Use Case
Tutorial

Version X.x • April 18, 2016

Company Name Limited
Street
City, State ZIP Country
phone: +1 000 123 4567

Company Name Limited
Street
City, State ZIP Country
phone: +1 000 123 4567

Company Name Limited
Street
City, State ZIP Country
phone: +1 000 123 4567

www.website.com

Table of Contents

Introduction

Full more tutorials, please see: http://openlearn.open.ac.uk/mod/resource/view.php?id=190548

1. Use cases and activity diagrams

1.1. Use case modelling

In this section, we take a closer look at *use case modelling*, and show you how it can be used to model the requirements for a product that includes the development of a software application or, simply, a system. Use case models act as a discussion tool between the requirements analyst and stakeholders, and offer a common language for agreeing the functions of a proposed system. In this discussion, we shall use the Unified Modelling Language (UML) notation (diagrams) for use cases to reflect the fact that the development team are the stakeholders as well as the client and the intended users.

The use cases for a system are a record of the intended behaviour of the system that is visible to its users. This behaviour is what the system does when responding to the events that arise from its interactions with a set of actors. The people who use the software system will be one group of actors, but there may be other systems (some of which could be software based) and devices (including computers) that must interact with the intended (software) system, which are also actors.

An **actor** is anything outside a software system that interacts with it. For example, in a system that allows people to buy goods over the Internet, the human users will be significant actors, but so too will be the credit card system that enables users to pay for their purchases. Representing other systems as actors lets you focus upon your area of concern. In UML, all actors (human or otherwise) are represented by stick figures as illustrated in Figure 2.

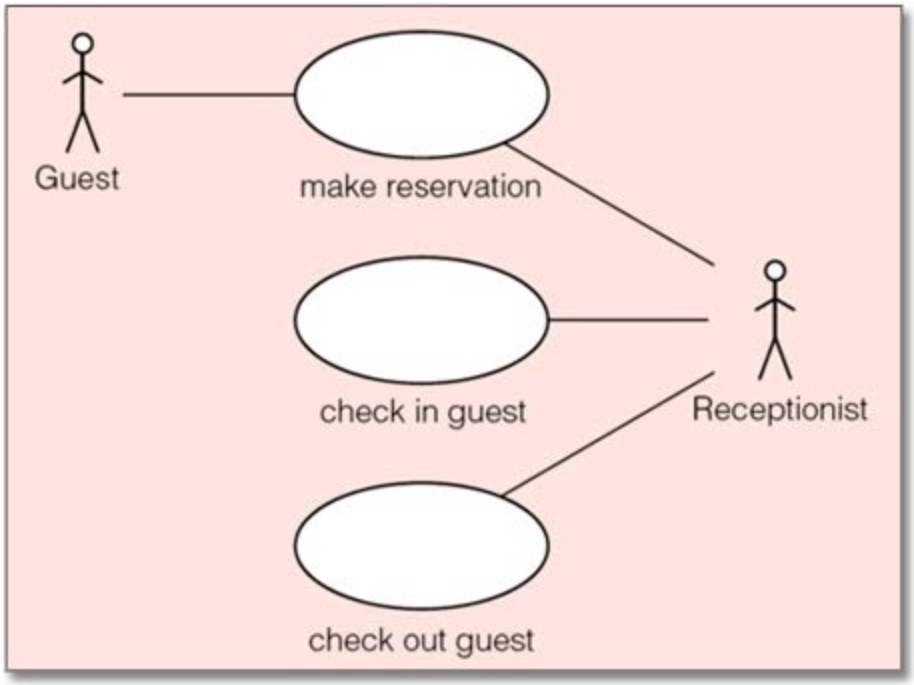

Figure 2 A use case model for a system for checking in and out of a hotel

The contents of the use case ovals represent some *tasks* or *coherent units of functionality*, as the UML defines it, which the system performs. The detailed description of each use case is held elsewhere. An actor, shown by a stick figure, represents the *role* that a human or non-human entity outside the system, often called a user, might play. The line connecting an actor figure and a use case oval indicates an association between them, which represents communication between the actor and the use case. It means that the actor *may* be involved in carrying out the task; it does not mean that it *must* be involved, as we shall explain later.

