

http://software-engineering-book.com/web/dfds/

5.4 Behavioral models

Behavioral models are models of the dynamic behavior of a 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. These stimuli may be either data or events:

1. Data becomes available that has to be processed by the system. The availability of the data triggers the processing.
2. An event happens that triggers system processing. Events may have associated data, although this is not always the case.

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 for that customer.

By contrast, real-time systems are usually event-driven, with limited data processing. For example, a landline phone switching system responds to events such as "handset activated" by generating a dial tone, pressing keys on a handset by capturing the phone number, and so on.

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 can be used during the analysis of requirements as they show end-to-end processing in a system. That is, they show the entire sequence of actions that takes place from an initial input being processed to the corresponding output, which is the system's response.

Data-driven models were among the first graphical software models. In the 1970s, structured design methods used data-flow diagrams (DFDs) as a way to illustrate the

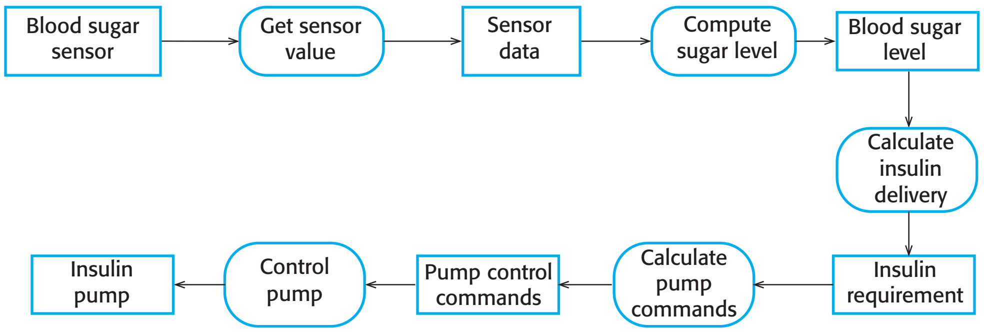


Figure 5.14 An activity model of the insulin pump's operation

processing steps in a system. Data-flow models are useful because tracking and documenting how data associated with a particular process moves through the system help analysts and designers understand what is going on in the process. DFDs are simple and intuitive and so are more accessible to stakeholders than some other types of model. It is usually possible to explain them to potential system users who can then participate in validating the model.

Data-flow diagrams can be represented in the UML using the activity diagram type, described in Section 5.1. Figure 5.14 is a simple activity diagram that shows the chain of processing involved in the insulin pump software. You can see the processing steps, represented as activities (rounded rectangles), and the data flowing between these steps, represented as objects (rectangles).

An alternative way of showing the sequence of processing in a system is to use UML sequence diagrams. You have seen how these diagrams can be used to model interaction, but if 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 processing an order and sending it to a supplier. Sequence models highlight objects in a system, whereas data-flow diagrams highlight the operations or activities. In practice, nonexperts seem to find data-flow diagrams more intuitive, but engineers prefer sequence diagrams.

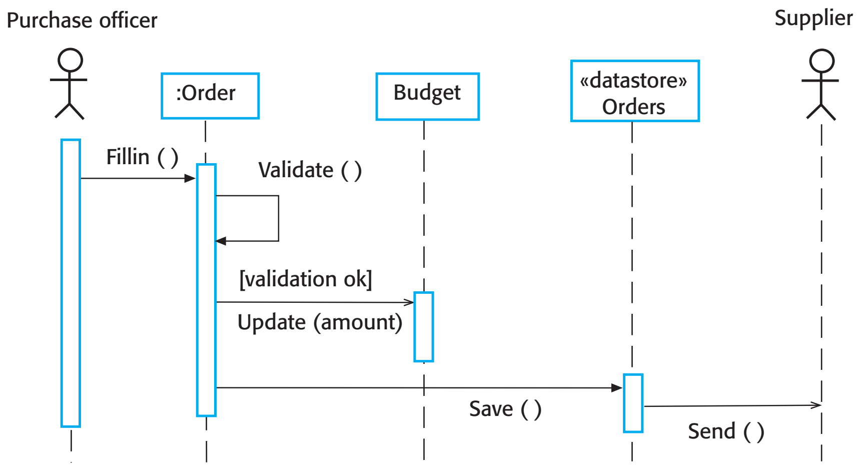


Figure 5.15 Order processing

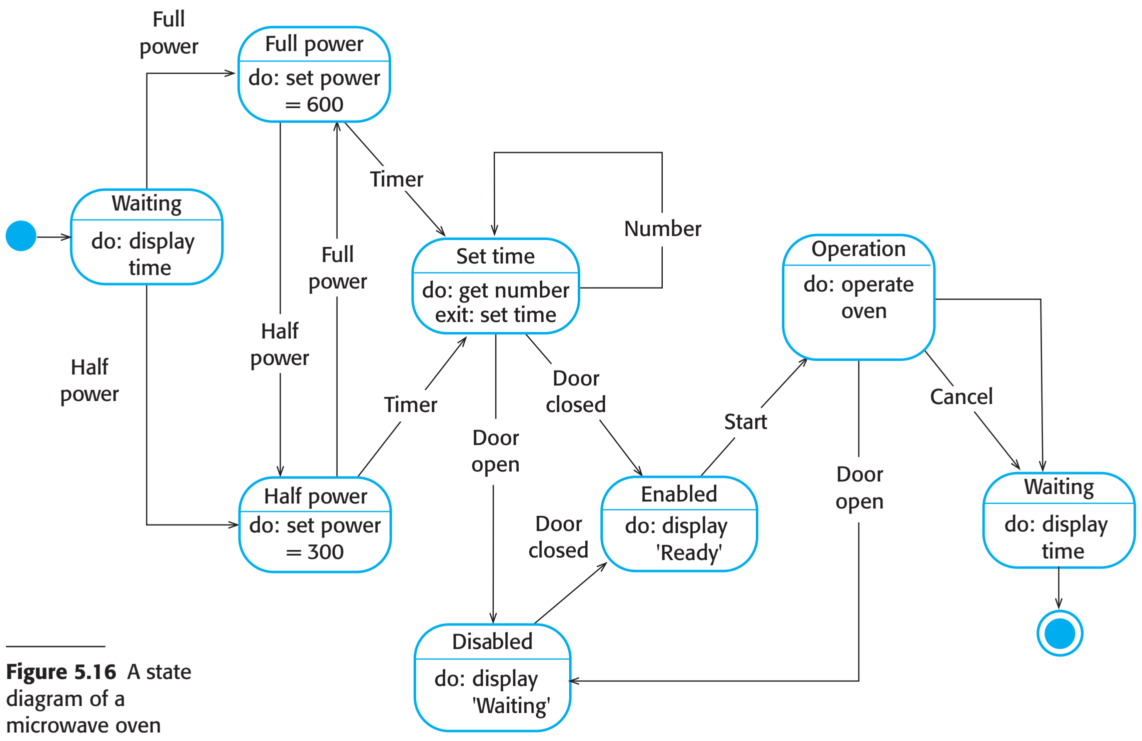


Figure 5.16 A state diagram of a microwave oven

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-driven modeling is used extensively when designing and documenting real-time systems (Chapter 21).

The UML supports event-based modeling using state diagrams, which are based on Statecharts (Harel 1987). State diagrams show system states and events that cause 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 (Figure 5.16). Real microwave ovens are much more complex than this system, but the simplified system is easier to understand. This simple oven 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.

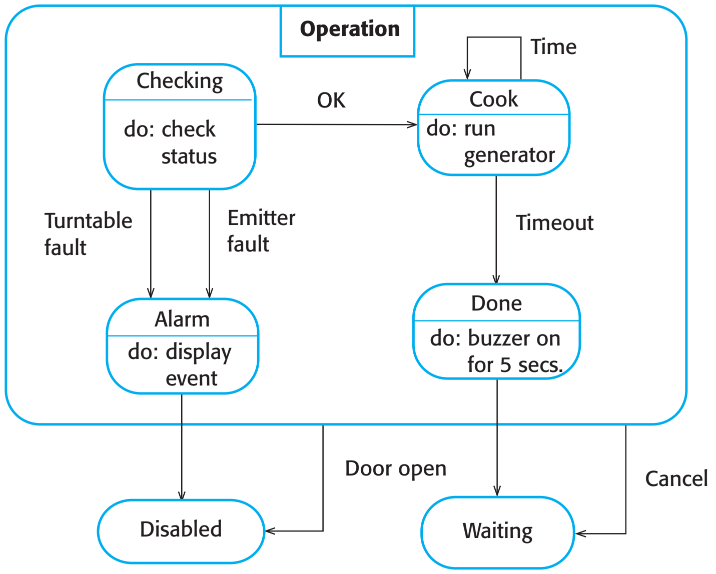


Figure 5.17 A state model of the Operation state

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

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 simple 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 their minds after selecting one of these and may 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 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 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 5.16. This is a superstate that can be expanded, as shown in Figure 5.17.

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 5 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 5.18 States and stimuli for the microwave oven

The Operation state includes a number of substates. 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 5.17.

State models of a system provide an overview of event processing, but you normally have to extend this with a more detailed description of the stimuli and the system states. You may use a table to list the states and events that stimulate state transitions along with a description of each state and event. Figure 5.18 shows a tabular description of each state and how the stimuli that force state transitions are generated.

5.4.3 Model-driven engineering

Model-driven engineering (MDE) is an approach to software development whereby models rather than programs are the principal outputs of the development process

(Brambilla, Cabot, and Wimmer 2012). The programs that execute on a hardware/software platform are 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 was developed from the idea of model-driven architecture (MDA). This was proposed by the Object Management Group (OMG) as a new software development paradigm (Mellor, Scott, and Weise 2004). 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 are not considered in MDA.

MDA as an approach to system engineering has been adopted by a number of large companies to support their development processes. This section focuses on the use of MDA for software implementation rather than discuss more general aspects of MDE. The take-up of more general model-driven engineering has been slow, and few companies have adopted this approach throughout their software development life cycle. In his blog, den Haan discusses possible reasons why MDE has not been widely adopted (den Haan 2011).

5.5 Model-driven architecture

Model-driven architecture (Mellor, Scott, and Weise 2004; Stahl and Voelter 2006) is a model-focused approach to software design and implementation that uses a subset of UML models to describe a system. Here, models at different 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)* CIMs model the important domain abstractions used in a system and so 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 and consultations.

2. *A platform-independent model (PIM)* PIMs model the operation of the system without reference to its implementation. A PIM is usually described using UML models that show the static system structure and how it responds to external and internal events.

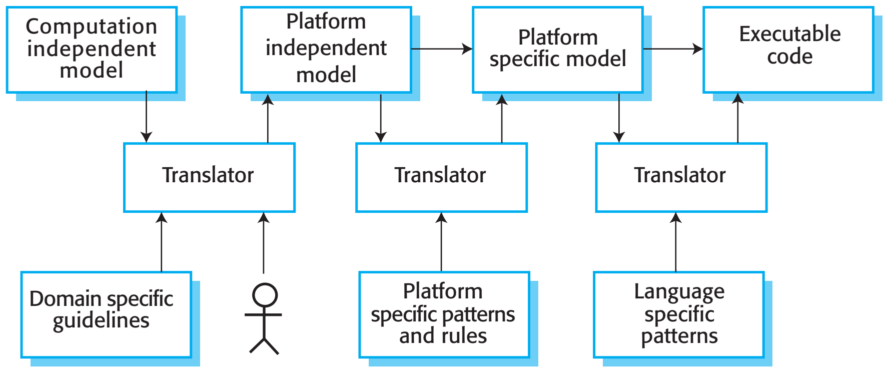

Computation independent model
Platform independent model
Platform specific model
Executable code
Translator
Translator
Translator
Domain specific guidelines
Platform specific patterns and rules
Language specific patterns

Figure 5.19 MDA transformations

3. *Platform-specific models (PSM)* PSMs 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.

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, you write a model translator for that platform. When this is available, all platform-independent models can then be rapidly re-hosted on the new platform.

Fundamental to MDA is the notion that transformations between models can be defined and applied automatically by software tools, as illustrated in Figure 5.19. This diagram also shows a final level of automatic transformation where a transformation is applied to the PSM to generate the executable code that will run on the designated software platform. Therefore, in principle at least, executable software can be generated from a high-level system model.

In practice, completely automated translation of models to code is rarely possible. The translation of high-level CIM to PIM models remains a research problem, and for production systems, human intervention, illustrated using a stick figure in Figure 5.19, is normally required. A particularly difficult problem for automated model transformation is the need to link the concepts used in different CIMS. For example, the concept of a role in a security CIM that includes role-driven access control may have to be mapped onto the concept of a staff member in a hospital CIM. Only a person who understands both security and the hospital environment can make this mapping.

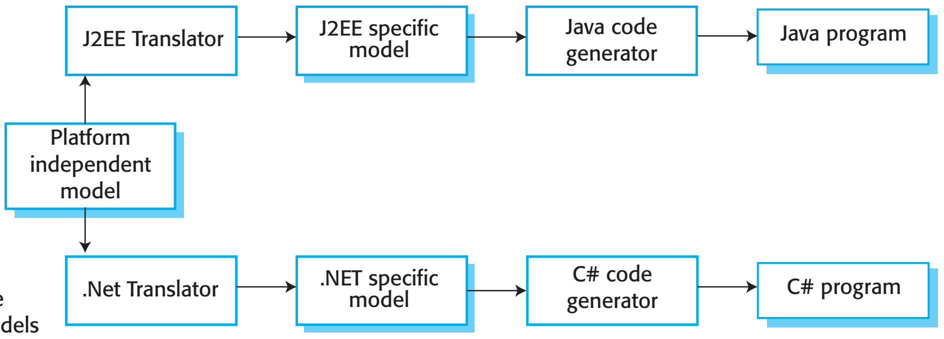


Figure 5.20 Multiple platform-specific models

The translation of platform-independent to platform-specific models is a simpler technical problem. Commercial tools and open-source tools (Koegel 2012) are available that provide translators from PIMS to common platforms such as Java and J2EE. These use an extensive library of platform-specific rules and patterns to convert a PIM to a 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, in principle, you only have to maintain a single PIM. The PSMs for each platform are automatically generated (Figure 5.20).

Although MDA support tools include platform-specific translators, these sometimes only offer partial support for translating PIMS to PSMs. The execution environment for a system is more than the standard execution platform, such as J2EE or Java. It also includes other application systems, specific application libraries that may be created for a company, external services, and user interface libraries.

These vary from one company to another, so off-the-shelf tool support is not available that takes these into account. Therefore, when MDA is introduced into an organization, special-purpose translators may have to be created to make use of the facilities available in the local environment. This is one reason why many companies have been reluctant to take on model-driven approaches to development. They do not want to develop or maintain their own tools or to rely on small software companies, who may go out of business, for tool development. Without these specialist tools, model-based development requires additional manual coding which reduces the cost-effectiveness of this approach.

I believe that there are several other reasons why MDA has not become a mainstream approach to software development.

1. Models are a good way of facilitating discussions about a software design. However, it does not always follow that the abstractions that are useful for discussions are the right abstractions for implementation. You may decide to use a completely different implementation approach that is based on the reuse of off-the-shelf application systems.

2. For most complex systems, implementation is not the major problem—requirements engineering, security and dependability, integration with legacy

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 with clearly defined meanings that can be compiled to executable code. 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).

http://software-engineering-book.com/web/xuml/

systems and testing are all more significant. Consequently, the gains from the use of MDA are limited.

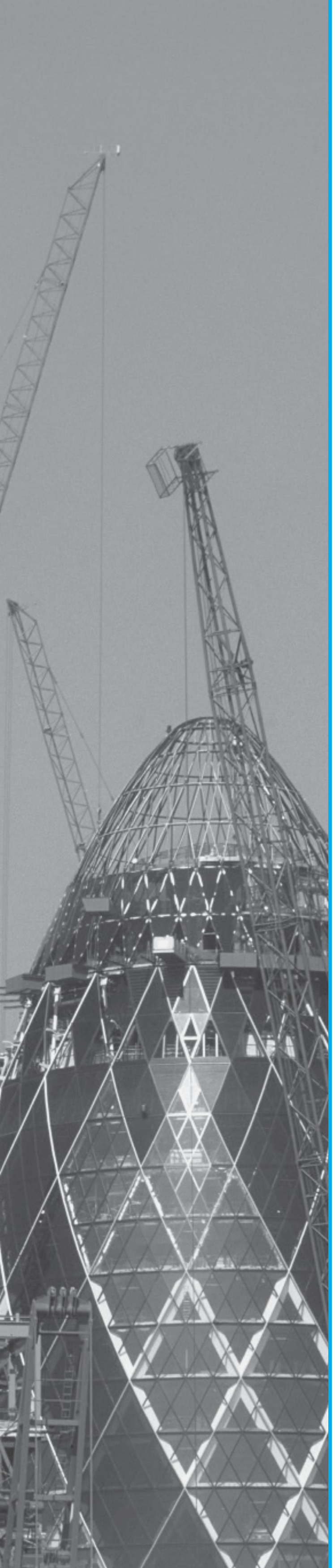
3. The arguments for platform independence are only valid for large, long-lifetime systems, where the platforms become obsolete during a system's lifetime. For software products and information systems that are developed for standard platforms, such as Windows and Linux, the savings from the use of MDA are likely to be outweighed by the costs of its introduction and tooling.

4. The widespread adoption of agile methods over the same period that MDA was evolving has diverted attention away from model-driven approaches.

The success stories for MDA (OMG 2012) have mostly come from companies that are developing systems products, which include both hardware and software. The software in these products has a long lifetime and may have to be modified to reflect changing hardware technologies. The domain of application (automotive, air traffic control, etc.) is often well understood and so can be formalized in a CIM.

Hutchinson and his colleagues (Hutchinson, Rouncefield, and Whittle 2012) report on the industrial use of MDA, and their work confirms that successes in the use of model-driven development have been in systems products. Their assessment suggests that companies have had mixed results when adopting this approach, but the majority of users report that using MDA has increased productivity and reduced maintenance costs. They found that MDA was particularly useful in facilitating reuse, and this led to major productivity improvements.

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. Ambler, a pioneer in the development of agile methods, suggests that some aspects of MDA can be used in agile processes (Ambler 2004) but considers automated code generation to be impractical. However, Zhang and Patel report on Motorola's success in using agile development with automated code generation (Zhang and Patel 2011).



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 software architecture during the architectural design process;
- have been introduced to the idea of Architectural patterns, well-tried ways of organizing software architectures that can be reused in system designs;
- understand how Application-Specific Architectural patterns may be used in transaction processing and language processing systems.

Contents

Architectural design is concerned with understanding how a software system should be organized and designing the overall structure of that system. In the model of the software development process that I described 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 an agile development process should focus on designing an overall system architecture. Incremental development of architectures is not usually successful. Refactoring components in response to changes is usually relatively easy. However, refactoring the system architecture is expensive because you may need to modify most system components to adapt them to the architectural changes.

To help you understand what I mean by system architecture, look at Figure 6.1. This diagram shows an abstract model of the architecture for a packing robot system. This robotic system can pack different kinds of objects. 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

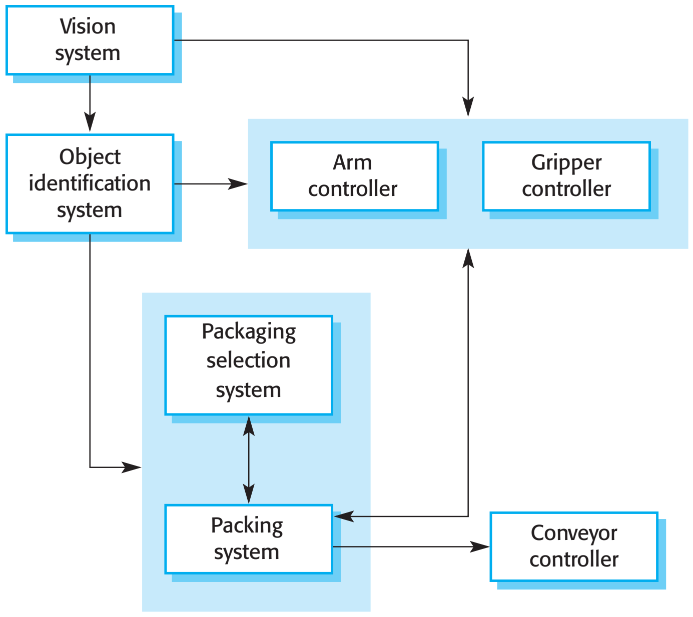


Figure 6.1 The architecture of a packing robot control system

include any design information. This ideal is unrealistic, however, except for very small systems. You need to identify the main architectural components as these reflect the high-level features of the system. 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 subsystems. You then use this decomposition to discuss the requirements and more detailed 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 may be distributed over different computers, which may be owned and managed by different companies. (I cover architecture in the large in Chapters 17 and 18.)

Software architecture is important because it affects the performance, robustness, distributability, and maintainability of a system (Bosch 2000). As Bosch explains, individual components implement the functional system requirements, but the dominant influence on the non-functional system characteristics is the system's architecture. Chen et al. (Chen, Ali Babar, and Nuseibeh 2013) confirmed this in a study of "architecturally significant requirements" where they found that non-functional requirements had the most significant effect on the system's architecture.

Bass et al. (Bass, Clements, and Kazman 2012) suggest that explicitly designing and documenting software architecture has three advantages:

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* An architectural model 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 15, product-line architectures are an approach to reuse where the same architecture is reused across a range of related systems.

System architectures are often modeled informally 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 subcomponents. 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 handbook of software architecture (Booch 2014).

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. In spite of their widespread use, Bass et al. (Bass, Clements, and Kazman 2012) 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 architectural theory and industrial practice arise because there are two ways in which an architectural model of a program is used:

1. *As a way of encouraging discussions 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 that 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 such a model is that such a detailed architectural description makes it easier to understand and evolve the system.

Block diagrams are a good way of supporting communications between the people involved in the software design process. They are intuitive, and domain experts and software engineers can relate to them and participate in discussions about the system. Managers find them helpful in planning the project. For many projects, block diagrams are the only architectural description.

Ideally, if the architecture of a system is to be documented in detail, it is better to use a more rigorous notation for architectural description. Various architectural description languages (Bass, Clements, and Kazman 2012) have been developed for this purpose. A more detailed and complete description means that there is less scope for misunderstanding the relationships between the architectural components. However, developing a detailed architectural description is an expensive and time-consuming process. It is practically impossible to know whether or not it is cost-effective, so this approach is not widely used.

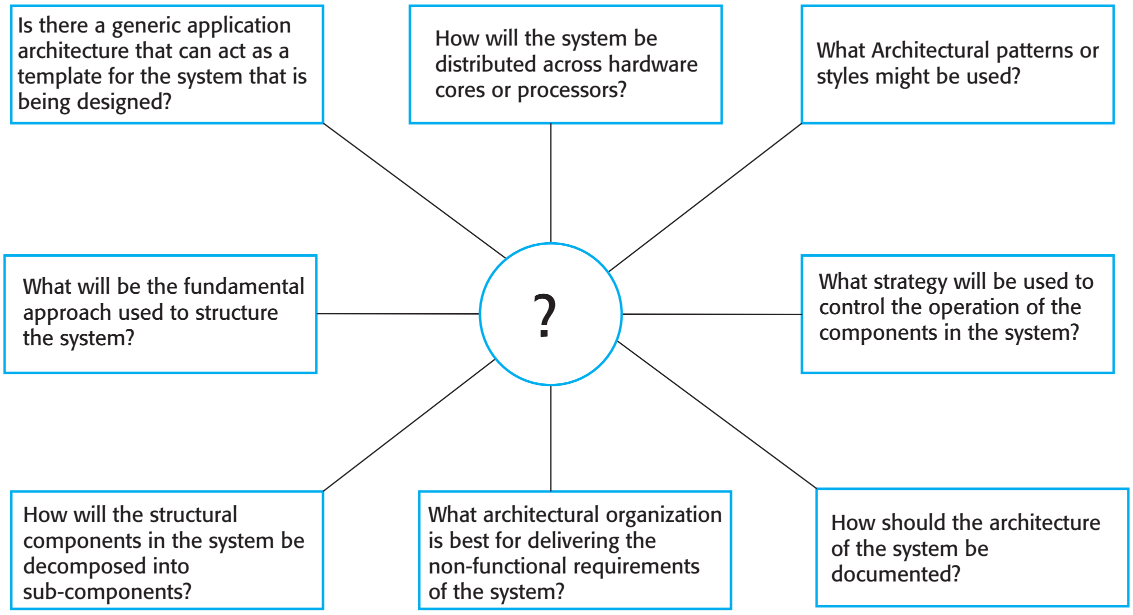

Is there a generic application architecture that can act as a template for the system that is being designed?
How will the system be distributed across hardware cores or processors?
What Architectural patterns or styles might be used?
What will be the fundamental approach used to structure the system?
?
What strategy will be used to control the operation of the components in the system?
How will the structural components in the system be decomposed into sub-components?
What architectural organization is best for delivering the non-functional requirements of the system?
How should the architecture of the system be documented?

Figure 6.2 Architectural design decisions

6.1 Architectural design decisions

Architectural design is a creative process in which you design a system organization that will satisfy the functional and non-functional requirements of a system. There is no formulaic architectural design process. It depends on the type of system being developed, the background and experience of the system architect, and the specific requirements for the system. Consequently, I think it is best to consider 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 fundamental questions shown in Figure 6.2.

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.

For embedded systems and apps designed for personal computers and mobile devices, you do not have to design a distributed architecture for the system. However, most large systems are 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 that I cover in Chapter 17.

The architecture of a software system may be based on a particular Architectural pattern or style (these terms have come to mean the same thing). 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 cover several frequently used patterns in Section 6.3.

Garlan and Shaw's notion of an architectural style covers questions 4 to 6 in the list of fundamental architectural questions shown in Figure 6.2. 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 a strategy for decomposing components into subcomponents. Finally, in the control modeling process, you develop a general model of the control relationships between the various parts of the system and make decisions about how the execution of components is controlled.

