**Developing requirements**

4

In the previous two chapters, you learned about technologies that software engineers need to master before developing applications. Now, we can start thinking about the particular problem we wish to solve. We will first put effort into understanding the background of the problem, a process called domain analysis. Then we will look at the information you have to gather so that you can describe the problem and its proposed solution. Finally, we will discuss some techniques for gathering and analyzing that information.

In this chapter you will learn about the following

* Domain analysis: learning background knowledge so that you can communicate with users and make more intelligent decisions.
* Understanding the customer’s problem and setting the scope for the project.
* What exactly is a requirement, as well as the various types of requirements.
* Requirements documents and what should be put in them.
* How to go about gathering requirements.
* How to model users’ tasks using use case diagrams and detailed descriptions of use cases.
* How to review a set of requirements.
  1. Domain analysis

*Domain analysis* is the process by which a software engineer learns background information. He or she has to learn sufficient information so as to be able to understand the problem and make good decisions during requirements analysis and other stages of the software engineering process. The word ‘domain’ in this case means the general field of business or technology in which the customers expect to be using the software.

110

Chapter 4

**Developing requirements**

Some domains might be very broad, such as ‘airline reservations’, ‘medical diagnosis’, and ‘financial analysis’. Others are narrower, such as ‘the manufacturing of paint’ or ‘scheduling meetings’. People who work in a domain and who have a deep knowledge of it (or part of it) are called *domain experts*. Many of these people may become customers or users.

To perform domain analysis, you gather information from whatever sources of information are available: these include the domain experts; any books about the domain; any existing software and its documentation; and any other documents you can find. The interviewing, brainstorming and use case analysis techniques discussed later in this chapter can help with domain analysis. Object-oriented modeling, discussed in Chapter 5, can also be of assistance.

As a software engineer, you are not expected to become an expert in the

domain; nevertheless, domain analysis can involve considerable work. The

following benefits will make this work worthwhile:

* + - **Faster development**. You will be able to communicate with the stakeholders more effectively, hence you will be able to establish requirements more rapidly. Having performed domain analysis will help you to focus on the most important issues.
    - **Better system**. Knowing the subtleties of the domain will help ensure that the solutions you adopt will more effectively solve the customer’s problem. You will make fewer mistakes, and will know which procedures and standards to follow. The analysis will give you a global picture of the domain of application; this will lead to better abstractions and hence improved designs.
    - **Anticipation of extensions**. Armed with domain knowledge, you will obtain insights into emerging trends and you will notice opportunities for future development. This will allow you to build a more adaptable system.

It is useful to write a summary of the information found during domain analysis. The process of organizing and writing this summary can help you gain a better grasp of the knowledge; the resulting document can help educate other software engineers who join the team later.

We suggest that a domain analysis document should be divided into sections

such as the following:

1. **Introduction**. Name the domain, and give the motivation for performing the analysis. The motivation normally is that you are preparing to solve a particular problem by development or extension of a software system.
2. **Glossary**. Describe the meanings of all terms used in the domain that are either not part of everyday language or else have special meanings. You must master this terminology if you want to be able to communicate with your customers and users. The terminology is likely to appear in the user interface of the software as well as in the documentation. You may be able to refer to an existing glossary in some other document, rather than writing a new glossary.

114

Chapter 4

**Developing requirements**

systems (see Appendix C for the descriptions of systems).

* 1. The police information system.
  2. The household alarm system.
  3. The GPS automobile navigation system.
  4. The investments system.
  5. The woodworking design system.

**E54** Write a short domain analysis for one of the systems listed in Exercise E53 using the format we proposed in this section. Record only the most important information a software engineer would need to know in order to develop your chosen system. Gather whatever information you can from several sources. Be resourceful in your hunt for information! Do not forget to consider including any general or specific knowledge you might already possess.

* 1. The starting point for software projects

When a development team starts work on a software project, their starting point

can vary considerably. We can distinguish different types of project, based on

whether or not *software* exists at the outset, and whether or not *requirements*

exist at the outset. The four broad categories of starting point are illustrated in

Figure 4.1.

New development

Requirements must be determined

Clients have produced requirements

Evolution of existing system

|  |  |
| --- | --- |
| A | B |
| C | D |

Figure 4.1 Starting points for software projects

In projects of type A or B, the development team starts to develop new software from scratch – this is sometimes called *green-field* development, alluding to constructing a new building where none existed before. In cases C and D the team evolves an existing system, a rather more common situation.

In cases A and C, the development team has to determine the requirements for the software – they either have a bright idea for something that might sell, or else they are asked to solve a problem and have to work out the best way to solve

Section 4.3

**Defining the problem and the scope**

**115**

it. In cases B and D, on the other hand, the development team is contracted to

design and implement a very specific set of requirements. In these latter cases, the customer’s organization has normally done the requirements analysis, perhaps using in-house software engineers or consultants specializing in requirements analysis.