The simple notations, like those in Figure 2, for the elements of a use case diagram are intended to be intuitive, even for a lay person who is unfamiliar with the notation. It is possible to exploit this simplicity to represent the main functions of a particular business. For example, if your business was a lending library, then its main functions to be represented in a use case diagram would be the borrowing of books, videos and CDs by its members. Copies of books, films and music are the things handled or used by people; they are examples of *business objects*.

Use case diagrams deal with functional requirements (things that the system must do) alone, but it is often recommended that, as you develop a use case diagram and come across nonfunctional requirements (qualities that the product should have such as how responsive the system should be), you should record them by annotating the use case diagram with a descriptive note. In the UML, a note is shown as a rectangle with the top, right-hand corner folded down (see Figure 3). Dashed lines are used to attach the note to the model element(s) to which it refers.

The system boundary, denoted in the UML by a rectangle surrounding the use cases is an important conceptual line that separates the system we are interested in from the rest of the world. By drawing the boundary around the system represented in a use case diagram, you are setting the scope of your solution. Figure 3 shows a boundary for the hotel chain use cases. In simple use case diagrams, it is common to omit the system boundary as in Figure 2.

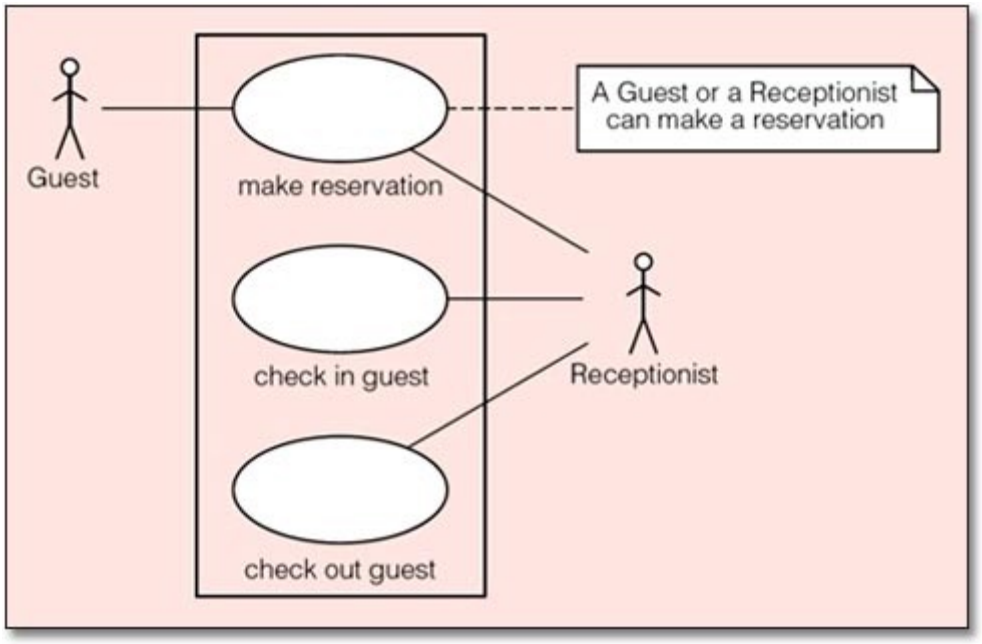

Figure 3 A use case model for a system for checking in and out of a hotel

Figure 3 illustrates a common problem. From the diagram alone, it is not clear which of the two actors, *Guest* or *Receptionist*, initiates the *make reservation* use case. It may even be that both are needed to complete the use case successfully. The UML provides several ways to deal with this problem. The simplest, which is shown in Figure 3, is to use a note to record this observation; you would then refine (that is to say, amend, improve or extend) the model when you know more about the use case.

The work context diagrams in *MRP* are related to use case diagrams through business events. Inside the system boundary is the work, and the actors represent either people or autonomous cooperative adjacent systems. The actors represent the context within which the work exists. The lines joining an actor to a use case represent communication (the flow of data). In this section, we are not only interested in the context of the work, we also want to look more closely at the work itself.

1.2. Use cases and activity diagrams

1.3. Actors

Iteration is a natural part of the modelling process. It does not matter whether you start by looking for the actors or the use cases. We have chosen to begin with the actors, since it is a way of expressing the system boundary implicitly and identifying the different views that need to be taken into account. In practice, you are likely to find that the actors are to be found in the roles that people play as employees in the problem domain, such as the hotel's receptionist or manager.

