Chapter 3. Requirements Engineering

Table of Contents

Objectives 1
Requirements engineering 1
What is requirements engineering? 1
The steps in detail 2
- Inception 2
- Elicitation 3
- Elaboration 3
- Negotiation 3
- Specification 4
- Validation 4
- Management 4

Use case modeling 4
- Use case modeling in the UML specification 4

Review 18
- Questions 18
- Answers 19

Objectives

At the end of this chapter you should be able to:

- Explain the need for requirements engineering.
- Give a series of steps for use in performing requirements engineering.
- Create and interpret use case models.

Requirements engineering

The first two activities in the generic process framework is that of *communication* and *modelling*. A large portion of these activities are concerned with discovering the requirements of the software which the customer is asking to have developed. This chapter deals with this process of *requirements engineering*.

Note

As with all other activities in a process model, requirements engineering should be tailored to fit the developers creating the software, the product being created, and the overall process model being employed. In the previous chapter you could already see this happening with the *extreme programming* software process model, which limits modelling to the creation of CRC cards (see ???).

Requirements engineering is concerned with understanding the software system that the customer has requested. It provides the base on which software design and programming can proceed. Importantly, if the developers do not adequately understand the requirements, it is very likely that the software will not meet the customer's needs. This makes understanding the customer's requirements important to the success of the development project.

What is requirements engineering?

At its most essential, requirements engineering is focused on discovering *what* it is that should be developed (and not *how* it should be developed). There are a number of aspects to this:

- What does the customer want?
- What does the user require in order to use the system?
- What will the software's impact on the users be?

To discover this information, requirements engineering contains a number of overlapping steps:

1. **Inception**, in which the nature and scope of the system is defined.
2. **Elicitation**, in which the requirements for the software are initially gathered.
3. **Elaboration**, in which the gathered requirements are refined.
4. **Negotiation**, in which the priorities of each requirement is determined, the essential requirements are noted, and, importantly, conflicts between the requirements are resolved.
5. **Specification**, in which the requirements are gathered into a single product, being the result of the requirements engineering.
6. **Validation**, in which the quality of the requirements (i.e., are they unambiguous, consistent, complete, etc.), and the developer's interpretation of them, are assessed.
7. **Management**, in which the changes that the requirements must undergo during the project's lifetime are managed.

Requirements engineering will usually result in one or more *work products* being produced. These products, taken together, represent the software's *specification* (see the specification step previously mentioned, and detailed below). These work products, however, do not have to be formal, written documents — indeed, the work products can be a set of models, a formal mathematical specification, a collection of use cases or user stories, or even a software prototype.

Note

These steps are *overlapping* for a variety of reasons. You should be able to notice that some of them, such as management, must occur throughout the *communication* and *modelling* activities. Negotiation will also occur at each of the various requirements engineering steps. More importantly, a process model which understands that requirements may be (initially) poorly understood, and that they may change through the project's lifetime, will also iteratively collect and detail the use cases (consider the unified process model from the previous chapter). This iterative process will forcefully overlap all of these steps.

The steps in detail

Inception

The requirements engineering process begins by examining the problem which the software should solve and gaining an understanding of both the problem's nature and the nature of the desired solution. This should be done in a context-free manner, that is, a manner which does not presume to know anything concerning the problem, the customers, the users, and the requested solution. The following questions may be asked:

- Who is requesting the software?
- Who will use the software?
- What is the benefit that the software will bring?

It is important to identify the **stakeholders** in the project. Stakeholders are the people who will find benefit in the project and the software being developed. They may include:

- Customers
- End users
- Business operations managers
- Product managers
- Advertising and marketing staff
- Software engineers
- Support engineers

Each of the stakeholders will have a different view on what the software product should do and on what the software engineers should focus on. This might be on creating “sexy” features (from the marketing department), staying within budgets and deadlines (from managers), maintainability (from support engineers) and so on. Out of these views, the requirements engineer should determine which requirements there are a consensus on, and on which requirements the stakeholders disagree. Resolving disagreements between stakeholders makes up the negotiation step.

Something to consider during inception is the effectiveness of the communication between the requirements engineer and the customers. This may be done by, for example, asking the customer if they feel that they have been asked appropriate questions concerning their problems, and if the person communicating with the requirements engineer feels that they are (or are not) the person who should be answering the engineer's questions.

Elicitation

This step is concerned with identifying the overall problem the software is attempting to solve, proposing solutions, negotiating between the differing approaches to solving the problem, and finally specifying a basic set of requirements.

This can be done by calling a meeting between all of the stakeholders. It is important to nominate someone to act as a *facilitator*, who will guide the meeting. Each of the attendees should bring to the meeting a list of:

- objects that make up the system's operating environment
- objects used by the system (such as those things which make up the input to the system)
- objects produced by the system
- the services that interact with these objects
- various constraints, such as time and budget constraints, interoperability constraints, performance restraints, usability constraints, and so on.

Elaboration

This step involves expanding on the requirements defined in the previous two steps, and from these requirements producing an *analysis model*, which is a technical model of the software and its functions.

The construction of analysis models will be discussed in detail in the following two chapters.

Negotiation

This step involves negotiating between the various stakeholders in order to remove any conflict in the requirements. A useful technique for resolving these conflicting requirements is to provide each of the stakeholders with a finite number of *priority points*. They may then allocate points between the conflicting requirements as they see fit. The overall importance of any requirement can then be determined by the number of priority points that it has received.

Specification

The specification step produces the final product of the requirements engineering process. It describes the software, both its functions and constraints. The specification need not be a written document, but could also be a graphical model (such as those produced using the UML), software prototype or formal model, or a collection of these.

Validation

This step is concerned with ensuring that the gathered requirements in the software specification meet certain standards of quality. For example, have the requirements been written to the proper level of abstraction, or do they provide too much technical detail for the given stage of development? Is the requirement necessary, or something not essential to the software? Are the requirements unambiguous? Do requirements contradict other requirements?

A useful action during validation is to ensure that each requirement has a source attributed to it. In this way, if more information is required, the requirements engineers know who to contact.

Management

Requirements change over time; requirements management is concerned with controlling and tracking change in the requirements.