Projects where the requirements are pre-specified should be handled carefully. If the customer has not done a good job of analysis and specification, the requirements are likely to be poor. For example, the customer may be proposing a system that is far too large, or that does not address a clear problem (we will discuss both of these issues later). As a software engineer, you have a professional responsibility to ensure that the requirements on which you base your work are of good quality, even when they were developed by others. You should evaluate such requirements yourself, and work with the customers to resolve any problems. You should not accept a contract where you are required to implement requirements with no changes allowed.

In the next few sections, we will largely be assuming that you are working on a project of type A or C and therefore have to develop your own requirements. However, the matters we discuss will also be helpful if you are reviewing requirements produced by others.

* 1. Defining the problem and the scope

Once you have learned enough about the domain, you can begin to determine the requirements. The first step in this process is to work out an initial definition of the *problem* to be solved.

A problem can be expressed as a *difficulty* the users or customers are facing, or as an *opportunity* that will result in some benefit such as improved productivity or sales. The solution to the problem will normally entail developing software, although you may decide that it is better to purchase software or to develop a non-software solution.

You should write the problem as a simple statement. Careful attention to the problem statement is important since, later on, the requirements will be evaluated based on the question: ‘are we adequately solving the problem?’

A good problem statement is short and succinct – one or two sentences is best. For example, if you were developing a new student registration system, you might express the problem as follows: ‘The system will allow students to register for courses, and change their registration, as simply and rapidly as possible. It will help students achieve their personal goals of obtaining their degree in the shortest reasonable time while taking courses that they find most interesting and fulfilling.’

If the problem is broad, or contains a long list of sub-problems, then the system will have a broad scope, and hence be more complex. An important objective is to narrow the scope by defining a more precise problem. In the

116

Chapter 4

**Developing requirements**

above example, if we had stated: ‘the system will automate all the functions of the registrar’s office’, that leaves open the possibility of including such features as fee payment, printing class lists and allocating rooms to courses.

One way to set the scope is to list all the sub-problems you might imagine the system attacking. To narrow the scope, you can then exclude some of these sub-problems – perhaps they can be left for another project. Figure 4.2 illustrates this.

Initial list of problems with very broad scope

Narrowed scope

Scope of another system

browsing courses

room allocation

registering

fee payment

exam scheduling

browsing courses

room allocation

registering exam scheduling

fee payment

Figure 4.2 Narrowing the project’s scope

Sometimes, an inappropriate choice of problem statement can result in a scope that is too narrow or completely wrong. To determine whether this is the case, think about what will be the user’s ultimate *high-level goal* when they use the system, and the customer’s high-level goal for having it developed.

In the university registration example you could consider a student’s goal to be ‘completing the registration process’. However, you can see that the student’s higher-level goal might be, ‘obtaining their degree in the shortest reasonable time while taking courses that they find most interesting and fulfilling’. This new goal sheds a different light on the problem; you might consider adding features to the system that would not otherwise have occurred to you, such as actively proposing courses based on an analysis of the student’s academic and personal-interest profiles.

All the requirements gathering and analysis techniques described later in this chapter can help in defining the problem and hence the system’s scope. Interviewing can give you the stakeholders’ personal perspectives; brainstorming can generate lists of ideas from which you can extract a suitable problem or problems; use case analysis can give you a list of the possible things the system could do; and prototyping can give everybody a better perspective about what might be possible.

It is a good idea to define the problem and scope as early as possible, before getting deeper into analysis of the detailed requirements. This prevents you from working on unnecessary requirements. However, as with domain analysis, your perspective on the problem will improve as analysis continues, hence the problem statement may need to be refined several times.

Section 4.3

**Defining the problem and the scope**

**117**

*Example 4.3*

*Define the problem and scope for a system that handles university degree*

*requirements and registrations. Then develop a requirements statement from this.*

A university registrar’s office handles a large number of functions. Below, we have listed some of the functions that you might consider building into such a system.

The initial, overly broad, problem statement might be: ‘to automate all the functions of the registrar’s office’.

You would have extensive discussions with stakeholders, and then agree on a narrower problem statement such as the following: ‘Helping university administrators manage lists of courses, degree requirements, registration and academic results. Helping students choose and register in courses in which they are interested that will lead to their degree.’

You can then determine which functions should be included in the system. The functions marked with a ‘’ will be included, while those marked with a ‘

’ will be excluded. Note that there will still have to be systems to handle the functions marked ‘’, but these will be left to others to develop, or to a later project.

 Fee payments and related accounting and billing

 Applications for admission

 Editing and querying the list of available courses, including their descriptions and lists of prerequisites

 Editing and querying the requirements for obtaining a degree

 Editing and querying the list of courses to be taught in a given semester

 Scheduling the times that courses will be offered

 Allocating courses or exams to rooms

 Helping students determine which courses they could take by analyzing their degree requirements, the courses they have previously taken, their schedule, and their preferences

 Registering students

 Recording marks

 Printing transcripts

*Example 4.4 You are asked to improve a data entry program used to enter a patient’s personal information when he or she is first admitted to a hospital. Admission clerks have to enter each new patient’s name, address, telephone number, and various other pieces of data. The customer tells you that the admissions clerks make an unacceptable number of mistakes that contaminate the database and cause administrative problems. You are told that the problem is lack of clarity in the*

Section 4.5

**Types of requirements**

**119**

**E56** Give precise problem statements for the systems listed in E55. Remember to think about high-level goals.

* 1. What is a requirement?