Actors are not intended to represent a particular individual, rather they tell us about a particular role that someone or something might adopt in order to interact with a system. For example, someone who works as a receptionist in one hotel might want to stay in another hotel as part of his or her holiday. Thus the same person will act in the role of receptionist at some times but will adopt the role of a guest at other times. Hence the two roles are modelled as different actors.

You could begin the task of identifying the actors by looking for the groups of people who use the current system in order to do their work. It might be easy to find those who perform the main tasks, such as the receptionists who work on the front desk of a hotel. But it might be harder to find those who use the system in support of their administrative or maintenance tasks. For example, would the maintenance engineers and cleaners in the hotel have to be considered? The answer will clearly depend on the scope of the problem being solved.

You can use an actor to represent an external software system. In the case of the hotel chain, it is likely that you will need to pass information about a guest's stay to an accounting system. At some later point, you may be asked to provide an interface to the restaurant side of the hotel in order to associate the costs of any meals with the guests who ate them. When the guest leaves the hotel, there may be a requirement to collect payment for the guest's stay from an external banking or credit card system. In each case, you should consider whether or not there is some value in an exchange between the use case and any identified external system. For the actors that you have chosen to include, treat them as though they were an autonomous black box. You do not need to know how they work. You only need to know about the shared phenomena that are relevant to the exchange between your system and the external system.

It is important to distinguish between an actor and the way that actor communicates with the system. For example, when analysing a system you should not be concerned with the mechanism used by the receptionist to check guests in and out of the hotel system. It could involve the use of a paper diary and a pen, a keyboard and a mouse to interact with a series of screens on a personal computer (PC) or even include a network connection or voice recognition software. You should concentrate on the meaning of the stimuli and the responses for any given use case, not the communication mechanism that is used. That mechanism is part of the solution, which you intend to provide.

In situations where two or more actors are associated with a use case, one of them must initiate the actions. The other actor(s) will play a passive role. For example, when a guest checks into a hotel in person, the receptionist typically performs the checking-in process and the guest has no

direct interaction with the system. In *MRP* this is described as a business event: the work learns that such an event has happened by the arrival of an incoming flow of data and responds to this business event.

When it comes to describing a use case, treat the system as a black box, as in the case of an external system. It will accept stimuli from the actors and generate responses. That is, information (data) flows between the actors and their associated use cases.

1.4. Describing use cases

To understand the work, you need a good idea of what each use case means. To get a feel for what this might entail, look again at Figure 3 (reproduced below) which shows a simple use case model for a hotel chain reservation system. Note that Figure 3 is not intended to be an exhaustive model of the hotel domain; the scope of the problem to be solved is confined to reservations and the processes of checking in and out.