Because of the close relationship between non-functional system characteristics and software architecture, the choice of architectural style and structure should depend on the non-functional requirements of the system:

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 deployed on the same computer rather than distributed across the network. This may mean using a few relatively large components rather than small, finer-grain components. Using large components reduces the number of component communications, as most of the interactions between related system features take place within a component. You may also consider runtime 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 and 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 co-located in a single component or in a small number of components. This reduces the costs and problems of safety validation and may make 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 fault-tolerant system architectures for high-availability systems in Chapter 11.

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, however, then some compromise must be found. You can sometimes do this by using different Architectural patterns or styles for separate parts of the system. Security is now almost always a critical requirement, and you have to design an architecture that maintains security while also satisfying other non-functional requirements.

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 description (Bosch 2000) of the non-functional characteristics of some Architectural patterns can 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 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 diagram, as a graphical model can only show one view or perspective of the system. It might show how a system is decomposed into modules, how the runtime processes interact, or the different ways in which system components are distributed across a network. Because all of these are useful at different times, 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 (Krutchen 1995) in his well-known 4+1 view model of software architecture, suggests that there should

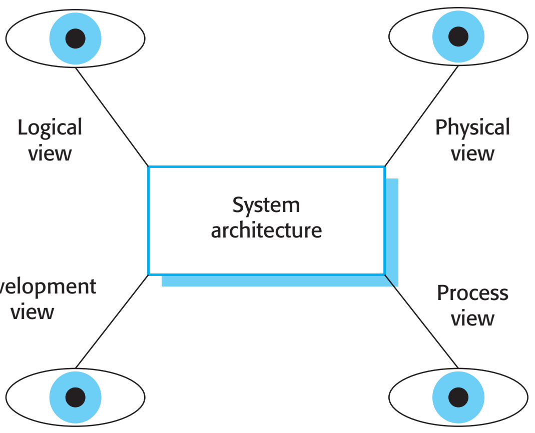


Figure 6.3 Architectural views

be four fundamental architectural views, which can be linked through common use cases or scenarios (Figure 6.3). He suggests the following views:

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 runtime, 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. (Hofmeister, Nord, and Soni 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 15) 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 of a system's architecture are almost always developed during the design process. They are used to explain the system architecture to stakeholders and to inform architectural decision making. During the design process, some of the other views may also be developed when different aspects of the system are discussed, but it is rarely necessary to develop 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 describing and documenting software architectures. A survey in 2006 (Lange, Chaudron, and Muskens 2006) showed that, when the UML was used, it was mostly applied in an 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 that 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 (Bass, Clements, and Kazman 2012) have proposed the use of more specialized architectural description languages (ADLs) 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 ADLs are specialist languages, domain and application specialists find it hard to understand and use ADLs. There may be some value in using domain-specific ADLs as part of model-driven development, but I do not think they will become part of mainstream software engineering practice. 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 these documents. I largely agree with this view, and I think that, except for critical systems, it is not worth developing a detailed architectural description from Krutchen's four perspectives. You should develop the views that are useful for communication and not worry about whether or not your architectural documentation is complete.

6.3 Architectural patterns

The idea of patterns as a way of presenting, sharing, and reusing knowledge about software systems has been adopted in a number of areas of software engineering. The trigger for this was the publication of a book on object-oriented design patterns (Gamma et al. 1995). This prompted the development of other types of patterns, such as patterns for organizational design (Coplien and Harrison 2004), usability patterns (Usability Group 1998), patterns of cooperative interaction (Martin and Sommerville 2004), and configuration management patterns (Berczuk and Appleton 2002).

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

In this section, I introduce Architectural patterns and briefly describe a selection of Architectural patterns that are commonly used. Patterns may be described in a standard way (Figures 6.4 and 6.5) using a mixture of narrative description and diagrams.

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 6.5.
Example	Figure 6.6 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	May involve additional code and code complexity when the data model and interactions are simple.

Figure 6.4 The Model-View-Controller (MVC) pattern

For more detailed information about patterns and their use, you should refer to the 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 on when it is and is not appropriate to use that pattern, and details on the pattern's strengths and weaknesses.

Figure 6.4 describes the well-known Model-View-Controller pattern. This pattern is the basis of interaction management in many web-based systems and is supported by most language frameworks. The stylized pattern description includes the pattern

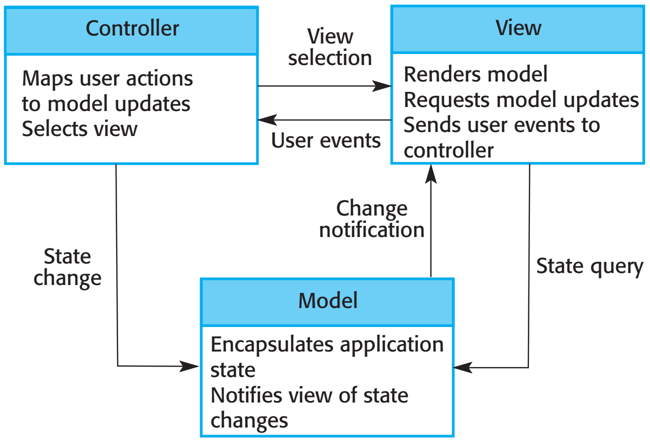


Figure 6.5 The organization of the Model-View-Controller

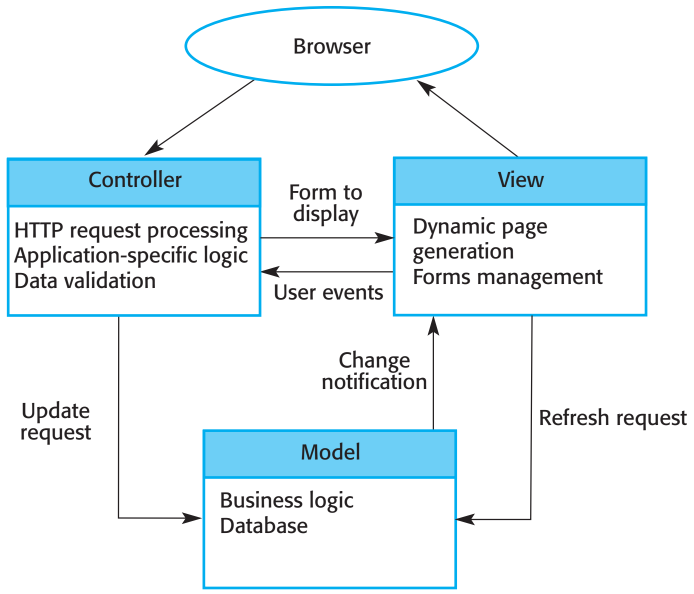


Figure 6.6 Web application architecture using the MVC pattern

name, a brief description, a graphical model, and an example of the type of system where the pattern is used. 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 6.5 and 6.6. These present the architecture from different views: Figure 6.5 is a conceptual view, and Figure 6.6 shows a runtime system architecture when this pattern is used for interaction management in a web-based system.

In this short space, it is impossible to describe all of the generic patterns that can be used in software development. Instead, I present some selected examples of patterns that are widely used and that capture good architectural design principles.

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 6.4, 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 6.7. 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. If its interface is unchanged, a new layer with extended functionality can replace an existing layer

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 6.8.
Example	A layered model of a digital learning system to support learning of all subjects in schools (Figure 6.9).
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 multilevel security.
Advantages	Allows replacement of entire layers as 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 6.7 The Layered Architecture pattern

without changing other parts of the system. 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, this makes it easier to provide multi-platform implementations of an application system. Only the machine-dependent layers need be reimplemented to take account of the facilities of a different operating system or database.

Figure 6.8 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, which includes the components concerned with the application functionality and utility components used by other application components.

The third layer is concerned with user interface 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.

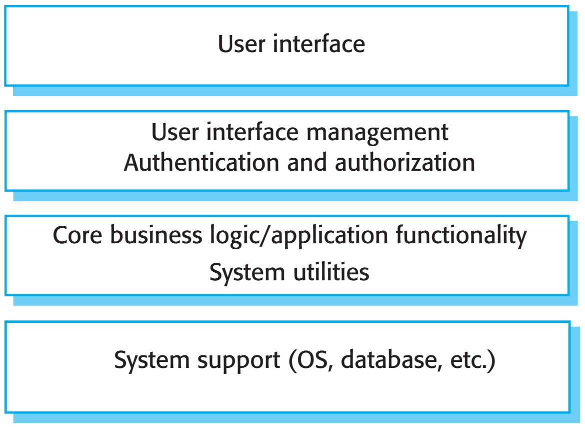

User interface
User interface management
Authentication and authorization
Core business logic/application functionality
System utilities
System support (OS, database, etc.)

Figure 6.8 A generic layered architecture

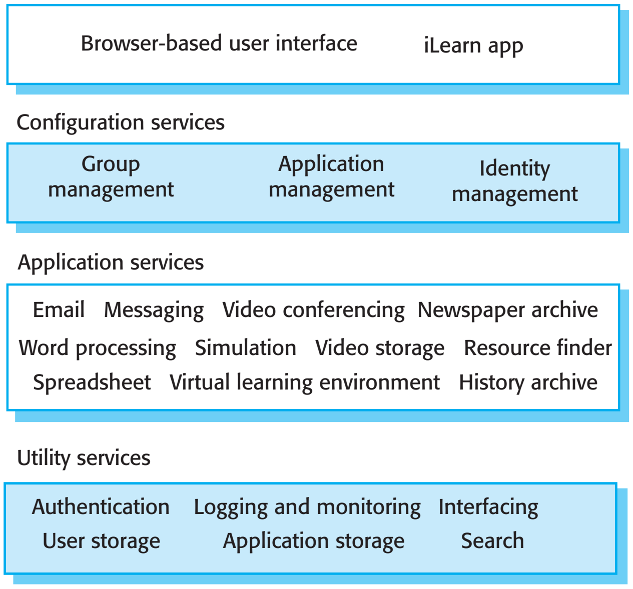


Figure 6.9 The architecture of the iLearn system

Figure 6.9 shows that the iLearn digital learning system, introduced in Chapter 1, has a four-layer architecture that follows this pattern. You can see another example of the Layered Architecture pattern in Figure 6.19 (Section 6.4, which shows the organization of the Mentcare 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 6.10), describes how a set of interacting components can share data.

Figure 6.10 The Repository pattern

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 6.11 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.

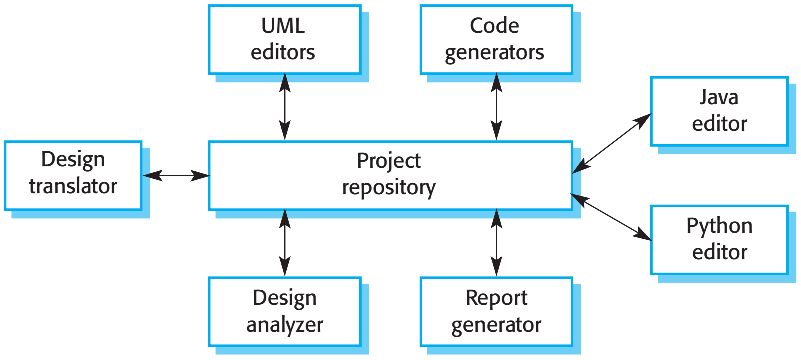


Figure 6.11 A repository architecture for an IDE

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

Figure 6.11 illustrates 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 rollback to earlier versions.

Organizing tools around a repository is an efficient way of sharing 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, this involves maintaining multiple copies of data. Keeping these consistent and up to date adds more overhead to the system.

In the repository architecture shown in Figure 6.11, the repository is passive and control is the responsibility of the components using the repository. An alternative approach, which has been derived for artificial intelligence (AI) systems, uses a "blackboard" model that triggers components when particular data become available. This is appropriate when the data in the repository is unstructured. Decisions about which tool is to be activated can only be made when the data has been analyzed. This model was introduced by Nii (Nii 1986), and Bosch (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 runtime organization. My next example, the Client–Server pattern (Figure 6.12), illustrates a commonly used runtime organization for distributed

Name	Client–server
Description	In a client–server architecture, the system is presented as a set of services, with each service delivered by a separate server. Clients are users of these services and access servers to make use of them.
Example	Figure 6.13 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 and so is susceptible to denial-of-service attacks or server failure. Performance may be unpredictable because it depends on the network as well as the system. Management problems may arise if servers are owned by different organizations.

Figure 6.12 The Client–Server pattern

systems. 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 that offers programming language compilation services. Servers are software components, and several servers may run on the same computer.

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. Client–server systems are usually 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 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 http), where a client makes a request to a server and waits until it receives a reply from that server.

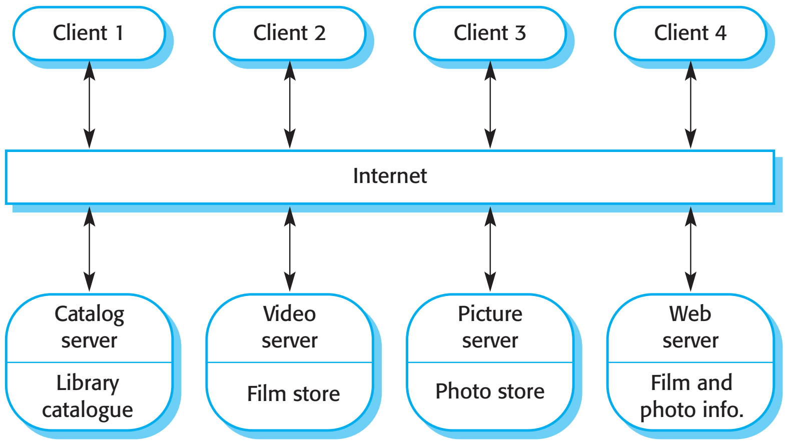


Figure 6.13 A client–server architecture for a film library

Figure 6.13 is an example of a system that is based on the client–server model. This is a multiuser, 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 include data about the film and video clips, and an e-commerce system that supports the sale of photographs, film, and video clips. The 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 cover distributed architectures in Chapter 17, where I explain the client–server model and its variants in more detail.

6.3.4 Pipe and filter architecture

My final example of a general Architectural pattern is the Pipe and Filter pattern (Figure 6.14). This is a model of the runtime 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.

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 6.15 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 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 architectural components that use incompatible data structures.

Figure 6.14 The Pipe and Filter pattern

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 such as billing systems. The architecture of an embedded system may also be organized as a process pipeline, with each process executing concurrently. I cover use of this pattern in embedded systems in Chapter 21.

An example of this type of system architecture, used in a batch processing application, is shown in Figure 6.15. An organization has issued invoices to customers. Once a week, payments that have been made are reconciled with the invoices. For

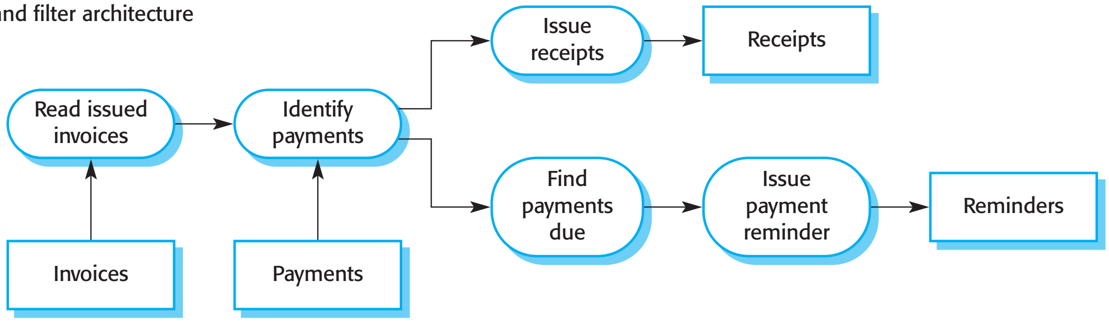


Figure 6.15 An example of the pipe and filter architecture

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://software-engineering-book.com/web/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.

Pipe and filter systems are best suited to batch processing systems and embedded systems where there is limited user interaction. Interactive systems are difficult to write using the pipe and filter model because of the need for a stream of data to be processed. While 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 implement this as a sequential stream that conforms to the pipe and filter model.

6.4 Application architectures

Application systems are intended to meet a business or an 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 and meter calls, manage their network and issue bills to customers. 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 reimplemented when developing new systems. However, for many business systems, application architecture reuse is implicit when generic application systems are configured to create a new application. We see this in the widespread use of Enterprise Resource Planning (ERP) systems and off-the-shelf configurable application systems, such as systems for accounting and stock control. These systems have a standard architecture and components. The components are configured and adapted to create a specific business application.

Application architectures

There are several examples of application architectures on the book's website. These include descriptions of batch data-processing systems, resource allocation systems, and event-based editing systems.

http://software-engineering-book.com/web/apparch/

For example, a system for supply chain management can be adapted for different types of suppliers, goods, and contractual arrangements.

As a software designer, you can use models of application architectures in a number of ways:

1. *As a starting point for the architectural design process* If you are unfamiliar with the type of application that you are developing, you can base your initial design on a generic application architecture. You then specialize this for the specific system that is being developed.

2. *As a design checklist* If you have developed an architectural design for an application system, you can compare this with the generic application architecture. You can check that your design is consistent with the generic architecture.

3. *As a way of organizing the work of the development team* The application architectures identify stable structural features of the system architectures, and in many cases, it is possible to develop these in parallel. You can assign work to group members to implement different components within the architecture.

4. *As a means of assessing components for reuse* If you have components you might be able to reuse, you can compare these with the generic structures to see whether there are comparable components in the application architecture.

5. *As a vocabulary for talking about applications* If you are discussing a specific application or trying to compare applications, then you can use the concepts identified in the generic architecture to talk about these applications.

There are many types of application system, and, in some cases, they may seem to be very different. However, superficially dissimilar applications may have much in common and thus share an abstract application architecture. I illustrate this by describing the architectures of two types of application:

1. *Transaction processing applications* Transaction processing applications are database-centered applications that process user requests for information and update the information in a database. These are the most common types of interactive business systems. They are organized in such a way that user actions can't interfere with each other and the integrity of the database is maintained. This class of system includes interactive banking systems, e-commerce systems, information systems, and booking systems.

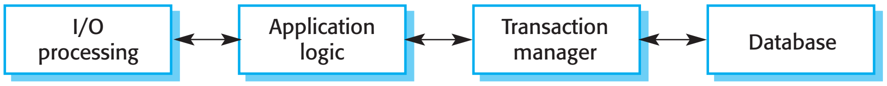


Figure 6.16 The structure of transaction processing applications

2. *Language processing systems* Language processing systems are systems in which the user's intentions are expressed in a formal language, such as a programming language. The language processing system processes this language into an internal format and then interprets this internal representation. The best-known language processing systems are compilers, which translate high-level language programs into machine code. However, language processing systems are also used to interpret command languages for databases and information systems, and markup languages such as XML.

I have chosen these particular types of system because a large number of web-based business systems are transaction processing systems, and all software development relies on language processing systems.

6.4.1 Transaction processing systems

Transaction processing systems are designed to process user requests for information from a database, or requests to update a database (Lewis, Bernstein, and Kifer 2003). Technically, a database transaction is part of a sequence of operations and is treated as a single unit (an atomic unit). All of the operations in a transaction have to be completed before the database changes are made permanent. This ensures that failure of operations within a transaction does not lead to inconsistencies in the database.

From a user perspective, a transaction is any coherent sequence of operations that satisfies a goal, such as "find the times of flights from London to Paris." If the user transaction does not require the database to be changed, then it may not be necessary to package this as a technical database transaction.

An example of a database transaction is a customer request to withdraw money from a bank account using an ATM. This involves checking the customer account balance to see if sufficient funds are available, modifying the balance by the amount withdrawn and sending commands to the ATM to deliver the cash. Until all of these steps have been completed, the transaction is incomplete and the customer accounts database is not changed.

Transaction processing systems are usually interactive systems in which users make asynchronous requests for service. Figure 6.16 illustrates the conceptual architectural structure of transaction processing applications. First, a user makes a request to the system through an I/O processing component. The request is processed by some application-specific logic. A transaction is created and passed to a transaction manager, which is usually embedded in the database management system. After the transaction manager 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

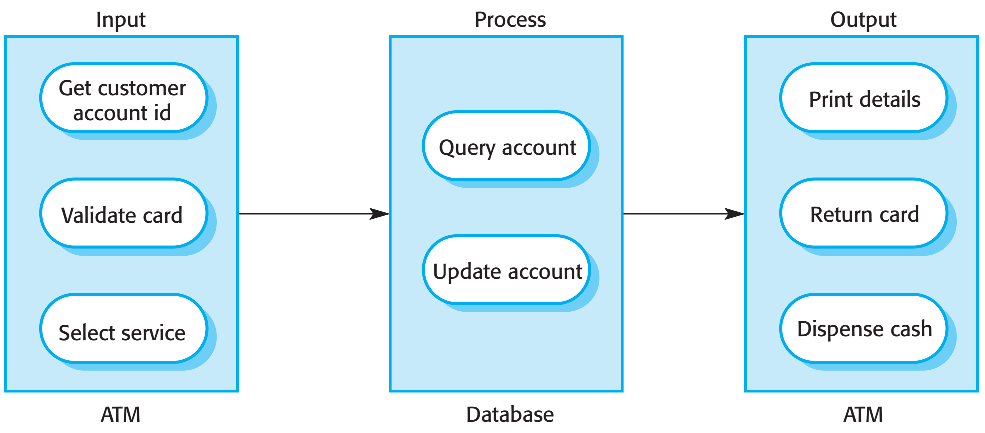


Figure 6.17 The software architecture of an ATM system

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 6.17 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. Information systems are almost always web-based systems, where the user interface is implemented in a web browser.

Figure 6.18 presents 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

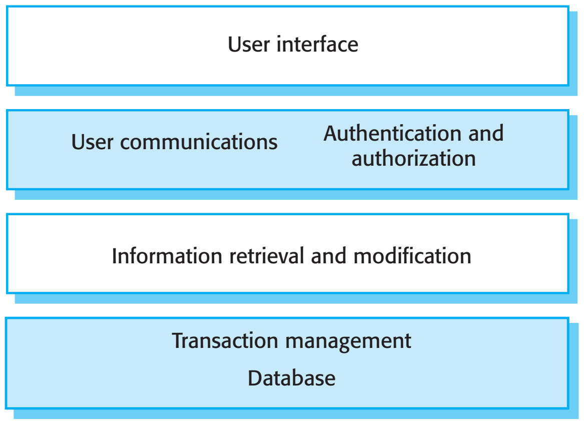

User interface
User communications
Authentication and authorization
Information retrieval and modification
Transaction management
Database

Figure 6.18 Layered information system architecture

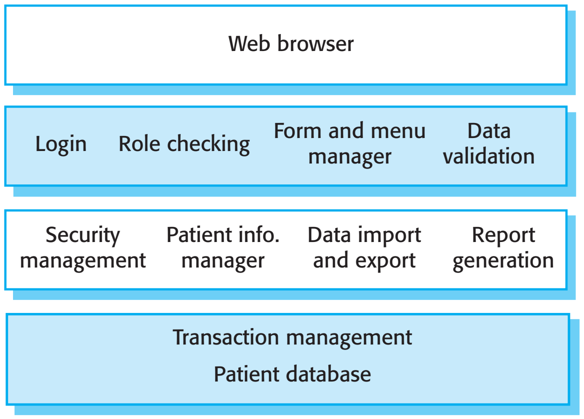


Figure 6.19 The architecture of the Mentcare system

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. The layers in this model can map directly onto servers in a distributed Internet-based system.

As an example of an instantiation of this layered model, Figure 6.19 shows the architecture of the Mentcare system. Recall that this system maintains and manages details of patients who are consulting specialist doctors about mental health problems. I have added detail to each layer in the model by identifying the components that support user communications and information retrieval and access:

1. The top layer is a browser-based user interface.
2. The second layer provides the user interface functionality that is delivered through the web browser. It includes components to allow users to log in to the system and checking components that ensure that the operations they use are allowed by their role. This layer includes form and menu management components that present information to users, and data validation components that check information consistency.
3. The third layer implements the functionality of the system and provides components that implement system security, patient information creation and updating, import and export of patient data from other databases, and report generators that create management reports.
4. Finally, the lowest layer, which is built using a commercial database management system, provides transaction management and persistent data storage.

Information and resource management systems are sometimes also transaction processing systems. For example, e-commerce systems are Internet-based resource management systems that accept electronic orders for goods or services and then arrange delivery of these goods or services to the customer. In an e-commerce

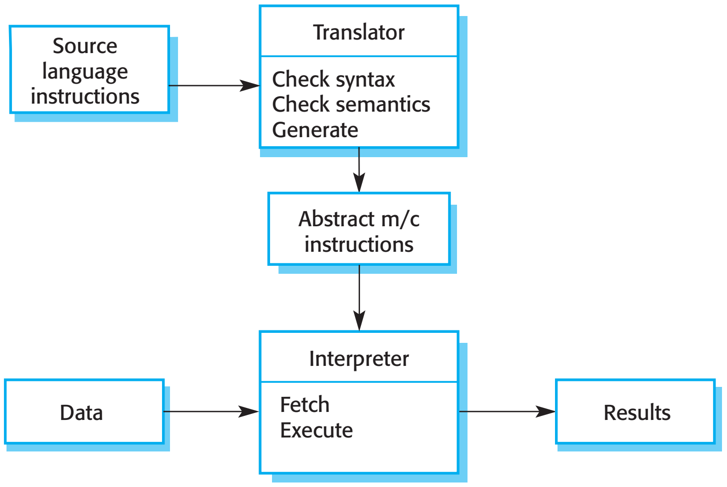


Figure 6.20 The architecture of a language processing system

system, the application-specific layer includes additional functionality supporting a "shopping cart" in which users can place a number of items in separate transactions, then pay for them all together in a single transaction.

The organization of servers in these systems usually reflects the four-layer generic model presented in Figure 6.18. These systems are often implemented as distributed systems with a multitier client server/architecture

1. The web server is responsible for all user communications, with the user interface implemented using a web browser;
2. The application server is responsible for implementing application-specific logic as well as information storage and retrieval requests;
3. The database server moves information to and from the database and handles transaction management.

Using multiple servers allows high throughput and makes it possible to handle thousands of transactions per minute. As demand increases, servers can be added at each level to cope with the extra processing involved.