Requirements management proceeds by associating requirements with various aspects of the software engineering process. As these aspects are changed, the requirements associated with them can be easily identified and changed. As these requirements are changed, all aspects of development associated with the modified requirements can be examined, and in this way the changes can more easily be propagated through the project.

Such an association between requirements and aspects of the project can be done using a table: each row in the table represents a specific requirement, each column an aspect of the software project. The entries mark whether a requirement is associated with that aspect.

Use case modeling

Use case modelling is a useful tool for requirements elicitation. It provides a graphical representation of the software system's requirements.

The key elements in a use case model are **actors** (external entities), and the **use cases** themselves. In outline, a use case is a unit of functionality (a requirement), or a service, in the system. A use case is not a process, or program, or function.

Because use case models are simple both in concept and appearance, it is relatively easy to discuss the correctness of a use case model with a non-technical person (such as a customer).

Use case modeling effectively became a practicable analysis technique with the publication of Ivar Jacobson's (1991) book “Object-oriented software engineering: a use case driven approach”. Jacobson has continued to promote this approach to system analysis to the present day, and it has now been formalised as part of the UML. However, use case modeling is not very different in its purpose and strategy from earlier techniques, such as structured viewpoint analysis.

Use case modeling in the UML specification

The *Unified Modeling Language* (UML) represents a deliberate attempt to standardise the modeling notation used in software engineering, particularly object-oriented development. The widespread uptake of the UML is a result largely of two factors. First, it is driven by some of the most influential proponents of object-oriented development, including James Rumbaugh, Grady Booch, and Ivar Jacobson. Second, it has broad support from major business concerns in the software industry, including Microsoft, IBM, Hewlett-Packard and Oracle.

The notation specified for use case modeling by the UML is not very different from that originally proposed by Jacobson, so early books and articles on use case modeling that follow the Jacobson strategy are still useful reading.

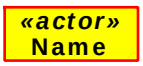
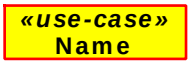
It is only fair to point out that not all experts support the UML effort, and it comes under regular and harsh criticism, some of it fair. For example, one criticism is that there is not good enough integration between the different components of the UML (e.g., between use case and class modeling). No doubt this will improve in time. In due course you will be able to make your own judgement on this issue, but is important to keep sight of the fact that the UML is an international standard for software modeling, and any software professional needs to understand it.

The UML is under continuous development, and at the time of writing the latest version is 2.2. The definitive reference for the UML notation is the UML specification, which is available from the Object Management Group's Web site [http://www.omg.org/technology/documents/formal/uml.htm]. However, while this is an authoritative document about the UML, it is not a good document from which to learn about the UML.

It is important to understand that the UML is a specification for a modeling language. It is most emphatically not a software design methodology. Although the UML states the symbols that are to be used in use case modeling, and how they are to be interpreted, it does not say when, or even if, use case modeling should be applied. We shall have more to say about this later.

Use case modeling symbols

This section presents an overview of the symbols used in use case modeling; the important ones will all be discussed in detail later.

Symbol	Name	Interpretation
 or 	Actor	An entity (human or otherwise) external to the system, and which interacts with it
 or 	Use case	A service or unit of functionality
	System boundary	Indicates the division between the system being designed and the rest of the world
	Communication association	The line indicates that a particular use case is associated with a particular actor. The name

Symbol	Name	Interpretation
		is optional and often omitted. An arrow can also optionally be used; where present it does *not* indicate a flow of information (such as in a data flow diagram)
	Use case association	Indicates that to use cases are related in a particular way, e.g., the one use case's behaviour includes the behaviour of another use case
«name»	Stereotype	Indicates that the symbol it is attached to belongs to a particular category
	Generalisation	Indicates that the two symbols it connects are related by a generalisation-specialisation relationship. For example, one actor is a sub-type of another, or one use case is a type of another. Both the name and the stereotype are option
	Note	The designer may, and should, qualify any part of the model with a textual note if it improves the clarity of the design

Note

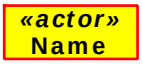
The above images were created using the Umbrello Software package [http://uml.sourceforge.net].

Actors

An actor is any entity, human or otherwise, that is external to the system being designed. Two symbols are available in the UML specification:

Figure 3.1. Actor representations

An actor can be represented by a stick figure.



Alternatively, an actor may be represented by a class with the «actor» stereotype.

The “stick figure” symbol is the more expressive, but can lead to confusion if the actor is not in fact a human but a machine. The rectangle symbol is the standard UML symbol for a class. What this symbol says is that the entity is a class that is a member of the category “use case”. The reason why an actor is a type of class will be discussed later. Because of its greater expressive power (that is, an ability to make a more immediate impression on the viewer) it is probably best to use the stick person figure where possible.

Since use cases are technically classes, and by convention class names start with a capital letter, names of use cases should also begin with a capital letter. The UML specification does not insist on this, but it is common practice.

Two interesting philosophical issues are associated with actors. First, should the actor be a person, or the part of the computer with which they interact? For example would we ever want an actor called “Keyboard” or “Printer”? In designing a word processor, for instance, there is only one human user and they fulfil all the roles within the system. But you may wish to distinguish between, say keyboard, mouse, and printer actions. However, distinguishing between different pieces of hardware is probably inappropriate at this level, and it could be argued anyway that use case modelling is not a helpful way to begin the design of a word processor. On the other hand, a supermarket stock control system may accept input from a bar-code scanner at the checkout, when the cashier registers the prices of the customer's purchases. Is the actor here the cashier, or the bar-code scanner? Perhaps neither is appropriate, as the cashier can probably enter the identities of products manually if the bar-code scanner does not work. Do we want to distinguish between different techniques for entering item details at the highest level of analysis? Probably not. In this case we could perhaps invent a more abstract actor (called, perhaps, “point-of-sale”) that provides the key message: that something at the point-of-sale interacts with the system whenever a purchase is made. In summary, then, when a human being interacts with a computer system, it is not necessarily the case that the human is identified as the actor. It is often clearer to use a non-specific entity as the actor.

The second philosophical issue is concerned with whether an actor has to be an active participant in an interaction. For example, in a computer-based building security system, do we want “Burglar” as an actor? A burglar does not actively interact with the system; indeed they would probably rather not interact with it at all. Moreover, a burglar may interact with the system in a number of different ways, all passively. In this case, it may well be better to identify the sensor devices as actors. What about the fire alarm? Presumably “Fire Detector” is a better actor than “Fire”?