**Definition:** a *requirement* is a statement describing either 1) an aspect of what the

proposed system must do, or 2) a constraint on the system’s development. In

either case, it must contribute in some way towards adequately solving the

customer’s problem; the set of requirements as a whole represents a

negotiated agreement among all stakeholders.

Let us dissect this definition in order to better understand it:

* + - *A requirement is a statement…*: this means that each requirement is a relatively short and concise piece of information, expressed as a fact. It can be written as a sentence or can be expressed using some kind of diagram. We will call a collection of requirements a requirements document.
    - *…an aspect of what the proposed system must do…*: most requirements say something about the tasks the system is supposed to accomplish. They do not describe the domain.
    - …*a constraint on the system’s development…*: requirements often specify the quality levels required. They may also specify details such as the programming language to be used if this is truly important to the customer. They should, however, avoid discussing incidental aspects of the design.
    - *…contribute … towards adequately solving the customer’s problem*: a requirement should only be present if it helps solve the customer’s problem – as we discussed in Chapter 1, this is what software engineering is all about.
    - *…a negotiated agreement among all stakeholders…*: a statement about the system should not be declared to be an official requirement until all the stakeholders (users, customers, developers and their management) have reviewed it, have negotiated any needed changes, and have agreed that it is valid.
  1. Types of requirements

Requirements can be divided into four major types: functional, quality, platform and process. Requirements documents normally include at least the first two types.

Functional requirements

Functional requirements describe what the system should do; in other words, they describe the *services* provided for the users and for other systems. The

120

Chapter 4

**Developing requirements**

functional requirements should include 1) everything that a *user* of the system would need to know regarding what the system does, and 2) everything that would concern any *other system* that has to interface to this system. Details that go any deeper into how the requirements are implemented should be left out.

The functional requirements can be further categorized as follows:

* + - What *inputs* the system should accept, and under what conditions. This includes data and commands both from the users and from other systems.
    - What *outputs* the system should produce, and under what conditions. Outputs can be to the screen or printed. They can also be transmitted to other systems, such as special I/O devices, clients or servers.
    - What data the system should *store* that other systems might use. This is really a special kind of output that will eventually become an input to other systems. Data which is stored for the exclusive use of this system (e.g. the specifics of a file format used to temporarily back up some data) can be ignored until the design stage.
    - What *computations* the system should perform. The computations should be described at a level that all the readers can understand. For example, you would

describe a sorting process by saying that the result is to be ordered in ascending sequence according to the account number. You would not normally specify the particular algorithm to be used.

* + - The *timing and synchronization* of the above. Not all systems involve timing and synchronization – this category of functional requirements is of most importance in hard real-time systems that do such things as control hardware devices (e.g. telecommunications systems, systems that control power plants or factories, and systems that run automobiles and airplanes).

An individual requirement often covers more than one of the above categories.

For example, the requirements for a word processor might say, ‘when the user selects “word count”, the system displays a dialog box listing the number of characters, words, sentences, lines, paragraphs, pages, and words per sentence in the current document.’ This requirement clearly describes input (selecting ‘word count’), output (what is displayed) and computation (counting all the necessary information, and computing the average words per sentence).

*Example 4.6 Summarize the functional requirements for an embedded software system that allows a user to control a microwave oven. The system as a whole consists of:*

❏ *A keypad, with the following buttons that deliver an interrupt to the software when they are pressed: 0 to 9, five power-level buttons (‘hi’, ‘med-hi’, ‘med’, ‘med-low’, ‘low’), three temperature buttons (‘frozen’, ‘refrigerated’, ‘room temperature’), and five action buttons (‘AUTO-DEFROST’, ‘AUTO-REHEAT’, ‘START’, ‘CANCEL’ and ‘TIME OF DAY’).*

Section 4.5

**Types of requirements**

**121**

* *A door sensor that delivers an interrupt to the software when the door is opened or closed.*
* *A steam sensor that can be queried by the software, and which indicates the amount of steam being released from the food.*
* *A digital LCD display on which the system can display output.*
* *A sound generator that the system can use to generate various tones.*
* *The cooking hardware (microwave emitter, fan and turntable). The software can run this at the five different power levels, and can turn it off.*

The following summarizes the main functional requirements. This is intended to illustrate the different categories of functional requirements, which we have marked in italics. It is not a complete requirements document; examples of more complete documents are found in Sections 4.11 and 4.12. There are a few deliberate weaknesses in these requirements, which are left for you to find in later exercises.

