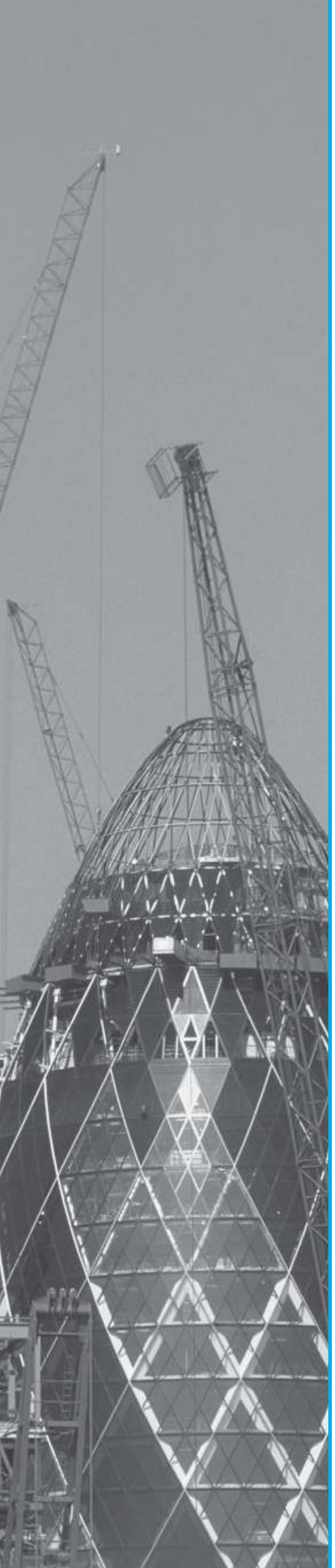


4 Requirements engineering

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,

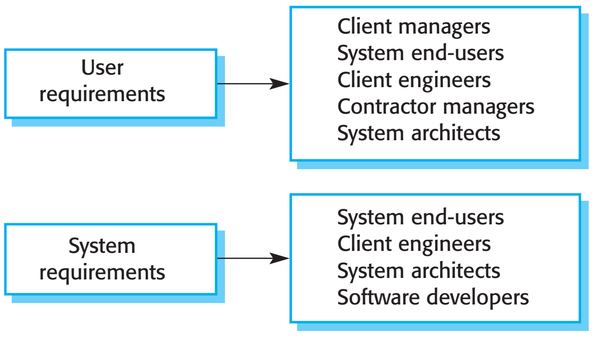


Figure 4.2 Readers of different types of requirements specification

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.

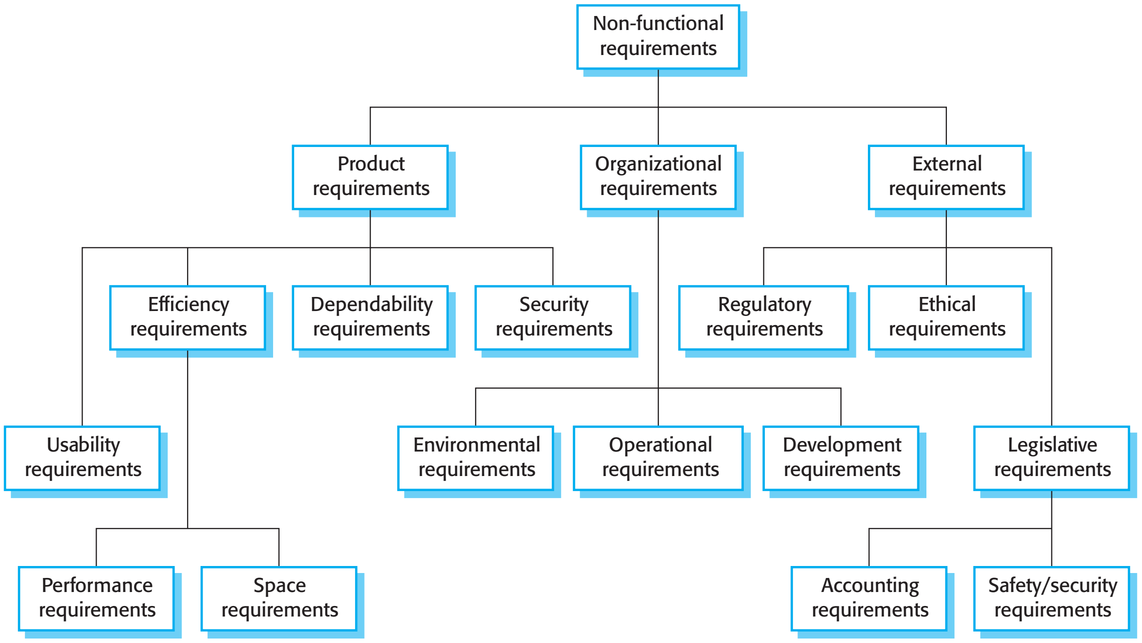


Figure 4.3 Types of non-functional requirements

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.

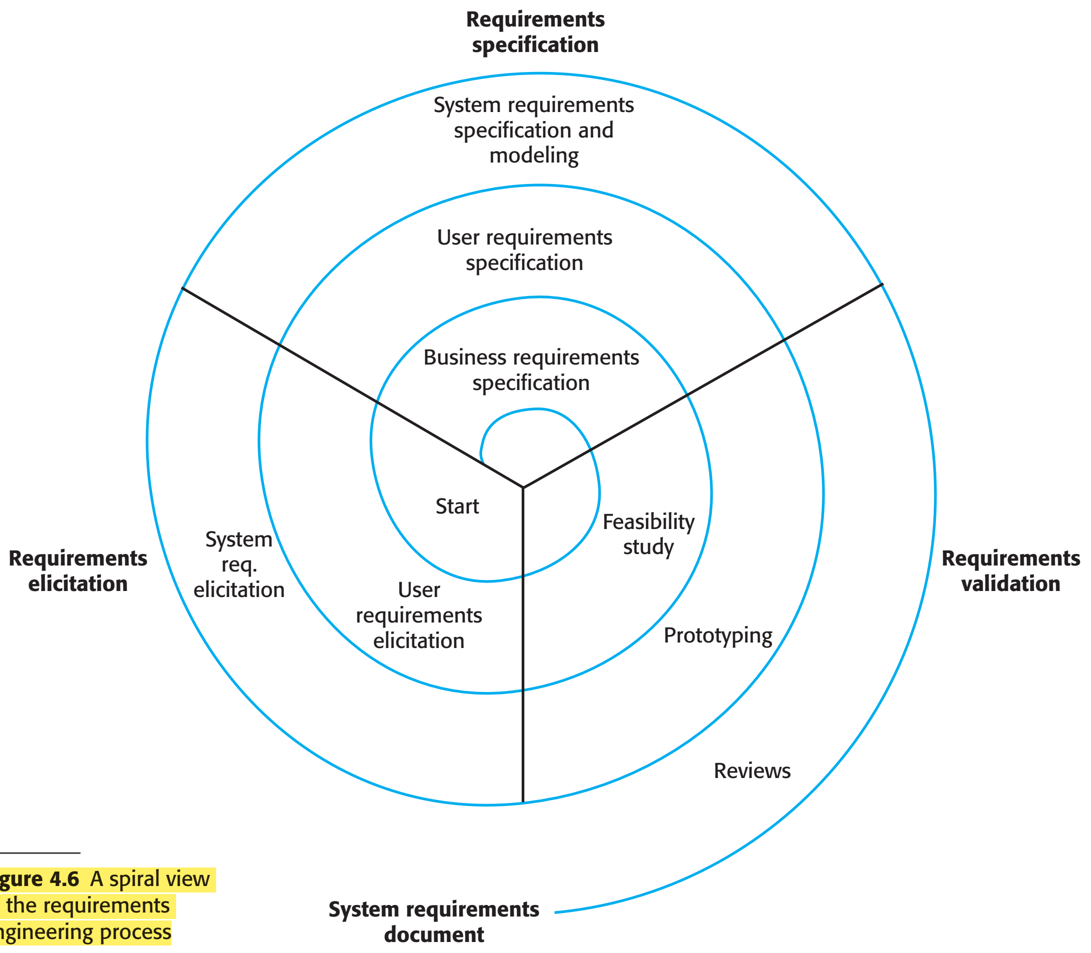

Requirements specification
System requirements specification and modeling
User requirements specification
Business requirements specification
Start
Feasibility study
Prototyping
Reviews
Requirements elicitation
System req. elicitation
User requirements elicitation
Requirements validation
System requirements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