The UML gives us no guidance on these issues; although the letter of the UML specification is that actors can be any external entity, the spirit of the standard seems to be that actors are people who interact with the system in a rich, complex way. There is little concern for “trivial” actors like printers and fire detectors.

These complications notwithstanding, actors do not have to be human beings; they may be external computer systems. For example, we may be designing a system for allocating staff work timetables that draws information from a central record of employees which is part of the payroll system. In this case, “Payroll System” would be an actor.

You should bear in mind that you will never implement an actor; by definition an actor is external to your system. You will, however, implement the interfaces used by actors to interact with the system.

Actors as roles

A decision that the use case modeller has to make is whether to treat (human) actors as expressions of a person, or as expressions of a role within an organisation. Perhaps an example will help to clarify this issue. Suppose that you are designing a computer system to automate the operations of a large library. The system should maintain the library catalogue, provide information to staff and readers, check books in and out, provide guidance on re-shelving returned books, manage inter-library loans, warn users about overdue books, and manage collection of fines and subscriptions.

Most people faced with this problem begin by considering the types of people who will use the system. Two such groups come to mind immediately: library users and library staff. In a library one will often find that staff members have very general job descriptions, and will take a hand in most of the normal operations of the library. We will refer to these people as “librarians”. So far our use case model has two actors: Librarian and User, and these two actors interact with all the use cases that we could identify. In this case we are treating “Librarian” as an expression of a person. We could get a lot of information about what services the computer system should provide by reading the job description of a librarian.

The problem with this approach is that it has no high-level structure, and is therefore not very expressive. We could convey much the same information as the use case model by merely writing a list of library services. There is no scope for simplification and management of the model by generalisation (see later). Furthermore, it carries the implication that librarians are interchangeable, and any librarian can do the work of any other. Even if this is true at the time the system is designed, it may not continue to be true. For example, more junior staff members may not have authority within the system to order new books for the library. A person managing the stock of books is interacting with the system in a totally different way to the person who is, for example, signing up new library users. The fact that these to job functions may perhaps be carried out by the same physical human being is irrelevant; they are totally different roles within the system, and should be considered to be different actors.

Of course, we could now go to the opposite extreme and create a new actor for every service the system provides, but this leap from the frying pan into the fire would still result in a model with no structure that is difficult to simplify. In addition, there may be a loss of expressiveness. For example, in the library system it is quite likely that both staff and users can browse the catalogue of books. They will quite possibly do this in an identical way, and see exactly the same information. Moreover, this activity is completely separate from any other functionality the system may provide. So we may be tempted to create a new actor (perhaps called “Catalogue Browser”) to model this role within the system. But we will have lost a very important piece of information by doing this: the fact that both staff and users can browse the catalogue implies that we have to put computer terminals or workstations on both sides of the counter, so to speak.

So what is the correct approach? In short it is the one which leads to the simplest correct model which is still adequately expressive. Of course it is more important that the model is correct than that it is simple. To arrive at this point may require that your choice of actors be modified several times during the course of construction of the use case model.

Identifying actors

In the last section we considered the distinction between actors as people and actors as roles, and clearly this is an issue that needs to be taken seriously by the requirements engineer. However, it only becomes an issue when we know enough about the way the system is to be used to have a potential set of actors to hand. We now have to consider the situation where the analyst knows nothing at all about the system to be developed, and doesn't have the first idea what the actors are. Many development jobs begin like this.

The analyst's first recourse is to the stakeholders identified through the earlier requirements engineering steps. The stakeholders should certainly be on the initial list of actors.

You may then ask what information is to be manipulated by the system, and where that information will come from. All information that is not present in the system at the instant it begins work will be coming from an actor of some sort. If there is not enough information to do this, then this is a sign that further information must be elicited from the customers.

Other potential actors include all the computer systems with which the system will interact, and any other hardware devices (including hardware such as printers, bar-code readers, and so on).

There are also various “standard actors” that all systems of any complexity are going to need, and which may not have been considered by the clients. These include the person (or rather the role) which maintains the system after it has been put into service, the person who performs software upgrades and tests, the person who carries out and checks backups (if not automatic), and so on.

When developing a software system to replace a manual procedure, or an older software system, actors may be identified by watching people go about their daily work, and by speaking to people who are experts on the business the system has to carry out.

It is probably not possible to identify actors without some consideration of the services that the system provides. In practice design work will switch between consideration of the actors and consideration of the use cases.

Individual actors and classes of actors

All of the forgoing was, we hope, largely common sense. Now it is time discuss a technicality. We will state the principle, and then go on to explain it.

The principle is this: the UML specifies that an actor in a use case is a class, and individuals are instances of that class.

The actors in a use case model do not represent individuals (individual humans or individual computers), but classes of individuals. For example, an actor called “Customer” models all those properties that customers have in common; it is, in effect, the class of all customers. It will be assumed by anybody who reads your model that the ways in which the Customer actor interacts with your system will apply to all customers. So if some customer or group of customers is expected to behave differently, then we need a different actor to handle this case.

This may seem trivially obvious, but the complication is that the above principle is true even if there is only one instance of an actor. For example, a company may have only one managing director, and may be constrained by law only to have one, but if the managing director interacts with the system you are designing in a specific way, then they are a class, not an individual. The problem is that although the designer understands intellectually that actors are classes, they may still have a specific individual in mind, leading to a bad model. The reason is this: suppose Joe Bloggs is a bank clerk; there can be different types of bank clerk, but it is meaningless to talk of different types of Joe Bloggs. And the ability to simplify a model by identifying where one thing is a type of another thing is a key feature of use case modeling, and indeed of all types of object-oriented design.

System boundary

The system boundary demarcates the system being designed from the rest of the world. It is denoted simply by a box with the name of the system in the corner.

Figure 3.2. The system boundary

Use cases are inside the box, actors are outside. Because different symbols appear inside the box and outside, in practice it is usually unnecessary to show the system boundary explicitly. One circumstance in which it is necessary to show it is if you wish to show the use case models of two or more different systems on the same diagram. This may be necessary if you are designing a system which interacts with another in a way that is too complex for you to show the external system as simply an actor.