1. The system can be in the following modes (*conditions under which input and output can occur*):
   * ‘idle’: this is entered when the system is switched on, when cooking is complete or when ‘CANCEL’ is pressed. This mode is exited when the system starts accepting input.
   * ‘accepting input’: this is entered if the system was in idle mode and the user presses any button, except ‘CANCEL’ and ‘START’. This mode is exited when the system enters ‘cooking’ mode, the user presses ‘cancel’, or the user completes the process of setting the time of day.
   * ‘cooking’: this is entered if the door is closed, while the system is in ‘suspended’ or ‘accepting input’ mode, and the user then presses ‘start’. This is exited when the user opens the door or presses ‘cancel’.
   * ‘suspended’: this is entered if the user opens the door while cooking. This mode is exited when the user presses ‘cancel’; or closes the door and then presses ‘start’.
2. The user specifies a valid cooking method in one of the following ways (*input*):
   * By pressing a sequence of up to five digits indicating minutes and seconds. The last two digits are the seconds, the previous digits (if any) are the minutes. The user may optionally then press one of the power-level keys.
   * By pressing ‘AUTO-DEFROST’ followed by an optional sequence of digits indicating the weight in pounds. If the user omits the weight, then the default is 1.
   * By pressing ‘AUTO-REHEAT’ followed optionally by one of the temperature buttons. If the user omits the temperature, then the default is ‘refrigerated’.

122

Chapter 4

**Developing requirements**

1. After specifying a cooking method, the user must press ‘START’ to initiate cooking (*input*).
2. The user sets the time of day by pressing ‘TIME OF DAY’ followed by four digits indicating the hours and minutes, followed by ‘TIME OF DAY’ again (*input*).
3. When in idle mode, the system displays the time of day using a 12-hour clock, without any ‘a.m.’ or ‘p.m.’ (*output*).
4. Each time the user presses a button, the system generates a tone. If the button is valid, the tone is high-pitched. If any button is pressed in an invalid sequence (e.g. the user presses ‘START’ while the door is open, or presses ‘AUTO-DEFROST’ followed by ‘AUTO-REHEAT’) the tone is low-pitched (*output and conditions for the output*).
5. When the system is in ‘accepting input’ mode, the system indicates on the display the buttons the user presses (*output – in a full document, more details would be needed*).
6. When the user specifies ‘AUTO-DEFROST’ cooking, the system computes the required heating time and power level from the entered weight of the food (*computation – in a full requirements document the formulas used would have to be specified*).
7. When the system enters ‘cooking’ mode, the system sends a signal to the cooking hardware to start cooking at the specified or computed power-level (*output*).
8. If the system is cooking, and ‘AUTO-REHEAT’ has been specified:
   * The system stops cooking when it detects sufficient steam, indicating that the food is hot (*input and output – in a full requirements document more details of required steam levels would be required*).
   * The system displays an estimate of the remaining cooking time from the initial temperature of the food, as specified by the user, and the amount of steam it detects. The system constantly updates this estimate (*computation and output*).
   * As a safety measure, if the system has reached its time estimate and has not detected any increase in steam, then it stops cooking (*computation, timing and output*).
9. When the system is in ‘cooking’ mode and either a simple time-period or ‘AUTO-DEFROST’ has been specified:
   * The system displays the power level and the cooking time remaining (*output*).
   * The system stops cooking when the time remaining reaches zero (*timing and output*).

Section 4.5

**Types of requirements**

**123**

Exercise

1. When the system is to stop cooking, it sends a signal to the cooking hardware to switch it off, and sounds three short high-pitched beeps (*output*).

**E57** Using the same format as Example 4.6, describe a set of functional requirements for systems that would solve the problem statements you devised in Exercise E56. If you find that a list of functional requirements is getting too extensive, you may further narrow the problem statement.

Quality requirements

Quality requirements ensure the system possesses quality attributes such as the five discussed in Chapter 1: usability, efficiency, reliability, maintainability and

reusability. These requirements *constrain* the design to meet specified levels of

quality.

One of the most important things about quality requirements is to make them

*verifiable*. By this, we mean that it should be possible, after the system is implemented, to determine whether they have, in fact, been adhered to. The verification is normally done by measuring various aspects of the system and seeing if the measurements conform to the requirements.

The following are some of the main categories of quality requirements, although this is not an exclusive list.

* + - **Response time**. For systems that process a lot of data or use a network extensively, you should require that the system gives results or feedback to the user in a certain minimum time. For example, you might write that a result

must be calculated in less than three seconds, or that feedback about the progress of a search must appear within one second. In Chapter 7, we will discuss usability guidelines for response time. Remember, however, that for hard real-time systems, response time requirements should be considered to be functional – the system would not work unless they are adhered to.

* + - **Throughput**. For number-crunching programs that may take hours, or for servers that continually respond to client requests, it is a good idea to specify *throughput*, in terms of computations or transactions per minute.
    - **Resource usage**. For systems that use non-trivial amounts of such resources as memory and network bandwidth, you should

**Non-functional requirements**

Quality, platform and process requirements used to be collectively called *non-functional* requirements. However, that term has fallen into disfavor and we have therefore stopped using it in this book.

specify the maximum amount of these resources that the system will consume. This allows others to plan hardware upgrades. For

example, you could specify that no more than

50 MB of memory is to be used by the system,

and that the system must consume less than

10% of the CPU’s time when run on a 1.8GHz machine under a certain operating system.