6.4.3 Language processing systems

Language processing systems translate one language into an alternative representation of that language and, for programming languages, may also execute the resulting code. Compilers translate a programming language into machine code. Other language processing systems may translate an XML data description into commands to query a database or to an alternative XML representation. Natural language processing systems may translate one natural language to another, for example, French to Norwegian.

A possible architecture for a language processing system for a programming language is illustrated in Figure 6.20. The source language instructions define the

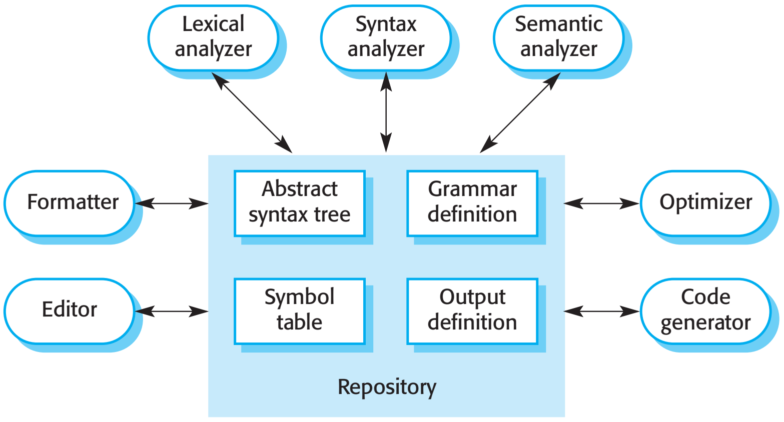


Figure 6.21 A repository architecture for a language processing system

program to be executed, and a translator converts these into instructions for an abstract machine. These instructions are then interpreted by another component that fetches the instructions for execution and executes them using (if necessary) data from the environment. The output of the process is the result of interpreting the instructions on the input data.

For many compilers, the interpreter is the system hardware that processes machine instructions, and the abstract machine is a real processor. However, for dynamically typed languages, such as Ruby or Python, the interpreter is a software component.

Programming language compilers that are part of a more general programming environment have a generic architecture (Figure 6.21) that includes the following components:

1. A lexical analyzer, which takes input language tokens and converts them into an internal form.
2. A symbol table, which holds information about the names of entities (variables, class names, object names, etc.) used in the text that is being translated.
3. A syntax analyzer, which checks the syntax of the language being translated. It uses a defined grammar of the language and builds a syntax tree.
4. A syntax tree, which is an internal structure representing the program being compiled.
5. A semantic analyzer, which uses information from the syntax tree and the symbol table to check the semantic correctness of the input language text.
6. A code generator, which "walks" the syntax tree and generates abstract machine code.

Other components might also be included that analyze and transform the syntax tree to improve efficiency and remove redundancy from the generated machine code.

Reference architectures

Reference architectures capture important features of system architectures in a domain. Essentially, they include everything that might be in an application architecture, although, in reality, it is very unlikely that any individual application would include all the features shown in a reference architecture. The main purpose of reference architectures is to evaluate and compare design proposals, and to educate people about architectural characteristics in that domain.

http://software-engineering-book.com/web/refarch/

In other types of language processing system, such as a natural language translator, there will be additional components such as a dictionary. The output of the system is translation of the input text.

Figure 6.21 illustrates how a language processing system can be part of an integrated set of programming support tools. In this example, the symbol table and syntax tree act as a central information repository. Tools or tool fragments communicate through it. Other information that is sometimes embedded in tools, such as the grammar definition and the definition of the output format for the program, have been taken out of the tools and put into the repository. Therefore, a syntax-directed editor can check that the syntax of a program is correct as it is being typed. A program formatter can create listings of the program that highlight different syntactic elements and are therefore easier to read and understand.

Alternative Architectural patterns may be used in a language processing system (Garlan and Shaw 1993). Compilers can be implemented using a composite of a repository and a pipe and filter model. In a compiler architecture, the symbol table is a repository for shared data. The phases of lexical, syntactic, and semantic analysis are organized sequentially, as shown in Figure 6.22, and communicate through the shared symbol table.

This pipe and filter model of language compilation is effective in batch environments where programs are compiled and executed without user interaction; for example, in the translation of one XML document to another. It is less effective when a compiler is integrated with other language processing tools such as a structured editing system, an interactive debugger, or a program formatter. In this situation, changes from one component need to be reflected immediately in other components. It is better to organize the system around a repository, as shown in Figure 6.21 if you are implementing a general, language-oriented programming environment.

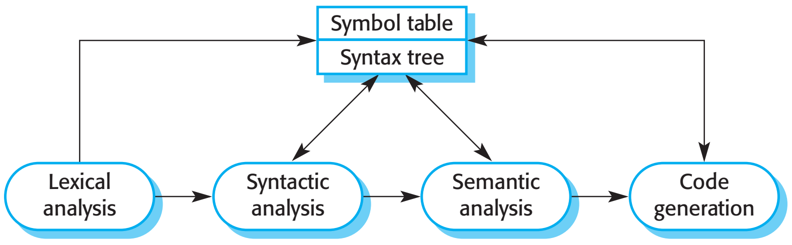
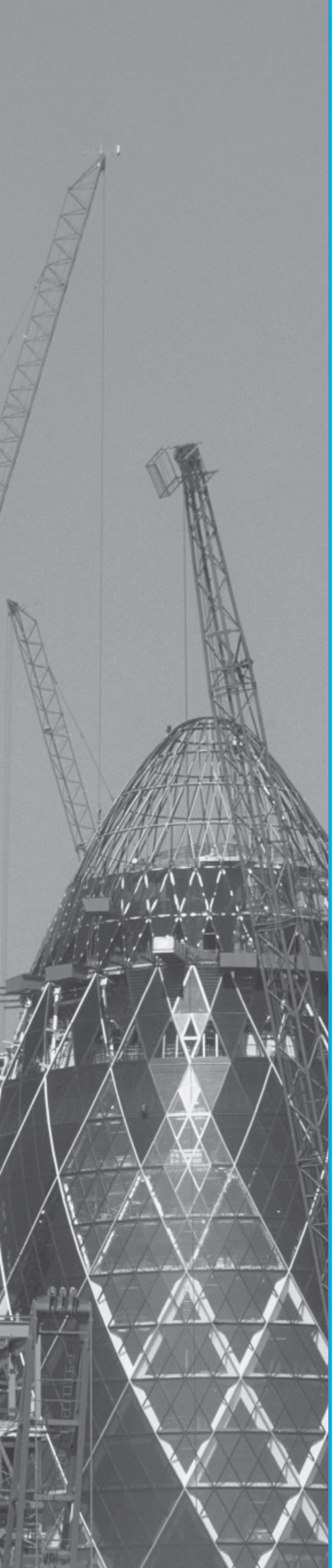


Figure 6.22 A pipe and filter compiler architecture



7 Design and implementation

Objectives

The objectives of this chapter are to introduce object-oriented software design using the UML and highlight important implementation concerns. When you have read this chapter, you will:

- understand the most important activities in a general, object-oriented design process;
- understand some of the different models that may be used to document an object-oriented design;
- know about the idea of design patterns and how these are a way of reusing design knowledge and experience;
- have been introduced to key issues that have to be considered when implementing software, including software reuse and open-source development.

Contents

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 engineering means software design and implementation and all other software engineering activities are merged with this process. However, for large systems, software design and implementation is only one of a number of software engineering processes (requirements engineering, verification and validation, etc.).

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 using a dynamically typed language like Python. There is no point in using the UML 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 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 to build or to buy the application software. For many types of application, it is now possible to buy off-the-shelf application systems 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 is usually cheaper and faster to use this approach rather than developing a new system in a conventional programming language.

When you develop an application system by reusing an off-the-shelf product, the design process focuses on how to configure the system product to meet the application requirements. You don't develop design models of the system, such as models of the system objects and their interactions. I discuss this reuse-based approach to development in Chapter 15.

I assume that most readers of this book 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.

2. To introduce important implementation issues that are not usually covered in programming books. These include software reuse, configuration management and open-source development.

As there are a vast number of different development platforms, the chapter is not biased toward any particular programming language or implementation technology. Therefore, I have presented all examples using the UML rather than a programming language such as Java or Python.

7.1 Object-oriented design using the UML

An object-oriented system is made up of interacting objects that maintain their own local state and provide operations on that state. The representation of the state is private and cannot be accessed directly from outside the object. Object-oriented design processes involve designing object classes and the relationships between these classes. These classes define the objects in the system and their interactions. When the design is realized as an executing program, the objects are created dynamically from these class definitions.

Objects include both data and operations to manipulate that data. They may therefore be understood and modified as stand-alone entities. Changing the implementation of an object or adding services should not affect other system objects. Because objects are associated with things, there is often a clear mapping between real-world entities (such as hardware components) and their controlling objects in the system. This improves the understandability, and hence the maintainability, of the design.

To develop a system design from concept to detailed, object-oriented design, you need to:

1. Understand and define the context and the external interactions with the system.
2. Design the system architecture.
3. Identify the principal objects in the system.
4. Develop design models.
5. Specify interfaces.

Like all creative activities, design is not a clear-cut, sequential process. You develop a design by getting ideas, proposing solutions, and refining these solutions as information becomes available. You inevitably have to backtrack and retry when problems arise. Sometimes you explore options in detail to see if they work; at other times you ignore details until late in the process. Sometimes you use notations, such as the UML, precisely to clarify aspects of the design; at other times, notations are used informally to stimulate discussions.

I explain object-oriented software design by developing a design for part of the embedded software for the wilderness weather station that I introduced in Chapter 1. Wilderness weather stations are deployed in remote areas. Each weather station

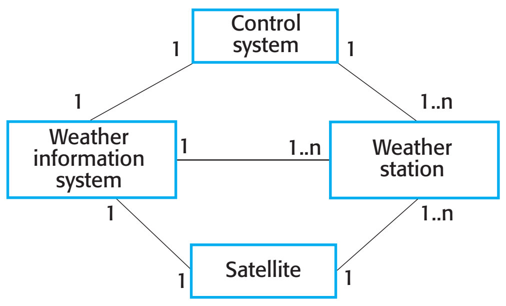


Figure 7.1 System context for the weather station

records local weather information and periodically transfers this to a weather information system, using a satellite link.

7.1.1 System context and interactions

The first stage in any software design process is to develop an understanding of the relationships between the software that is being designed and its external environment. This is essential for deciding how to provide the required system functionality and how to structure the system to communicate with its environment. As I discussed in Chapter 5, understanding the context also lets you establish the boundaries of the system.

Setting the system boundaries helps you decide what features are implemented in the system being designed and what features are in other associated systems. In this case, you need to decide how functionality is distributed between the control system for all of the weather stations and the embedded software in the weather station itself.

System context models and interaction models present complementary views of the relationships between a system and its environment:

1. A system context model is a structural model that demonstrates the other systems in the environment of the system being developed.
2. An interaction model is a dynamic model that shows how the system interacts with its environment as it is used.

The context model of a system may be represented using associations. Associations simply show that there are some relationships between the entities involved in the association. You can document the environment of the system using a simple block diagram, showing the entities in the system and their associations. Figure 7.1 shows that the systems in the environment of each weather station are a weather information system, an onboard satellite system, and a control system. The cardinality information on the link shows that there is a single control system but several weather stations, one satellite, and one general weather information system.

When you model the interactions of a system with its environment, you should use an abstract approach that does not include too much detail. One way to do this is to use a use case model. As I discussed in Chapters 4 and 5, each use case represents

Weather station use cases

Report weather–send weather data to the weather information system

Report status–send status information to the weather information system

Restart–if the weather station is shut down, restart the system

Shutdown–shut down the weather station

Reconfigure–reconfigure the weather station software

Powersave–put the weather station into power-saving mode

Remote control–send control commands to any weather station subsystem

http://software-engineering-book.com/web/ws-use-cases/

an interaction with the system. Each possible interaction is named in an ellipse, and the external entity involved in the interaction is represented by a stick figure.

The use case model for the weather station is shown in Figure 7.2. This shows that the weather station interacts with the weather information system to report weather data and the status of the weather station hardware. Other interactions are with a control system that can issue specific weather station control commands. The stick figure is used in the UML to represent other systems as well as human users.

Each of these use cases should be described in structured natural language. This helps designers identify objects in the system and gives them an understanding of what the system is intended to do. I use a standard format for this description that clearly identifies what information is exchanged, how the interaction is initiated, and so on. As I explain in Chapter 21, embedded systems are often modeled by describing

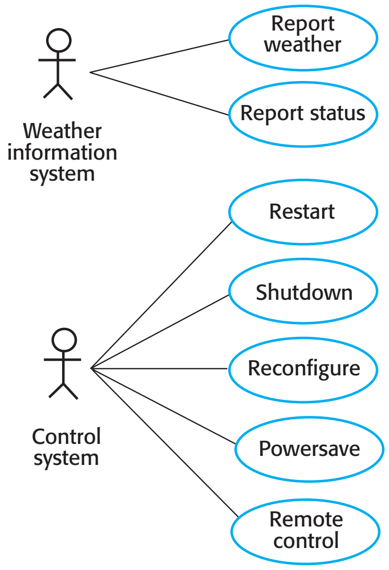


Figure 7.2 Weather station use cases

System	Weather station
Use case	Report weather
Actors	Weather information system, Weather station
Data	The weather station sends a summary of the weather data that has been collected from the instruments in the collection period to the weather information system. The data sent are the maximum, minimum, and average ground and air temperatures; the maximum, minimum, and average air pressures; the maximum, minimum and average wind speeds; the total rainfall; and the wind direction as sampled at 5-minute intervals.
Stimulus	The weather information system establishes a satellite communication link with the weather station and requests transmission of the data.
Response	The summarized data is sent to the weather information system.
Comments	Weather stations are usually asked to report once per hour, but this frequency may differ from one station to another and may be modified in future.

Figure 7.3 Use case description—Report weather

how they respond to internal or external stimuli. Therefore, the stimuli and associated responses should be listed in the description. Figure 7.3 shows the description of the Report weather use case from Figure 7.2 that is based on this approach.

7.1.2 Architectural design

Once the interactions between the software system and the system's environment have been defined, you use this information as a basis for designing the system architecture. Of course, you need to combine this knowledge with your general knowledge of the principles of architectural design and with more detailed domain knowledge. You identify the major components that make up the system and their interactions. You may then design the system organization using an architectural pattern such as a layered or client–server model.

The high-level architectural design for the weather station software is shown in Figure 7.4. The weather station is composed of independent subsystems that communicate

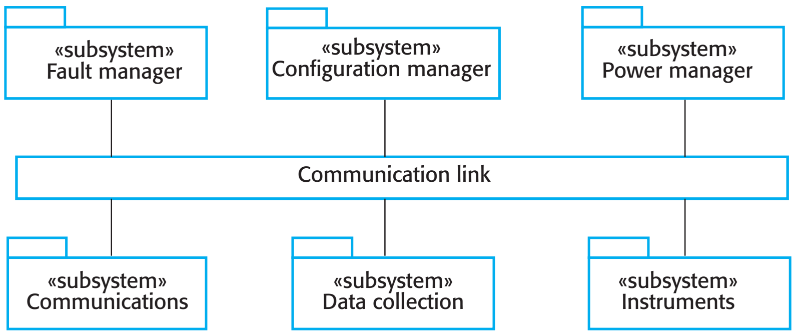


Figure 7.4 High-level architecture of weather station

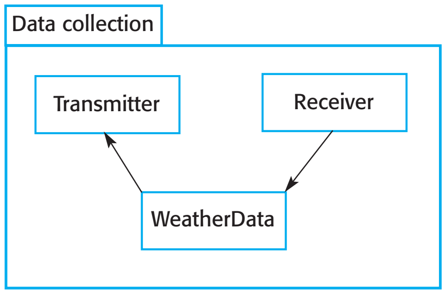


Figure 7.5 Architecture of data collection system

by broadcasting messages on a common infrastructure, shown as **Communication** link in Figure 7.4. Each subsystem listens for messages on that infrastructure and picks up the messages that are intended for them. This "listener model" is a commonly used architectural style for distributed systems.

When the communications subsystem receives a control command, such as shutdown, the command is picked up by each of the other subsystems, which then shut themselves down in the correct way. The key benefit of this architecture is that it is easy to support different configurations of subsystems because the sender of a message does not need to address the message to a particular subsystem.

Figure 7.5 shows the architecture of the data collection subsystem, which is included in Figure 7.4. The **Transmitter** and **Receiver** objects are concerned with managing communications, and the **WeatherData** object encapsulates the information that is collected from the instruments and transmitted to the weather information system. This arrangement follows the producer–consumer pattern, discussed in Chapter 21.

7.1.3 Object class identification

By this stage in the design process, you should have some ideas about the essential objects in the system that you are designing. As your understanding of the design develops, you refine these ideas about the system objects. The use case description helps to identify objects and operations in the system. From the description of the Report weather use case, it is obvious that you will need to implement objects representing the instruments that collect weather data and an object representing the summary of the weather data. You also usually need a high-level system object or objects that encapsulate the system interactions defined in the use cases. With these objects in mind, you can start to identify the general object classes in the system.

As object-oriented design evolved in the 1980s, various ways of identifying object classes in object-oriented systems were suggested:

1. Use a grammatical analysis of a natural language description of the system to be constructed. Objects and attributes are nouns; operations or services are verbs (Abbott 1983).

2. Use tangible entities (things) in the application domain such as aircraft, roles such as manager, events such as request, interactions such as meetings, locations

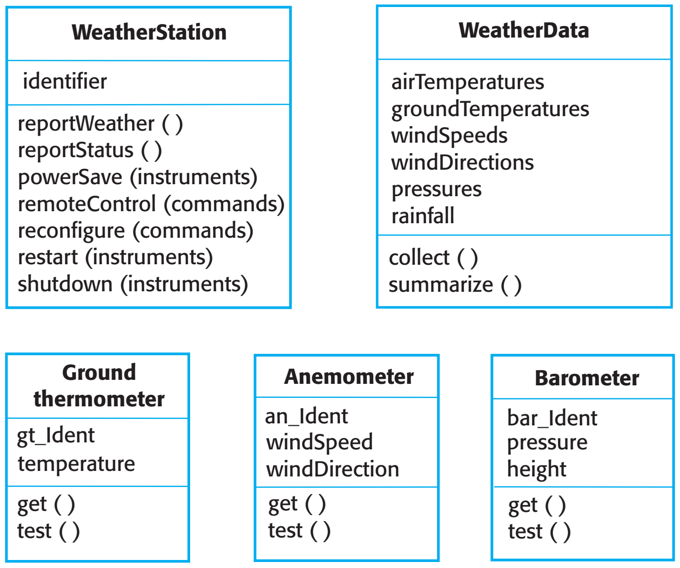


Figure 7.6 Weather station objects

such as offices, organizational units such as companies, and so on (Wirfs-Brock, Wilkerson, and Weiner 1990).

3. Use a scenario-based analysis where various scenarios of system use are identified and analyzed in turn. As each scenario is analyzed, the team responsible for the analysis must identify the required objects, attributes, and operations (Beck and Cunningham 1989).

In practice, you have to use several knowledge sources to discover object classes. Object classes, attributes, and operations that are initially identified from the informal system description can be a starting point for the design. Information from application domain knowledge or scenario analysis may then be used to refine and extend the initial objects. This information can be collected from requirements documents, discussions with users, or analyses of existing systems. As well as the objects representing entities external to the system, you may also have to design "implementation objects" that are used to provide general services such as searching and validity checking.

In the wilderness weather station, object identification is based on the tangible hardware in the system. I don't have space to include all the system objects here, but I have shown five object classes in Figure 7.6. The Ground thermometer, Anemometer, and Barometer objects are application domain objects, and the WeatherStation and WeatherData objects have been identified from the system description and the scenario (use case) description:

1. The WeatherStation object class provides the basic interface of the weather station with its environment. Its operations are based on the interactions shown in Figure 7.3. I use a single object class, and it includes all of these interactions. Alternatively, you could design the system interface as several different classes, with one class per interaction.

2. The **WeatherData** object class is responsible for processing the report weather command. It sends the summarized data from the weather station instruments to the weather information system.

3. The **Ground thermometer**, **Anemometer**, and **Barometer** object classes are directly related to instruments in the system. They reflect tangible hardware entities in the system and the operations are concerned with controlling that hardware. These objects operate autonomously to collect data at the specified frequency and store the collected data locally. This data is delivered to the **WeatherData** object on request.

You use knowledge of the application domain to identify other objects, attributes. and services:

1. Weather stations are often located in remote places and include various instruments that sometimes go wrong. Instrument failures should be reported automatically. This implies that you need attributes and operations to check the correct functioning of the instruments.

2. There are many remote weather stations, so each weather station should have its own identifier so that it can be uniquely identified in communications.

3. As weather stations are installed at different times, the types of instrument may be different. Therefore, each instrument should also be uniquely identified, and a database of instrument information should be maintained.

At this stage in the design process, you should focus on the objects themselves, without thinking about how these objects might be implemented. Once you have identified the objects, you then refine the object design. You look for common features and then design the inheritance hierarchy for the system. For example, you may identify an **Instrument** superclass, which defines the common features of all instruments, such as an identifier, and get and test operations. You may also add new attributes and operations to the superclass, such as an attribute that records how often data should be collected.

7.1.4 Design models

Design or system models, as I discussed in Chapter 5, show the objects or object classes in a system. They also show the associations and relationships between these entities. These models are the bridge between the system requirements and the implementation of a system. They have to be abstract so that unnecessary detail doesn't hide the relationships between them and the system requirements. However, they also have to include enough detail for programmers to make implementation decisions.

The level of detail that you need in a design model depends on the design process used. Where there are close links between requirements engineers, designers and programmers, then abstract models may be all that are required. Specific design decisions may be made as the system is implemented, with problems resolved through informal discussions. Similarly, if agile development is used, outline design models on a whiteboard may be all that is required.

However, if a plan-based development process is used, you may need more detailed models. When the links between requirements engineers, designers, and programmers are indirect (e.g., where a system is being designed in one part of an organization but implemented elsewhere), then precise design descriptions are needed for communication. Detailed models, derived from the high-level abstract models, are used so that all team members have a common understanding of the design.

An important step in the design process, therefore, is to decide on the design models that you need and the level of detail required in these models. This depends on the type of system that is being developed. A sequential data-processing system is quite different from an embedded real-time system, so you need to use different types of design models. The UML supports 13 different types of models, but, as I discussed in Chapter 5, many of these models are not widely used. Minimizing the number of models that are produced reduces the costs of the design and the time required to complete the design process.

When you use the UML to develop a design, you should develop two kinds of design model:

1. *Structural models,* which describe the static structure of the system using object classes and their relationships. Important relationships that may be documented at this stage are generalization (inheritance) relationships, uses/used-by relationships, and composition relationships.
2. *Dynamic models,* which describe the dynamic structure of the system and show the expected runtime interactions between the system objects. Interactions that may be documented include the sequence of service requests made by objects and the state changes triggered by these object interactions.

I think three UML model types are particularly useful for adding detail to use case and architectural models:

1. *Subsystem models,* which show logical groupings of objects into coherent subsystems. These are represented using a form of class diagram with each subsystem shown as a package with enclosed objects. Subsystem models are structural models.
2. *Sequence models,* which show the sequence of object interactions. These are represented using a UML sequence or a collaboration diagram. Sequence models are dynamic models.
3. *State machine models,* which show how objects change their state in response to events. These are represented in the UML using state diagrams. State machine models are dynamic models.

A subsystem model is a useful static model that shows how a design is organized into logically related groups of objects. I have already shown this type of model in Figure 7.4 to present the subsystems in the weather mapping system. As well as subsystem models, you may also design detailed object models, showing the objects in the systems and their associations (inheritance, generalization, aggregation, etc.). However, there is a danger

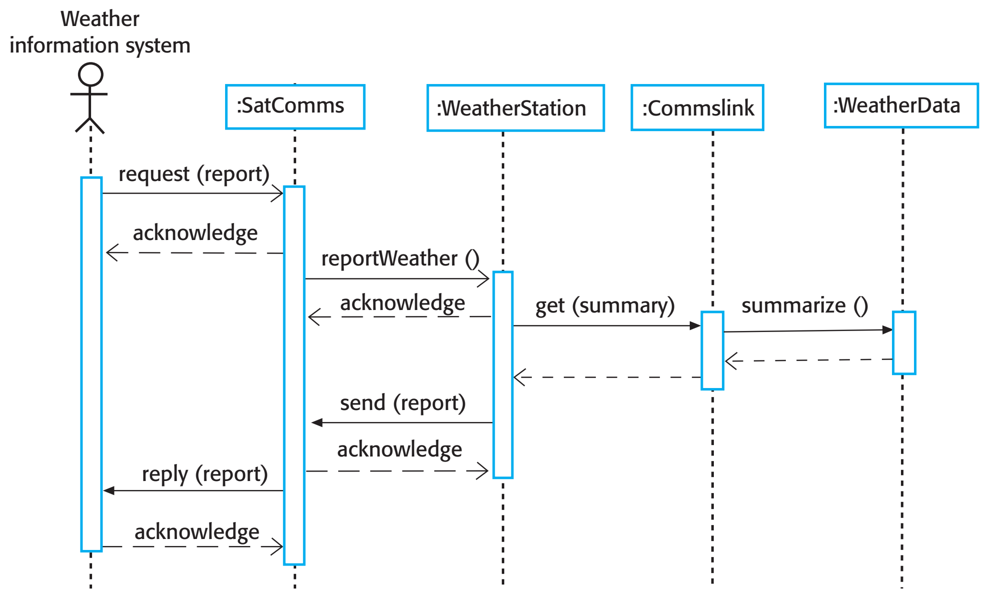


Figure 7.7 Sequence diagram describing data collection

in doing too much modeling. You should not make detailed decisions about the implementation that are really best left until the system is implemented.

Sequence models are dynamic models that describe, for each mode of interaction, the sequence of object interactions that take place. When documenting a design, you should produce a sequence model for each significant interaction. If you have developed a use case model, then there should be a sequence model for each use case that you have identified.

Figure 7.7 is an example of a sequence model, shown as a UML sequence diagram. This diagram shows the sequence of interactions that take place when an external system requests the summarized data from the weather station. You read sequence diagrams from top to bottom:

1. The SatComms object receives a request from the weather information system to collect a weather report from a weather station. It acknowledges receipt of this request. The stick arrowhead on the sent message indicates that the external system does not wait for a reply but can carry on with other processing.