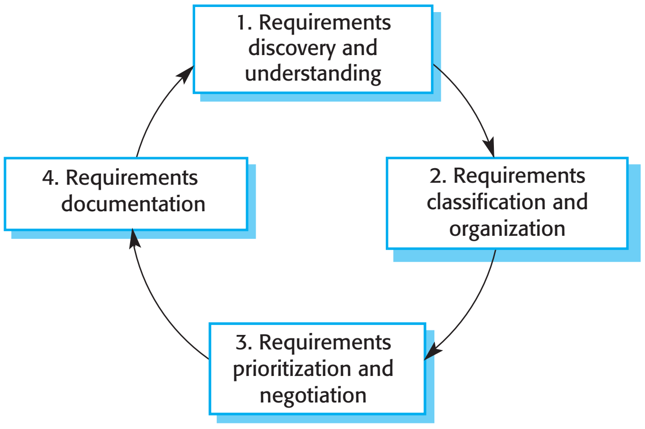

1. Requirements discovery and understanding
2. Requirements classification and organization
3. Requirements prioritization and negotiation
4. Requirements documentation

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).

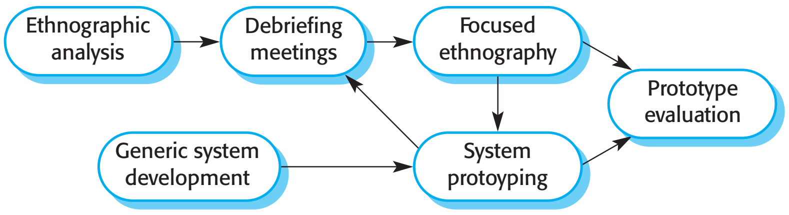


Figure 4.8 Ethnography and prototyping for requirements analysis

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

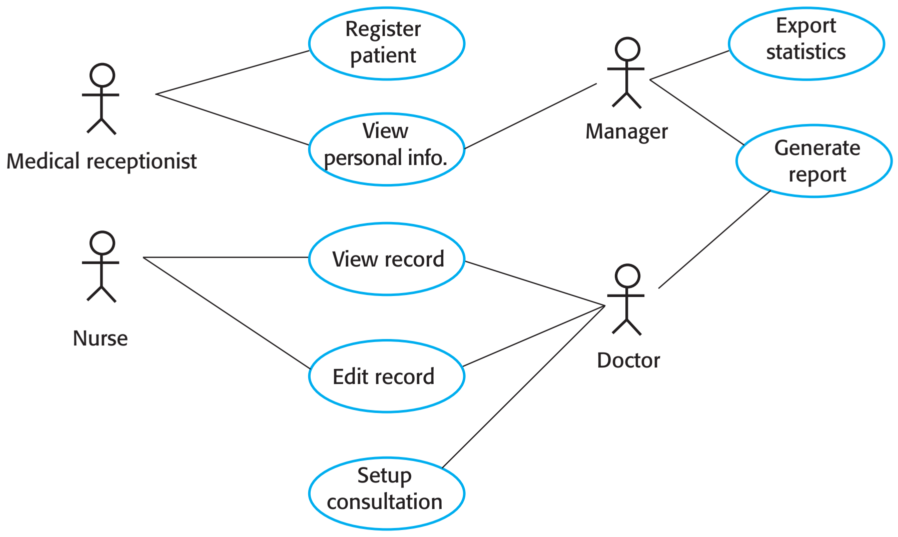


Figure 4.15 Use cases for the Mentcare system

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.

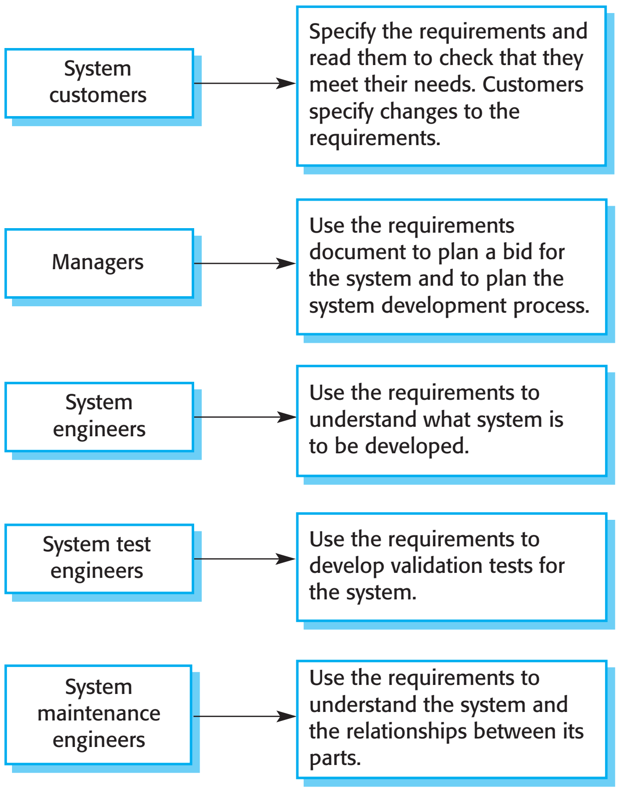

System customers
Specify the requirements and read them to check that they meet their needs. Customers specify changes to the requirements.
Managers
Use the requirements document to plan a bid for the system and to plan the system development process.
System engineers
Use the requirements to understand 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

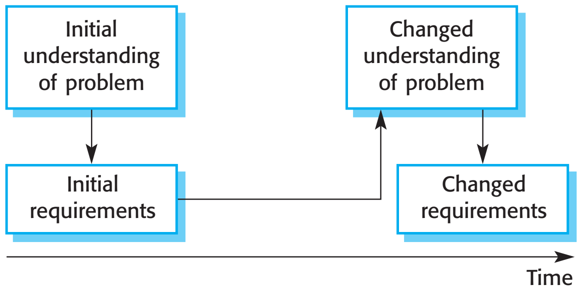

Initial understanding of problem
Initial requirements
Changed understanding of problem
Changed requirements
Time

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.

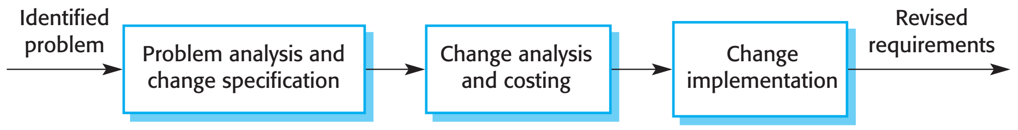

Identified problem
Problem analysis and change specification
Change analysis and costing
Change implementation
Revised requirements

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.