124

Chapter 4

**Developing requirements**

* + - **Reliability**. *Reliability* is measured as the average amount of time between failures or the probability of a failure in a given period. It is a good idea to set strong but realistic targets for this. For example, you might specify that a

continuously running server must not suffer more than one failure in a six-

month period. It is necessary to define what you mean by a failure: normally it means much more than just crashes; failures normally include any difficulties users have getting their work done which are attributable to defects.

* + - **Availability**. *Availability* measures the amount of time that a server is running and available to respond to users. As with reliability, you should set a target for this. For example, you might specify that a server must be available over 99% of the time, and that no period of downtime may exceed 1 minute. Telecommunications systems have very rigorous availability criteria: for example, you might specify that such a system must not be down more than 10 minutes in its 20-year life-span. This is also often called ‘6-nines’ availability, since it is equivalent to saying that the system must be up 99.9999% of the time.
    - **Recovery from failure**. Requirements in this category constrain the allowed impact of a failure. They state that if the hardware or software crashes, or the power fails, then the system will be able to recover within a certain amount of time, and with a certain minimal loss of data. For example, the requirements

for a word processor might state: ‘the system will allow users to continue their work after a failure with the loss of no more than 20 words of typing or 20 formatting commands.’ Note that the detailed *procedure* for recovery from failure is a functional requirement.

* + - **Allowances for maintainability and enhancement**. In order to ensure that the system can be adapted in the future, you should describe changes that are anticipated for subsequent releases. This constrains design and improves quality without adding explicit new functional requirements.
    - **Allowances for reusability**. Similarly to the previous category, it is desirable in many cases to specify that a certain percentage of the system, e.g. 40%, measured in terms of lines of code, must be designed generically so that it can be reused. This will help break the reuse vicious cycle discussed in the previous chapter.

Platform requirements

This type of requirement constrains the environment and technology of the system:

* + - **Computing platform**. It is normally important to make it clear what hardware and operating system the software must be able to work on. Such requirements specify the *least* powerful platforms and declare that it must work on anything more recent or more powerful. For example, you might declare that certain software must run on any computer operating under Mac OS X version 10.2 or MS-Windows 98 or higher, with 128 MB of RAM or more, and 100 MB of free

Section 4.5

**Types of requirements**

**125**

disk space. This is quite different from the resource usage constraint on memory above: a resource usage constraint specifies that the system *will not use* more than a certain amount of resources, whereas a platform constraint says it *is not guaranteed to run* if inadequate or incorrect resources are available.

* + - **Technology to be used**. While it is wise to give the designers as much flexibility as possible in choosing how to implement the system, sometimes constraints must be imposed. Common examples are to specify the programming language or database system. Such requirements are normally stated to ensure that all systems in an organization use the same technology – this reduces the need to train people in different technologies. The company might have also spent considerable money on a certain technology and wants to get the best value for that money.

Process requirements

The final type of requirements constrains the project plan and development methods:

* + - **Development process (methodology) to be used**. In order to ensure quality, some requirements documents specify that certain processes be followed; for example, particular approaches to testing. The details of the process should not be included in the requirements; instead a reference should be made to other documents that describe the process.
    - **Cost and delivery date**. These are important constraints. However, they are usually not placed in the requirements document, but are found in the contract for the system or are left to a separate project plan document, which we will discuss in Chapter 11.

In most cases, the boundaries between the functional requirements and other requirements types are clear. But sometimes it is unclear in which category a requirement should fit. For example, if you were designing a word processor you would likely build in an ‘auto-save’ feature so that, if the computer was turned off or crashed, very little work would be lost. The question is, is the auto-save feature a fundamental part of the word processor’s functionality (a functional requirement), or is it a constraint on quality (a quality requirement)? You would probably conclude that it is a functional requirement.

Example 4.7

*Classify the following aspects of an airline reservation system into F for*

*functional, Q for quality, PL for platform, PR for process, and X for ‘should not be*

*a requirement’. Also indicate the subcategory of requirement. For something that*

*should not be a requirement, explain why not.*

* How information about flights, passengers and bookings are entered. F: *Input*.
* What information appears on tickets and reports. F: *Output*.
* How fares are calculated. F: *Computation*.

Section 4.6

**Use cases: describing how the user will use the system**

**127**

1. The system must be produced at minimum cost.
2. The automatic teller machine should be fast.

**E60** Write a set of quality and platform requirements for the microwave oven system of Example 4.6.

**E61** Add quality and platform requirements to the systems you worked on in Exercise E57.

* 1. Use cases: describing how the user will use the system

*Use case analysis* is a systematic approach to working out what users should be able to do with the software you are developing. It is one of the key activities in requirements analysis.