2. SatComms sends a message to WeatherStation, via a satellite link, to create a summary of the collected weather data. Again, the stick arrowhead indicates that SatComms does not suspend itself waiting for a reply.

3. WeatherStation sends a message to a Commslink object to summarize the weather data. In this case, the squared-off style of arrowhead indicates that the instance of the WeatherStation object class waits for a reply.

4. Commslink calls the summarize method in the object WeatherData and waits for a reply.

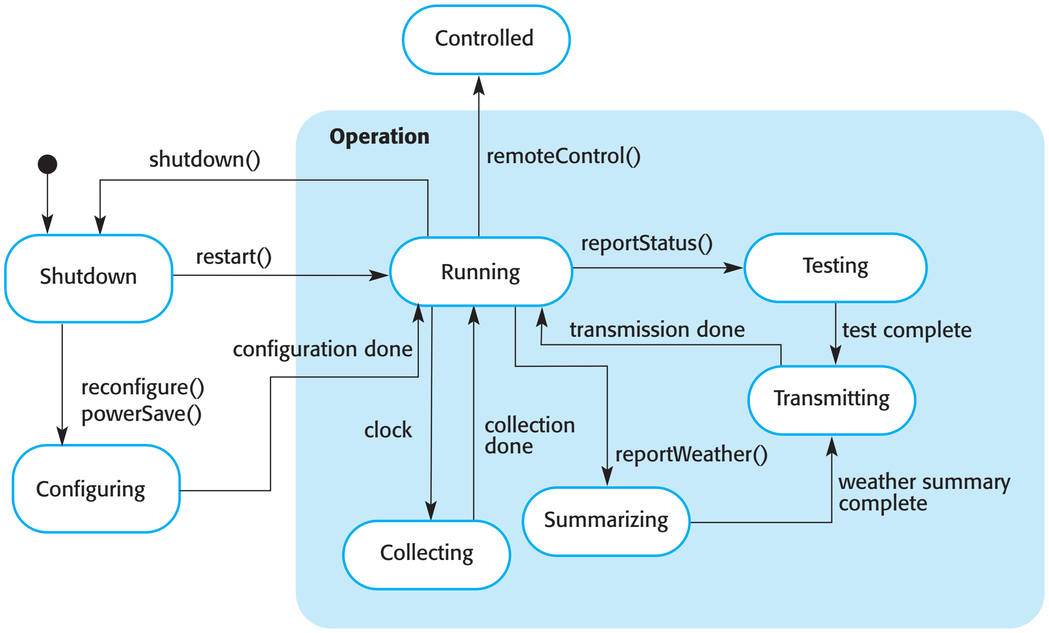


Figure 7.8 Weather station state diagram

5. The weather data summary is computed and returned to **WeatherStation** via the **Commslink** object.

6. **WeatherStation** then calls the **SatComms** object to transmit the summarized data to the weather information system, through the satellite communications system.

The **SatComms** and **WeatherStation** objects may be implemented as concurrent processes, whose execution can be suspended and resumed. The **SatComms** object instance listens for messages from the external system, decodes these messages, and initiates weather station operations.

Sequence diagrams are used to model the combined behavior of a group of objects, but you may also want to summarize the behavior of an object or a subsystem in response to messages and events. To do this, you can use a state machine model that shows how the object instance changes state depending on the messages that it receives. As I discuss in Chapter 5, the UML includes state diagrams to describe state machine models.

Figure 7.8 is a state diagram for the weather station system that shows how it responds to requests for various services.

You can read this diagram as follows:

1. If the system state is **Shutdown**, then it can respond to a **restart**(), a **reconfigure**() or a **powerSave**() message. The unlabeled arrow with the black blob indicates that the **Shutdown** state is the initial state. A **restart**() message causes a transition to normal operation. Both the **powerSave**() and **reconfigure**() messages cause a transition to a state in which the system reconfigures itself. The state diagram shows that reconfiguration is allowed only if the system has been shut down.

2. In the **Running** state, the system expects further messages. If a **shutdown()** message is received, the object returns to the shutdown state.
3. If a **reportWeather()** message is received, the system moves to the **Summarizing** state. When the summary is complete, the system moves to a **Transmitting** state where the information is transmitted to the remote system. It then returns to the **Running** state.
4. If a signal from the clock is received, the system moves to the **Collecting** state, where it collects data from the instruments. Each instrument is instructed in turn to collect its data from the associated sensors.
5. If a **remoteControl()** message is received, the system moves to a controlled state in which it responds to a different set of messages from the remote control room. These are not shown on this diagram.

State diagrams are useful high-level models of a system or an object's operation. However, you don't need a state diagram for all of the objects in the system. Many system objects in a system are simple, and their operation can be easily described without a state model.

7.1.5 Interface specification

An important part of any design process is the specification of the interfaces between the components in the design. You need to specify interfaces so that objects and subsystems can be designed in parallel. Once an interface has been specified, the developers of other objects may assume that interface will be implemented.

Interface design is concerned with specifying the detail of the interface to an object or to a group of objects. This means defining the signatures and semantics of the services that are provided by the object or by a group of objects. Interfaces can be specified in the UML using the same notation as a class diagram. However, there is no attribute section, and the UML stereotype «interface» should be included in the name part. The semantics of the interface may be defined using the object constraint language (OCL). I discuss the use of the OCL in Chapter 16, where I explain how it can be used to describe the semantics of components.

You should not include details of the data representation in an interface design, as attributes are not defined in an interface specification. However, you should include operations to access and update data. As the data representation is hidden, it can be easily changed without affecting the objects that use that data. This leads to a design that is inherently more maintainable. For example, an array representation of a stack may be changed to a list representation without affecting other objects that use the stack. By contrast, you should normally expose the attributes in an object model, as this is the clearest way of describing the essential characteristics of the objects.

There is not a simple 1:1 relationship between objects and interfaces. The same object may have several interfaces, each of which is a viewpoint on the methods that it provides. This is supported directly in Java, where interfaces are declared separately from objects and objects "implement" interfaces. Equally, a group of objects may all be accessed through a single interface.

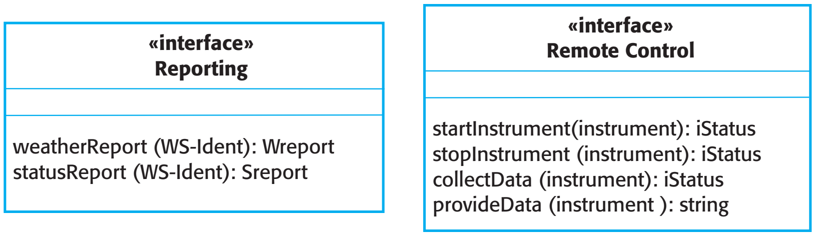

«interface»
Reporting
weatherReport (WS-Ident): Wreport
statusReport (WS-Ident): Sreport
«interface»
Remote Control
startInstrument(instrument): iStatus
stopInstrument (instrument): iStatus
collectData (instrument): iStatus
provideData (instrument): string

Figure 7.9 Weather station interfaces

Figure 7.9 shows two interfaces that may be defined for the weather station. The left-hand interface is a reporting interface that defines the operation names that are used to generate weather and status reports. These map directly to operations in the WeatherStation object. The remote control interface provides four operations, which map onto a single method in the WeatherStation object. In this case, the individual operations are encoded in the command string associated with the remoteControl method, shown in Figure 7.6.

7.2 Design patterns

Design patterns were derived from ideas put forward by Christopher Alexander (Alexander 1979), who suggested that there were certain common patterns of building design that were inherently pleasing and effective. The pattern is a description of the problem and the essence of its solution, so that the solution may be reused in different settings. The pattern is not a detailed specification. Rather, you can think of it as a description of accumulated wisdom and experience, a well-tried solution to a common problem.

A quote from the Hillside Group website (hillside.net/patterns/), which is dedicated to maintaining information about patterns, encapsulates their role in reuse:

> *Patterns and Pattern Languages are ways to describe best practices, good designs, and capture experience in a way that it is possible for others to reuse this experience*†.

Patterns have made a huge impact on object-oriented software design. As well as being tested solutions to common problems, they have become a vocabulary for talking about a design. You can therefore explain your design by describing the patterns that you have used. This is particularly true for the best known design patterns that were originally described by the "Gang of Four" in their patterns book, published in 1995 (Gamma et al. 1995). Other important pattern descriptions are those published in a series of books by authors from Siemens, a large European technology company (Buschmann et al. 1996; Schmidt et al. 2000; Kircher and Jain 2004; Buschmann, Henney, and Schmidt 2007a, 2007b).

Patterns are a way of reusing the knowledge and experience of other designers. Design patterns are usually associated with object-oriented design. Published patterns often rely on object characteristics such as inheritance and polymorphism to provide generality. However, the general principle of encapsulating experience in a pattern is

†The HIllside Group: hillside.net/patterns

Pattern name: Observer

Description: Separates the display of the state of an object from the object itself and allows alternative displays to be provided. When the object state changes, all displays are automatically notified and updated to reflect the change.

Problem description: In many situations, you have to provide multiple displays of state information, such as a graphical display and a tabular display. Not all of these may be known when the information is specified. All alternative presentations should support interaction and, when the state is changed, all displays must be updated.

This pattern may be used in situations where more than one display format for state information is required and where it is not necessary for the object that maintains the state information to know about the specific display formats used.

Solution description: This involves two abstract objects, Subject and Observer, and two concrete objects, ConcreteSubject and ConcreteObject, which inherit the attributes of the related abstract objects. The abstract objects include general operations that are applicable in all situations. The state to be displayed is maintained in ConcreteSubject, which inherits operations from Subject allowing it to add and remove Observers (each observer corresponds to a display) and to issue a notification when the state has changed.

The ConcreteObserver maintains a copy of the state of ConcreteSubject and implements the Update() interface of Observer that allows these copies to be kept in step. The ConcreteObserver automatically displays the state and reflects changes whenever the state is updated.

The UML model of the pattern is shown in Figure 7.12.

Consequences: The subject only knows the abstract Observer and does not know details of the concrete class. Therefore there is minimal coupling between these objects. Because of this lack of knowledge, optimizations that enhance display performance are impractical. Changes to the subject may cause a set of linked updates to observers to be generated, some of which may not be necessary.

Figure 7.10 The Observer pattern

one that is equally applicable to any kind of software design. For instance, you could have configuration patterns for instantiating reusable application systems.

The Gang of Four defined the four essential elements of design patterns in their book on patterns:

1. A name that is a meaningful reference to the pattern.
2. A description of the problem area that explains when the pattern may be applied.
3. A solution description of the parts of the design solution, their relationships and their responsibilities. This is not a concrete design description. It is a template for a design solution that can be instantiated in different ways. This is often expressed graphically and shows the relationships between the objects and object classes in the solution.
4. A statement of the consequences—the results and trade-offs—of applying the pattern. This can help designers understand whether or not a pattern can be used in a particular situation.

Gamma and his co-authors break down the problem description into motivation (a description of why the pattern is useful) and applicability (a description of situations in which the pattern may be used). Under the description of the solution, they describe the pattern structure, participants, collaborations, and implementation.

To illustrate pattern description, I use the Observer pattern, taken from the Gang of Four's patterns book. This is shown in Figure 7.10. In my description, I use the

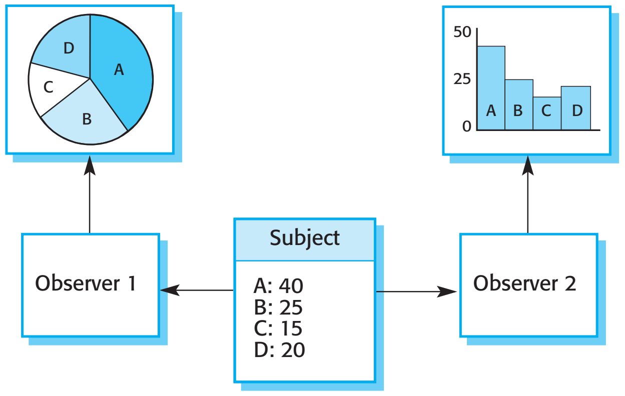


Figure 7.11 Multiple displays

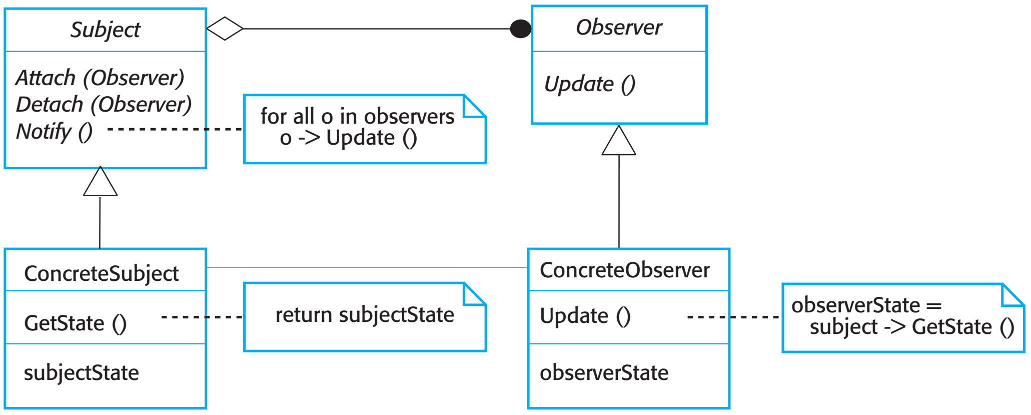
four essential description elements and also include a brief statement of what the pattern can do. This pattern can be used in situations where different presentations of an object's state are required. It separates the object that must be displayed from the different forms of presentation. This is illustrated in Figure 7.11, which shows two different graphical presentations of the same dataset.

Graphical representations are normally used to illustrate the object classes in patterns and their relationships. These supplement the pattern description and add detail to the solution description. Figure 7.12 is the representation in UML of the Observer pattern.

To use patterns in your design, you need to recognize that any design problem you are facing may have an associated pattern that can be applied. Examples of such problems, documented in the Gang of Four's original patterns book, include:

1. Tell several objects that the state of some other object has changed (Observer pattern).
2. Tidy up the interfaces to a number of related objects that have often been developed incrementally (Façade pattern).

Figure 7.12 A UML model of the Observer pattern



3. Provide a standard way of accessing the elements in a collection, irrespective of how that collection is implemented (Iterator pattern).

4. Allow for the possibility of extending the functionality of an existing class at runtime (Decorator pattern).

Patterns support high-level, concept reuse. When you try to reuse executable components you are inevitably constrained by detailed design decisions that have been made by the implementers of these components. These range from the particular algorithms that have been used to implement the components to the objects and types in the component interfaces. When these design decisions conflict with your requirements, reusing the component is either impossible or introduces inefficiencies into your system. Using patterns means that you reuse the ideas but can adapt the implementation to suit the system you are developing.

When you start designing a system, it can be difficult to know, in advance, if you will need a particular pattern. Therefore, using patterns in a design process often involves developing a design, experiencing a problem, and then recognizing that a pattern can be used. This is certainly possible if you focus on the 23 general-purpose patterns documented in the original patterns book. However, if your problem is a different one, you may find it difficult to find an appropriate pattern among the hundreds of different patterns that have been proposed.

Patterns are a great idea, but you need experience of software design to use them effectively. You have to recognize situations where a pattern can be applied. Inexperienced programmers, even if they have read the pattern books, will always find it hard to decide whether they can reuse a pattern or need to develop a special-purpose solution.

7.3 Implementation issues

Software engineering includes all of the activities involved in software development from the initial requirements of the system through to maintenance and management of the deployed system. A critical stage of this process is, of course, system implementation, where you create an executable version of the software. Implementation may involve developing programs in high- or low-level programming languages or tailoring and adapting generic, off-the-shelf systems to meet the specific requirements of an organization.

I assume that most readers of this book will understand programming principles and will have some programming experience. As this chapter is intended to offer a language-independent approach, I haven't focused on issues of good programming practice as language-specific examples need to be used. Instead, I introduce some aspects of implementation that are particularly important to software engineering and that are often not covered in programming texts. These are:

1. *Reuse* Most modern software is constructed by reusing existing components or systems. When you are developing software, you should make as much use as possible of existing code.

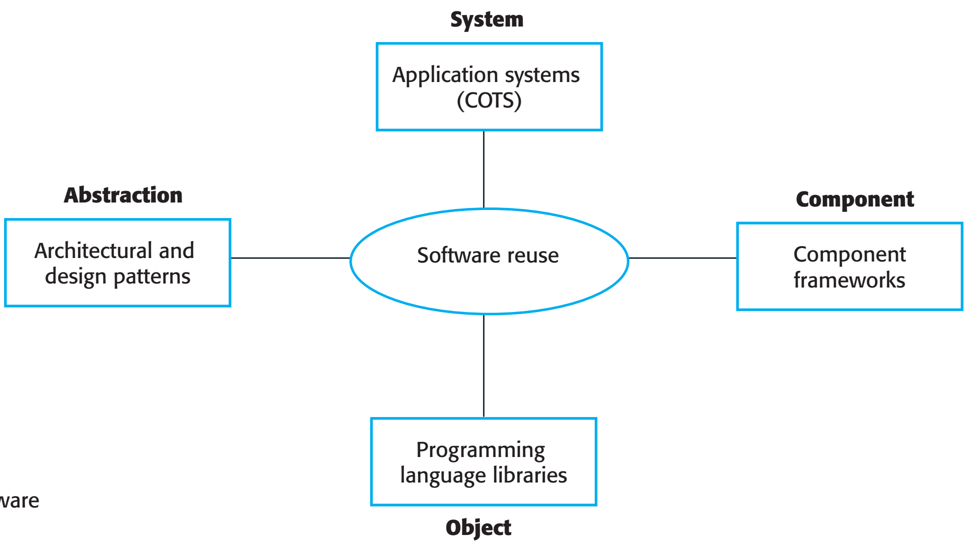

System
Application systems (COTS)
Abstraction
Architectural and design patterns
Software reuse
Component
Component frameworks
Programming language libraries
Object

Figure 7.13 Software reuse

2. *Configuration management* During the development process, many different versions of each software component are created. If you don't keep track of these versions in a configuration management system, you are liable to include the wrong versions of these components in your system.

3. *Host-target development* Production software does not usually execute on the same computer as the software development environment. Rather, you develop it on one computer (the host system) and execute it on a separate computer (the target system). The host and target systems are sometimes of the same type, but often they are completely different.

7.3.1 Reuse

From the 1960s to the 1990s, most new software was developed from scratch, by writing all code in a high-level programming language. The only significant reuse or software was the reuse of functions and objects in programming language libraries. However, costs and schedule pressure meant that this approach became increasingly unviable, especially for commercial and Internet-based systems. Consequently, an approach to development based on the reuse of existing software is now the norm for many types of system development. A reuse-based approach is now widely used for web-based systems of all kinds, scientific software, and, increasingly, in embedded systems engineering.

Software reuse is possible at a number of different levels, as shown in Figure 7.13:

1. *The abstraction level* At this level, you don't reuse software directly but rather use knowledge of successful abstractions in the design of your software. Design patterns and architectural patterns (covered in Chapter 6) are ways of representing abstract knowledge for reuse.

2. *The object level* At this level, you directly reuse objects from a library rather than writing the code yourself. To implement this type of reuse, you have to find appropriate libraries and discover if the objects and methods offer the functionality that you need. For example, if you need to process email messages in a Java program, you may use objects and methods from a JavaMail library.

3. *The component level* Components are collections of objects and object classes that operate together to provide related functions and services. You often have to adapt and extend the component by adding some code of your own. An example of component-level reuse is where you build your user interface using a framework. This is a set of general object classes that implement event handling, display management, etc. You add connections to the data to be displayed and write code to define specific display details such as screen layout and colors.

4. *The system level* At this level, you reuse entire application systems. This function usually involves some kind of configuration of these systems. This may be done by adding and modifying code (if you are reusing a software product line) or by using the system's own configuration interface. Most commercial systems are now built in this way where generic application systems systems are adapted and reused. Sometimes this approach may involve integrating several application systems to create a new system.

By reusing existing software, you can develop new systems more quickly, with fewer development risks and at lower cost. As the reused software has been tested in other applications, it should be more reliable than new software. However, there are costs associated with reuse:

1. The costs of the time spent in looking for software to reuse and assessing whether or not it meets your needs. You may have to test the software to make sure that it will work in your environment, especially if this is different from its development environment.

2. Where applicable, the costs of buying the reusable software. For large off-the-shelf systems, these costs can be very high.

3. The costs of adapting and configuring the reusable software components or systems to reflect the requirements of the system that you are developing.

4. The costs of integrating reusable software elements with each other (if you are using software from different sources) and with the new code that you have developed. Integrating reusable software from different providers can be difficult and expensive because the providers may make conflicting assumptions about how their respective software will be reused.

How to reuse existing knowledge and software should be the first thing you should think about when starting a software development project. You should consider the

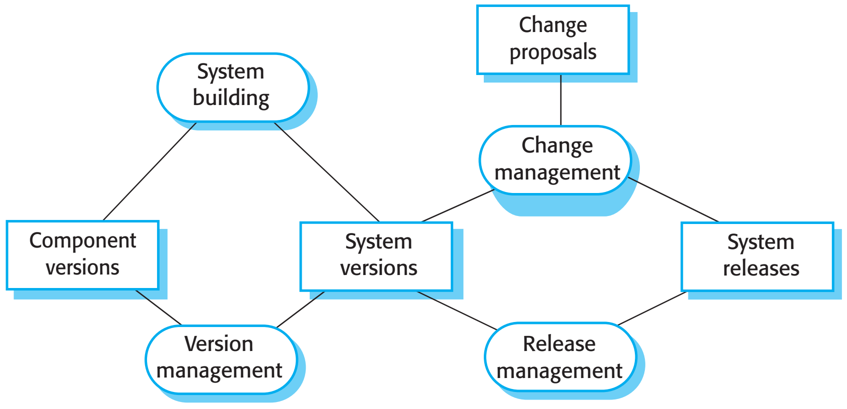


Figure 7.14 Configuration management

possibilities of reuse before designing the software in detail, as you may wish to adapt your design to reuse existing software assets. As I discussed in Chapter 2, in a reuse-oriented development process, you search for reusable elements, then modify your requirements and design to make the best use of these.

Because of the importance of reuse in modern software engineering, I devote several chapters in Part 3 of this book to this topic (Chapters 15, 16, and 18).

7.3.2 Configuration management

In software development, change happens all the time, so change management is absolutely essential. When several people are involved in developing a software system, you have to make sure that team members don't interfere with each other's work. That is, if two people are working on a component, their changes have to be coordinated. Otherwise, one programmer may make changes and overwrite the other's work. You also have to ensure that everyone can access the most up-to-date versions of software components; otherwise developers may redo work that has already been done. When something goes wrong with a new version of a system, you have to be able to go back to a working version of the system or component.

Configuration management is the name given to the general process of managing a changing software system. The aim of configuration management is to support the system integration process so that all developers can access the project code and documents in a controlled way, find out what changes have been made, and compile and link components to create a system. As shown in Figure 7.14, there are four fundamental configuration management activities:

1. *Version management,* where support is provided to keep track of the different versions of software components. Version management systems include facilities to coordinate development by several programmers. They stop one developer from overwriting code that has been submitted to the system by someone else.

2. *System integration,* where support is provided to help developers define what versions of components are used to create each version of a system. This

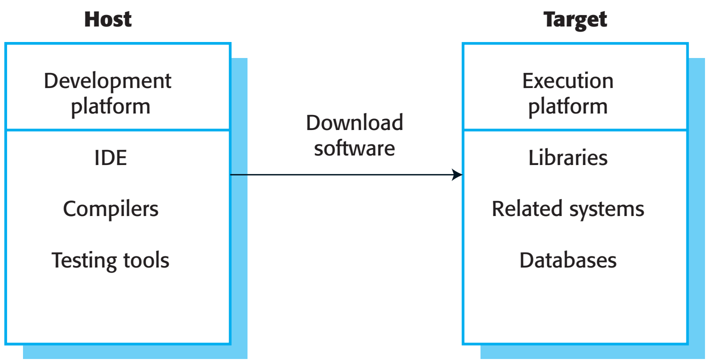


Figure 7.15 Host-target development

description is then used to build a system automatically by compiling and linking the required components.

3. *Problem tracking,* where support is provided to allow users to report bugs and other problems, and to allow all developers to see who is working on these problems and when they are fixed.

4. *Release management,* where new versions of a software system are released to customers. Release management is concerned with planning the functionality of new releases and organizing the software for distribution.

Software configuration management tools support each of the above activities. These tools are usually installed in an integrated development environment, such as Eclipse. Version management may be supported using a version management system such as Subversion (Pilato, Collins-Sussman, and Fitzpatrick 2008) or Git (Loeliger and McCullough 2012), which can support multi-site, multi-team development. System integration support may be built into the language or rely on a separate toolset such as the GNU build system. Bug tracking or issue tracking systems, such as Bugzilla, are used to report bugs and other issues and to keep track of whether or not these have been fixed. A comprehensive set of tools built around the Git system is available at Github (http://github.com).

Because of its importance in professional software engineering, I discuss change and configuration management in more detail in Chapter 25.

7.3.3 Host-target development

Most professional software development is based on a host-target model (Figure 7.15). Software is developed on one computer (the host) but runs on a separate machine (the target). More generally, we can talk about a development platform (host) and an execution platform (target). A platform is more than just hardware. It includes the installed operating system plus other supporting software such as a database management system or, for development platforms, an interactive development environment.

Sometimes, the development platform and execution platform are the same, making it possible to develop the software and test it on the same machine. Therefore, if you develop in Java, the target environment is the Java Virtual Machine. In principle, this is the same on every computer, so programs should be portable from one machine to another. However, particularly for embedded systems and mobile systems, the development and the execution platforms are different. You need to either move your developed software to the execution platform for testing or run a simulator on your development machine.

Simulators are often used when developing embedded systems. You simulate hardware devices, such as sensors, and the events in the environment in which the system will be deployed. Simulators speed up the development process for embedded systems as each developer can have his or her own execution platform with no need to download the software to the target hardware. However, simulators are expensive to develop and so are usually available only for the most popular hardware architectures.