Some modeling software will draw the system boundary automatically.

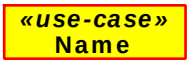
In the design of a large, complex system, it is said that the boundary may “move” during the design operation. In fact what is happening is that decisions are being made about what functionality is the responsibility of the current design exercise, and what is not. For example, it may turn out that during the analysis of a staff payroll system, management of staff health records — which was previously the responsibility of another system — is seen to be more appropriately part of the payroll system. In effect, the system boundary has “moved” to encompass part of another system. Needless to say, such movement needs to be settled as early in the development process as possible.

Use cases

In the UML, a use case can be represented in two different ways:

Figure 3.3. Representations of use cases

Use cases may be represented by a title within an ellipse.



Use cases may be represented by a class with the «use case» stereotype.

Most designers will immediately recognise the oval as a use case (the oval symbol has no other meaning in the UML) so this symbol is to be preferred where available. The other symbol reflects the fact use cases are technically classes (just as actors are), whose category is “use case”. We will discuss later what is meant by a class of use cases.

Although use cases are central to use case modeling — and indeed to many object-oriented development strategies — there is surprisingly little general agreement on what a use case is. In fact, there are people currently working on doctoral theses whose subject is what a use case is supposed to be showing. The following definition is taken from version 0.8 of the UML specification:

> A use case is a generic description of an entire transaction involving several objects.
>
> —UML 0.8

This rather terse definition was all the UML had to say about the nature of use cases at that point. In version 1.3, there is the following, somewhat expanded, definition:

> The purpose of a use case is to define a piece of behaviour of an entity without revealing the internal structure of the entity. The entity specified in this way may be a system or any model element that contains behaviour, like a subsystem or a class, in a model of a system. Each use case specifies a service the entity provides to its users, i.e. a specific way of using the entity. The service, which is initiated by a user, is a complete sequence. This implies that after its performance the entity will in general be in a state in which the sequence can be initiated again. A use case describes the interactions between the users and the entity as well as the responses performed by the entity, as these responses are perceived from the outside of the entity . A use case also includes possible variants of this sequence, e.g. alternative sequences, exceptional behaviour, error handling etc. The complete set of use cases specifies all different ways to use the entity, i.e. all behaviour of the entity is expressed by its use cases.
>
> —UML 1.3

Versions 2.2 states:

> A use case is the specification of a set of actions performed by a system, which yields an observable result that is, typically, of value for one or more actors or other stakeholders of the system.
>
> —UML 2.2

In fact, these are very general descriptions, and not particularly helpful to the practitioner; here are some more pragmatic views of the use case.

Use cases as services

In this view, each use case provides a particular service to the users of the system. Examples of services may include withdrawing money from an account, printing a report, reserving a theatre ticket, and

ordering a book. Every time an actor interacts with a use case a transaction occurs; this transaction takes time and may have a number of alternative paths.

Use cases as business processes

This view of use cases seems to be favoured by people with an interest in business process modeling. In this view, a use case is an activity of the business, such as ordering stock, issuing a check to clear an account, or checking a customer's credit-worthiness. Superficially this view is not that different from the view of use cases as services, but the implication here is that use cases are best characterised by a discrete series of actions, perhaps illustrated by a flowchart (the UML equivalent of a flowchart is an activity diagram; expanding use cases into activity diagrams is quite common practice).

Use cases as increments of functionality

In this view, adding a use case to the system is equivalent to adding some extra functionality, with the proviso that the new functionality be largely self-contained.

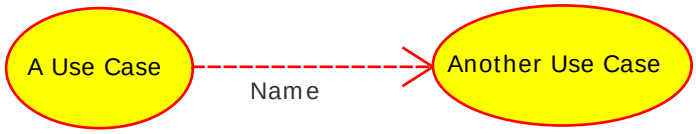
These views are not mutually exclusive, and all are compatible with the UML definition. However, the view that the designer adopts will subtly influence the character of the model produced.

Whichever view of you take of use cases, it is important to remember that a use case is a thing, not an event or a process. It is not uncommon for novices to generate use cases with names like “print” or “mouse moved”. “Print” is a process or action. It may be a valid use case, if you mean by “print”, “the facility to allow a user to obtain a printed output”. Some experts recommend that you force the names of use cases to be nouns to reinforce this meaning. In this case “Print Service” or “Print Facility” might be better names. “moused moved” is an event, a thing that happens to the system.

Relationships between use cases

It is possible, and usual, to show that one use case is related to another. The standard notation for this is a dashed line:

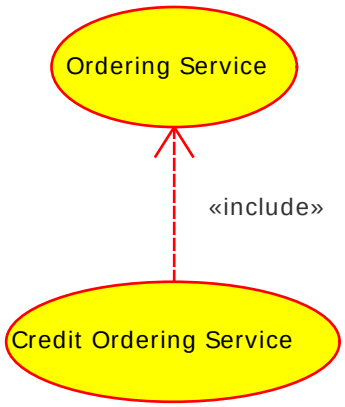
Figure 3.4. Use case association



An association between use cases is typically shown with a dotted line.

A stereotype can also, optionally, be given to the relationship. This indicates the type of relationship, of which the most important are «include» and «extend». An example of an «include» relationship is show below.

Figure 3.5. Use of stereotypes in use case relationships



In this example, “Credit Ordering Service” includes “Ordering Service”, that is, all the behaviour of “Ordering Service” will be invoked whenever “Credit Ordering Service” is invoked. This is sensible if ordering something on credit is the same as ordering something with payment, plus a bit extra.

Typography

Stereotypes in the UML are enclosed in guillemets: « and ». Often, if the typist or application is unable to represent these symbols, << and >> (two less-than or greater-than signs) are used instead. However, now that Unicode display is available on most operating systems, the guillemets are to be preferred. The Unicode code points for « is u+00AB, and » is u+00BB. Your operating system will no doubt have an easier input method available to you than entering Unicode code points directly.

In the UML, «extend» is similar to «include», with the distinction that with «extend» the behaviour of the extending use case will not always be invoked.