The first step in use case analysis is to determine the types of users or other systems that will use the facilities of this system. These are called *actors*. An actor

is a *role* that a user or some other system plays when interacting with your system; each actor will need to interact with the system in different ways.

Most of the actors will be users; a given user may be considered as several different actors if they play different roles from time to time – that is, if they have different job functions. Other actors will be systems that automatically exchange information with your system. If you performed domain analysis, you will have already listed the different types of users

*Example 4.8 You are developing a system for managing the processes of a small town public library. List all the actors for this system.*

The actors might include the following: Librarian, Checkout Clerk, Borrower.

The second step in use case analysis is to determine the tasks that each actor will need to do with the system. Each task is called a *use case* because it represents one particular way the system will be used.

**Definition:** a *use case* is a typical sequence of actions that an actor performs in order to

complete a given task.

When listing use cases, make sure you respect the system scope, as discussed in Section 4.3. In other words, only list use cases that actors will need to do when they are using the system to solve the customer’s problem. You can also list a set of use cases to help define the system’s scope.

*Example 4.9 List a minimal set of use cases for the following actors in a library system: Borrower, Checkout Clerk, Librarian, Accounting System.*

128

Chapter 4

**Developing requirements**

Borrower:

* Search for items by title.
* … by author.
* … by subject.
* Place a book on hold if it is on loan to somebody else.
* Check the borrower’s personal information and list of books currently borrowed.

Checkout Clerk:

* All the Borrower use cases, plus
* Check out an item for a borrower.
* Check in an item that has been returned.
* Renew an item.
* Record that a fine has been paid.
* Add a new borrower.
* Update a borrower’s personal information (address, telephone number etc.).

Librarian:

* All of the Borrower and Checkout Clerk use cases, plus
* Add a new item to the collection.
* Delete an item from the collection.
* Change the information the system has recorded about an item.

Accounting System (acting autonomously):

* Obtain the amount of overdue fines paid by borrowers.

Exercises

**E62** For the following systems list the actors and, for each of these actors, list as many use cases as you can think of.

1. A system to handle the functions of a mail-order company that manages a warehouse of goods, takes orders from customers by telephone, and ships goods overnight to customers.

Section 4.6

**Use cases: describing how the user will use the system**

**129**

1. A system to handle electronic voting. The system will allow electors to vote for a specific number of candidates, and only vote once. At the end of the voting period, the system displays the result of the vote.
2. A camping reservation system for multiple campgrounds. Campground managers register many details of their site in the system, including maps of camping locations and services available. Campers use the system to select and reserve a camping location.
3. The microwave oven system of Example 4.6.

Building a use case model

So far, we have discussed the first two steps of use case analysis, listing the actors and listing the use cases. To build a complete *use case model*, we now need to describe the use cases in more detail. A use case model consists of a set of use cases, and optional descriptions or diagrams indicating how they are related.

How to describe a single use case

The following is how we suggest you describe a complete use case. Only the

name and steps are essential – you may choose to provide a simplified use case description that omits the other components.

1. **Name**. Give a short, descriptive name to the use case. This should be a verb phrase describing the action the user will do with the system. It is also useful to include a number as a unique identifier for each use case.
2. **Actors**. List the actor or actors who can perform this use case. For example, in a library system both a borrower and a librarian can check out a book.
3. **Goals**. Explain what the actor or actors are trying to achieve. For example, in a library system, the goal of checking out a book would be to borrow the book in order to read it.
4. **Preconditions.** Describe the state of the system before the use case occurs by listing any conditions that must be true before an actor can initiate this use case. For example, to be able to check out a book, the book must be available and the client must not have any

overdue fines.

**Use case versus task analysis**

Use case analysis is similar to *task anal- ysis*.Task analysis has been traditionally used to understand and improve the efficiency of the tasks that users per- form, whether or not they are using a computer. User interface designers sometimes continue to use the term task analysis.

1. **Summary**. Summarize what occurs as the actor or actors perform the use case.
2. **Related use cases**. List use cases that may be generalizations, specializa- tions, extensions or inclusions of this one. Later, we will explain these rela- tionships.

130

Chapter 4

**Developing requirements**

1. **Steps**. Describe each step of the use case using a two-column format, with the left column showing the actions taken by the actor, and the right column

showing the system’s responses.

1. **Postconditions**. What state is the system in following the completion of this use case.

In general, a use case should cover the full sequence of steps from the beginning of a task until the end.

A use case should describe the user’s interaction with the system, not the

computations the system performs. For example in a use case for withdrawing money from an automated teller machine, you would describe the fact that the user inserts his or her card, responds to various prompts by pressing some buttons, and then removes his or her card and money. You would not describe how communication with the bank is established or how the system computes any fees it charges. The latter information is clearly important, but it belongs in a different part of the functional requirements.

A use case should also be written so as to be as independent as possible from any particular user interface design. In Example 4.10, for example, instead of writing, ‘Push the “Open…” button’ as the first steps, we write ‘Choose the “Open…” command’. The command could then be implemented as a button, a menu item, a keystroke or a voice command. Similarly, we have not specified whether the user types the file name or uses the mouse to select it from a list. Nor have we indicated what the ‘File open’ dialog looks like.