Requirements traceability

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

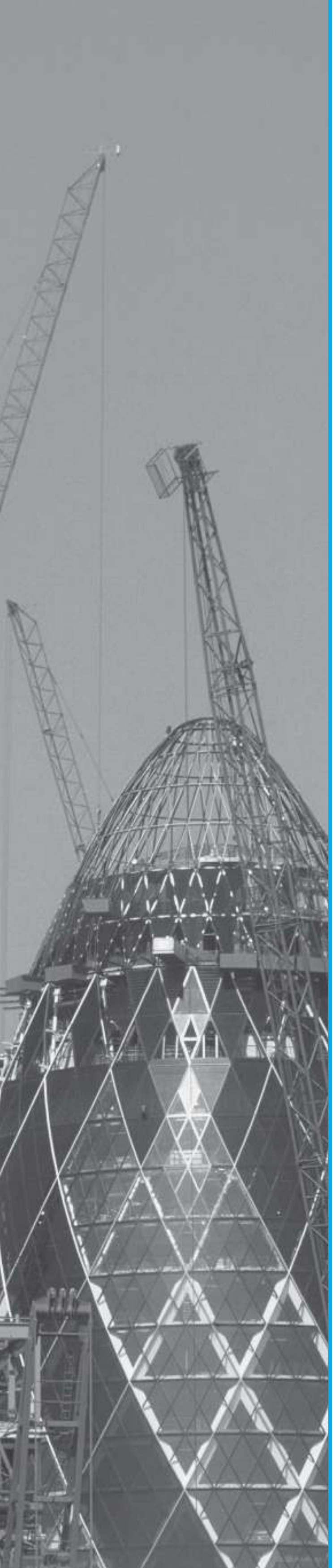
http://software-engineering-book.com/web/traceability/

3. *Change implementation* The requirements document and, where necessary, the system design and implementation, are modified. You should organize the requirements document so that you can make changes to it without extensive rewriting or reorganization. As with programs, changeability in documents is achieved by minimizing external references and making the document sections as modular as possible. Thus, individual sections can be changed and replaced without affecting other parts of the document.

If a new requirement has to be urgently implemented, there is always a temptation to change the system and then retrospectively modify the requirements document. This almost inevitably leads to the requirements specification and the system implementation getting out of step. Once system changes have been made, it is easy to forget to include these changes in the requirements document. In some circumstances, emergency changes to a system have to be made. In those cases, it is important that you update the requirements document as soon as possible in order to include the revised requirements.

KEY POINTS

- Requirements for a software system set out what the system should do and define constraints on its operation and implementation.
- Functional requirements are statements of the services that the system must provide or are descriptions of how some computations must be carried out.
- Non-functional requirements often constrain the system being developed and the development process being used. These might be product requirements, organizational requirements, or external requirements. They often relate to the emergent properties of the system and therefore apply to the system as a whole.
- The requirements engineering process includes requirements elicitation, requirements specification, requirements validation, and requirements management.
- Requirements elicitation is an iterative process that can be represented as a spiral of activities—requirements discovery, requirements classification and organization, requirements negotiation, and requirements documentation.



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

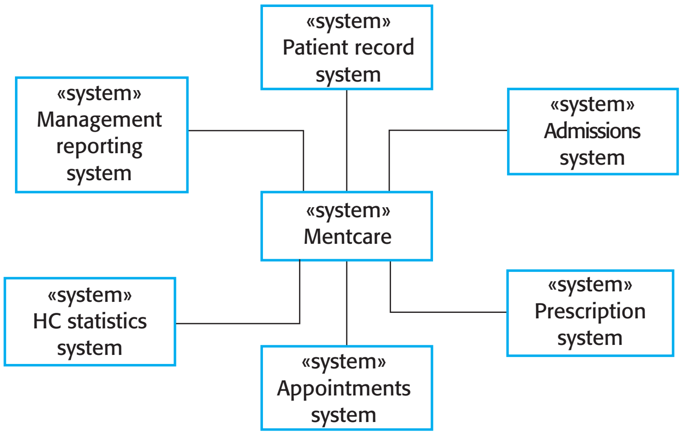

«system» Patient record system
«system» Management reporting system
«system» Admissions system
«system» Mentcare
«system» HC statistics system
«system» Prescription system
«system» Appointments system

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

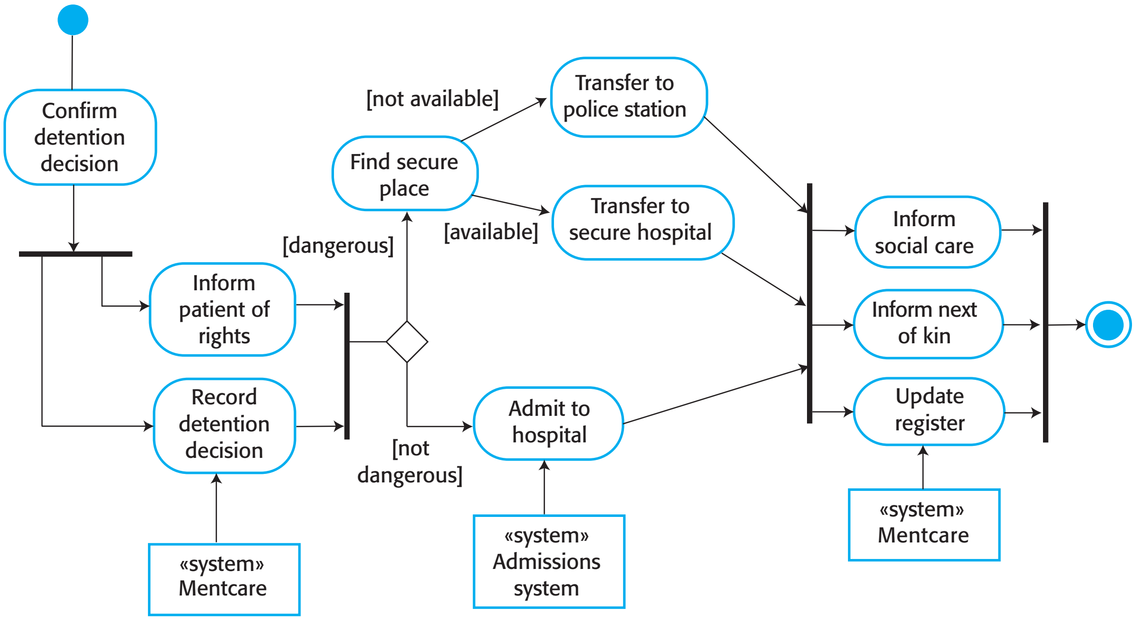


Figure 5.2 A process model of involuntary detention

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

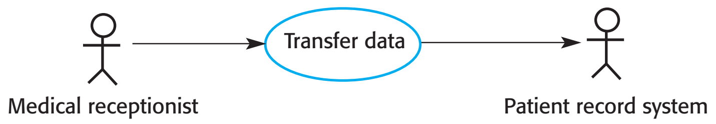


Figure 5.3 Transfer-data use case

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

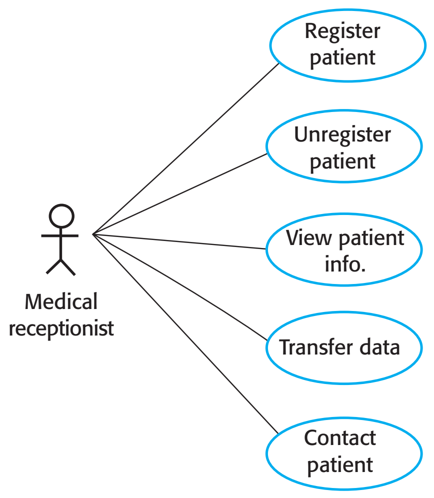


Figure 5.5 Use cases involving the role "Medical receptionist"