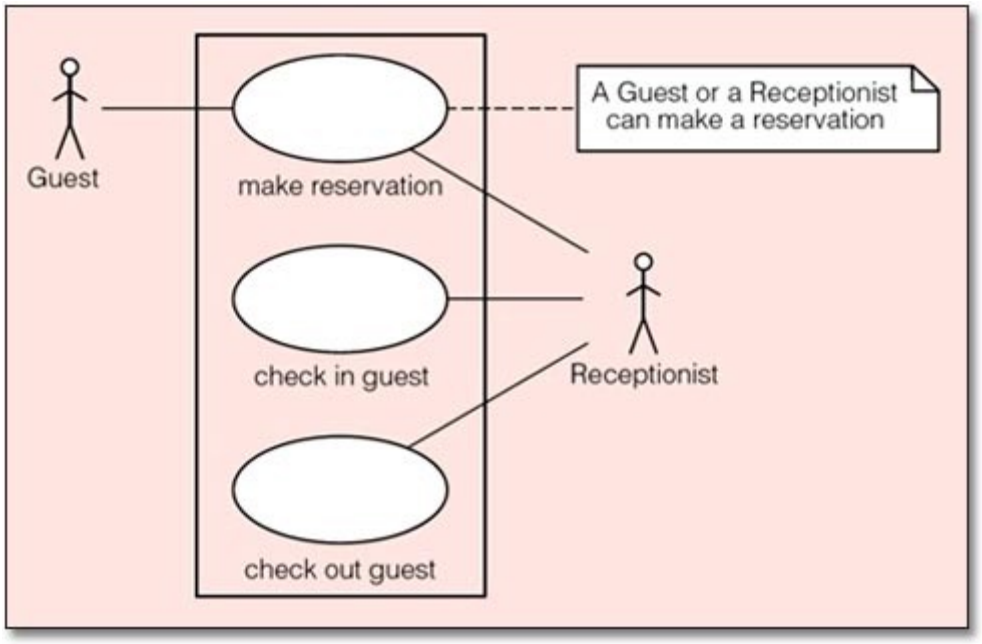

Figure 3 A use case model for a system for checking in and out of a hotel

What do the use cases *make reservation, check in guest* and *check out guest* mean? No doubt, using your own experience of reserving rooms at hotels, the names of the use cases are quite indicative of what they represent. However, you should never rely on intuition or personal experience but rather create a description of the use cases that you and the stakeholders can agree upon. For example, suppose that the following is a description of the *check in guest* use case (for a particularly simple system).

Upon arrival at a hotel, a guest provides the reference number for his or her reservation to the hotel's receptionist, who uses it to find the details of that reservation so that each guest can confirm them. The receptionist allocates an appropriate room to that guest and opens a bill for the duration of the stay. The receptionist issues a key for the room.

Having read this description, you are probably thinking of all the deficiencies it contains. This is just what should happen: you need to check with the receptionist to clarify that the description contains all the necessary information. We shall not pursue this line of thought further now because we want to draw your attention to the following observations about the requirements of the *check in guest* use case.

There is a condition, known as a **pre-condition**, that must hold *before* a room can be allocated to a guest. It is as follows.

There must be a reservation for the guest and there must be at least one room available (of the desired type) and the hotel must be confident that the guest is able to pay for the room.

There is another condition, known as a **post-condition**, that must hold *after* a room has been allocated to a guest. It is as follows.

The guest will have been allocated to a room for the period identified in the reservation; the room will have been identified as being in use for a specific period and a bill will have been opened for the duration of the stay.

In other words, we have captured the meaning of *check in guest* in terms of two statements: one that must be true before the use case can be carried out – the precondition, and one that must be true once the use case has been completed – the postcondition. At some later point, the developer must decide how these conditions can be met as part of the design activity.

The advantage of describing a use case in terms of a pre-condition and a post-condition is that, when you go on to elaborate the use case in more detail, that is, to describe its components, you know that the components must also satisfy these same conditions. For example, it is no use if one of the components ignores the fact that there must be a vacant room *on the dates requested* and allows the reservation to go ahead, or if a component fails to indicate that the room, once booked, cannot be booked again until it becomes free.

Notice that such a specification does not say *how* a reservation must be performed; simply *what* conditions should be satisfied. The advantage of thinking in this way is that it avoids all issues to do with software and leaves the developers (who may specialise in design and programming) to choose an appropriate implementation, which is their field of expertise.

The two descriptions: the first, which is entirely prose, and the second, which is more formal using pre- and post-conditions, are both equivalent descriptions (specifications) of the requirements represented by the *check in guest* use case.

When something is described using natural language we often say that it is an *informal* description. When we use a more structured approach to descriptions, such as the way in which we described the pre- and post-conditions for the *check in guest* use case, we say that we are being more *formal*. The ultimate level of formality, when we want to be as precise and

unambiguous as possible, is the use of mathematics. There are times when the use of mathematics is essential; typically when dealing with software that controls situations which, if incorrect, can lead to death (for example, aircraft and nuclear power stations). However, you would not want to be too formal in most situations otherwise stakeholders are very unlikely to understand your requirements.

1.5. Scenarios

The purpose of a use case is to meet the goal of its associated actor(s), such as a guest making a reservation with a hotel. This implies that a use case should include everything that must be done to meet that goal. For example, if it is necessary to check the availability of rooms in the hotel for the desired length of stay before accepting a reservation, then we expect the use case to contain that check. In general, a use case contains a narrative about the flow of events that specifies a particular use of the software system.