*Example 4.10 Describe in a simplified format a use case for opening a file in an application.*

**Use case:** Open file

Steps:

*Actor actions System responses*

* 1. Choose ‘Open…’ command. 2. Display ‘File open’ dialog.

1. Specify filename.
2. Confirm selection. 5. Remove dialog from display.

*Example 4.11 Briefly describe a use case for leaving a particular automated car park (parking lot).*

**Use case:** Exit car park, paying cash

**Actors:** Car drivers

**Goals:** To leave the parking lot after having paid the amount due.

**Preconditions:** The driver must have entered the car park with his or her car, and must have picked up a ticket upon entry.

Section 4.6

**Use cases: describing how the user will use the system**

**131**

**Summary:** When a driver wishes to exit the car park, he or she must bring his or her car to the exit barrier and interact with a machine to pay the amount due.

**Related use case:** Exit car park by paying using a debit card

Steps:

*Actor actions System responses*

1. Drive to exit barrier, triggering a sensor.

2a. Detect presence of a car.

2b. Prompt driver to insert his or her card.

3. Insert ticket. 4. Display amount due.

5. Insert money into slot. 6a. Return any change owing.

6b Prompt driver to take the change (if any).

6c. Raise barrier.

1. Drive through barrier, triggering a sensor.
2. Lower barrier.

Note that we have not dealt with the case where the user has not entered enough money at step 5. We will deal with this case later.

A use case should normally include only actions in which the actor interacts with the system. For example, when developing use cases for a library system, you would not include actions such as ‘Get a book from the shelves’ or ‘Read the book’. However, if there is a manual task that must be done between two interactions with the computer, then this can be part of the use case; for example, ‘Stamp the book with the due date’ is a valid action in Example 4.12.

*Example 4.12 Describe in detail the ‘Check out an item for a borrower’ use case as performed by the checkout clerks at the circulation desk of a library. This is one of the use cases listed in Example 4.9.*

**Use case:** Check out an item for a borrower

**Actors:** Checkout clerk (regularly), chief librarian (occasionally)

**Goals:** To help the borrower to borrow the item if they are allowed, and to ensure a proper record is entered of the loan.

**Preconditions:** The borrower must have a valid card and not owe any fines. The item must have a valid barcode and not be from the reference section.

Steps:

*Actor actions System responses*

* 1. Scan item’s barcode and barcode of the borrower’s card.
  2. Display confirmation that the loan is allowed.

132

Chapter 4

**Developing requirements**

* 1. Stamp item with the due date.
  2. Confirm that the loan is to be initiated.
  3. Display confirmation that the loan has been recorded.

Exercise

**Postconditions:** The system has a record of the fact that the item is borrowed, and the date it is due.

**E63** Write use case descriptions for the following activities:

* + 1. Paying a bill at an automatic teller machine.
    2. Creating a table in a word processor.
    3. Programming a microwave oven to turn on in five hours and heat some food.
    4. Read messages in a voice mail system.
    5. Programming a thermostat to set the day and night temperatures.

Use case diagrams

Use case diagrams are UML’s notation for showing the relationships among a set of use cases and actors. They help a software engineer to convey a high-level

picture of the functionality of a system.

It is not necessary to create a use case diagram for every system or subsystem. For a small system, or a system with just one or two actors, a simple list of use cases will suffice.

As Figure 4.3 shows, there are two main symbols in use case diagrams: an actor is shown as a stick person and a use case is shown as an ellipse. Lines indicate which actors perform which use cases. You do not actually need to write the word ‘Actor’ in each actor’s name; however, we find it useful to do this when it helps prevent confusion with classes of the same names.

Exercise

**E64** For the systems of Exercise E62, draw a use case diagram that shows which actors perform which use cases.

Extension, generalization and inclusion of use cases

You may want to develop a group of distinct but related use cases. For example, when an actor interacts with a system to achieve a particular goal, he or she may select different options, perform some action repetitively, provide different inputs, or answer too slowly causing a time-out error. Each variant or repetitive

Section 4.6

**Use cases: describing how the user will use the system**

Add course offering

**133**



Register in course

Add course

Registrar Actor

Student Actor

Enter grade for course

Find information about course

Professor Actor

Figure 4.3 A simple use case diagram showing three actors and five use cases

pattern of interaction can be represented as a separate use case. Similarly, the system might have a special reaction when a file cannot be found. The use case modeler can use *extensions*, *generalizations* or *inclusions* to represent different types of relationships among use cases.

* + - * **Extensions** are used to make *optional* interactions explicit or to handle

*exceptional* cases. A use case extension can, for example, describe what happens

if an actor provides a wrong filename to access a given file or describes the

extra interaction that occurs if the actor decides to browse in order to locate the

required file instead of simply typing a filename. By creating separate extension use cases, the description of the basic use case remains simple. In the extension, you should indicate which step is the *extension point* – the point at which the extension changes the basic sequence.