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

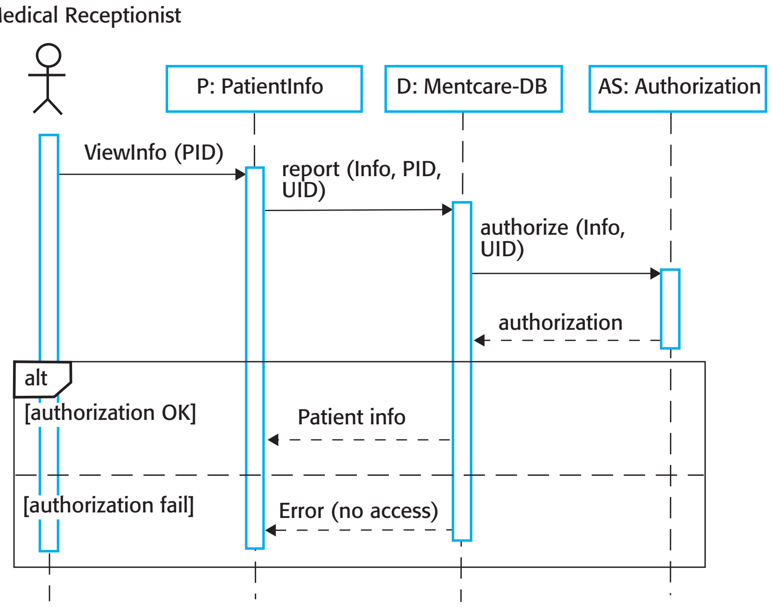


Figure 5.6 Sequence diagram for View patient information

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

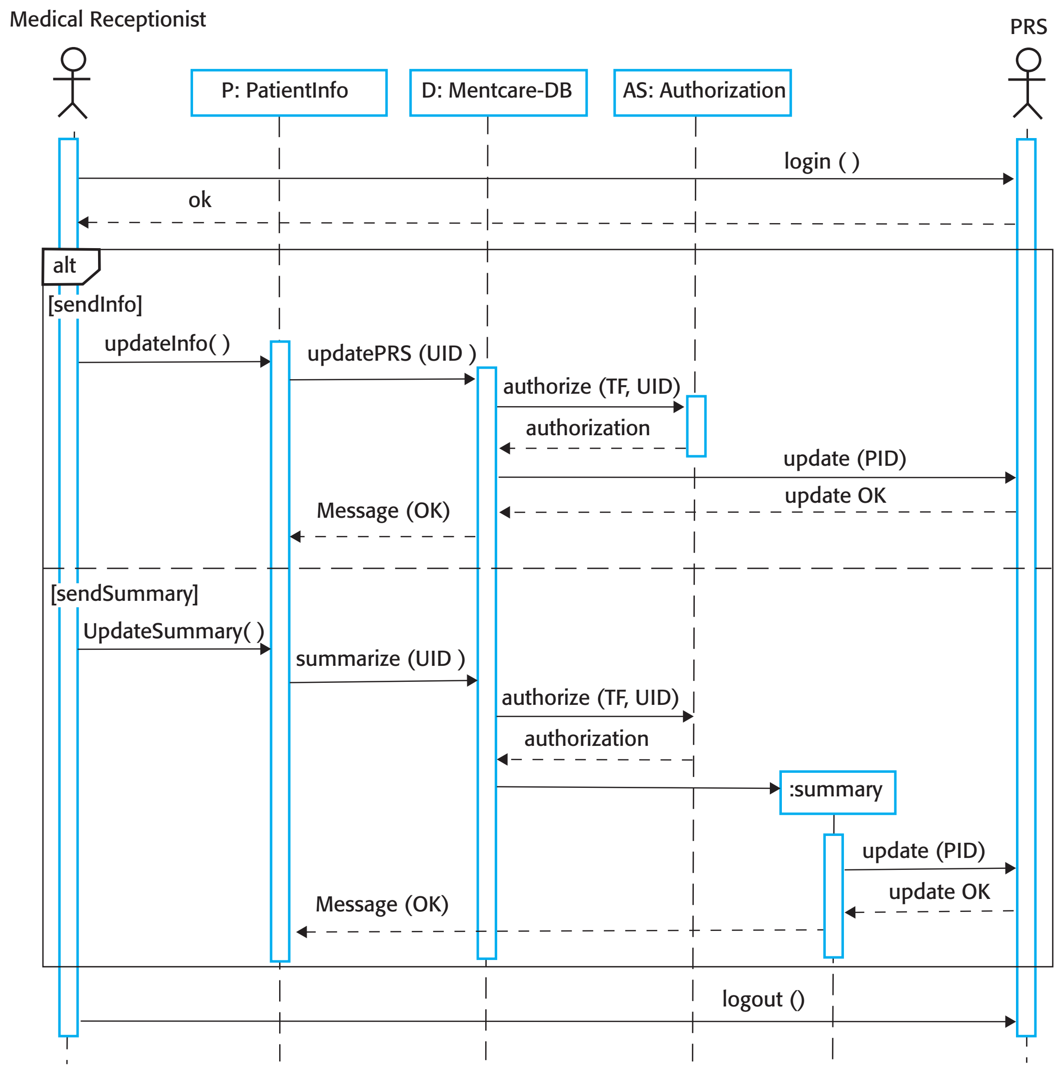


Figure 5.7 Sequence diagram for Transfer Data

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

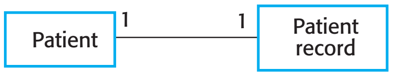


Figure 5.8 UML Classes and association

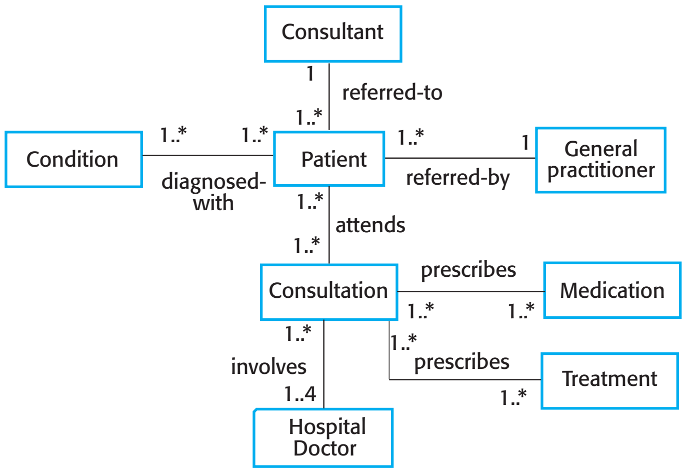


Figure 5.9 Classes and associations in the Mentcare system

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.

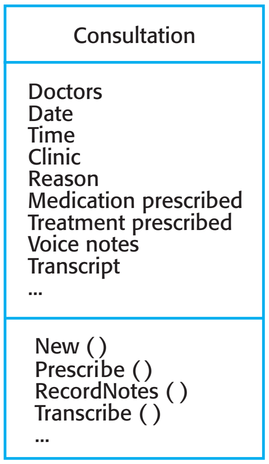


Figure 5.10 A Consultation class

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.

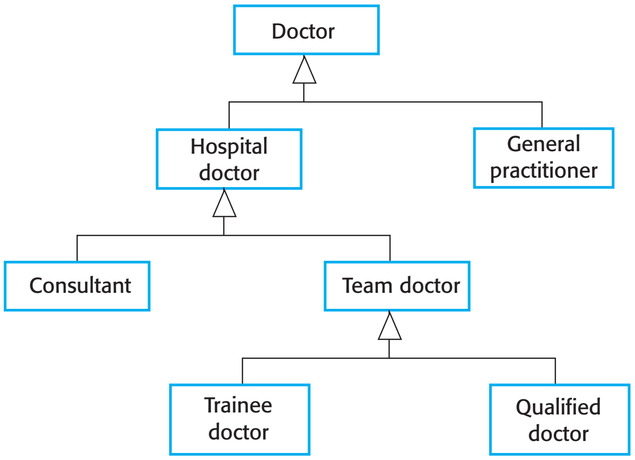


Figure 5.11 A generalization hierarchy

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.

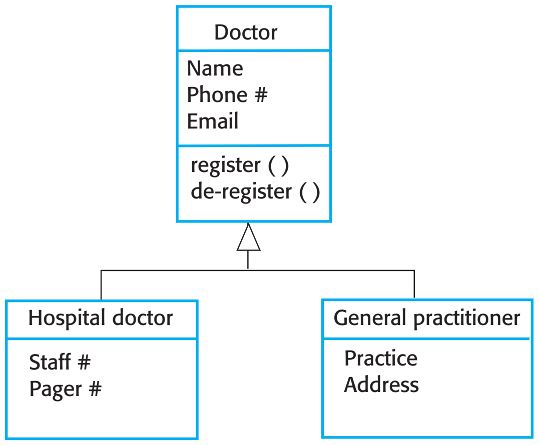


Figure 5.12 A generalization hierarchy with added detail

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.