If the target system has installed middleware or other software that you need to use, then you need to be able to test the system using that software. It may be impractical to install that software on your development machine, even if it is the same as the target platform, because of license restrictions. If this is the case, you need to transfer your developed code to the execution platform to test the system.

A software development platform should provide a range of tools to support software engineering processes. These may include:

1. An integrated compiler and syntax-directed editing system that allows you to create, edit, and compile code.
2. A language debugging system.
3. Graphical editing tools, such as tools to edit UML models.
4. Testing tools, such as JUnit, that can automatically run a set of tests on a new version of a program.
5. Tools to support refactoring and program visualization.
6. Configuration management tools to manage source code versions and to integrate and build systems.

In addition to these standard tools, your development system may include more specialized tools such as static analyzers (discussed in Chapter 12). Normally, development environments for teams also include a shared server that runs a change and configuration management system and, perhaps, a system to support requirements management.

Software development tools are now usually installed within an integrated development environment (IDE). An IDE is a set of software tools that supports different aspects of software development within some common framework and user interface. Generally, IDEs are created to support development in a specific programming

UML deployment diagrams

UML deployment diagrams show how software components are physically deployed on processors. That is, the deployment diagram shows the hardware and software in the system and the middleware used to connect the different components in the system. Essentially, you can think of deployment diagrams as a way of defining and documenting the target environment.

http://software-engineering-book.com/web/deployment/

language such as Java. The language IDE may be developed specially or may be an instantiation of a general-purpose IDE, with specific language-support tools.

A general-purpose IDE is a framework for hosting software tools that provides data management facilities for the software being developed and integration mechanisms that allow tools to work together. The best-known general-purpose IDE is the Eclipse environment (http://www.eclipse.org). This environment is based on a plug-in architecture so that it can be specialized for different languages, such as Java, and application domains. Therefore, you can install Eclipse and tailor it for your specific needs by adding plug-ins. For example, you may add a set of plug-ins to support networked systems development in Java (Vogel 2013) or embedded systems engineering using C.

As part of the development process, you need to make decisions about how the developed software will be deployed on the target platform. This is straightforward for embedded systems, where the target is usually a single computer. However, for distributed systems, you need to decide on the specific platforms where the components will be deployed. Issues that you have to consider in making this decision are:

1. *The hardware and software requirements of a component* If a component is designed for a specific hardware architecture, or relies on some other software system, it must obviously be deployed on a platform that provides the required hardware and software support.

2. *The availability requirements of the system* High-availability systems may require components to be deployed on more than one platform. This means that, in the event of platform failure, an alternative implementation of the component is available.

3. *Component communications* If there is a lot of intercomponent communication, it is usually best to deploy them on the same platform or on platforms that are physically close to one another. This reduces communications latency—the delay between the time that a message is sent by one component and received by another.

You can document your decisions on hardware and software deployment using UML deployment diagrams, which show how software components are distributed across hardware platforms.

If you are developing an embedded system, you may have to take into account target characteristics, such as its physical size, power capabilities, the need for real-time responses to sensor events, the physical characteristics of actuators and its real-time operating system. I discuss embedded systems engineering in Chapter 21.

7.4 Open-source development

Open-source development is an approach to software development in which the source code of a software system is published and volunteers are invited to participate in the development process (Raymond 2001). Its roots are in the Free Software Foundation (www.fsf.org), which advocates that source code should not be proprietary but rather should always be available for users to examine and modify as they wish. There was an assumption that the code would be controlled and developed by a small core group, rather than users of the code.

Open-source software extended this idea by using the Internet to recruit a much larger population of volunteer developers. Many of them are also users of the code. In principle at least, any contributor to an open-source project may report and fix bugs and propose new features and functionality. However, in practice, successful open-source systems still rely on a core group of developers who control changes to the software.

Open-source software is the backbone of the Internet and software engineering. The Linux operating system is the most widely used server system, as is the open-source Apache web server. Other important and universally used open-source products are Java, the Eclipse IDE, and the mySQL database management system. The Android operating system is installed on millions of mobile devices. Major players in the computer industry such as IBM and Oracle, support the open-source movement and base their software on open-source products. Thousands of other, lesser-known open-source systems and components may also be used.

It is usually cheap or even free to acquire open-source software. You can normally download open-source software without charge. However, if you want documentation and support, then you may have to pay for this, but costs are usually fairly low. The other key benefit of using open-source products is that widely used open-source systems are very reliable. They have a large population of users who are willing to fix problems themselves rather than report these problems to the developer and wait for a new release of the system. Bugs are discovered and repaired more quickly than is usually possible with proprietary software.

For a company involved in software development, there are two open-source issues that have to be considered:

1. Should the product that is being developed make use of open-source components?
2. Should an open-source approach be used for its own software development?

The answers to these questions depend on the type of software that is being developed and the background and experience of the development team.

If you are developing a software product for sale, then time to market and reduced costs are critical. If you are developing software in a domain in which there are high-quality open-source systems available, you can save time and money by using these systems. However, if you are developing software to a specific set of organizational requirements, then using open-source components may not be an option. You may have to integrate your software with existing systems that are incompatible with available open-source systems. Even then, however, it could be quicker and cheaper to modify the open-source system rather than redevelop the functionality that you need.

Many software product companies are now using an open-source approach to development, especially for specialized systems. Their business model is not reliant on selling a software product but rather on selling support for that product. They believe that involving the open-source community will allow software to be developed more cheaply and more quickly and will create a community of users for the software.

Some companies believe that adopting an open-source approach will reveal confidential business knowledge to their competitors and so are reluctant to adopt this development model. However, if you are working in a small company and you open source your software, this may reassure customers that they will be able to support the software if your company goes out of business.

Publishing the source code of a system does not mean that people from the wider community will necessarily help with its development. Most successful open-source products have been platform products rather than application systems. There are a limited number of developers who might be interested in specialized application systems. Making a software system open source does not guarantee community involvement. There are thousands of open-source projects on Sourceforge and GitHub that have only a handful of downloads. However, if users of your software have concerns about its availability in future, making the software open source means that they can take their own copy and so be reassured that they will not lose access to it.

7.4.1 Open-source licensing

Although a fundamental principle of open-source development is that source code should be freely available, this does not mean that anyone can do as they wish with that code. Legally, the developer of the code (either a company or an individual) owns the code. They can place restrictions on how it is used by including legally binding conditions in an open-source software license (St. Laurent 2004). Some open-source developers believe that if an open-source component is used to develop a new system, then that system should also be open source. Others are willing to allow their code to be used without this restriction. The developed systems may be proprietary and sold as closed-source systems.

Most open-source licenses (Chapman 2010) are variants of one of three general models:

1. The GNU General Public License (GPL). This is a so-called reciprocal license that simplistically means that if you use open-source software that is licensed under the GPL license, then you must make that software open source.

2. The GNU Lesser General Public License (LGPL). This is a variant of the GPL license where you can write components that link to open-source code without having to publish the source of these components. However, if you change the licensed component, then you must publish this as open source.

3. The Berkley Standard Distribution (BSD) License. This is a nonreciprocal license, which means you are not obliged to re-publish any changes or modifications made to

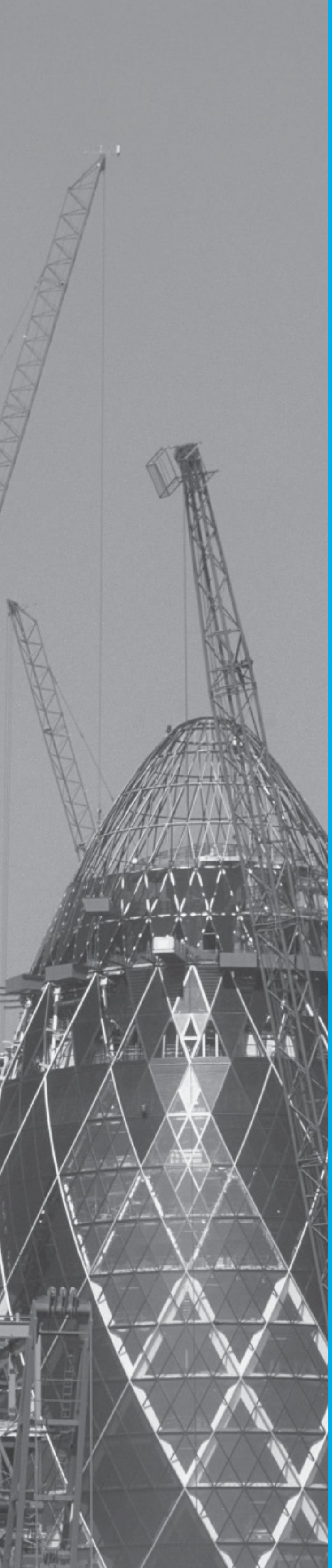
open-source code. You can include the code in proprietary systems that are sold. If you use open-source components, you must acknowledge the original creator of the code. The MIT license is a variant of the BSD license with similar conditions.

Licensing issues are important because if you use open-source software as part of a software product, then you may be obliged by the terms of the license to make your own product open source. If you are trying to sell your software, you may wish to keep it secret. This means that you may wish to avoid using GPL-licensed open-source software in its development.

If you are building software that runs on an open-source platform but that does not reuse open-source components, then licenses are not a problem. However, if you embed open-source software in your software, you need processes and databases to keep track of what's been used and their license conditions. Bayersdorfer (Bayersdorfer 2007) suggests that companies managing projects that use open source should:

1. Establish a system for maintaining information about open-source components that are downloaded and used. You have to keep a copy of the license for each component that was valid at the time the component was used. Licenses may change, so you need to know the conditions that you have agreed to.
2. Be aware of the different types of licenses and understand how a component is licensed before it is used. You may decide to use a component in one system but not in another because you plan to use these systems in different ways.
3. Be aware of evolution pathways for components. You need to know a bit about the open-source project where components are developed to understand how they might change in future.
4. Educate people about open source. It's not enough to have procedures in place to ensure compliance with license conditions. You also need to educate developers about open source and open-source licensing.
5. Have auditing systems in place. Developers, under tight deadlines, might be tempted to break the terms of a license. If possible, you should have software in place to detect and stop this.
6. Participate in the open-source community. If you rely on open-source products, you should participate in the community and help support their development.

The open-source approach is one of several business models for software. In this model, companies release the source of their software and sell add-on services and advice in association with this. They may also sell cloud-based software services—an attractive option for users who do not have the expertise to manage their own open-source system and also specialized versions of their system for particular clients. Open-source is therefore likely to increase in importance as a way of developing and distributing software.



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 about three distinct types of testing—component testing, system testing, and release testing;
- understand the distinctions between development testing and user testing.

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.

When you test software, you are trying to do two things:

1. 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 that will be included in the product release. You may also test combinations of features to check for unwanted interactions between them.

2. Find inputs or input sequences where the behavior of the software is incorrect, undesirable, or does not conform to its specification. These are caused by defects (bugs) in the software. When you test software to find defects, you are trying to root out undesirable system behavior such as system crashes, unwanted interactions with other systems, incorrect computations, and data corruption.

The first of these is validation testing, where you expect the system to perform correctly using a set of test cases that reflect the system's expected use. The second is 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.

Figure 8.1 shows 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 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 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. Barry Boehm, a pioneer of software engineering, succinctly expressed the difference between them (Boehm 1979):

[†]Dijkstra, E. W. 1972. "The Humble Programmer." Comm. ACM 15 (10): 859–66. doi:10.1145/355604.361591

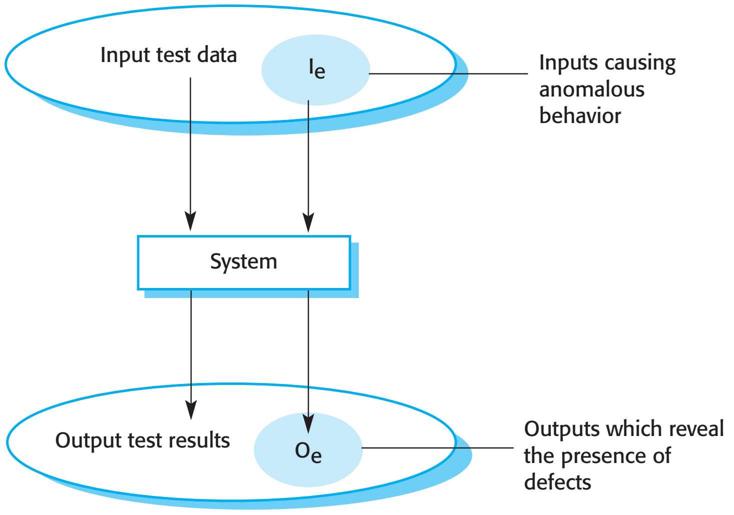

Input test data
I_e
Inputs causing anomalous behavior
System
Output test results
O_e
Outputs which reveal the presence of defects

Figure 8.1 An input–output model of program testing

- **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.

Software verification is the process of checking that the software meets its stated functional and non-functional requirements. Validation is a more general process. The aim of software validation is to ensure that the software meets the customer's expectations. It goes beyond 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, statements of requirements do not always reflect the real wishes or needs of system customers and users.

The 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 it is 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 demonstrator system that prototypes new product ideas.
2. *User expectations* Because of their previous experiences with buggy, unreliable software, users sometimes have low expectations of software quality. They are not surprised when their software fails. When a new system is installed, users

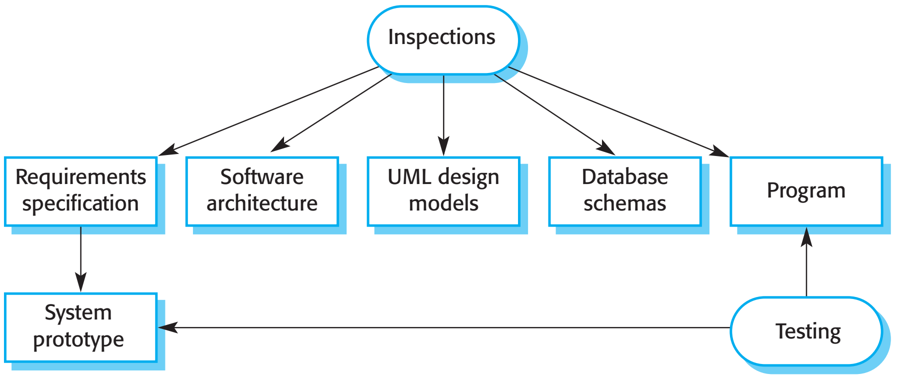


Figure 8.2 Inspections and testing

may tolerate failures because the benefits of use outweigh the costs of failure recovery. However, as a software product becomes more established, users expect it to become more reliable. Consequently, more thorough testing of later versions of the system may be required.

3. *Marketing environment* When a software company brings a system to market, it 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, the company may decide to release a program before it has been fully tested and debugged because it wants to be the first into the market. If a software product or app 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 "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.

Software inspection has three advantages 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 doesn't involve executing the system, you don't have to worry about interactions between errors. Consequently, a single inspection session can discover many errors in a system.

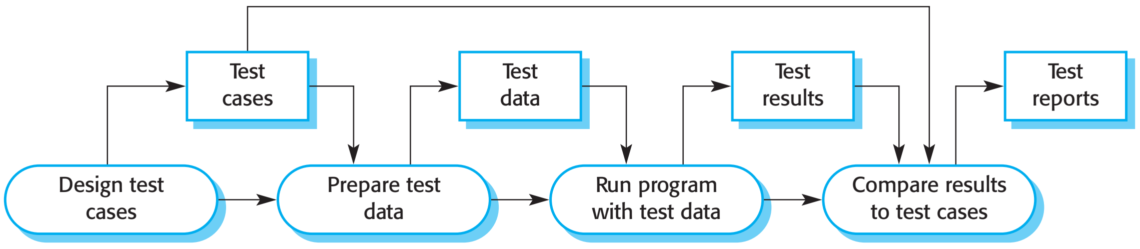


Figure 8.3 A model of the software testing process

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 several studies and experiments have shown that inspections are more effective for defect discovery than program testing. Fagan (Fagan 1976) 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. In small companies or development groups, it can be difficult and expensive to put together a separate inspection team as all potential team members may also be developers of the software.

I discuss reviews and inspections in more detail in Chapter 24 (Quality Management). Static analysis, where the source text of a program is automatically analyzed to discover anomalies, is explained in Chapter 12. 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. 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 test 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.

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 that 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://software-engineering-book.com/web/test-planning/

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 the 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 that decides 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 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.

Unfortunately, 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 unanticipated 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. Some development processes use programmer/tester pairs (Cusamano and Selby 1998) where each programmer has an associated tester who

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. When you are debugging a program, you usually use interactive tools that provide extra information about program execution.

http://software-engineering-book.com/web/debugging/

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. This group is responsible for developing tests and maintaining detailed records of test results.

There are three stages of development testing:

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 the component interfaces that provide access to the component functions.
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; and put the object into all possible states. This means that you should simulate all events that cause a state change.

Consider, for example, the weather station object from the example that I discussed in Chapter 7. The attributes and operations of this object are shown in Figure 8.4.

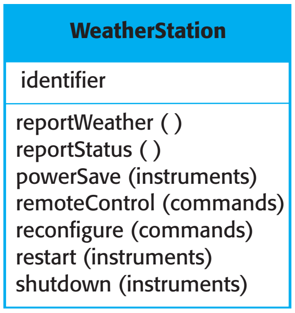


Figure 8.4 The weather station object interface

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** and **reportStatus**. Ideally, you should test methods in isolation, but, in some cases, 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 all of the subclasses that inherit the operation. The operation that is inherited may make assumptions about other operations and attributes. These assumptions may not be valid in some subclasses that inherit the operation. You therefore have to test the inherited operation everywhere that it is used.

To test the states of the weather station, you can use a state model as discussed in Chapter 7 (Figure 7.8). Using this model, you 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 (Tahchiev et al. 2010) 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 graphical unit interface (GUI), on the success or otherwise of the tests. An entire test suite can often be run in a few seconds, so it is possible to execute all 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 implemented or whose use slows down the testing process. For example, if an object calls a database, this may involve a slow setup process before it can be used. In such 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. For example, a mock object simulating a database may have only a few data items that are organized in an array. They can be accessed quickly, without the overheads of calling a database and accessing disks. Similarly, mock objects can be used to simulate abnormal operations 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 design 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.

Two strategies that can be effective in helping you choose test cases 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.

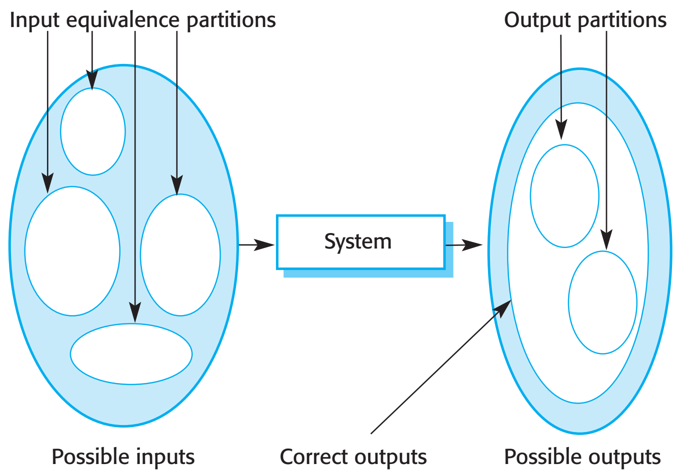


Figure 8.5 Equivalence partitioning

The input data and output results of a program can be thought of as members of sets with common characteristics. Examples of these sets are positive numbers, negative numbers, and menu selections. Programs normally behave in a comparable way for all members of a set. That is, if you test a program that does a computation and requires two positive numbers, then you would expect the program to behave in the same way for all positive numbers.

Because of this equivalent behavior, these classes are sometimes called equivalence partitions or domains (Bezier 1990). One systematic approach to test-case design is based on identifying all input and output partitions for a system or component. Test cases are designed so that the inputs or outputs lie within these partitions. Partition testing can be used to design test cases for both systems and components.

In Figure 8.5, the large shaded ellipse on the left represents the set of all possible inputs to the program that is being tested. The smaller unshaded ellipses represent equivalence partitions. A program being tested should process all of the members of an input equivalence partition in the same way.

Output equivalence partitions are partitions within which all of the outputs have something in common. Sometimes there is a 1:1 mapping between input and output equivalence partitions. However, this is not always the case; you may need to define a separate input equivalence partition, where the only common characteristic of the inputs is that they generate outputs within the same output partition. The shaded area in the left ellipse represents inputs that are invalid. The shaded area in the right ellipse represents exceptions that may occur, that is, responses to invalid inputs.

Once you have identified a set of partitions, you choose test cases from each of these partitions. A good rule of thumb for test-case selection is to choose test cases on the boundaries of the partitions, plus cases close to the midpoint of the partition. The reason for this is that designers and programmers tend to consider typical values of inputs when developing a system. You test these by choosing the midpoint of the partition. Boundary values are often atypical (e.g., zero may behave differently from other non-negative numbers) and so are sometimes overlooked by developers. Program failures often occur when processing these atypical values.

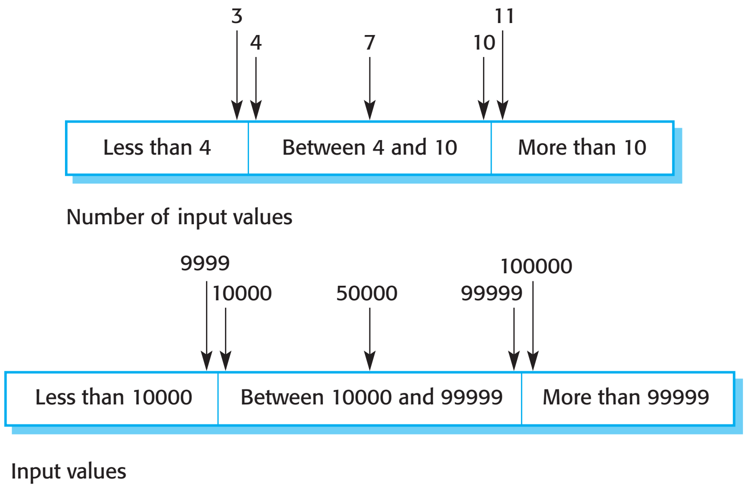


Figure 8.6 Equivalence partitions

You identify partitions by using the program specification or user documentation and from experience where you predict the classes of input value that are likely to detect errors. For example, say a program specification states that the program accepts four to eight inputs which are five-digit integers greater than 10,000. You use this information to identify the input partitions and possible test input values. These are shown in Figure 8.6.

When you use the specification of a system to identify equivalence partitions, this is called black-box testing. You don't need any knowledge of how the system works. It is sometimes useful to supplement the black-box tests with "white-box testing," where you look at the code of the program to find other possible tests. For example, your code may include exceptions to handle incorrect inputs. You can use this knowledge to identify "exception partitions"—different ranges where the same exception handling should be applied.

Equivalence partitioning is an effective approach to testing because it helps account for errors that programmers often make when processing inputs at the edges of partitions. You can also use testing guidelines to help choose test cases. Guidelines encapsulate knowledge of what kinds of test cases are effective for discovering errors. For example, when you are testing programs with sequences, arrays, or lists, guidelines that could help reveal defects include:

1. Test software with sequences that have only a single value. Programmers naturally think of sequences as made up of several values, and sometimes they embed this assumption in their programs. Consequently, if presented with a single-value sequence, a program may not work properly.
2. Use different sequences of different sizes in different tests. This decreases the chances that a program with defects will accidentally produce a correct output because of some accidental characteristics of the input.
3. Derive tests so that the first, middle, and last elements of the sequence are accessed. This approach reveals problems at partition boundaries.

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 to test the methods associated with objects.

http://software-engineering-book.com/web/path-testing/

Whittaker's book (Whittaker 2009) 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.

8.1.3 Component testing

Software components are often 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 component interfaces (see Chapter 7). Testing composite components should therefore focus on showing that the component interface or interfaces behave 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.

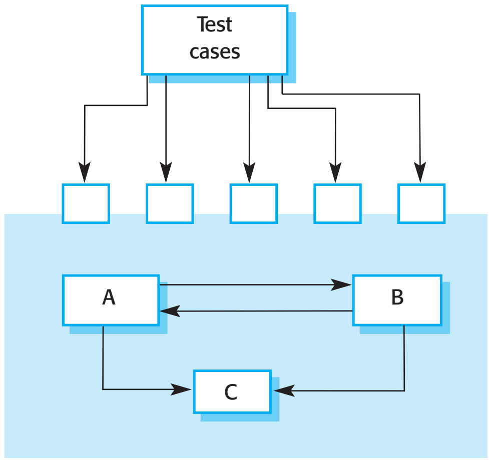


Figure 8.7 Interface testing

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 subsystems. This type of interface is 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 in 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 operate at different speeds. Unless particular care is taken in the interface design, the consumer can access out-of-date information because the producer of the information has not updated the shared interface information.

Testing for interface defects is difficult because some interface faults may only manifest themselves under unusual conditions. For example, say an object implements a queue as a fixed-length data structure. A calling object may assume that the queue is implemented as an infinite data structure, and so it does not check for queue overflow when an item is entered.

This condition can only be detected during testing by designing a sequence of test cases that force the queue to overflow. The tests should check how calling objects handle that overflow. However, as this is a rare condition, testers may think that this isn't worth checking when writing the test set for the queue object.

A further problem may arise because of interactions between faults in different modules or objects. Faults in one object may only be detected when some other object behaves in an unexpected way. Say an object calls another object to receive some service and the calling object assumes that the response is correct. If the called service is faulty in some way, the returned value may be valid but incorrect. The problem is therefore not immediately detectable but only becomes obvious when some later computation, using the returned value, goes wrong.

Some general guidelines for interface testing are:

1. Examine the code to be tested and identify each call to an external component. Design a set of tests in which the values of the parameters to the external components are at the extreme ends of their ranges. These extreme values are most likely to reveal interface inconsistencies.

2. Where pointers are passed across an interface, always test the interface with null pointer parameters.

3. Where a component is called through a procedural interface, design tests that deliberately cause the component to fail. Differing failure assumptions are one of the most common specification misunderstandings.