* + - * **Generalizations** work the same way as in a class diagram and use the same triangle symbol: several similar use cases can be shown along with a common generalized use case. In Example 4,13, the general ‘open file’ use case has two sub use cases: ‘open file by typing name’, and ‘open file by browsing’.
      * **Inclusions** allow you to express a *part* of a use case so that you can capture commonality between several different use cases. Even very different use cases can share a sequence of actions. For example, many different use cases might require an actor to specify a password, to browse through a list of items, or to open a file. Rather than repeating the details of such common interactions in multiple use cases, you can create a special use case that will be included in other use cases. Such a use case represents the performing of a lower-level task with a lower-level goal.

In practice, it can be difficult to decide whether to use specialization or extension. Although it is worth trying to understand the distinction, it is not worth wasting time in any particular model if you have trouble choosing between these constructs.

134

Chapter 4

**Developing requirements**

*Example 4.13 Describe related use cases that have to do with opening a file in an application.*

**Use case:** Open file

Related use cases:

**Generalization of**:

|  |  |  |
| --- | --- | --- |
| ❏ | open file by typing name | |
| ❏ | open file by browsing |  |

**Steps**:

*Actor actions System responses*

1. Choose ‘Open…’ command. 2. Display ‘File open’ dialog.

3. Specify filename.

4. Confirm selection. 5. Remove dialog from display.

**Use case:** Open file by typing name

Related use cases:

**Generalization:** Open file

**Steps**:

*Actor actions System responses*

1. Choose ‘Open…’ command. 2. Display ‘File open’ dialog. 3a. Select text field.

3b. Type file name.

4. Click ‘Open’. 5. Remove dialog from display.

**Use case:** Open file by browsing

Related use cases:

**Generalization:** Open file

**Includes:** Browse for file

**Steps**:

*Actor actions System responses*

1. Choose ‘Open…’ command. 2. Display ‘File open’ dialog.

1. Browse for file (included use case).
2. Confirm selection. 5. Remove dialog from display.

Section 4.6

**Use cases: describing how the user will use the system**

**135**

**Use case:** Attempt to open file that does not exist

Related use cases:

**Extension of:** Open file by typing name (extension point: step 4: Click ‘Open’)

*Actor actions System responses*

4. Click ‘Open’. 4a. Indicate that file does not exist. 4b. Correct the file name.

4c. Click ‘Open’. 5. Remove dialog from display.

**Use case:** Browse for file (inclusion)

Steps:

*Actor actions System responses*

1. If the desired file is not displayed, select a directory.
2. Repeat step 1 until the desired file is displayed.
3. Select a file.
4. Display directory.

Exercises

The graphical notation for showing extension, generalization and inclusion is illustrated in Figure 4.4. The open triangle points to a generalization. The «extend» and «include» stereotypes show the other relationships between use cases.

Note that actors can also be arranged in a generalization hierarchy. In Figure 4.4, ‘System Administrator’ is a sub-actor of ‘Ordinary User’. This means that all System Administrators can also act as ordinary users, and do such things as open files.

**E65** Write the following use cases, which are related to the ‘Exit car park, paying cash’ use case of Example 4.11.

* 1. Exit car park by paying using a debit card.
  2. Attempt to exit car park without initially entering enough money.
  3. Exit car park.

**E66** Draw a use case diagram showing the relationships among the use cases of the last exercise.

**E67** Create a complete use case model for the systems you worked on in Exercises

E55 to E57.

136

Chapter 4

**Developing requirements**

Open file



Ordinary User

Open file by typing name

Open file by browsing

«extend» «include»



System

Administrator

Attempt to open file that does not exist

Browse for file

Figure 4.4 Extension, generalization and inclusion in a use case diagram

The modeling processes: choosing use cases on which to focus

You should identify all the use cases associated with the software product, but you have to put varying emphasis on them when actually developing the system. Often one use case (or a very small number) can be identified as *central* to the system. For example, in an airline reservation system, the central use case will be ‘Reserve a seat on a flight’. Once identified, the system can be built around this particular use case.

There are also other reasons for focusing on particular use cases:

* + - You may identify some use cases as *high risk* because you expect to have problems implementing them. For example, in the GPS-based Automobile Navigation Assistant (GANA) described in Section 4.11, you may identify ‘Enter navigation mode’ and ‘Speak now’ as high risk. These use cases will require exploratory prototypes, since it will be challenging to ensure that the voice output is easy to understand and is spoken to the user at the right time.
    - Some use cases will have high political or commercial value. For example, in an online stock exchange system, a use case in which the user can see, in real time, the evolution of the value of a given stock would not be essential – the users could do their jobs without it. Nevertheless, it might be given high priority because of its appeal when presenting the product prototype to potential clients.

Exercise

**E68** For the following systems create a list of use cases and describe each of them in detail. Then identify the central ones, and any other ones that should be given high priority.

(a) A web-application to pay bills from your bank account.