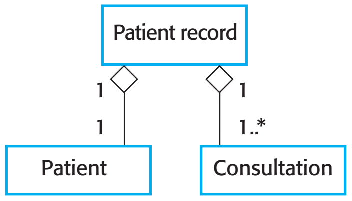


Figure 5.13 The aggregation association

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

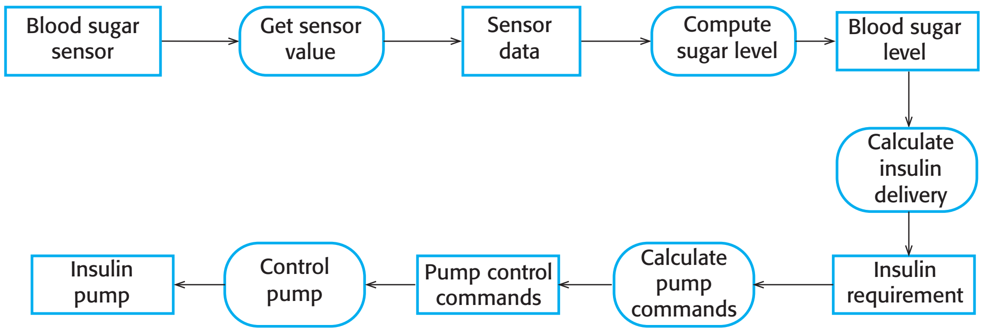


Figure 5.14 An activity model of the insulin pump's operation

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.

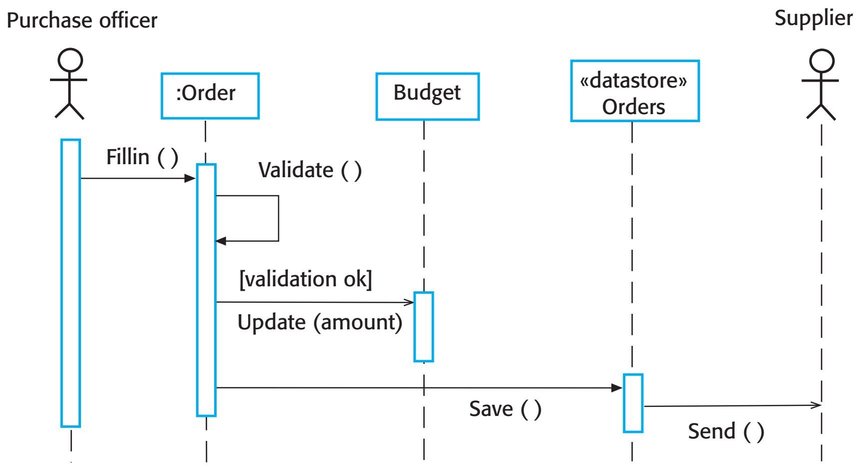


Figure 5.15 Order processing

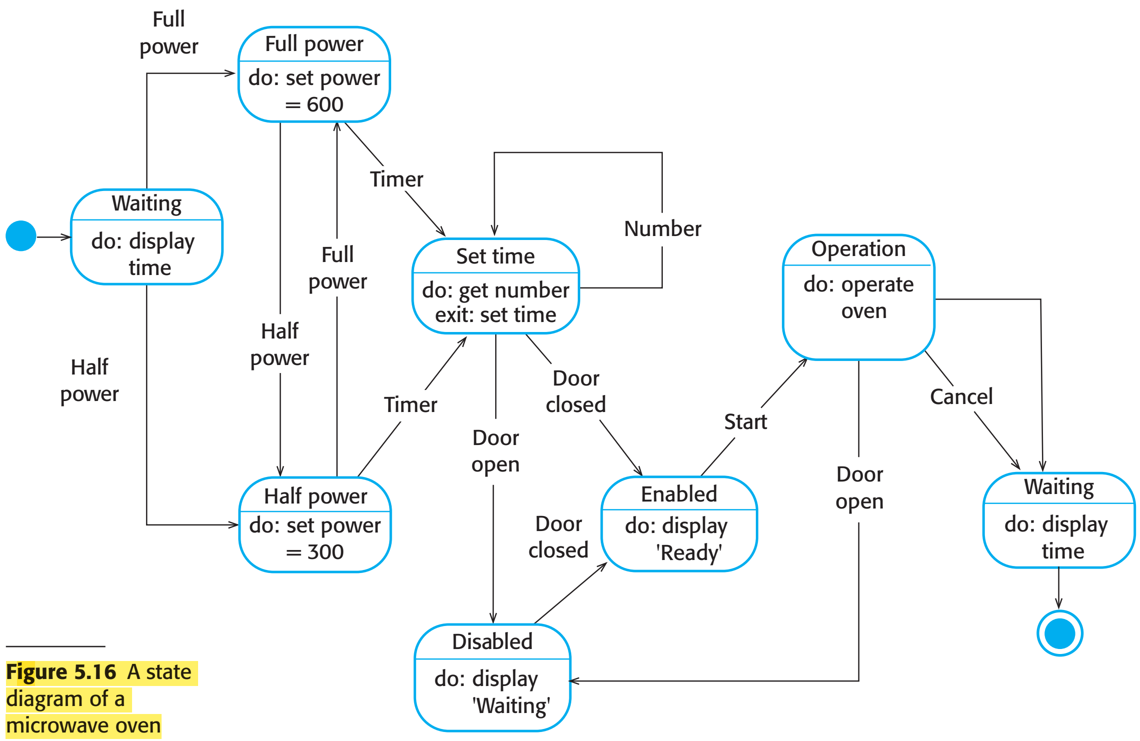


Figure 5.16 A state diagram of a microwave oven

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.

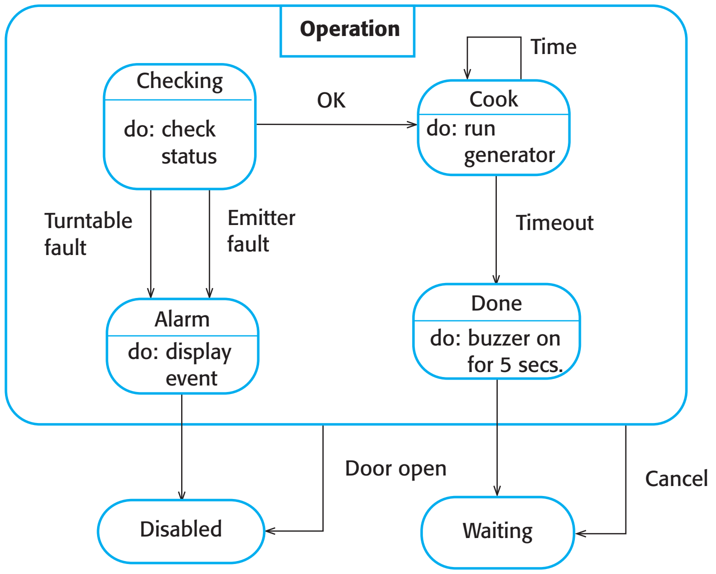


Figure 5.17 A state model of the Operation state

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.