A **scenario** is a description of a sequence of actions that illustrate a piece of interesting behaviour. In the UML, a scenario is said to be an **instance** of a use case (implying that there could be several such instances, each one describing a different situation). So, a scenario describes the interaction and dialogue between the users of a system (its actors) and the system itself. For a given use case, we expect to see one **main scenario** that describes the flow of events leading to a *successful* conclusion. There may be other scenarios that describe alternative or additions to the main scenario. Here, for example, are two possible scenarios for making a reservation at a hotel.

1. Jill wants to reserve a room at the Ritz Hotel for 14 July. A room is available for that date and so the receptionist makes a reservation for the guest, Jill.
2. Jack wants to reserve a room at the Savoy Hotel for the first week of August. There is no single room that is free for seven days in August, but there is one room available for four days followed by another of the same type for three days. The receptionist presents that option to Jack, who rejects it.

Both scenarios are possible instances of the *make reservation* use case. Their interactions and outcomes are different. In the first, there is a description of the use case leading to a successful outcome. In the second, there is an exception to the main success scenario. Exceptions to the normal behaviour for a use case are common, especially where actors decide to abort a use case without completing it. However, the common theme among all the scenarios is the intent of an actor to reach the goal defined by a use case. In the unsuccessful scenario above, Jack was trying to make a reservation at the Savoy Hotel. Perhaps he didn't like the idea of changing rooms during his stay. Hence, a use case should include any unusual or alternative courses of action.

You could start an investigation by simply identifying a use case and its main success scenario, and later refine or adapt it. You will need to decide the way in which you record the information for each use case not just for the main success scenario, but for all the relevant scenarios. At its simplest, you can record a textual description (narrative) for each use case that details each scenario together with its outcome.

Remember that your description of a use case expresses what the system should do without constraining how it should do it. Since the description takes an external viewpoint, all the

behaviour is in the form of observable results. Later on, a developer will choose an architecture for the solution and produce a workable design and implementation. While the structure and format of a use case description may vary among the different development processes, we suggest that you include the following items as minimum details.

- A **unique identifier** for the use case that allows traceability throughout development.
- The **name** of the actor that initiates the use case as well as the identity of any other actors that may be associated with the main success scenario.
- A short description of the **goal** of the use case.
- A single sequence of **steps** that describe the main success scenario. You may also find it helpful to number these steps for traceability, in cases where you need to identify any extensions or variations that occur as a result of the other scenarios of a use case (we discuss this in more detail later).
- A textual description of the **pre-** and **post-conditions**.

In some circumstances, you may have to add other information. For example, the identity of the authors may be required where there is a large team of developers. In a risk-driven process, you might be required to record an assessment of the risks, assumptions and outstanding issues to support the decision-making process. For example, it helps to record the things that the authors of a use case had assumed to be true during their analysis.

Opinions vary about the correct format of the description of a use case. One development process might require a detailed structure with tightly controlled phrasing and numbering of each item in the description. Another might place few or no limitations such that each use case reads like a story – with a beginning, a middle and an end.

However, you should use the language of the domain to formulate the use cases and identify requirements. Each requirement is part of the contract between the developer (as supplier) and the customer. Both parties need to have a clear understanding of what is captured in each use case and of what it means. Table 2 illustrates one way to structure and record a use case.

The main success scenario in Table 2 contains a stepwise description of what happens when nothing goes wrong; usually the most common case. It is assumed that the steps are performed in the order described with no concurrent (simultaneous) behaviour. In the next section, you will see how each step can be used as a decision point to deal with exceptional circumstances. For example, what should be done if there is no available room for the desired stay? Later, you will see how the UML can be used to model the conditional and concurrent activities of a business process. For example, what should be done at step 5 if the ‘guest’ had stayed in the hotel chain before and had already provided these details?

Table 2 Textual description of a use case in the hotel domain

Identifier and name	UC_1 Make reservation.
Initiator	Receptionist or Guest.
Goal	Reserve a room at a hotel for a guest.
Pre-condition	None (that is, there are no conditions to be satisfied prior to carrying out this use case).
Post-condition	A room of the desired type will have been reserved for the guest for the requested period and the room will be occupied for that period.
Assumptions	A guest can make a reservation via the Internet. The guest is not already known to the hotel's software system (see main success scenario, step 5).