A note on terminology and semantics

The UML uses the term «extend» here because Jacobson used this term in his earlier work on use case modeling. Unfortunately, many authors use the term “extend” to mean a generalisation relationship — in other words, that one thing is a type of another thing. This is particularly relevant in Java programming, where the word “extend” has this (generalisation) sense, and not the sense defined in the UML. To further complicate matters, the «include» relationship does denote a kind of generalisation: a “credit ordering service” is a type of “ordering service”. These complications have little impact on use case modeling or on programming in practice, but they do cause confusion among students (and others), and it is good to be aware of them.

Describing and specifying use cases

A use case diagram on its own may well be a useful method for describing the large-scale structure of a system; however, it is of very limited use as the input to a more detailed design operation. In practice it is necessary to describe use cases in a more detailed fashion if the model is to be interpreted properly.

Such a description is not only an aid to communication, it is an aid to *validation*. **Validation** is the process of ensuring that a specification is correct, in contrast to **verification**, which is the process of ensuring the product meets its specification (see ???). If the analyst cannot describe a use case in a way that makes sense to someone else, then one of two things has happened:

- The analyst has made an error: the use case is not valid and should be removed and its functionality placed elsewhere;
- The analyst has insufficient information about the system being developed.

Either case needs to be corrected.

How do we describe use cases? Two approaches are in widespread use.

- Plain text. This is the approach recommended by the UML specification. Normally a few paragraphs of text should be adequate. Most experts recommend that the text used should be in language terms which the customer uses; that is, a system for controlling a steel mill should be written in the language of a steel producer, not a software engineer, even though the latter may allow greater technical accuracy. The reason for this is simple: only the clients can confirm that the use case model is correct, and that they meet the requirements of their application. While the analyst can be sure that the model is logically consistent, and can be implemented, this is not good enough grounds for proceeding with development.
- Using a sub-model. By this we mean using a further, more detailed model, to clarify the use case. Ultimately, as we shall see, a use case will be expanded into a *collaboration diagram*, that is, an interacting group of classes. However, this does not describe the use case in the sense meant above;

this will probably not help customers to determine whether the model represents a system that meets their needs. A common choice of a sub-model is the *activity diagram*. This diagram is a relative newcomer to the UML, and is not very different from a traditional flowchart. It shows the sequence of steps that are carried out when a use case/actor transaction occurs, and can show alternative sequences of operations which are selected according to some condition of the system. The use of activity diagrams is beyond the scope of this chapter, but should be explained in any UML textbook and, of course, in the UML specification.

Individual use cases and use case classes

Use cases are technically classes. Thus a use case does not represent a particular delivery of a service, or use of some functionality, but *all* conceivable deliveries of the service.

If a use case is a class, then the individuals / instances are the specific cases of a transaction occurring between the use case and one or more actors.

If all interactions between the actor(s) and the use case are identical, then the distinction between the use case class and its individual instances is not important to the analyst. It becomes important when a use case can behave in many different ways, that is, the individual instances of the use case class are different. If these individuals are important enough to be documented, they are called scenarios.

For example, suppose we are developing a computer system that allows people to place credit-card orders for our customers' products using a Web browser. We have identified a use case called “Place Order”, which represents an ordering transaction between the customer and the system. This use case has very complex behaviour, because there are many things that can go wrong while placing the order. For example, the credit card company may not authorise the funds transfer, the credit card number may be invalid, the connection to the credit card company may be out of service, and so on. In all cases the use case must exhibit behaviour that tries to recover from these errors. For example, if the user enters an invalid card number, they must be given the opportunity to correct it and try again.

For the purpose of documenting the use case we may use a model like an activity diagram to show the general behaviour, but support it by describing a number of possible complete transactions, where different errors are encountered and corrected. These descriptions are the scenarios of the use case. Including scenarios for complex use cases helps the analyst to be sure that the use case is properly specified, because the clients will be able to understand the scenarios (which are in plain language), and the analyst and their colleagues will be able to check that the scenarios are compatible with the general pattern of behaviour specified for the use case.

The concept of generalisation

Generalisation is one of the most important concepts in use case modeling, and indeed in object orientation in general. When stated as a principle it appears trivially obvious, but it has profound implications. The principle is this:

Generalisation

Entity A is a **generalisation** of entities B, C... if the behaviour, attributes and associations of B, C... are found in A, and no properties of A are absent from B, C...

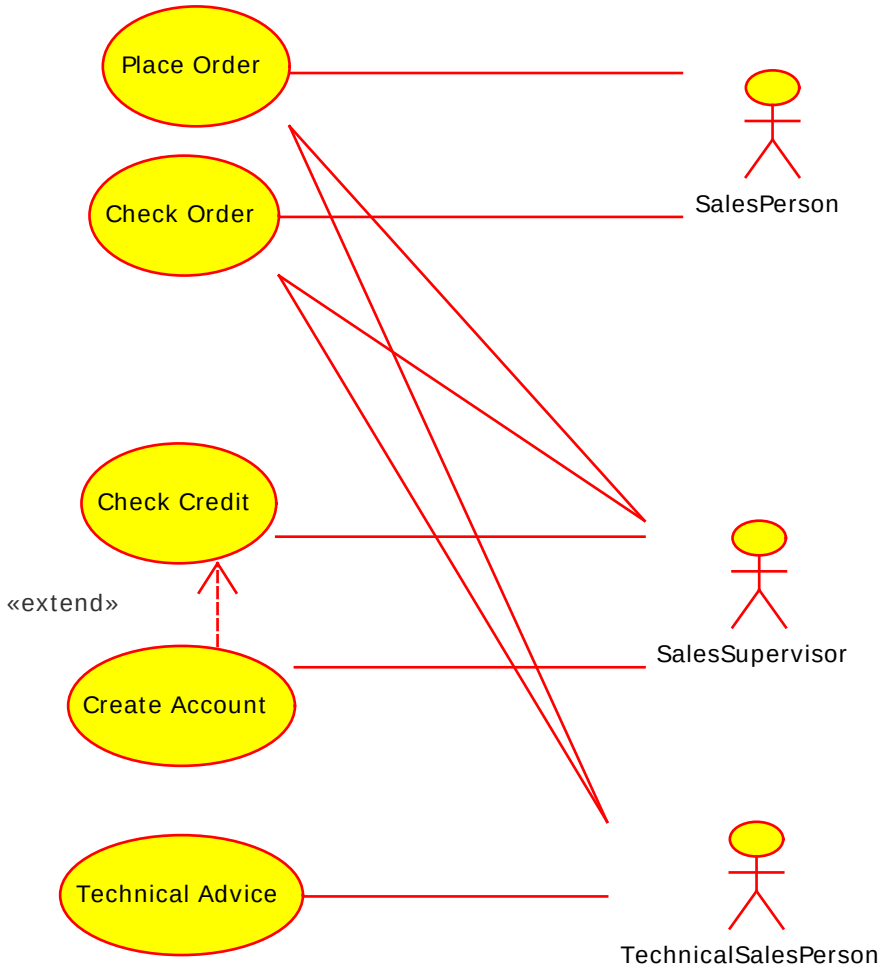
We say that entities B, C, and so on, *inherit* the properties of A. This can also be stated as: A is a generalisation of B and C if B and C are *types* of A (we could also say *sub-types* or *sub-classes*); A is a generalisation of B and C if B and C *extend* the functionality of A, while retaining all of A's functionality