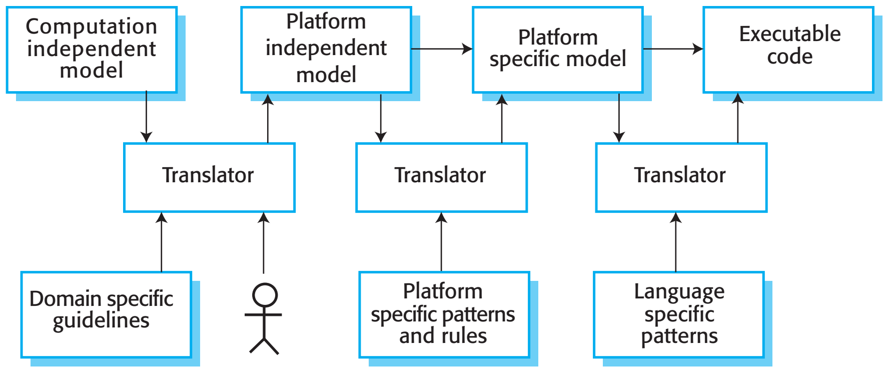

Computation independent model
Platform independent model
Platform specific model
Executable code
Translator
Translator
Translator
Domain specific guidelines
Platform specific patterns and rules
Language specific patterns

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.

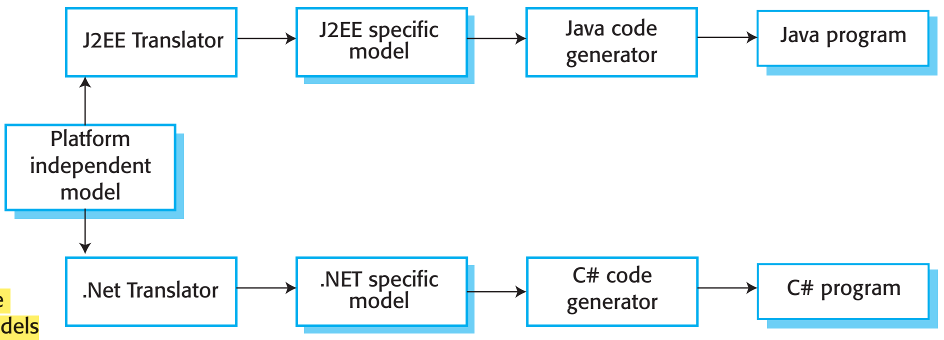


Figure 5.20 Multiple platform-specific models

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.

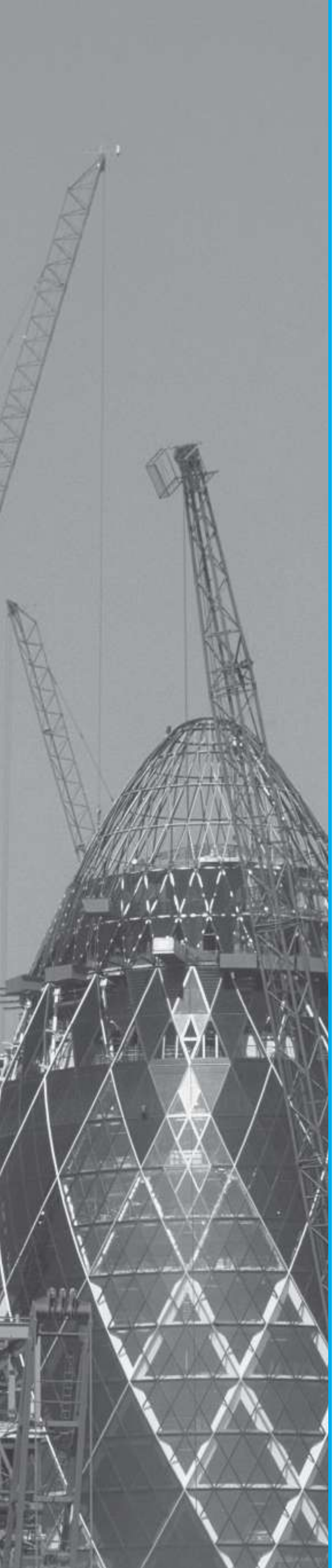
3. The arguments for platform independence are only valid for large, long-lifetime systems, where the platforms become obsolete during a system's lifetime. For software products and information systems that are developed for standard platforms, such as Windows and Linux, the savings from the use of MDA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

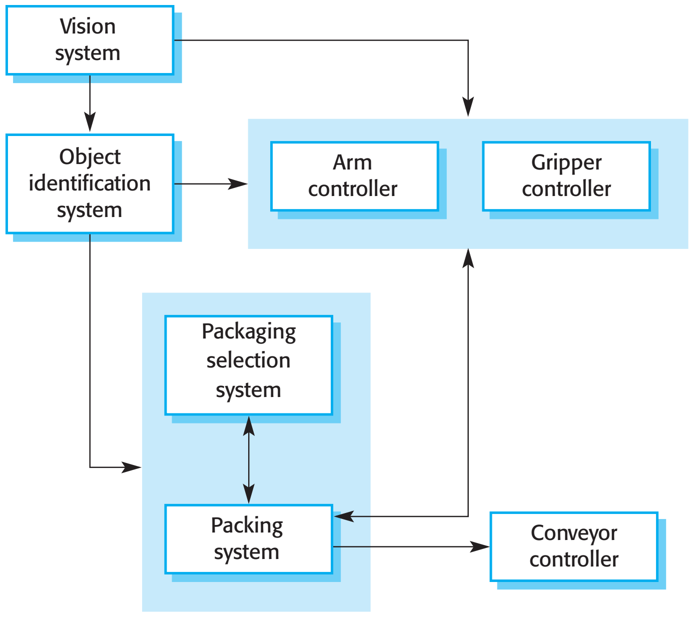


Figure 6.1 The architecture of a packing robot control system

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.

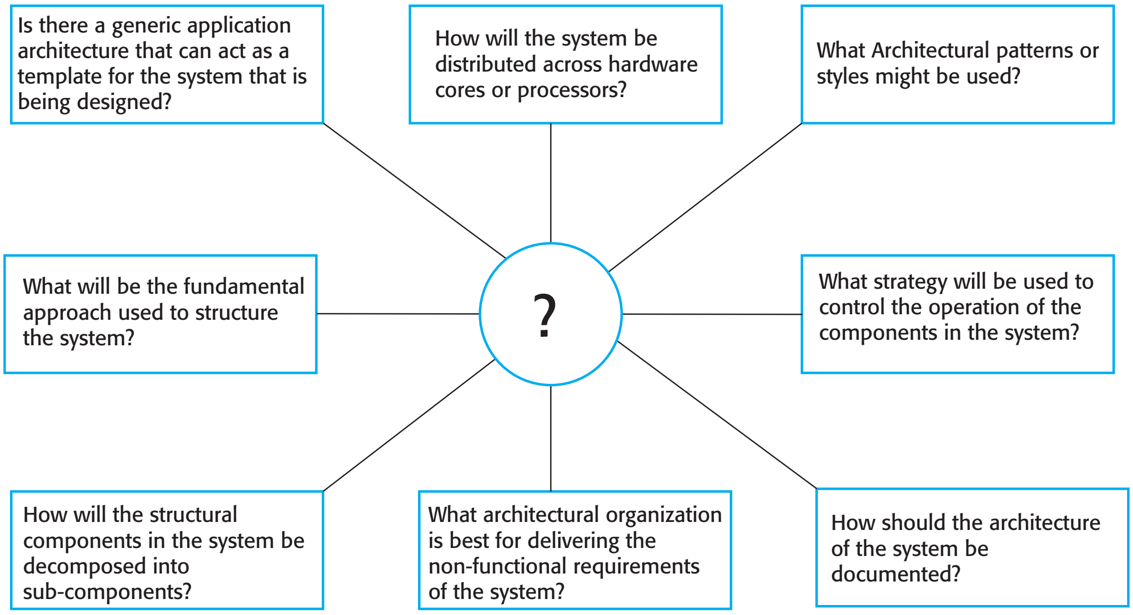

Is there a generic application architecture that can act as a template for the system that is being designed?
How will the system be distributed across hardware cores or processors?
What Architectural patterns or styles might be used?
What will be the fundamental approach used to structure the system?
?
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