Main success scenario

1. The guest requests a reservation.
2. The guest selects the desired hotel, dates and type of room.
3. The hotel receptionist provides the availability and price for the request (an offer is made).
4. The guest agrees to proceed with the offer.
5. The guest provides identification and contact details for the hotel's records.
6. The guest provides payment details.
7. The hotel receptionist creates a reservation and gives it an identifier.
8. The hotel receptionist reveals the identifier to the guest.
9. The hotel receptionist creates a confirmation of the reservation and sends it to the guest.

1.6. More about actors

In the hotel example, you saw two actors in the use case diagram shown in Figure 3 (reproduced below). Why is the actor *Guest* associated with the use case for making a reservation but not associated with the use cases for checking in and out? The answer comes from an understanding of what happens when someone, a guest, arrives at a hotel. Hotels are service oriented. That is to say, they offer certain services to their guests with the intention of earning money for the business. A hotel employs its staff on this basis. In particular, a hotel will employ a receptionist, who will be the real user of the proposed software system, to deal with guests on their arrival and departure; a guest will not use the system.

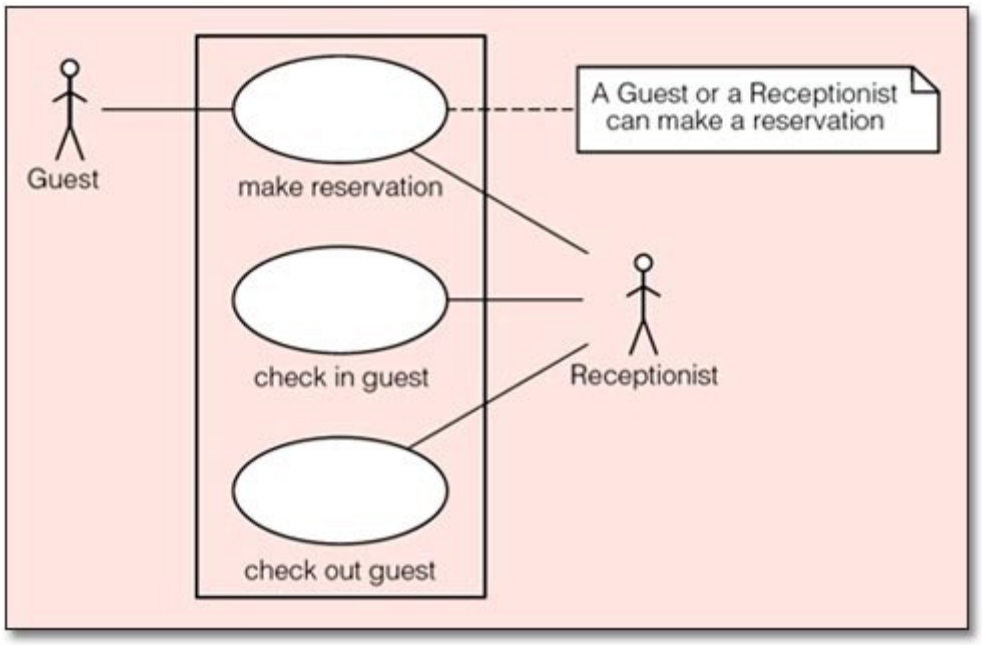

Figure 3 A use case model for a system for checking in and out of a hotel

However, if a goal of the new system is to allow potential guests to use their web browser to make reservations in addition to contacting a hotel directly, the potential guest will be a user of the proposed software system. Hence the use case diagram must show *Guest* as an actor for the *make reservation* use case as in Figure 3. Since people will still be using other methods of requesting a room, such as by telephoning or sending a letter to the hotel, we should allow for a member of the hotel's staff to perform the service. Hence the use case diagram in Figure 3 and the use case description in Table 2 include both the *Receptionist* and *Guest* actors.

If you need to include some significant information about the roles that actors play, you can do so by expanding the use case diagram in the following way. Since either a guest or a receptionist can make a reservation it may be better to think of a new kind of actor, a *Reserver*, say, who

could be either a *Guest* or a *Receptionist*. The use case shown in Figure 3 can then be modified to the one shown in Figure 4.