4. Use stress testing in message passing systems. This means that you should design tests that generate many more messages than are likely to occur in practice. This is an effective way of revealing timing problems.

5. Where several components interact through shared memory, design tests that vary the order in which these components are activated. These tests may reveal implicit assumptions made by the programmer about the order in which the shared data is produced and consumed.

Sometimes it is better to use inspections and reviews rather than testing to look for interface errors. Inspections can concentrate on component interfaces and questions about the assumed interface behavior asked during the inspection process.

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 subteams 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.

All systems have emergent behavior. This means that some system functionality and characteristics 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 the system database. 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.

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. Several components or objects normally implement each use case in the system. Testing the use case forces these interactions to occur. If you have developed a sequence diagram to model the use case implementation, you can see the objects or components that are involved in the interaction.

In the wilderness weather station example, the system software reports summarized weather data to a remote computeras described in Figure 7.3. Figure 8.8 shows the sequence of operations in the weather station when it responds to a request to collect data for the mapping system. You can use this diagram to identify operations that will be tested and to help design the test cases to execute the tests. Therefore issuing a request for a report will result in the execution of the following thread of methods:

SatComms:request → WeatherStation:reportWeather → Commslink:Get(summary) → WeatherData:summarize

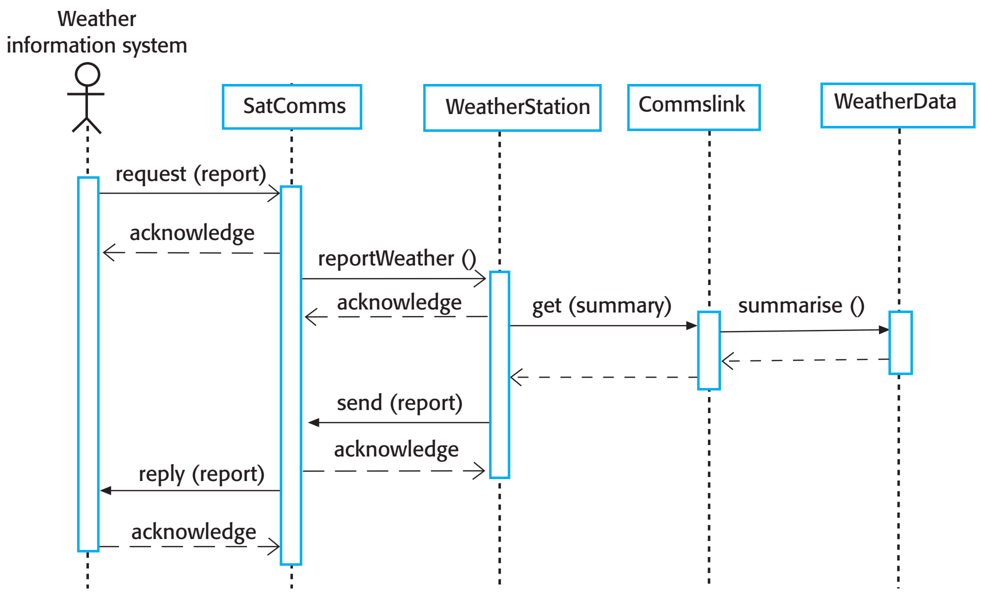


Figure 8.8 Collect weather data sequence chart

The sequence diagram helps you design the specific test cases that you need, as it shows what inputs are required and what outputs are created:

1. An input of a request for a report should have an associated acknowledgment. A report should ultimately be returned from the request. During testing, you should create summarized data that can be used to check that the report is correctly organized.

2. An input request for a report to WeatherStation results in a summarized report being generated. You can test this in isolation by creating raw data corresponding to the summary that you have prepared for the test of SatComms and checking that the WeatherStation object correctly produces this summary. This raw data is also used to test the WeatherData object.

Of course, I have simplified the sequence diagram in Figure 8.8 so that it does not show exceptions. A complete use case/scenario test must take these exceptions into account and ensure that they are correctly handled.

For most systems, it is difficult to know how much system testing is essential and when you should stop testing. Exhaustive testing, where every possible program execution sequence is tested, is impossible. Testing, therefore, has to be based on a subset of possible test cases. Ideally, software companies should have policies for choosing this subset. These policies might be based on general testing policies, such as a policy that all program statements should be executed at least once. Alternatively, they may be based on experience of system usage and focus on testing the features of the operational system. For example:

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 where you integrate a component, test the system, integrate another component, test again, and so on. If problems occur, they are probably due to interactions with the most recently integrated component.

Incremental integration and testing is fundamental to agile methods, where regression tests are run every time a new increment is integrated.

http://software-engineering-book.com/web/integration/

1. All system functions that are accessed through menus should be tested.
2. Combinations of functions (e.g., text formatting) that are accessed through the same menu must be tested.
3. Where user input is provided, all functions must be tested with both correct and incorrect input.

It is clear from experience with major software products such as word processors or spreadsheets that similar guidelines are normally used during product testing. When features of the software are used in isolation, they normally work. Problems arise, as Whittaker explains (Whittaker 2009), when combinations of less commonly used features have not been tested together. He gives the example of how, in a commonly used word processor, using footnotes with multicolumn layout causes incorrect layout of the text.

Automated system testing is usually more difficult than automated unit or component testing. Automated unit testing relies on predicting the outputs and then encoding these predictions in a program. The prediction is then compared with the result. However, the point of implementing a system may be to generate outputs that are large or cannot be easily predicted. You may be able to examine an output and check its credibility without necessarily being able to create it in advance.

8.2 Test-driven development

Test-driven development (TDD) is an approach to program development in which you interleave testing and code development (Beck 2002; Jeffries and Melnik 2007). You develop the code incrementally, along with a set of tests for that increment. You don't start working on the next increment until the code that you have developed passes all of its tests. Test-driven development was introduced as part of the XP agile development method. However, it has now gained mainstream acceptance and may be used in both agile and plan-based processes.

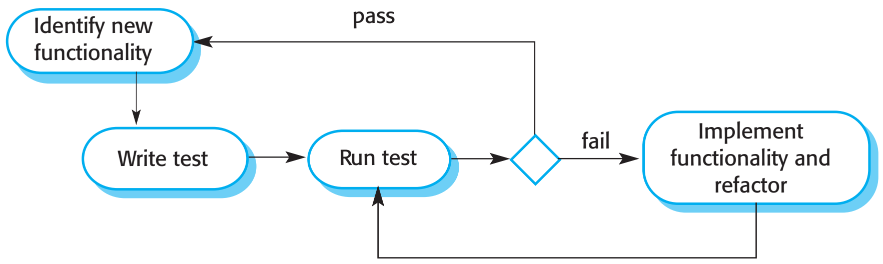

Identify new functionality
pass
Write test
Run test
fail
Implement functionality and refactor

Figure 8.9 Test-driven development

The fundamental TDD process is shown in Figure 8.9. The steps in the process are as follows:

1. You start by identifying the increment of functionality that is required. This should normally be small and implementable in a few lines of code.
2. You write a test for this functionality and implement it as an automated test. This means that the test can be executed and will report whether or not it has passed or failed.
3. You then run the test, along with all other tests that have been implemented. Initially, you have not implemented the functionality so the new test will fail. This is deliberate as it shows that the test adds something to the test set.
4. You then implement the functionality and re-run the test. This may involve refactoring existing code to improve it and add new code to what's already there.
5. Once all tests run successfully, you move on to implementing the next chunk of functionality.

An automated testing environment, such as the JUnit environment that supports Java program testing (Tahchiev et al. 2010) is essential for TDD. As the code is developed in very small increments, you have to be able to run every test each time that you add functionality or refactor the program. Therefore, the tests are embedded in a separate program that runs the tests and invokes the system that is being tested. Using this approach, you can run hundreds of separate tests in a few seconds.

Test-driven development helps programmers clarify their ideas of what a code segment is actually supposed to do. To write a test, you need to understand what is intended, as this understanding makes it easier to write the required code. Of course, if you have incomplete knowledge or understanding, then TDD won't help.

If you don't know enough to write the tests, you won't develop the required code. For example, if your computation involves division, you should check that you are not dividing the numbers by zero. If you forget to write a test for this, then the checking code will never be included in the program.

As well as better problem understanding, other benefits of test-driven development are:

1. *Code coverage* In principle, every code segment that you write should have at least one associated test. Therefore, you can be confident that all of the code in

the system has actually been executed. Code is tested as it is written, so defects are discovered early in the development process.

2. *Regression testing* A test suite is developed incrementally as a program is developed. You can always run regression tests to check that changes to the program have not introduced new bugs.

3. *Simplified debugging* When a test fails, it should be obvious where the problem lies. The newly written code needs to be checked and modified. You do not need to use debugging tools to locate the problem. Reports of the use of TDD suggest that it is hardly ever necessary to use an automated debugger in test-driven development (Martin 2007).

4. *System documentation* The tests themselves act as a form of documentation that describe what the code should be doing. Reading the tests can make it easier to understand the code.

One of the most important benefits of TDD is that it reduces the costs of regression testing. Regression testing involves running test sets that have successfully executed after changes have been made to a system. The regression test checks that these changes have not introduced new bugs into the system and that the new code interacts as expected with the existing code. Regression testing is expensive and sometimes impractical when a system is manually tested, as the costs in time and effort are very high. You have to try to choose the most relevant tests to re-run and it is easy to miss important tests.

Automated testing dramatically reduces the costs of regression testing. Existing tests may be re-run quickly and cheaply. After making a change to a system in test-first development, all existing tests must run successfully before any further functionality is added. As a programmer, you can be confident that the new functionality that you have added has not caused or revealed problems with existing code.

Test-driven development is of most value in new software development where the functionality is either implemented in new code or by using components from standard libraries. If you are reusing large code components or legacy systems, then you need to write tests for these systems as a whole. You cannot easily decompose them into separate testable elements. Incremental test-driven development is impractical. Test-driven development may also be ineffective with multithreaded systems. The different threads may be interleaved at different times in different test runs, and so may produce different results.

If you use TDD, you still need a system testing process to validate the system, that is, to check that it meets the requirements of all of the system stakeholders. System testing also tests performance, reliability, and checks that the system does not do things that it shouldn't do, such as produce unwanted outputs. Andrea (Andrea 2007) suggests how testing tools can be extended to integrate some aspects of system testing with TDD.

Test-driven development is now a widely used and mainstream approach to software testing. Most programmers who have adopted this approach are happy with it and find it a more productive way to develop software. It is also claimed that use of TDD encourages better structuring of a program and improved code quality. However, experiments to verify this claim have been inconclusive.

8.3 Release testing

Release testing is the process of testing a particular release of a system that is intended for use outside of the development team. Normally, the system release is for customers and users. In a complex project, however, the release could be for other teams that are developing related systems. For software products, the release could be for product management who then prepare it for sale.

There are two important distinctions between release testing and system testing during the development process:

1. The system development, team should not be responsible for release testing.
2. Release testing is a process of validation checking to ensure that a system meets its requirements and is good enough for use by system customers. System testing by the development team should focus on discovering bugs in the system (defect testing).

The primary goal of the release testing process is to convince the supplier of the system that it is good enough for use. If so, it can be released as a product or delivered to the customer. Release testing, therefore, has to show that the system delivers its specified functionality, performance, and dependability, and that it does not fail during normal use.

Release testing is usually a black-box testing process whereby tests are derived from the system specification. The system is treated as a black box whose behavior can only be determined by studying its inputs and the related outputs. Another name for this is functional testing, so-called because the tester is only concerned with functionality and not the implementation of the software.

8.3.1 Requirements-based testing

A general principle of good requirements engineering practice is that requirements should be testable. That is, the requirement should be written so that a test can be designed for that requirement. A tester can then check that the requirement has been satisfied. Requirements-based testing, therefore, is a systematic approach to test-case design where you consider each requirement and derive a set of tests for it. Requirements-based testing is validation rather than defect testing—you are trying to demonstrate that the system has properly implemented its requirements.

For example, consider the following Mentcare system requirements that are concerned with checking for drug allergies:

> *If a patient is known to be allergic to any particular medication, then prescription of that medication shall result in a warning message being issued to the system user.*
>
> *If a prescriber chooses to ignore an allergy warning, he or she shall provide a reason why this has been ignored.*

To check if these requirements have been satisfied, you may need to develop several related tests:

1. Set up a patient record with no known allergies. Prescribe medication for allergies that are known to exist. Check that a warning message is not issued by the system.
2. Set up a patient record with a known allergy. Prescribe the medication that the patient is allergic to and check that the warning is issued by the system.
3. Set up a patient record in which allergies to two or more drugs are recorded. Prescribe both of these drugs separately and check that the correct warning for each drug is issued.
4. Prescribe two drugs that the patient is allergic to. Check that two warnings are correctly issued.
5. Prescribe a drug that issues a warning and overrule that warning. Check that the system requires the user to provide information explaining why the warning was overruled.

You can see from this list that testing a requirement does not mean just writing a single test. You normally have to write several tests to ensure that you have coverage of the requirement. You should also keep traceability records of your requirements-based testing, which link the tests to the specific requirements that you have tested.

8.3.2 Scenario testing

Scenario testing is an approach to release testing whereby you devise typical scenarios of use and use these scenarios to develop test cases for the system. A scenario is a story that describes one way in which the system might be used. Scenarios should be realistic, and real system users should be able to relate to them. If you have used scenarios or user stories as part of the requirements engineering process (described in Chapter 4), then you may be able to reuse them as testing scenarios.

In a short paper on scenario testing, Kaner (Kaner 2003) suggests that a scenario test should be a narrative story that is credible and fairly complex. It should motivate stakeholders; that is, they should relate to the scenario and believe that it is

George is a nurse who specializes in mental health care. One of his responsibilities is to visit patients at home to check that their treatment is effective and that they are not suffering from medication side effects.

On a day for home visits, George logs into the Mentcare system and uses it to print his schedule of home visits for that day, along with summary information about the patients to be visited. He requests that the records for these patients be downloaded to his laptop. He is prompted for his key phrase to encrypt the records on the laptop.

One of the patients whom he visits is Jim, who is being treated with medication for depression. Jim feels that the medication is helping him but believes that it has the side effect of keeping him awake at night. George looks up Jim's record and is prompted for his key phrase to decrypt the record. He checks the drug prescribed and queries its side effects. Sleeplessness is a known side effect, so he notes the problem in Jim's record and suggests that he visit the clinic to have his medication changed. Jim agrees, so George enters a prompt to call him when he gets back to the clinic to make an appointment with a physician. George ends the consultation, and the system re-encrypts Jim's record.

After finishing his consultations, George returns to the clinic and uploads the records of patients visited to the database. The system generates a call list for George of those patients whom he has to contact for follow-up information and make clinic appointments.

Figure 8.10 A user story for the Mentcare system

important that the system passes the test. He also suggests that it should be easy to evaluate. If there are problems with the system, then the release testing team should recognize them.

As an example of a possible scenario from the Mentcare system, Figure 8.10 describes one way that the system may be used on a home visit. This scenario tests a number of features of the Mentcare system:

1. Authentication by logging on to the system.
2. Downloading and uploading of specified patient records to a laptop.
3. Home visit scheduling.
4. Encryption and decryption of patient records on a mobile device.
5. Record retrieval and modification.
6. Links with the drugs database that maintains side-effect information.
7. The system for call prompting.

If you are a release tester, you run through this scenario, playing the role of George and observing how the system behaves in response to different inputs. As George, you may make deliberate mistakes, such as inputting the wrong key phrase to decode records. This checks the response of the system to errors. You should carefully note any problems that arise, including performance problems. If a system is too slow, this will change the way that it is used. For example, if it takes too long to encrypt a record, then users who are short of time may skip this stage. If they then lose their laptop, an unauthorized person could then view the patient records.

When you use a scenario-based approach, you are normally testing several requirements within the same scenario. Therefore, as well as checking individual requirements, you are also checking that combinations of requirements do not cause problems.

8.3.3 Performance testing

Once a system has been completely integrated, it is possible to test for emergent properties, such as performance and reliability. Performance tests have to be designed to ensure that the system can process its intended load. This usually involves running a series of tests where you increase the load until the system performance becomes unacceptable.

As with other types of testing, performance testing is concerned both with demonstrating that the system meets its requirements and discovering problems and defects in the system. To test whether performance requirements are being achieved, you may have to construct an operational profile. An operational profile (see Chapter 11) is a set of tests that reflect the actual mix of work that will be handled by the system. Therefore, if 90% of the transactions in a system are of type A, 5% of type B, and the remainder of types C, D, and E, then you have to design the operational profile so that the vast majority of tests are of type A. Otherwise, you will not get an accurate test of the operational performance of the system.

This approach, of course, is not necessarily the best approach for defect testing. Experience has shown that an effective way to discover defects is to design tests around the limits of the system. In performance testing, this means stressing the system by making demands that are outside the design limits of the software. This is known as stress testing.

Say you are testing a transaction processing system that is designed to process up to 300 transactions per second. You start by testing this system with fewer than 300 transactions per second. You then gradually increase the load on the system beyond 300 transactions per second until it is well beyond the maximum design load of the system and the system fails.

Stress testing helps you do two things:

1. Test the failure behavior of the system. Circumstances may arise through an unexpected combination of events where the load placed on the system exceeds the maximum anticipated load. In these circumstances, system failure should not cause data corruption or unexpected loss of user services. Stress testing checks that overloading the system causes it to "fail-soft" rather than collapse under its load.

2. Reveal defects that only show up when the system is fully loaded. Although it can be argued that these defects are unlikely to cause system failures in normal use, there may be unusual combinations of circumstances that the stress testing replicates.

Stress testing is particularly relevant to distributed systems based on a network of processors. These systems often exhibit severe degradation when they are heavily loaded. The network becomes swamped with coordination data that the different processes must exchange. The processes become slower and slower as they wait for the required data from other processes. Stress testing helps you discover when the degradation begins so that you can add checks to the system to reject transactions beyond this point.

8.4 User testing

User or customer testing is a stage in the testing process in which users or customers provide input and advice on system testing. This may involve formally testing a system that has been commissioned from an external supplier. Alternatively, it may be an informal process where users experiment with a new software product to see if they like it and to check that it does what they need. User testing is essential, even when comprehensive system and release testing have been carried out. Influences from the user's working environment can have a major effect on the reliability, performance, usability, and robustness of a system.

It is practically impossible for a system developer to replicate the system's working environment, as tests in the developer's environment are inevitably artificial. For example, a system that is intended for use in a hospital is used in a clinical environment where other things are going on, such as patient emergencies and conversations with relatives. These all affect the use of a system, but developers cannot include them in their testing environment.

There are three different types of user testing:

1. *Alpha testing*, where a selected group of software users work closely with the development team to test early releases of the software.
2. *Beta testing*, where a release of the software is made available to a larger group of users to allow them to experiment and to raise problems that they discover with the system developers.
3. *Acceptance testing*, where customers test a system to decide whether or not it is ready to be accepted from the system developers and deployed in the customer environment.

In alpha testing, users and developers work together to test a system as it is being developed. This means that the users can identify problems and issues that are not readily apparent to the development testing team. Developers can only really work from the requirements, but these often do not reflect other factors that affect the practical use of the software. Users can therefore provide information about practice that helps with the design of more realistic tests.

Alpha testing is often used when developing software products or apps. Experienced users of these products may be willing to get involved in the alpha testing process because this gives them early information about new system features that they can exploit. It also reduces the risk that unanticipated changes to the software will have disruptive effects on their business. However, alpha testing may also be used when custom software is being developed. Agile development methods advocate user involvement in the development process, and that users should play a key role in designing tests for the system.

Beta testing takes place when an early, sometimes unfinished, release of a software system is made available to a larger group of customers and users for evaluation. Beta testers may be a selected group of customers who are early adopters of the system.

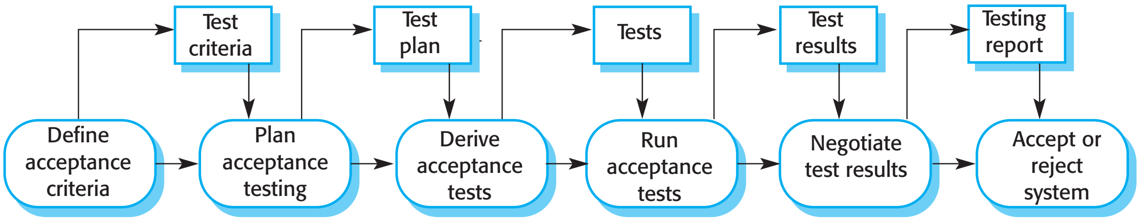


Figure 8.11 The acceptance testing process

Alternatively, the software may be made publicly available for use by anyone who is interested in experimenting with it.

Beta testing is mostly used for software products that are used in many different settings. This is important as, unlike custom product developers, there is no way for the product developer to limit the software's operating environment. It is impossible for product developers to know and replicate all the settings in which the software product will be used. Beta testing is therefore used to discover interaction problems between the software and features of its operational environment. Beta testing is also a form of marketing. Customers learn about their system and what it can do for them.

Acceptance testing is an inherent part of custom systems development. Customers test a system, using their own data, and decide if it should be accepted from the system developer. Acceptance implies that final payment should be made for the software.

Figure 8.11 shows that here are six stages in the acceptance testing process:

1. *Define acceptance criteria* This stage should ideally take place early in the process before the contract for the system is signed. The acceptance criteria should be part of the system contract and be approved by the customer and the developer. In practice, however, it can be difficult to define criteria so early in the process. Detailed requirements may not be available, and the requirements will almost certainly change during the development process.

2. *Plan acceptance testing* This stage involves deciding on the resources, time, and budget for acceptance testing and establishing a testing schedule. The acceptance test plan should also discuss the required coverage of the requirements and the order in which system features are tested. It should define risks to the testing process such as system crashes and inadequate performance, and discuss how these risks can be mitigated.

3. *Derive acceptance tests* Once acceptance criteria have been established, tests have to be designed to check whether or not a system is acceptable. Acceptance tests should aim to test both the functional and non-functional characteristics (e.g., performance) of the system. They should ideally provide complete coverage of the system requirements. In practice, it is difficult to establish completely objective acceptance criteria. There is often scope for argument about whether or not a test shows that a criterion has definitely been met.

4. *Run acceptance tests* The agreed acceptance tests are executed on the system. Ideally, this step should take place in the actual environment where the system will be used, but this may be disruptive and impractical. Therefore, a user testing

environment may have to be set up to run these tests. It is difficult to automate this process as part of the acceptance tests may involve testing the interactions between end-users and the system. Some training of end-users may be required.

5. *Negotiate test results* It is very unlikely that all of the defined acceptance tests will pass and that there will be no problems with the system. If this is the case, then acceptance testing is complete and the system can be handed over. More commonly, some problems will be discovered. In such cases, the developer and the customer have to negotiate to decide if the system is good enough to be used. They must also agree on how the developer will fix the identified problems.

6. *Reject/accept system* This stage involves a meeting between the developers and the customer to decide on whether or not the system should be accepted. If the system is not good enough for use, then further development is required to fix the identified problems. Once complete, the acceptance testing phase is repeated.

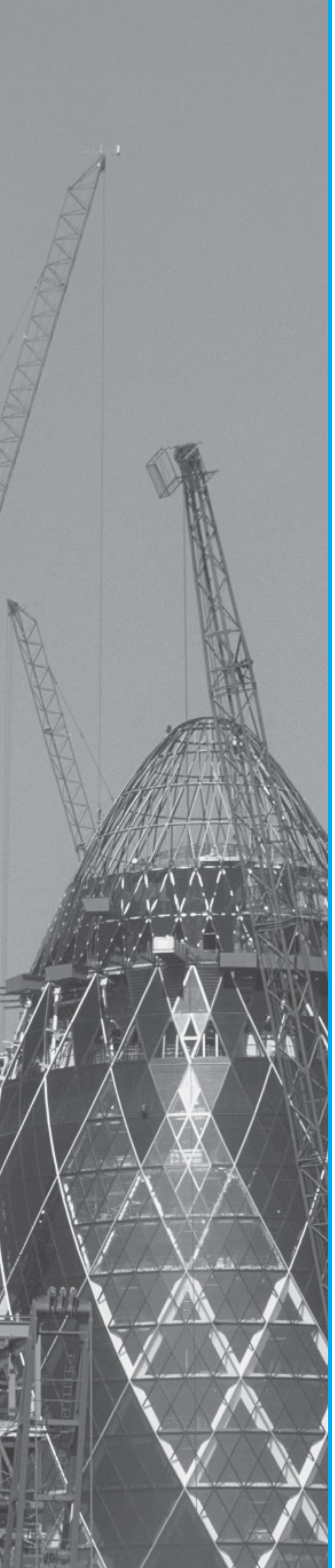
You might think that acceptance testing is a clear-cut contractual issue. If a system does not pass its acceptance tests, then it should not be accepted and payment should not be made. However, the reality is more complex. Customers want to use the software as soon as they can because of the benefits of its immediate deployment. They may have bought new hardware, trained staff, and changed their processes. They may be willing to accept the software, irrespective of problems, because the costs of not using the software are greater than the costs of working around the problems.

Therefore, the outcome of negotiations may be conditional acceptance of the system. The customer may accept the system so that deployment can begin. The system provider agrees to repair urgent problems and deliver a new version to the customer as quickly as possible.

In agile methods such as Extreme Programming, there may be no separate acceptance testing activity. The end-user is part of the development team (i.e., he or she is an alpha tester) and provides the system requirements in terms of user stories. He or she is also responsible for defining the tests, which decide whether or not the developed software supports the user stories. These tests are therefore equivalent to acceptance tests. The tests are automated, and development does not proceed until the story acceptance tests have successfully been executed.

When users are embedded in a software development team, they should ideally be "typical" users with general knowledge of how the system will be used. However, it can be difficult to find such users, and so the acceptance tests may actually not be a true reflection of how a system is used in practice. Furthermore, the requirement for automated testing limits the flexibility of testing interactive systems. For such systems, acceptance testing may require groups of end-users to use the system as if it was part of their everyday work. Therefore, while an "embedded user" is an attractive notion in principle, it does not necessarily lead to high-quality tests of the system.

The problem of user involvement in agile teams is one reason why many companies use a mix of agile and more traditional testing. The system may be developed using agile techniques, but, after completion of a major release, separate acceptance testing is used to decide if the system should be accepted.