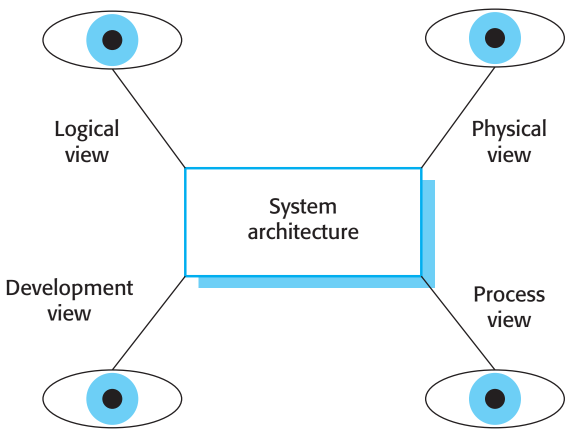


Figure 6.3 Architectural views

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

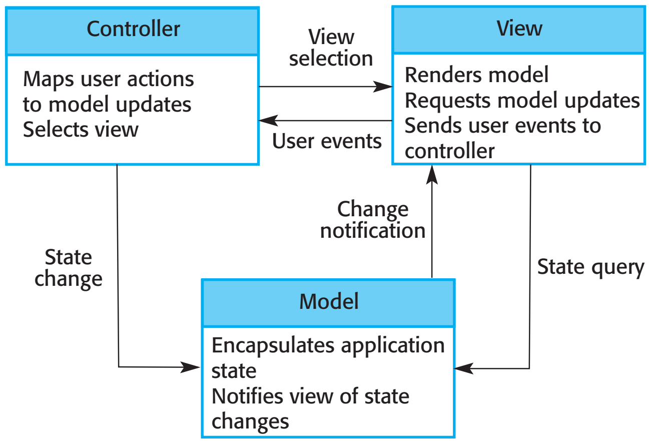


Figure 6.5 The organization of the Model-View-Controller

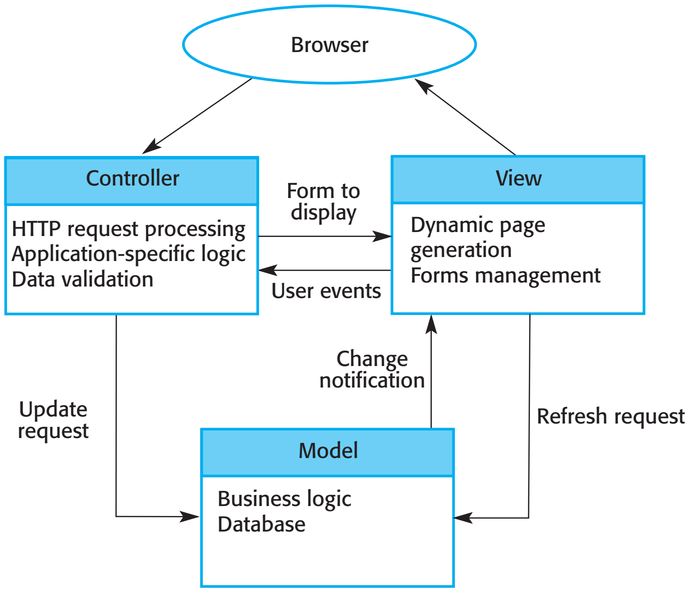


Figure 6.6 Web application architecture using the MVC pattern

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.

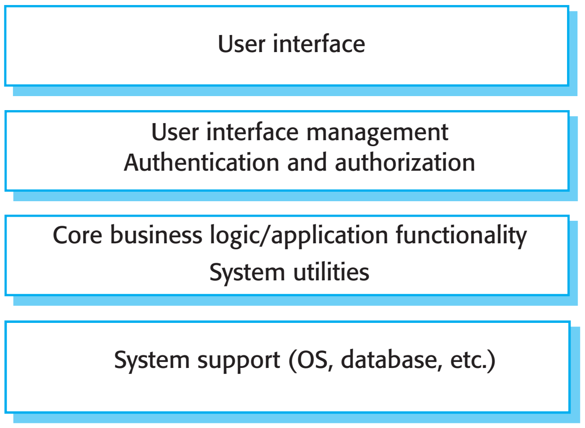

User interface
User interface management
Authentication and authorization
Core business logic/application functionality
System utilities
System support (OS, database, etc.)

Figure 6.8 A generic layered architecture

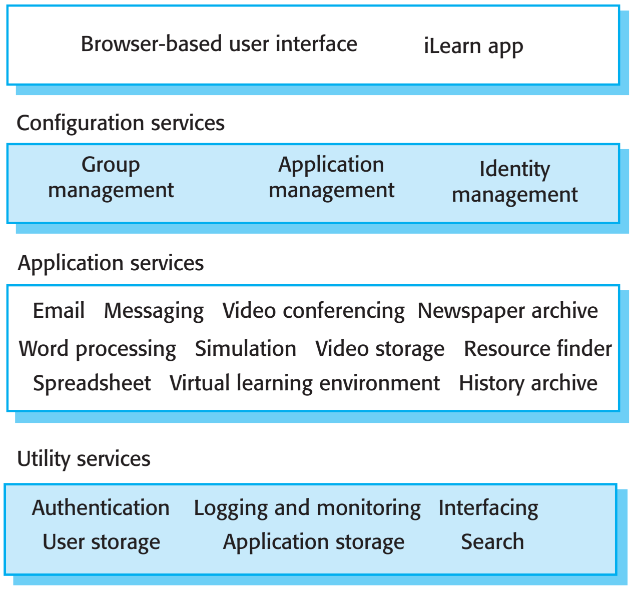


Figure 6.9 The architecture of the iLearn system

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.

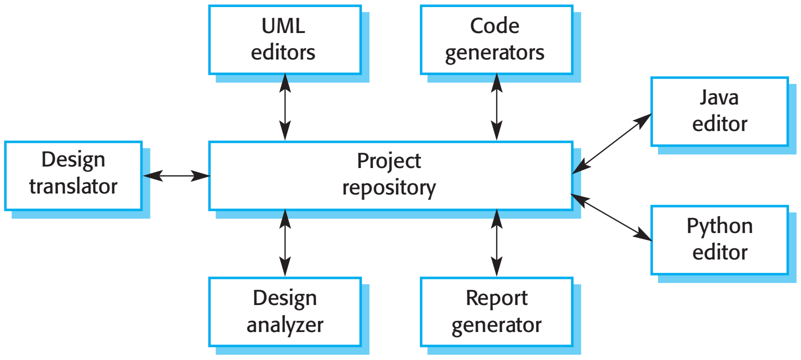


Figure 6.11 A repository architecture for an IDE

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.

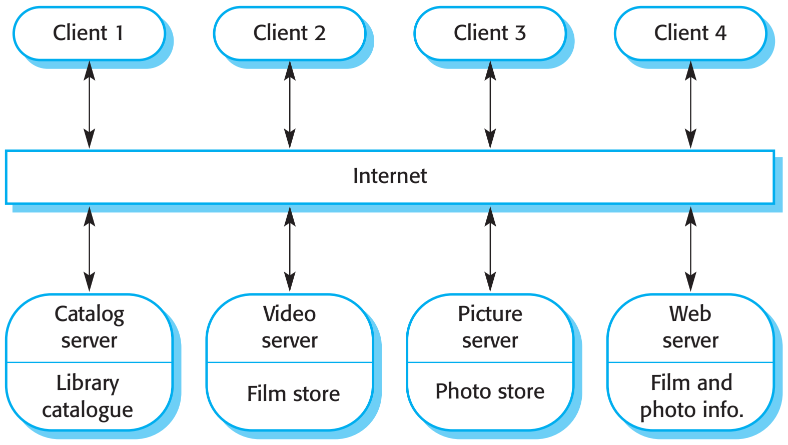


Figure 6.13 A client–server architecture for a film library

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.