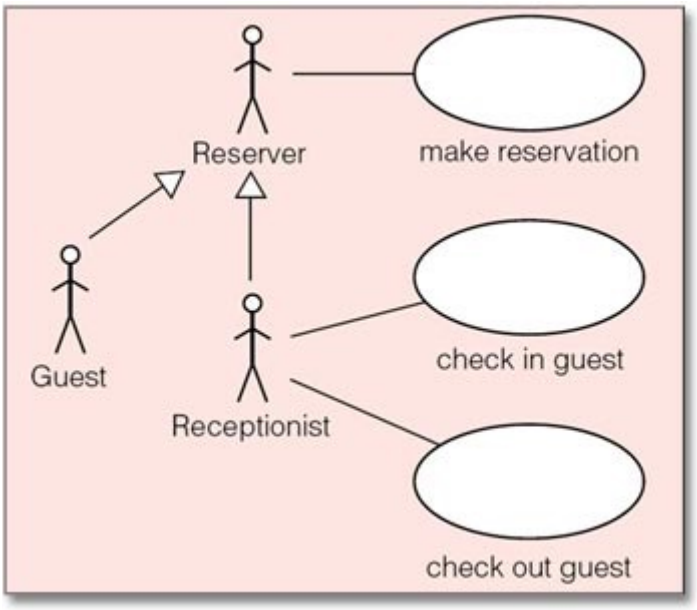

Figure 4 A use case model showing specialisation between actors

The notation used in Figure 4 indicates that *Guest* and *Receptionist* are **specialisations** of *Reserver* (or *Reserver* is a generalisation of *Guest* and *Receptionist*). That is, a *Guest* (or *Receptionist*) can do the same thing as a Reserver (but they can do other things as well).

1.7. Modelling the relationships between use cases

There are two situations when you would consider adding details to a use case diagram:

- to identify a common task (and its associated scenarios) that is shared between two or more use cases;
- to record any alternatives or additions to the main success scenario as separate use cases.

In both situations, the new tasks are shown as new use cases (ovals) and, as you will see below, the UML provides a suitable notation (known as a stereotype) to represent the relationship between the original use cases and the new ones.

The main disadvantage of this approach is the additional complexity they bring to a model in contrast to the simple use cases considered previously. The best advice for you as a requirements analyst is to remember why you are creating the model and who it is for.

1.8. Stereotypes

In the UML, a **stereotype** is a way of adding detail to any part (element) of a model. It is a way of expressing variation or a usage distinction that tells you more about the original element. For example, the line drawn between an actor and a use case indicates that there is an association between them. We could add the stereotype «communication» to such a line to emphasise the communication that takes place between the two. In practice, this stereotype is left out because it is the only type of association between an actor and a use case.

In general, stereotyping is a recognised way of extending the UML. You can define your own term and place it between the angle brackets (or guillemets: «»). However, there must be some agreement in the team about the existence and documentation of such new terms.

The UML includes some stereotypes that you cannot redefine. Two of them are used to describe dependencies between use cases and these are discussed in Subsections 6.8 and 6.9.

1.9. Sharing behaviour between use cases

For each use case there may be more than one scenario. In the process of requirements elicitation and specification, you may find a certain amount of common behaviour in two or more of your use cases. You may even find that an existing component can provide part or all of that common or **shared behaviour**. Indeed, if you do find such an existing component, this is an example of reusing requirements which is discussed more fully in *MRP*.

You can record the shared behaviour in a new use case and connect it to the use cases that it came from with an open-headed, dashed arrow pointing from the original use case to the new one. Think of the new use case as always being included in each of the originals. Hence the dependency arrow is labelled with the «include» stereotype. Figure 5 shows some examples from the hotel domain (rooms are not normally allocated until a guest checks in for a variety of reasons: rooms need servicing, guests extend their stay, and so forth). The «include» stereotype simply shows that a use case can contain one or more 'sub' use cases and that some such sub-use cases can be reused in two or more use cases.