9 Software evolution

Objectives

The objectives of this chapter are to explain why software evolution is such an important part of software engineering and to describe the challenges of maintaining a large base of software systems, developed over many years. When you have read this chapter, you will:

- understand that software systems have to adapt and evolve if they are to remain useful and that software change and evolution should be considered as an integral part of software engineering;
- understand what is meant by legacy systems and why these systems are important to businesses;
- understand how legacy systems can be assessed to decide whether they should be scrapped, maintained, reengineered, or replaced;
- have learned about different types of software maintenance and the factors that affect the costs of making changes to legacy software systems.

Contents

Large software systems usually have a long lifetime. For example, 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. Enterprise software costs a lot of money, so a company has to use a software system for many years to get a return on its investment. Successful software products and apps may have been introduced many years ago with new versions released every few years. For example, the first version of Microsoft Word was introduced in 1983, so it has been around for more than 30 years.

During their lifetime, operational software systems have to change if they are to emain useful. Business changes and changes to user expectations generate new requirements for the 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 products and apps have to evolve to cope with platform changes and new features introduced by their competitors. Software systems, therefore, adapt and evolve during their lifetime from initial deployment to final retirement.

Businesses have to change their software to ensure that they continue to get value from it. Their systems are critical business assets, and they have to invest in change to maintain the value of these assets. Consequently, most large companies spend more on maintaining existing systems than on new systems development. Historical data suggests that somewhere between 60% and 90% of software costs are evolution costs (Lientz and Swanson 1980; Erlikh 2000). Jones (Jones 2006) found that about 75% of development staff in the United States in 2006 were involved in software evolution and suggested that this percentage was unlikely to fall in the foreseeable future.

Software evolution is particularly expensive in enterprise systems when individual software systems are part of a broader "system of systems." In such cases, you cannot just consider the changes to one system; you also need to examine how these changes affect the broader system of systems. Changing one system may mean that other systems in its environment may also have to evolve to cope with that change.

Therefore, as well as understanding and analyzing the impact of a proposed change on the system itself, you also have to assess how this change may affect other systems in the operational environment. Hopkins and Jenkins (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 other software systems.

The requirements of installed software systems change as the business and its environment change, so new releases of the systems that incorporate changes and updates are usually created at regular intervals. Software engineering is therefore 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 later releases of the software may start development before the current version has even been released.

In the last 10 years, the time between iterations of the spiral has reduced dramatically. Before the widespread use of the Internet, new versions of a software system

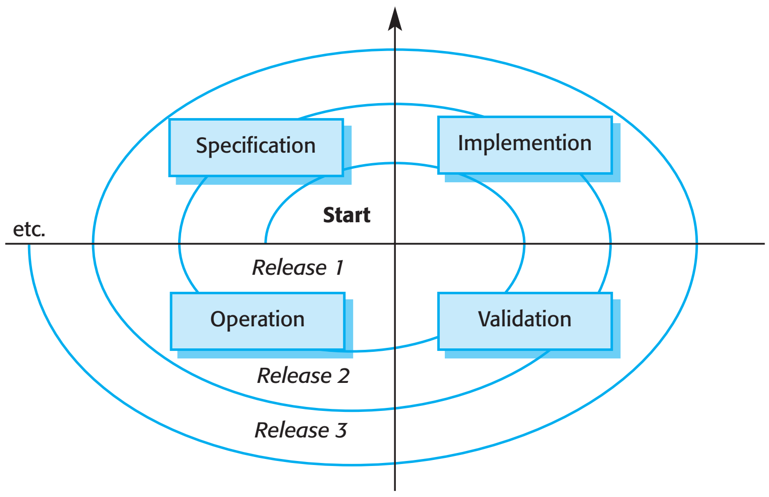


Figure 9.1 A spiral model of development and evolution

may only have been released every 2 or 3 years. Now, because of competitive pressures and the need to respond quickly to user feedback, the gap between releases of some apps and web-based systems may be weeks rather than years.

This model of software evolution is applicable when the same company is responsible for the software throughout its lifetime. There is a seamless transition from development to evolution, and the same software development methods and processes are applied throughout the lifetime of the software. Software products and apps are developed using this approach.

The evolution of custom software, however, usually follows a different model. The system customer may pay a software company to develop the software and then take over responsibility for support and evolution using its own staff. Alternatively, the software customer might issue a separate contract to a different software company for system support and evolution.

In this situation, there are likely to be discontinuities in the evolution process. Requirements and design documents may not be passed from one company to another. Companies may merge or reorganize, 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 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 (Rajlich and Bennett 2000) propose an alternative view of the software evolution life cycle for business systems. In this model, they distinguish between evolution and servicing. Evolution is the phase in which significant changes to the software architecture and functionality are made. During servicing, the only changes that are made are relatively small but essential changes. These phases overlap with each other, as shown in Figure 9.2.

According to Rajlich and Bennett, when software is first used successfully, many changes to the requirements by stakeholders are proposed and implemented. This is

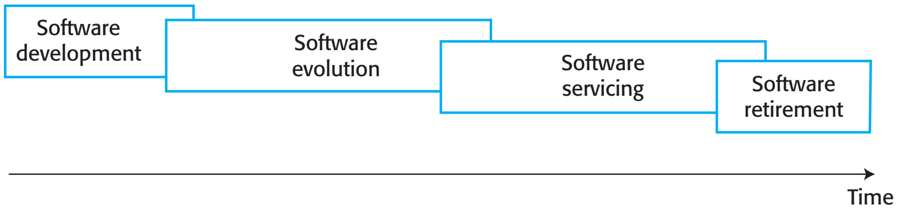


Figure 9.2 Evolution and servicing

the evolution phase. However, as the software is modified, its structure tends to degrade, and system 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 required. At some stage in the life cycle, the software reaches a transition point where significant changes and the implementation of new requirements become less and less cost-effective. At this stage, the software moves from evolution to servicing.

During the servicing phase, the software is still useful, but only small tactical changes are made to it. During this stage, the company is usually considering how the software can be replaced. In the final stage, the software may still be used, but only essential changes are made. Users have to work around problems that they discover. Eventually, the software is retired and taken out of use. This often incurs further costs as data is transferred from an old system to a newer replacement system.

9.1 Evolution processes

As with all software processes, there is no such thing as a standard software change or evolution process. The most appropriate evolution process for a software system depends on the type of software being maintained, the software development processes used in an organization, and the skills of the people involved. For some types of system, such as mobile apps, evolution may be an informal process, where change requests mostly come from conversations between system users and developers. For other types of systems, such as embedded critical systems, software evolution may be formalized, with structured documentation produced at each stage in the process.

Formal or informal system change proposals are the driver for system evolution in all organizations. In a change proposal, an individual or group suggests changes and updates to an existing software system. These proposals may be based on existing requirements that have not been implemented in the released system, requests for new requirements, bug reports from system stakeholders, and new ideas for software improvement from the system development team. The processes of change identification and system evolution are cyclical and continue throughout the lifetime of a system (Figure 9.3).

Before a change proposal is accepted, there needs to be an analysis of the software to work out which components need to be changed. This analysis allows the cost and the impact of the change to be assessed. This is part of the general process of change management, which should also ensure that the correct versions of

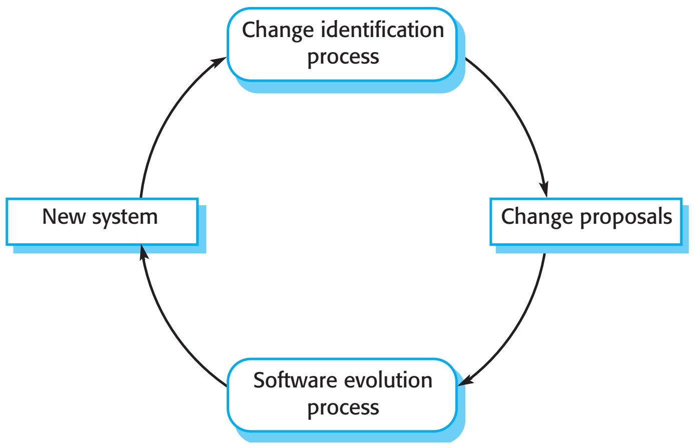


Figure 9.3 Change identification and evolution processes

components are included in each system release. I discuss change and configuration management in Chapter 25.

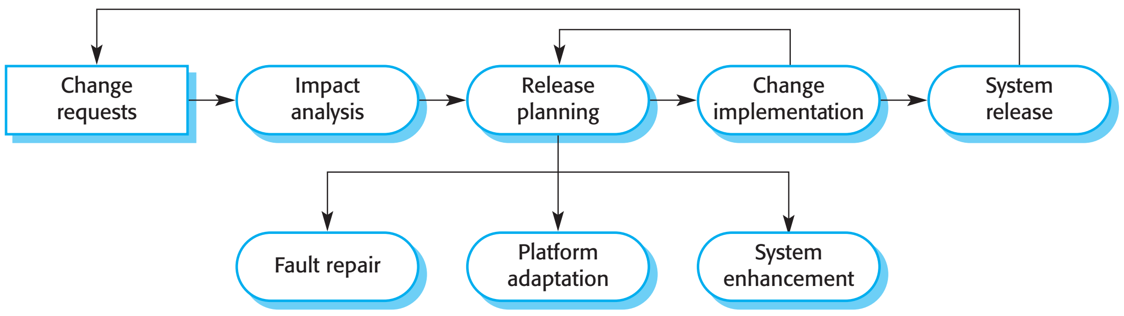
Figure 9.4 shows some of the activities involved in software evolution. The process includes the fundamental activities of change analysis, release planning, system implementation, and releasing a system to customers. The cost and impact of these changes are assessed to see how much of the system is affected by the change and how much it might cost to implement the change.

If the proposed changes are accepted, a new release of the system is planned. During release planning, all proposed changes (fault repair, adaptation, and new functionality) are considered. A decision is then made on which changes to implement in the next version of the system. The changes are implemented and validated, and a new version of the system is released. The process then iterates with a new set of changes proposed for the next release.

In situations where development and evolution are integrated, change implementation is simply an iteration of the development process. Revisions to the system are designed, implemented, and tested. The only difference between initial development and evolution is that customer feedback after delivery has to be considered when planning new releases of an application.

Where different teams are involved, a critical difference between development and evolution is that the first stage of change implementation requires program understanding.

Figure 9.4 A general model of the software evolution process



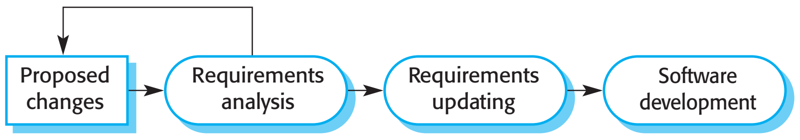


Figure 9.5 Change implementation

During the program understanding phase, new developers have to understand how the program is structured, how it delivers functionality, and how the proposed change might affect the program. They need this understanding to make sure that the implemented change does not cause new problems when it is introduced into the existing system.

If requirements specification and design documents are available, these should be updated during the evolution process to reflect the changes that are required (Figure 9.5). New software requirements should be written, and these should be analyzed and validated. If the design has been documented using UML models, these models should be updated. The proposed changes may be prototyped as part of the change analysis process, where you assess the implications and costs of making the change.

However, change requests sometimes relate to problems in operational systems that have to be tackled urgently. These urgent changes can arise for three reasons:

1. If a serious system fault is detected that has to be repaired to allow normal operation to continue or to address a serious security vulnerability.

2. If changes to the systems operating environment have unexpected effects that disrupt normal operation.

3. If there are unanticipated changes to the business running the system, such as the emergence of new competitors or the introduction of new legislation that affects the system.

In these cases, the need to make the change quickly means that you may not be able to update all of the software documentation. Rather than modify the requirements and design, you make an emergency fix to the program to solve the immediate problem (Figure 9.6). The danger here is that the requirements, the software design, and the code can become inconsistent. While you may intend to document the change in the requirements and design, additional emergency fixes to the software may then be needed. These take priority over documentation. Eventually, the original change is forgotten, and the system documentation and code are never realigned. This problem of maintaining multiple representations of a system is one of the arguments for minimal documentation, which is fundamental to agile development processes.

Emergency system repairs have to be completed as quickly as possible. You choose a quick and workable solution rather than the best solution as far as system structure is concerned. This tends to accelerate the process of software ageing so that future changes become progressively more difficult and maintenance costs increase. Ideally, after emergency code repairs are made, the new code should be refactored

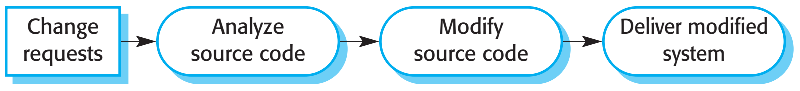


Figure 9.6 The emergency repair process

and improved to avoid program degradation. Of course, the code of the repair may be reused if possible. However, an alternative, better solution to the problem may be discovered when more time is available for analysis.

Agile methods and processes, discussed in Chapter 3, may be used for program evolution as well as program development. Because these methods are based on incremental development, making the transition from agile development to postdelivery evolution should be seamless.

However, problems may arise during the handover from a development team to a separate team responsible for system evolution. There are two potentially problematic situations:

1. Where the development team has used an agile approach but the evolution team prefers a plan-based approach. The evolution team may expect detailed documentation to support evolution, and this is rarely produced in agile processes. There may be no definitive statement of the system requirements that can be modified as changes are made to the system.

2. Where a plan-based approach has been used for development but the evolution team prefers to use agile methods. In this case, the evolution team may have to start from scratch developing automated tests. The code in the system may not have been refactored and simplified, as is expected in agile development. In this case, some program reengineering may be required to improve the code before it can be used in an agile development process.

Agile techniques such as test-driven development and automated regression testing are useful when system changes are made. System changes may be expressed as user stories, and customer involvement can help prioritize changes that are required in an operational system. The Scrum approach of focusing on a backlog of work to be done can help prioritize the most important system changes. In short, evolution simply involves continuing the agile development process.

Agile methods used in development may, however, have to be modified when they are used for program maintenance and evolution. It may be practically impossible to involve users in the development team as change proposals come from a wide range of stakeholders. Short development cycles may have to be interrupted to deal with emergency repairs, and the gap between releases may have to be lengthened to avoid disrupting operational processes.

9.2 Legacy systems

Large companies started computerizing their operations in the 1960s, so for the past 50 years or so, more and more software systems have been introduced. Many of these systems have been replaced (sometimes several times) as the business has changed and evolved. However, a lot of old systems are still in use and play a critical part in the running of the business. These older software systems are sometimes called legacy systems.

Legacy systems are older systems that rely on languages and technology that are no longer used for new systems development. Typically, they have been maintained over a long period, and their structure may have been degraded by the changes that have been made. Legacy software may be dependent on older hardware, such as mainframe computers and may have associated legacy processes and procedures. It may be impossible to change to more effective business processes because the legacy software cannot be modified to support new processes.

Legacy systems are not just software systems but are broader sociotechnical systems that include hardware, software, libraries, and other supporting software and business processes. Figure 9.7 shows the logical parts of a legacy system and their relationships.

1. *System hardware* Legacy systems may have been written for hardware that is no longer available, that is expensive to maintain, and that may not be compatible with current organizational IT purchasing policies.

2. *Support software* The legacy system may rely on a range of support software from the operating system and utilities provided by the hardware manufacturer through to the compilers used for system development. Again, these may be obsolete and no longer supported by their original providers.

3. *Application software* The application system that provides the business services is usually made up of a number of application programs that have been developed at different times. Some of these programs will also be part of other application software systems.

4. *Application data* These data are processed by the application system. In many legacy systems, an immense volume of data has accumulated over the lifetime of the system. This data may be inconsistent, may be duplicated in several files, and may be spread over a number of different databases.

5. *Business processes* These processes are used in the business to achieve some business objective. An example of a business process in an insurance company would be issuing an insurance policy; in a manufacturing company, a business process would be accepting an order for products and setting up the associated manufacturing process. Business processes may be designed around a legacy system and constrained by the functionality that it provides.

6. *Business policies and rules* These are definitions of how the business should be carried out and constraints on the business. Use of the legacy application system may be embedded in these policies and rules.

An alternative way of looking at these components of a legacy system is as a series of layers, as shown in Figure 9.8.

Each layer depends on the layer immediately below it and interfaces with that layer. If interfaces are maintained, then you should be able to make changes within a layer without affecting either of the adjacent layers. In practice, however, this simple encapsulation is an oversimplification, and changes to one layer of the system may

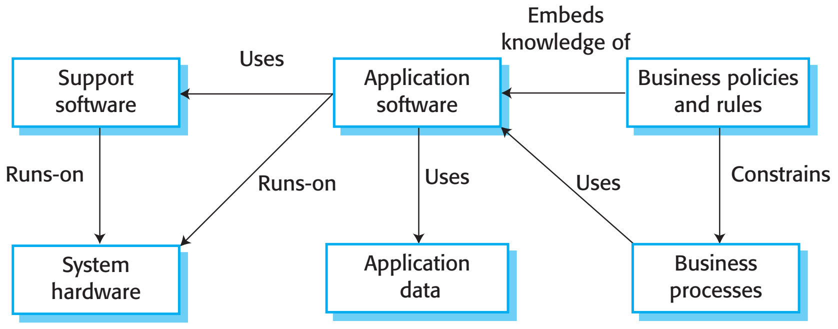


Figure 9.7 The elements of a legacy system

require consequent changes to layers that are both above and below the changed level. The reasons for this are as follows:

1. Changing one layer in the system may introduce new facilities, and higher layers in the system may then be changed to take advantage of these facilities. For example, a new database introduced at the support software layer may include facilities to access the data through a web browser, and business processes may be modified to take advantage of this facility.
2. Changing the software may slow the system down so that new hardware is needed to improve the system performance. The increase in performance from the new hardware may then mean that further software changes that were previously impractical become possible.
3. It is often impossible to maintain hardware interfaces, especially if new hardware is introduced. This is a particular problem in embedded systems where there is a tight coupling between software and hardware. Major changes to the application software may be required to make effective use of the new hardware.

It is difficult to know exactly how much legacy code is still in use, but, as an indicator, industry has estimated that there are more than 200 billion lines of COBOL code in current business systems. COBOL is a programming language designed for writing business systems, and it was the main business development language from the 1960s to the 1990s, particularly in the finance industry (Mitchell 2012). These programs still work effectively and efficiently, and the companies using them see no need to change them. A major problem that they face, however, is a shortage of COBOL programmers as the original developers of the system retire. Universities no longer teach COBOL, and younger software engineers are more interested in programming in modern languages.

Skill shortages are only one of the problems of maintaining business legacy systems. Other issues include security vulnerabilities because these systems were developed before the widespread use of the Internet and problems in interfacing with systems written in modern programming languages. The original software tool supplier may be out of business or may no longer maintain the support tools used to

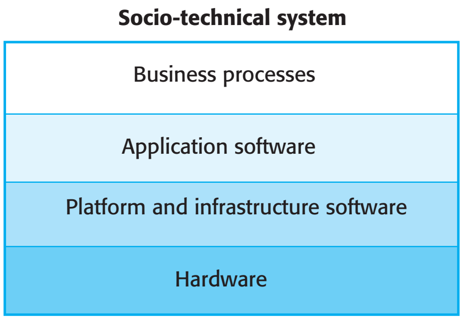


Figure 9.8 Legacy system layers

develop the system. The system hardware may be obsolete and so increasingly expensive to maintain.

Why then do businesses not simply replace these systems with more modern equivalents? The simple answer to this question is that it is too expensive and too risky to do so. If a legacy system works effectively, the costs of replacement may exceed the savings that come from the reduced support costs of a new system. Scrapping legacy systems and replacing them with more modern software open up the possibility of things going wrong and the new system failing to meet the needs of the business. Managers try to minimize those risks and therefore do not want to face the uncertainties of new software systems.

I discovered some of the problems of legacy system replacement when I was involved in analyzing a legacy system replacement project in a large organization. This enterprise used more than 150 legacy systems to run its business. It decided to replace all of these systems with a single, centrally maintained ERP system. For a number of business and technology reasons, the new system development was a failure, and it did not deliver the improvements promised. After spending more than £10 million, only a part of the new system was operational, and it worked less effectively than the systems it replaced. Users continued to use the older systems but could not integrate these with the part of the new system that had been implemented, so additional manual processing was required.

There are several reasons why it is expensive and risky to replace legacy systems with new systems:

1. There is rarely a complete specification of the legacy system. The original specification may have been lost. If a specification exists, it is unlikely that it has been updated with all of the system changes that have been made. Therefore, there is no straightforward way of specifying a new system that is functionally identical to the system that is in use.

2. Business processes and the ways in which legacy systems operate are often inextricably intertwined. These processes are likely to have evolved to take advantage of the software's services and to work around the software's shortcomings. If the system is replaced, these processes have to change with potentially unpredictable costs and consequences.

3. Important business rules may be embedded in the software and may not be documented elsewhere. A business rule is a constraint that applies to some business function, and breaking that constraint can have unpredictable consequences for the business. For example, an insurance company may have embedded its rules for assessing the risk of a policy application in its software. If these rules are not maintained, the company may accept high-risk policies that could result in expensive future claims.
4. New software development is inherently risky, so that there may be unexpected problems with a new system. It may not be delivered on time and for the price expected.

Keeping legacy systems in use avoids the risks of replacement, but making changes to existing software inevitably becomes more expensive as systems get older. Legacy software systems that are more than a few years old are particularly expensive to change:

1. The program style and usage conventions are inconsistent because different people have been responsible for system changes. This problem adds to the difficulty of understanding the system code.
2. Part or all of the system may be implemented using obsolete programming languages. It may be difficult to find people who have knowledge of these languages. Expensive outsourcing of system maintenance may therefore be required.
3. System documentation is often inadequate and out of date. In some cases, the only documentation is the system source code.
4. Many years of maintenance usually degrades the system structure, making it increasingly difficult to understand. New programs may have been added and interfaced with other parts of the system in an ad hoc way.
5. The system may have been optimized for space utilization or execution speed so that it runs effectively on older slower hardware. This normally involves using specific machine and language optimizations, and these usually lead to software that is hard to understand. This causes problems for programmers who have learned modern software engineering techniques and who don’t understand the programming tricks that have been used to optimize the software.
6. The data processed by the system may be maintained in different files that have incompatible structures. There may be data duplication, and the data itself may be out of date, inaccurate, and incomplete. Several databases from different suppliers may be used.

At same stage, the costs of managing and maintaining the legacy system become so high that it has to be replaced with a new system. In the next section, I discuss a systematic decision-making approach to making such a replacement decision.

9.2.1 Legacy system management

For new software systems developed using modern software engineering processes, such as agile development and software product lines, it is possible to plan how to integrate system development and evolution. More and more companies understand that the system development process is a whole life-cycle process. Separating software development and software evolution is unhelpful and leads to higher costs. However, as I have discussed, there is still a huge number of legacy systems that are critical business systems. These have to be extended and adapted to changing e-business practices.

Most organizations have a limited budget for maintaining and upgrading their portfolio of legacy systems. They have to decide how to get the best return on their investment. This involves making a realistic assessment of their legacy systems and then deciding on the most appropriate strategy for evolving these systems. There are four strategic options:

1. *Scrap the system completely* This option should be chosen when the system is not making an effective contribution to business processes. This usually occurs when business processes have changed since the system was installed and are no longer reliant on the legacy system.

2. *Leave the system unchanged and continue with regular maintenance* This option should be chosen when the system is still required but is fairly stable and the system users make relatively few change requests.

3. *Reengineer the system to improve its maintainability* This option should be chosen when the system quality has been degraded by change and where new change to the system is still being proposed. This process may include developing new interface components so that the original system can work with other, newer systems.

4. *Replace all or part of the system with a new system* This option should be chosen when factors, such as new hardware, mean that the old system cannot continue in operation, or where off-the-shelf systems would allow the new system to be developed at a reasonable cost. In many cases, an evolutionary replacement strategy can be adopted where major system components are replaced by off-the-shelf systems with other components reused wherever possible.

When you are assessing a legacy system, you have to look at it from both a business perspective and a technical perspective (Warren 1998). From a business perspective, you have to decide whether or not the business really needs the system. From a technical perspective, you have to assess the quality of the application software and the system's support software and hardware. You then use a combination of the business value and the system quality to inform your decision on what to do with the legacy system.

For example, assume that an organization has 10 legacy systems. You should assess the quality and the business value of each of these systems. You may then create a chart showing relative business value and system quality. An example of

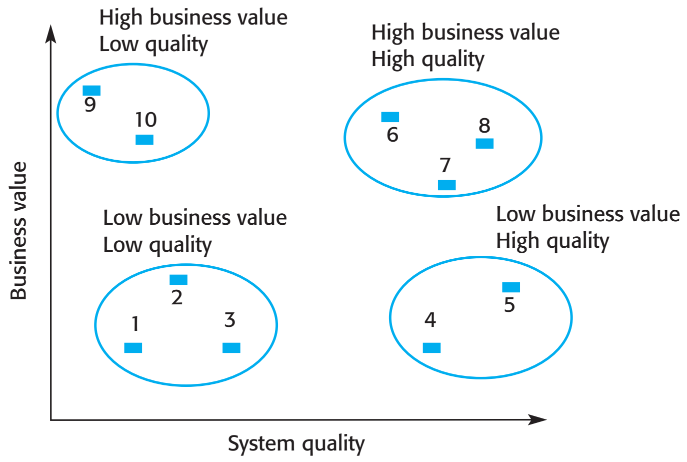


Figure 9.9 An example of a legacy system assessment

this is shown in Figure 9.9. From this diagram, you can see that there are four clusters of systems:

1. *Low quality, low business value* Keeping these systems in operation will be expensive, and the rate of the return to the business will be fairly small. These systems should be scrapped.
2. *Low quality, high business value* These systems are making an important business contribution, so they cannot be scrapped. However, their low quality means that they are expensive to maintain. These systems should be reengineered to improve their quality. They may be replaced, if suitable off-the-shelf systems are available.
3. *High quality, low business value* These systems don't contribute much to the business but may not be very expensive to maintain. It is not worth replacing these systems, so normal system maintenance may be continued if expensive changes are not required and the system hardware remains in use. If expensive changes become necessary, the software should be scrapped.
4. *High quality, high business value* These systems have to be kept in operation. However, their high quality means that you don't have to invest in transformation or system replacement. Normal system maintenance should be continued.