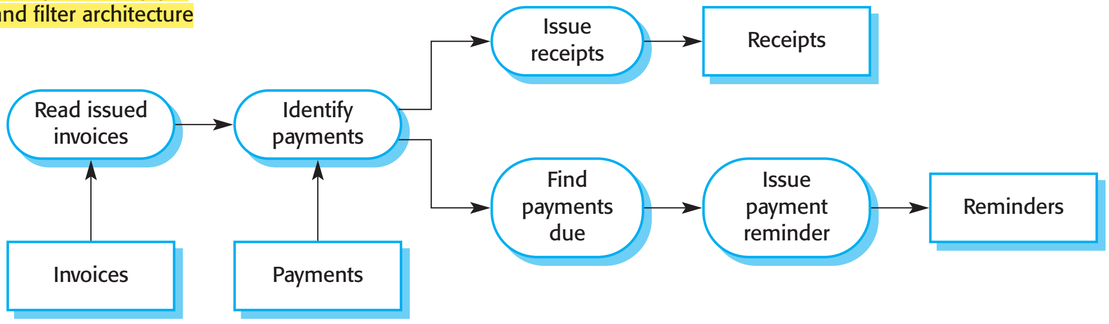
Figure 6.14 The Pipe and Filter pattern

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.

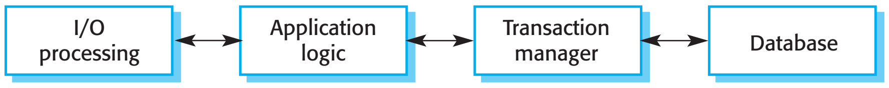


Figure 6.16 The structure of transaction processing applications

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

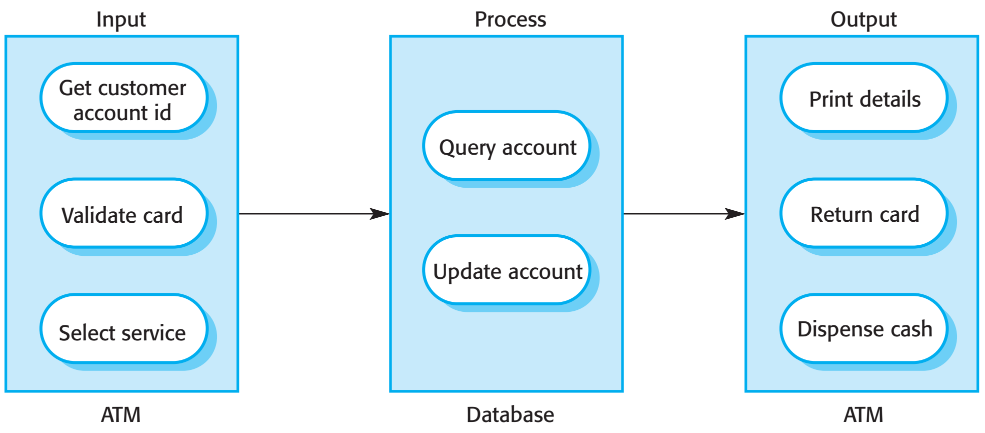


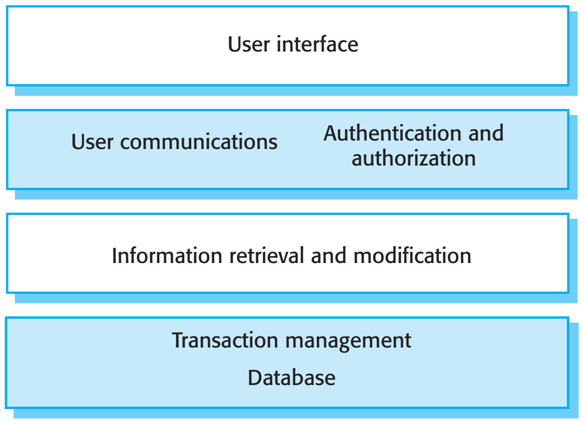
Figure 6.17 The software architecture of an ATM system

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

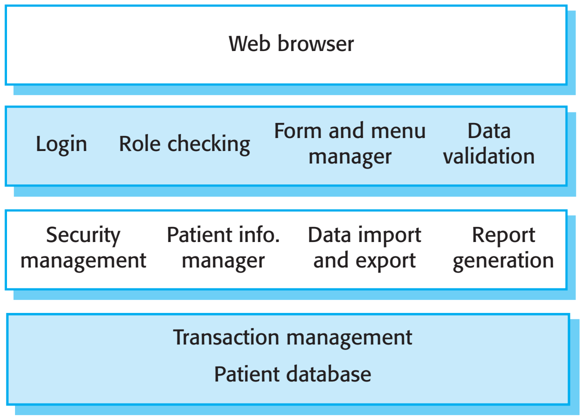


Figure 6.19 The architecture of the Mentcare system

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

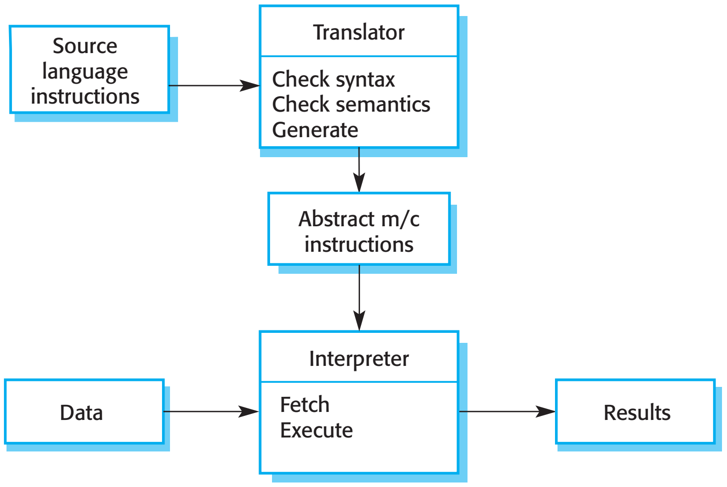


Figure 6.20 The architecture of a language processing system

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

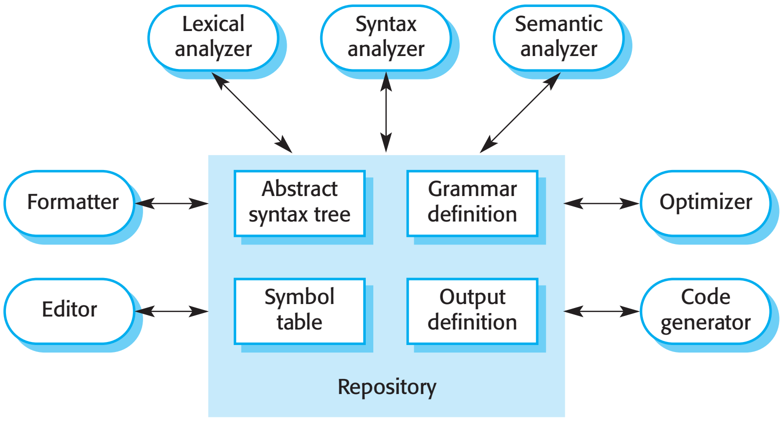


Figure 6.21 A repository architecture for a language processing system

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.

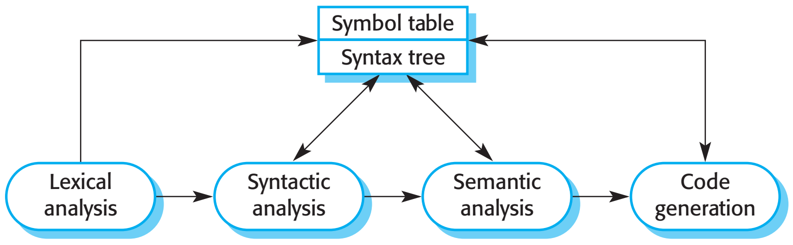


Figure 6.22 A pipe and filter compiler architecture