Here is trivial example: Dog and Cat are types of Mammal, because all dogs and cats share the basic properties of mammals (e.g., having fur, being warm-blooded, having four limbs). We may say that Mammal is a generalisation of Dog and Cat; alternatively we could say Dog and Cat are specialisations of Mammal, or types of Mammal, or sub-classes of Mammal. Note that when we say that Mammal is a generalisation of Dog and Cat we are not saying anything about whether there are other types of Mammal; there may be, but we haven not specified any.

Another example: A business wishes to automate some of its sales procedures. Preliminary interviews reveal that there are a number of staff roles in the Sales department. A salesperson can place orders on behalf of customers and check the status of these orders. A technical salesperson has the same duties, but additionally is able to provide customers with detailed technical advice (which we would not expect an ordinary salesperson to be able to do). A sales supervisor is a salesperson, with the additional responsibility of creating new customer accounts and checking their credit-worthiness. It is reasonable to assume that Salesperson is a Generalisation of Technical Salesperson and Sales Supervisor, because the technical salesperson and sales supervisor have all the properties of a salesperson, and some extra.

We can construct an outline use case model to show these relationships. We will assume for the sake of simplicity that the use cases are “Place Order”, “Check Order”, “Create Account”, “Check Credit”, and “Technical Advice”. Without generalisation, we obtain the following model:

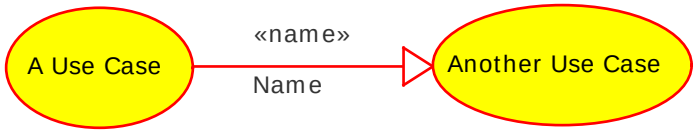
Figure 3.6. A use case example, without generalisation



While this is logically correct in that it accurately captures the information given in the text, the number of associations in the diagram makes it difficult to read. It provides no more insight into the system than does the textual description.

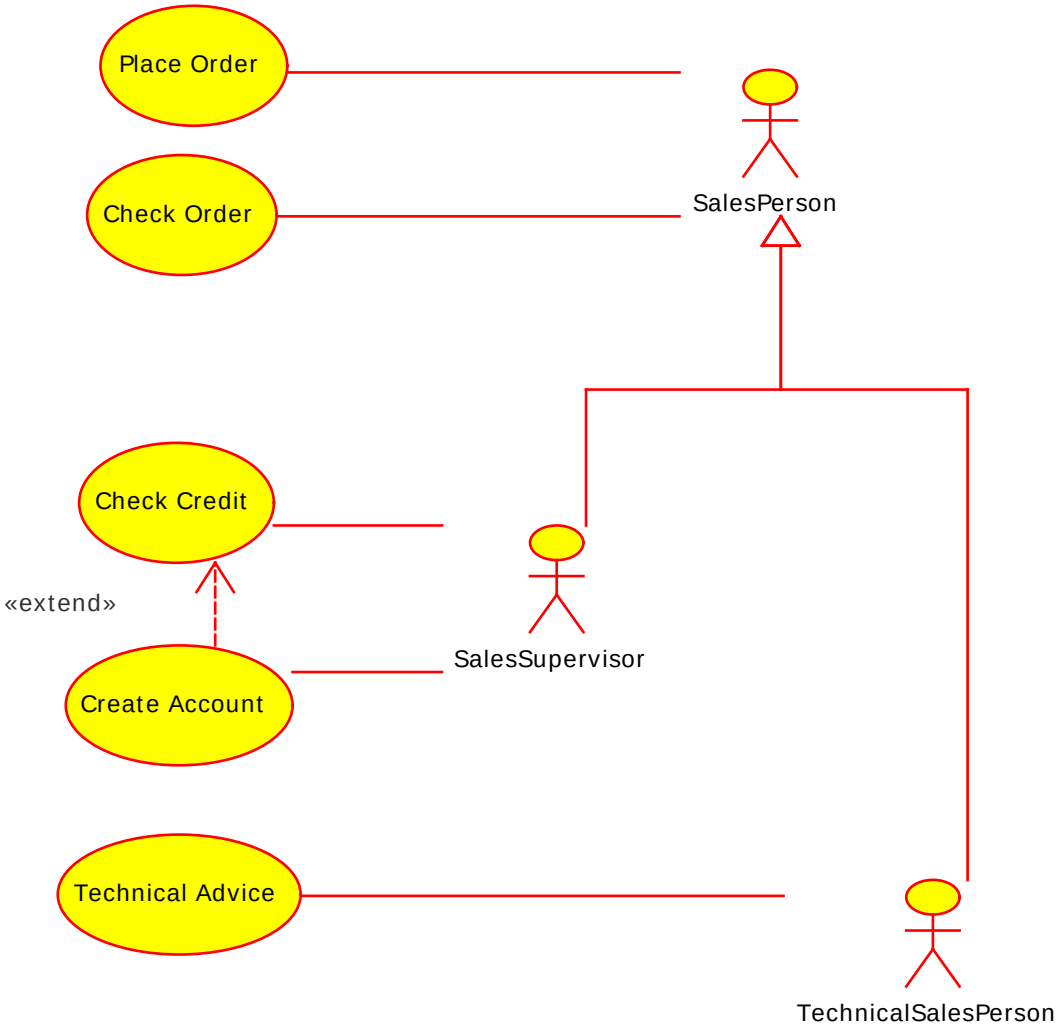
In the UML, the symbol for a generalisation is an arrow:

Figure 3.7. Use case generalisation



The arrowhead points to the more general entity. If we take account of the generalisation relationships present in the “Sales” example, we reach a model like this:

Figure 3.8. Use case example, with generalisation



This model does not have to show that “Technical Salesperson” and “Sales Supervisor” can check orders and place orders, because this is implied by their being sub-classes (specialisations) of “Salesperson”. Not only is this model easier to read, it gives a more immediate insight into the system being analysed.