The business value of a system is a measure of how much time and effort the system saves compared to manual processes or the use of other systems. To assess the business value of a system, you have to identify system stakeholders, such as the end-users of a system and their managers, and ask a series of questions about the system. There are four basic issues that you have to discuss:

1. *The use of the system* If a system is only used occasionally or by a small number of people, this may mean that it has a low business value. A legacy system may have been developed to meet a business need that has either changed or can now be met

more effectively in other ways. You have to be careful, however, about occasional but important use of systems. For example, a university system for student registration may only be used at the beginning of each academic year. Although it is used infrequently, it is an essential system with a high business value.

2. *The business processes that are supported* When a system is introduced, business processes are usually introduced to exploit the system's capabilities. If the system is inflexible, changing these business processes may be impossible. However, as the environment changes, the original business processes may become obsolete. Therefore, a system may have a low business value because it forces the use of inefficient business processes.

3. *System dependability* System dependability is not only a technical problem but also a business problem. If a system is not dependable and the problems directly affect business customers, or mean that people in the business are diverted from other tasks to solve these problems, the system has a low business value.

4. *The system outputs* The key issue here is the importance of the system outputs to the successful functioning of the business. If the business depends on these outputs, then the system has a high business value. Conversely, if these outputs can be cheaply generated in some other way, or if the system produces outputs that are rarely used, then the system has a low business value.

For example, assume that a company provides a travel ordering system that is used by staff responsible for arranging travel. They can place orders with an approved travel agent. Tickets are then delivered, and the company is invoiced for them. However, a business value assessment may reveal that this system is only used for a fairly small percentage of travel orders placed. People making travel arrangements find it cheaper and more convenient to deal directly with travel suppliers through their websites. This system may still be used, but there is no real point in keeping it—the same functionality is available from external systems.

Conversely, say a company has developed a system that keeps track of all previous customer orders and automatically generates reminders for customers to reorder goods. This results in a large number of repeat orders and keeps customers satisfied because they feel that their supplier is aware of their needs. The outputs from such a system are important to the business, so this system has a high business value.

To assess a software system from a technical perspective, you need to consider both the application system itself and the environment in which the system operates. The environment includes the hardware and all associated support software such as compilers, debuggers and development environments that are needed to maintain the system. The environment is important because many system changes, such as upgrades to the hardware or operating system, result from changes to the environment.

Factors that you should consider during the environment assessment are shown in Figure 9.10. Notice that these are not all technical characteristics of the environment. You also have to consider the reliability of the suppliers of the hardware and support software. If suppliers are no longer in business, their systems may not be supported, so you may have to replace these systems.

Factor	Questions
Supplier stability	Is the supplier still in existence? Is the supplier financially stable and likely to continue in existence? If the supplier is no longer in business, does someone else maintain the systems?
Failure rate	Does the hardware have a high rate of reported failures? Does the support software crash and force system restarts?
Age	How old is the hardware and software? The older the hardware and support software, the more obsolete it will be. It may still function correctly, but there could be significant economic and business benefits to moving to a more modern system.
Performance	Is the performance of the system adequate? Do performance problems have a significant effect on system users?
Support requirements	What local support is required by the hardware and software? If high costs are associated with this support, it may be worth considering system replacement.
Maintenance costs	What are the costs of hardware maintenance and support software licences? Older hardware may have higher maintenance costs than modern systems. Support software may have high annual licensing costs.
Interoperability	Are there problems interfacing the system to other systems? Can compilers, for example, be used with current versions of the operating system?

Figure 9.10 Factors used in environment assessment

In the process of environmental assessment, if possible, you should ideally collect data about the system and system changes. Examples of data that may be useful include the costs of maintaining the system hardware and support software, the number of hardware faults that occur over some time period and the frequency of patches and fixes applied to the system support software.

To assess the technical quality of an application system, you have to assess those factors (Figure 9.11) that are primarily related to the system dependability, the difficulties of maintaining the system, and the system documentation. You may also collect data that will help you judge the quality of the system such as:

1. *The number of system change requests* System changes usually corrupt the system structure and make further changes more difficult. The higher this accumulated value, the lower the quality of the system.

2. *The number of user interfaces* This is an important factor in forms-based systems where each form can be considered as a separate user interface. The more interfaces, the more likely it is that there will be inconsistencies and redundancies in these interfaces.

3. *The volume of data used by the system* As the volume of data (number of files, size of database, etc.) processed by the system increases, so too do the inconsistencies and errors in that data. When data has been collected over a long period of time, errors and inconsistencies are inevitable. Cleaning up old data is a very expensive and time-consuming process.

Factor	Questions
Understandability	How difficult is it to understand the source code of the current system? How complex are the control structures that are used? Do variables have meaningful names that reflect their function?
Documentation	What system documentation is available? Is the documentation complete, consistent, and current?
Data	Is there an explicit data model for the system? To what extent is data duplicated across files? Is the data used by the system up to date and consistent?
Performance	Is the performance of the application adequate? Do performance problems have a significant effect on system users?
Programming language	Are modern compilers available for the programming language used to develop the system? Is the programming language still used for new system development?
Configuration management	Are all versions of all parts of the system managed by a configuration management system? Is there an explicit description of the versions of components that are used in the current system?
Test data	Does test data for the system exist? Is there a record of regression tests carried out when new features have been added to the system?
Personnel skills	Are there people available who have the skills to maintain the application? Are there people available who have experience with the system?

Figure 9.11 Factors used in application assessment

Ideally, objective assessment should be used to inform decisions about what to do with a legacy system. However, in many cases, decisions are not really objective but are based on organizational or political considerations. For example, if two businesses merge, the most politically powerful partner will usually keep its systems and scrap the other company's systems. If senior management in an organization decides to move to a new hardware platform, then this may require applications to be replaced. If no budget is available for system transformation in a particular year, then system maintenance may be continued, even though this will result in higher long-term costs.

9.3 Software maintenance

Software maintenance is the general process of changing a system after it has been delivered. The term is usually applied to custom software, where separate development groups are involved before and after delivery. The changes made to the software may be simple changes to correct coding errors, more extensive changes to correct design errors, or significant enhancements to correct specification errors or to accommodate new requirements. Changes are implemented by modifying existing system components and, where necessary, by adding new components to the system.

Program evolution dynamics

Program evolution dynamics is the study of evolving software systems, pioneered by Manny Lehman and Les Belady in the 1970s. This led to so-called Lehman's Laws, which are said to apply to all large-scale software systems. The most important of these laws are:

1. A program must continually change if it is to remain useful.
2. As an evolving program changes, its structure is degraded.
3. Over a program's lifetime, the rate of change is roughly constant and independent of the resources available.
4. The incremental change in each release of a system is roughly constant.
5. New functionality must be added to systems to increase user satisfaction.

http://software-engineering-book.com/web/program-evolution-dynamics/

There are three different types of software maintenance:

1. *Fault repairs to fix bugs and vulnerabilities.* Coding errors are usually relatively cheap to correct; design errors are more expensive because they may involve rewriting several program components. Requirements errors are the most expensive to repair because extensive system redesign may be necessary.

2. *Environmental adaptation to adapt the software to new platforms and environments.* This type of maintenance is required when some aspect of a system's environment, such as the hardware, the platform operating system, or other support software, changes. Application systems may have to be modified to cope with these environmental changes.

3. *Functionality addition to add new features and to support new requirements.* This type of maintenance is necessary when 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 no clear-cut distinction between these types of maintenance. When you adapt a 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 these terms in this book.

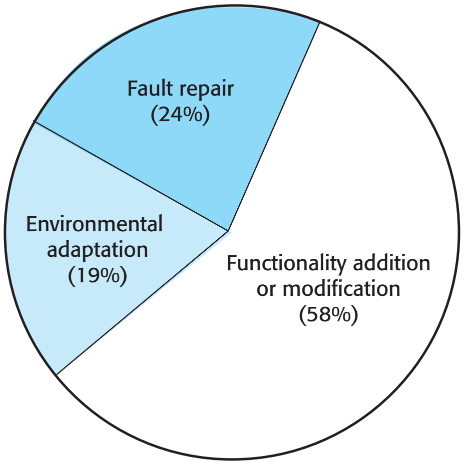


Figure 9.12
Maintenance effort distribution

Figure 9.12 shows an approximate distribution of maintenance costs, based on data from the most recent survey available (Davidsen and Krogstie 2010). This study compared maintenance cost distribution with a number of earlier studies from 1980 to 2005. The authors found that the distribution of maintenance costs had changed very little over 30 years. Although we don't have more recent data, this suggests that this distribution is still largely correct. Repairing system faults is not the most expensive maintenance activity. Evolving the system to cope with new environments and new or changed requirements generally consumes most maintenance effort.

Experience has shown that it is usually more expensive to add new features to a system during maintenance than it is to implement the same features during initial development. The reasons for this are:

1. *A new team has to understand the program being maintained.* After a system has been delivered, it is normal for the development team to be broken up and for people to work on new projects. The new team or the individuals responsible for system maintenance do not understand the system or the background to system design decisions. They need to spend time understanding the existing system before they can implement changes to it.

2. *Separating maintenance and development means there is no incentive for the development team to write maintainable software.* The contract to maintain a system is usually separate from the system development contract. A different company, rather than the original software developer, may be responsible for software maintenance. In those circumstances, a development team gets no benefit from investing effort to make the software maintainable. If a development team can cut corners to save effort during development it is worthwhile for them to do so, even if this means that the software is more difficult to change in future.

3. *Program maintenance work is unpopular.* Maintenance has a poor image among software engineers. It is seen as a less skilled process than system development

Documentation

System documentation can help the maintenance process by providing maintainers with information about the structure and organization of the system and the features that it offers to system users. While proponents of agile approaches suggest that the code should be the principal documentation, higher level design models and information about dependencies and constraints can make it easier to understand and make changes to that code.

http://software-engineering-book.com/web/documentation/ (web chapter)

and is often allocated to the least experienced staff. Furthermore, old systems may be written in obsolete programming languages. The developers working on maintenance may not have much experience of these languages and must learn these languages to maintain the system.

4. *As programs age, their structure degrades and they become harder to change.* As changes are made to programs, their structure tends to degrade. Consequently, they become harder to understand and change. Some systems have been developed without modern software engineering techniques. They may never have been well structured and were perhaps optimized for efficiency rather than understandability. System documentation may be lost or inconsistent. Old systems may not have been subject to stringent configuration management, so developers have to spend time finding the right versions of system components to change.

The first three of these problems stem from the fact that many organizations still consider software development and maintenance to be separate activities. Maintenance is seen as a second-class activity, and there is no incentive to spend money during development to reduce the costs of system change. The only long-term solution to this problem is to think of systems as evolving throughout their lifetime through a continual development process. Maintenance should have as high a status as new software development.

The fourth issue, the problem of degraded system structure, is, in some ways, the easiest problem to address. Software reengineering techniques (described later in this chapter) may be applied to improve the system structure and understandability. Architectural transformations can adapt the system to new hardware. Refactoring can improve the quality of the system code and make it easier to change.

In principle, it is almost always cost-effective to invest effort in designing and implementing a system to reduce the costs of future changes. Adding new functionality after delivery is expensive because you have to spend time learning the system and analyzing the impact of the proposed changes. Work done during development to structure the software and to make it easier to understand and change will reduce evolution costs. Good software engineering techniques such as precise specification, test-first development, the use of object-oriented development, and configuration management all help reduce maintenance cost.

These principled arguments for lifetime cost savings by investing in making systems more maintainable are, unfortunately, impossible to substantiate with real data. Collecting data is expensive, and the value of that data is difficult to judge; therefore, the vast majority of companies do not think it is worthwhile to gather and analyze software engineering data.

In reality, most businesses are reluctant to spend more on software development to reduce longer-term maintenance costs. There are two main reasons for their reluctance:

1. Companies set out quarterly or annual spending plans, and managers are incentivized to reduce short-term costs. Investing in maintainability leads to short-term cost increases, which are measurable. However, the long-term gains can't be measured at the same time, so companies are reluctant to spend money on something with an unknown future return.

2. Developers are not usually responsible for maintaining the system they have developed. Consequently, they don't see the point of doing additional work that might reduce maintenance costs, as they will not get any benefit from it.

The only way around this problem is to integrate development and maintenance so that the original development team remains responsible for software throughout its lifetime. This is possible for software products and for companies such as Amazon, which develop and maintain their own software (O'Hanlon 2006). However, for custom software developed by a software company for a client, this is unlikely to happen.

9.3.1 Maintenance prediction

Maintenance prediction is concerned with trying to assess the changes that may be required in a software system and with identifying those parts of the system that are likely to be the most expensive to change. If you understand this, you can design the software components that are most likely to change to make them more adaptable. You can also invest effort in improving those components to reduce their lifetime maintenance costs. By predicting changes, you can also assess the overall maintenance costs for a system in a given time period and so set a budget for maintaining the software. Figure 9.13 shows possible predictions and the questions that these predictions may answer.

Predicting the number of change requests for a system requires an understanding of the relationship between the system and its external environment. Some systems have a very complex relationship with their external environment, and changes to that environment inevitably result in changes to the system. To evaluate the relationships between a system and its environment, you should look at:

1. *The number and complexity of system interfaces* The larger the number of interfaces and the more complex these interfaces, the more likely it is that interface changes will be required as new requirements are proposed.

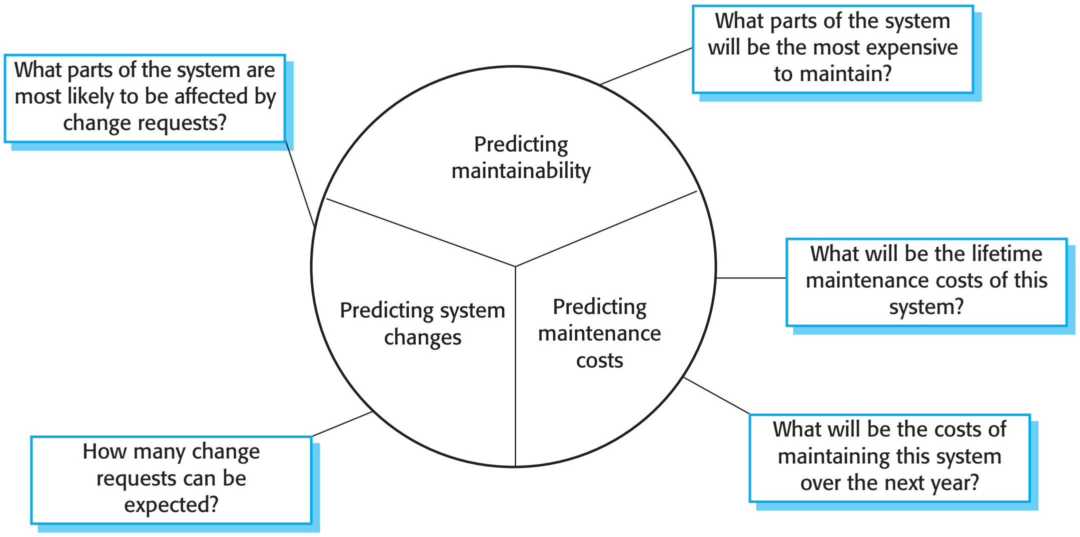

What parts of the system are most likely to be affected by change requests?
What parts of the system will be the most expensive to maintain?
Predicting maintainability
What will be the lifetime maintenance costs of this system?
Predicting system changes
Predicting maintenance costs
What will be the costs of maintaining this system over the next year?
How many change requests can be expected?

Figure 9.13 Maintenance prediction

2. *The number of inherently volatile system requirements* As I discussed in Chapter 4, requirements that reflect organizational policies and procedures are likely to be more volatile than requirements that are based on stable domain characteristics.

3. *The business processes in which the system is used* As business processes evolve, they generate system change requests. As a system is integrated with more and more business processes, there are increased demands for changes.

In early work on software maintenance, researchers looked at the relationships between program complexity and maintainability (Banker et al. 1993; Coleman et al. 1994; Kozlov et al. 2008). These studies found that the more complex a system or component, the more expensive it is to maintain. Complexity measurements are particularly useful in identifying program components that are likely to be expensive to maintain. Therefore, to reduce maintenance costs you should try to replace complex system components with simpler alternatives.

After a system has been put into service, you may be able to use process data to help predict maintainability. Examples of process metrics that can be used for assessing maintainability are:

1. *Number of requests for corrective maintenance* An increase in the number of bug and failure reports may indicate that more errors are being introduced into the program than are being repaired during the maintenance process. This may indicate a decline in maintainability.

2. *Average time required for impact analysis* This is related to the number of program components that are affected by the change request. If the time required for impact analysis increases, it implies that more and more components are affected and maintainability is decreasing.

3. *Average time taken to implement a change request* This is not the same as the time for impact analysis although it may correlate with it. This is the amount of time that you need to modify the system and its documentation, after you have assessed which components are affected. An increase in the time needed to implement a change may indicate a decline in maintainability.

4. *Number of outstanding change requests* An increase in this number over time may imply a decline in maintainability.

You use predicted information about change requests and predictions about system maintainability to predict maintenance costs. Most managers combine this information with intuition and experience to estimate costs. The COCOMO 2 model of cost estimation, discussed in Chapter 23, suggests that an estimate for software maintenance effort can be based on the effort to understand existing code and the effort to develop the new code.

9.3.2 Software reengineering

Software maintenance involves understanding the program that has to be changed and then implementing any required changes. However, many systems, especially older legacy systems, are difficult to understand and change. The programs may have been optimized for performance or space utilization at the expense of understandability, or, over time, the initial program structure may have been corrupted by a series of changes.

To make legacy software systems easier to maintain, you can reengineer these systems to improve their structure and understandability. Reengineering may involve redocumenting the system, refactoring the system architecture, translating programs to a modern programming language, or modifying and updating the structure and values of the system's data. The functionality of the software is not changed, and, normally, you should try to avoid making major changes to the system architecture.

Reengineering has two important advantages over replacement:

1. *Reduced risk* There is a high risk in redeveloping business-critical software. Errors may be made in the system specification or there may be development problems. Delays in introducing the new software may mean that business is lost and extra costs are incurred.

2. *Reduced cost* The cost of reengineering may be significantly less than the cost of developing new software. Ulrich (Ulrich 1990) quotes an example of a commercial system for which the reimplementation costs were estimated at $50 million. The system was successfully reengineered for $12 million. I suspect that, with modern software technology, the relative cost of reimplementation is probably less than Ulrich's figure but will still be more than the costs of reengineering.

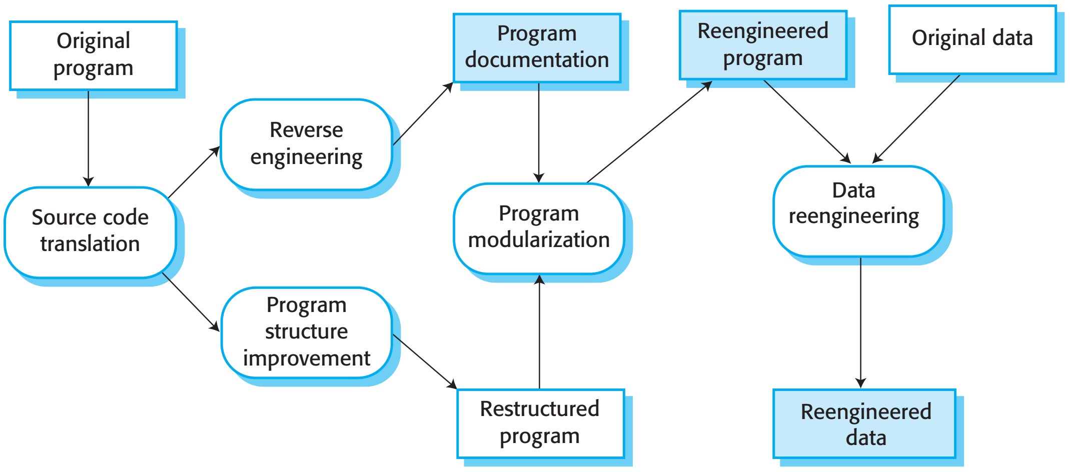


Figure 9.14 The reengineering process

Figure 9.14 is a general model of the reengineering process. The input to the process is a legacy program, and the output is an improved and restructured version of the same program. The activities in this reengineering process are:

1. *Source code translation* Using a translation tool, you can convert the program from an old programming language to a more modern version of the same language or to a different language.

2. *Reverse engineering* The program is analyzed and information extracted from it. This helps to document its organization and functionality. Again, this process is usually completely automated.

3. *Program structure improvement* The control structure of the program is analyzed and modified to make it easier to read and understand. This can be partially automated, but some manual intervention is usually required.

4. *Program modularization* Related parts of the program are grouped together, and, where appropriate, redundancy is removed. In some cases, this stage may involve architectural refactoring (e.g., a system that uses several different data stores may be refactored to use a single repository). This is a manual process.

5. *Data reengineering* The data processed by the program is changed to reflect program changes. This may mean redefining database schemas and converting existing databases to the new structure. You should usually also clean up the data. This involves finding and correcting mistakes, removing duplicate records, and so on. This can be a very expensive and prolonged process.

Program reengineering may not necessarily require all of the steps in Figure 9.11. You don't need source code translation if you still use the application's programming language. If you can do all reengineering automatically, then recovering documentation through reverse engineering may be unnecessary. Data reengineering is required only if the data structures in the program change during system reengineering.

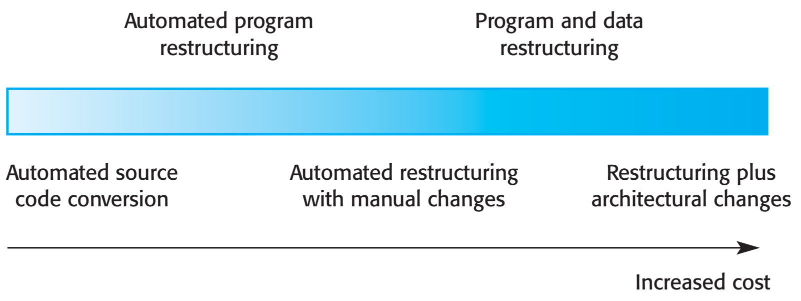

Automated program restructuring
Program and data restructuring
Automated source code conversion
Automated restructuring with manual changes
Restructuring plus architectural changes
Increased cost

Figure 9.15 Reengineering approaches

To make the reengineered system interoperate with the new software, you may have to develop adaptor services, as discussed in Chapter 18. These hide the original interfaces of the software system and present new, better-structured interfaces that can be used by other components. This process of legacy system wrapping is an important technique for developing large-scale reusable services.

The costs of reengineering obviously depend on the extent of the work that is carried out. There is a spectrum of possible approaches to reengineering, as shown in Figure 9.15. Costs increase from left to right so that source code translation is the cheapest option, and reengineering, as part of architectural migration, is the most expensive.

The problem with software reengineering is that there are practical limits to how much you can improve a system by reengineering. It isn't possible, for example, to convert a system written using a functional approach to an object-oriented system. Major architectural changes or radical reorganizing of the system data management cannot be carried out automatically, so they are very expensive. Although reengineering can improve maintainability, the reengineered system will probably not be as maintainable as a new system developed using modern software engineering methods.

9.3.3 Refactoring

Refactoring is the process of making improvements to a program to slow down degradation through change. It means modifying a program to improve its structure, reduce its complexity, or make it easier to understand. Refactoring is sometimes considered to be limited to object-oriented development, but the principles can in fact be applied to any development approach. When you refactor a program, you should not add functionality but rather should concentrate on program improvement. You can therefore think of refactoring as "preventative maintenance" that reduces the problems of future change.

Refactoring is an inherent part of agile methods because these methods are based on change. Program quality is liable to degrade quickly, so agile developers frequently refactor their programs to avoid this degradation. The emphasis on regression testing in agile methods lowers the risk of introducing new errors through refactoring. Any errors that are introduced should be detectable, as previously successful tests should then fail. However, refactoring is not dependent on other "agile activities."

Although reengineering and refactoring are both intended to make software easier to understand and change, they are not the same thing. Reengineering takes place after a system has been maintained for some time, and maintenance costs are increasing. You use automated tools to process and reengineer a legacy system to create a new system that is more maintainable. Refactoring is a continuous process of improvement throughout the development and evolution process. It is intended to avoid the structure and code degradation that increases the costs and difficulties of maintaining a system.

Fowler et al. (Fowler et al. 1999) suggest that there are stereotypical situations (Fowler calls them "bad smells") where the code of a program can be improved. Examples of bad smells that can be improved through refactoring include:

1. *Duplicate code* The same or very similar code may be included at different places in a program. This can be removed and implemented as a single method or function that is called as required.
2. *Long methods* If a method is too long, it should be redesigned as a number of shorter methods.
3. *Switch (case) statements* These often involve duplication, where the switch depends on the type of a value. The switch statements may be scattered around a program. In object-oriented languages, you can often use polymorphism to achieve the same thing.
4. *Data clumping* Data clumps occur when the same group of data items (fields in classes, parameters in methods) reoccurs in several places in a program. These can often be replaced with an object that encapsulates all of the data.
5. *Speculative generality* This occurs when developers include generality in a program in case it is required in the future. This can often simply be removed.

Fowler, in both his book and website, also suggests some primitive refactoring transformations that can be used singly or together to deal with bad smells. Examples of these transformations include Extract method, where you remove duplication and create a new method; Consolidate conditional expression, where you replace a sequence of tests with a single test; and Pull up method, where you replace similar methods in subclasses with a single method in a superclass. Interactive development environments, such as Eclipse, usually include refactoring support in their editors. This makes it easier to find dependent parts of a program that have to be changed to implement the refactoring.

Refactoring, carried out during program development, is an effective way to reduce the long-term maintenance costs of a program. However, if you take over a program for maintenance whose structure has been significantly degraded, then it may be practically impossible to refactor the code alone. You may also have to think about design refactoring, which is likely to be a more expensive and difficult problem. Design refactoring involves identifying relevant design patterns (discussed in Chapter 7) and replacing existing code with code that implements these design patterns (Kerievsky 2004).