You will not always be able to simplify a use case model using generalisation, but you should be on the alert for the opportunity to.

In the example above, “Salesperson” was an actor in its own right, as well as being a generalisation of other actors. That is, we would have identified “Salesperson” as an actor with or without generalisation. Sometimes, however, it is appropriate to “invent” actors simply to stand as generalisations of other actors, with the purpose of simplifying the model. These actors are referred to as “abstract”, because they abstract, or simplify, a system. An abstract actor is a special case of an abstract class. We shall have more to say about abstract classes in the next unit.

A simple example

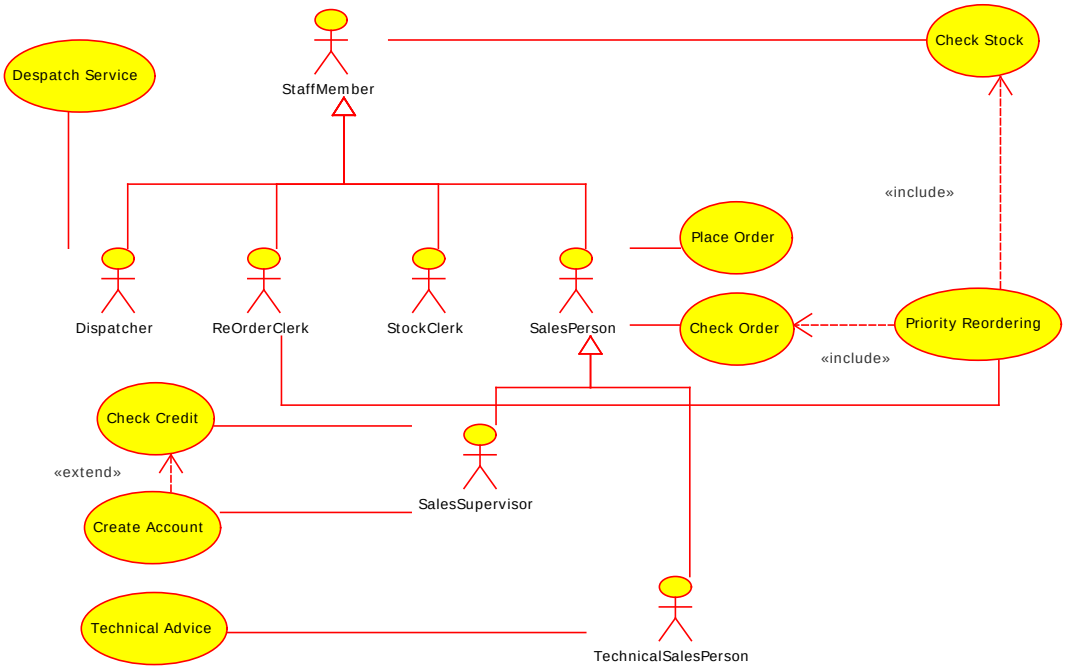
This is an example of a complete, simple use case diagram. It is based on the “Sales” example presented earlier. We will start with a description of the business, then present the use case diagram and the textual specification of the individual use cases. Note that a use case model is incomplete without this specification, either in plain text or something else.

A use case example

A retail business wishes to automate some of its sales procedures. The retailer buys items in bulk from various manufacturers and re-sells them to the public at a profit. Preliminary interviews reveal that there are number of staff roles in the Sales department. A salesperson can place orders on behalf of customers and check the status of these orders. A technical salesperson has the same duties, but additionally is able to provide customers with detailed technical advice (which we would not expect an ordinary salesperson to be able to do). A sales supervisor is a salesperson, with the additional responsibility of creating new customer accounts and checking their credit-worthiness. A dispatcher is responsible for collecting the goods ordered from the warehouse and packing them for dispatch to the customer. To assist in this operation, the computer system should be able to produce a list of unpacked orders as well as delete the orders from the list that the dispatcher has packed. All staff are able to find general details of the products stocked, including stock levels and locations in the warehouse. A re-ordering clerk is responsible for finding out which products are out of stock in the warehouse, and placing orders for these products from the manufacturers. If these products are required to satisfy an outstanding order, they are considered to be “priority” products, and are ordered first. The system should be able to advise the re-order clerk of which products are “priority” products. A stock clerk is responsible for placing items that arrive from manufacturers in their correct places in the warehouse. To do this the clerk needs to be able to find the correct warehouse location for each product from the computer system. Currently, the same person in the business plays the roles of stock clerks and re-order clerk.

Figure 3.9, “A full example”, shows the associated use case diagram.

Figure 3.9. A full example



Brief use case specifications

- **Check Stock**: Allows a user to check the levels of stock of any item held in the warehouse, and where that item is shelved. A particularly important scenario is that of obtaining a list a stock items for which the stock level is zero, that is, of which there is no stock in the warehouse.
- **Place Order**: A salesperson places an order on behalf of a client. This has the effect of making information about the order available to Dispatch Service. The order remains on the system until it has been packed and dispatched.

- **Dispatch Service**: Allows a list of outstanding orders to be obtained, and updated when orders are packed. An order is not available to this service if it cannot be satisfied because there is not enough stock in the warehouse.

- **Priority Reordering**: Obtains a list of items that need to be re-ordered urgently, as the business cannot satisfy its own orders without them. This use case makes use of Check Stock (to determine if an item is out of stock) and Check Order (to determine what stock is required to satisfy all current orders)

- **Check Order**: Obtains details about any outstanding orders, including what stock items are required to satisfy the order. This use case is used by salespeople to advise customers, and by the Priority Reorder use case to determine which items of stock must be replaced quickly.

- **Check Credit**: Used to find out whether it is safe to extend credit facilities to a client. This use case refers to external credit reference agencies (not shown).

- **Create Account**: Used to register a new customer. If a customer asks for credit facilities, this use case includes Check Credit. Otherwise it doesn't have to.

- **Technical Advice**: Provides technical specifications for selected products. Used by technical sales staff to provide advice to customers.

Some hints and warnings

Here are a few general hints, and warnings, concerning use case modeling.

- It will usually be necessary to modify and refine a use case model; even an experienced analyst will accept that the first attempt at such a model is unlikely to be optimal.

- A blank sheet of paper (or a blank computer screen) is not a good thing to be looking at when trying to identify use cases or actors. It is better to start by putting down a large number of potential use cases and actors, and perhaps remove or merge some of them later. The early stages of use case analysis can usefully be treated as a “brainstorming” procedure in which large numbers of ideas are floated, only some of which later turn out to be useful.

- A use case should usually provide a discrete, testable service to at least one actor. This makes it possible to implement and test use cases independently. The UML (version 1.3) specification says, “A pragmatic rule of use when defining use cases is that each use case should yield some kind of observable result of value to (at least) one of its actors”.

- Use case modeling is subject to the phenomenon known as “analysis paralysis”. This is a tendency to concentrate on one small part of a model, adding increasing amounts of detail, while neglecting the broader view. If you develop part of a model to a high level of detail, perhaps over an extended period of time, you will have an emotional disincentive to delete it later should it prove beneficial to do so. This tends to bring the whole process to a halt, as the analyst struggles vainly to complete a model that is too incorrect to be amenable to completion. The correct procedure is to start with a broad outline, and add detail later. For example, if you intend to document a use case with an activity diagram or some other model, it is best to avoid doing this until most of the use cases and actors are defined.

- When describing your use cases, however you choose to do it, you may well find that the process of description leads you to challenge your choice of use cases or actors. You should take this challenge seriously, and modify the model if necessary.

- You should consider using generalisation relationships to simplify a model, buts it is usually best to first identify most of the actors and use cases.

Review

Questions

Review Question 1

What is *requirements engineering*?

A discussion of this question can be found at the end of this chapter.

Review Question 2

Supply the seven steps that make up requirements engineering and briefly describe them.

A discussion of this question can be found at the end of this chapter.

Review Question 3

What advice do you have to offer between the two different representations available for *actors* in a use case?

A discussion of this question can be found at the end of this chapter.

Review Question 4

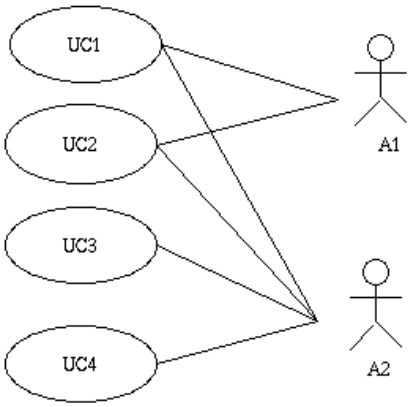
Construct a use case model that shows the requirements of a computer system that will automate the services of a large lending library. Assume that the library has separate adult and child services, orders its own books stocks, can obtain books from other libraries on request, has a catalogue on public access, and charges fines for overdue returns. Try to envisage all the services and users of the library, and capture them all in the diagram. Do not include services that don't relate to books (e.g., Internet access).

A discussion of this question can be found at the end of this chapter.

Review Question 5

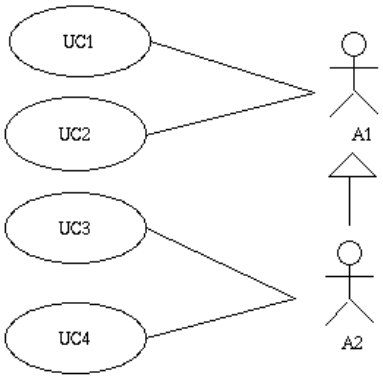
Can this use case model:

Figure 3.10. Without generalisation



be simplified using generalisation into the model shown below?

Figure 3.11. With generalisation



What reasons are there for thinking one way or the other?

A discussion of this question can be found at the end of this chapter.

Review Question 6

In a real-world design exercise, it can often be difficult to obtain the information needed to construct a complete use-case model. Why? (Try to think of at least ten reasons, and possible ways the problems can be overcome). It may help to refer to some general software-engineering books, like Sommerville for information.

A discussion of this question can be found at the end of this chapter.

Answers

Discussion of Review Question 1

Requirements engineering is concerned with discovering *what* it is that should be developed. Its goal is to develop the software's *specification*.

Discussion of Review Question 2

1. **Inception**, where the developers attempt to gain an understanding of the software and the problems that it has to solve.
2. **Elicitation** attempts to propose solutions to the problems that the software has to solve.
3. **Elaboration** expands the information from the previous two stages into an *analysis model*.
4. **Negotiation** is used to resolve any conflicts between the stakeholders in the project.
5. **Specification** involves producing the final specification from the previous steps.
6. **Validation** concerns itself with ensuring that the specification is of a high enough quality to be useful.
7. **Management** controls and tracks changes made to the specification.

Discussion of Review Question 3

The users of a system can be divided into two general classes: human users and other computer systems. Both of these are represented in use case diagrams. Representing human users using the standard stick figure can be very helpful, while representing other external computer systems as objects

with the «actor» stereotype can help to make a useful distinction between people and automated systems.

Discussion of Review Question 4

There is no single correct answer to this question. You are encouraged to discuss your solution with your tutor and/or your classmates.

Discussion of Review Question 5

If the actor A2 is genuinely a sub-type of A1, then the generalisation shown is logically correct. However, it is not possible to infer whether a generalisation exists from what associations an entity exhibits. Generalisation exists when one entity inherits all the properties of another, not just its associations.

Discussion of Review Question 6

There are many reasons why use-cases can be difficult to construct, and many of these reasons are related to requirements never being completely stable. Some examples to think about:

Use cases are written from the software's point of view, and not that of the actors. For instance, use cases should always describe a user's goal, and *not* a function of the software.

How the software is interacted with is poorly understood. This easily happens when the requirements for the software are novel, or are themselves poorly understood.

Stakeholders disagree. If the clients wanting the software cannot agree on what they want from the software, use-cases cannot be constructed.