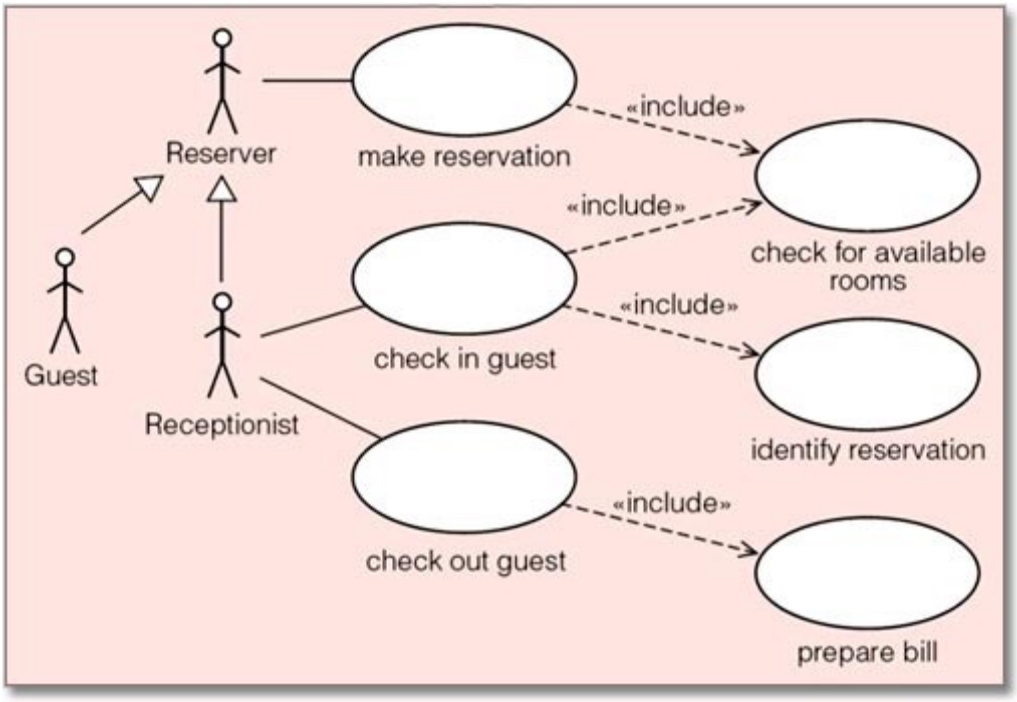

Figure 5 Shared behaviour in a hotel system

The *check for available rooms* sub-use case is a shared piece of behaviour, a common scenario, which can be developed separately from other use cases. Note that this is *unconditional* behaviour – the *check for available rooms* sub-use case *must* be performed whenever a reservation is made or a guest checks in.

By taking out any common or shared behaviour, you can benefit from a simplification of the original use cases and make them easier to understand. There is a further benefit in terms of the internal consistency of the final requirements specification. Instead of having two or more different scenarios for a room availability check, for example, there will be just one main success scenario for the new room availability check use case as shown in Figure 5.

In addition, there is a chance to consider the reuse of existing components and also the potential identification of new components.

1.10. Alternatives to the main success scenario

If a use case incorporates a scenario that is significantly different from the main success scenario, you may decide to create a new subsidiary use case. There may even be a need to create more than one subsidiary, depending on what happens in different circumstances. For example, when making a reservation in a typical hotel the receptionist would first determine whether the guest was already known to the hotel (among other advantages, this would speed up the reservation process since re-entering of all the guest's details would be avoided). Of course, in the case of a new guest and therefore not known to the hotel, all the guest's details would have to be entered.

Figure 6 shows a development of the hotel system use case diagram that identifies two new sub-use cases: *identify guest* and *create new guest*. The *identify guest* sub-use case is part of the *make reservation* use case and is connected to the original *make reservation* use case because it will have to be carried out every time a reservation is made (unconditional behaviour). However, the second new sub-use case, *create new guest*, which is connected to *identify guest* will not be carried out every time a reservation is made. Therefore, *create new guest* is connected to *identify guest* with an open-headed, dashed arrow labelled with the stereotype «extend». This is *conditional* behaviour as it is only performed when the guest is not already known to the hotel.

The UML allows a number of ways to record the event that triggers the subsidiary use case. In Figure 6, we have used the general purpose notation for a note to indicate that *create new guest* is performed when a guest is not already known to the hotel.

The textual description of new use case should record a description of the corresponding scenario. It should contain the following two key points:

- the condition that triggers the subsidiary use case, that is, the business event;
- the place(s) in the main success scenario where the condition is tested – these are called **extension point(s)**.

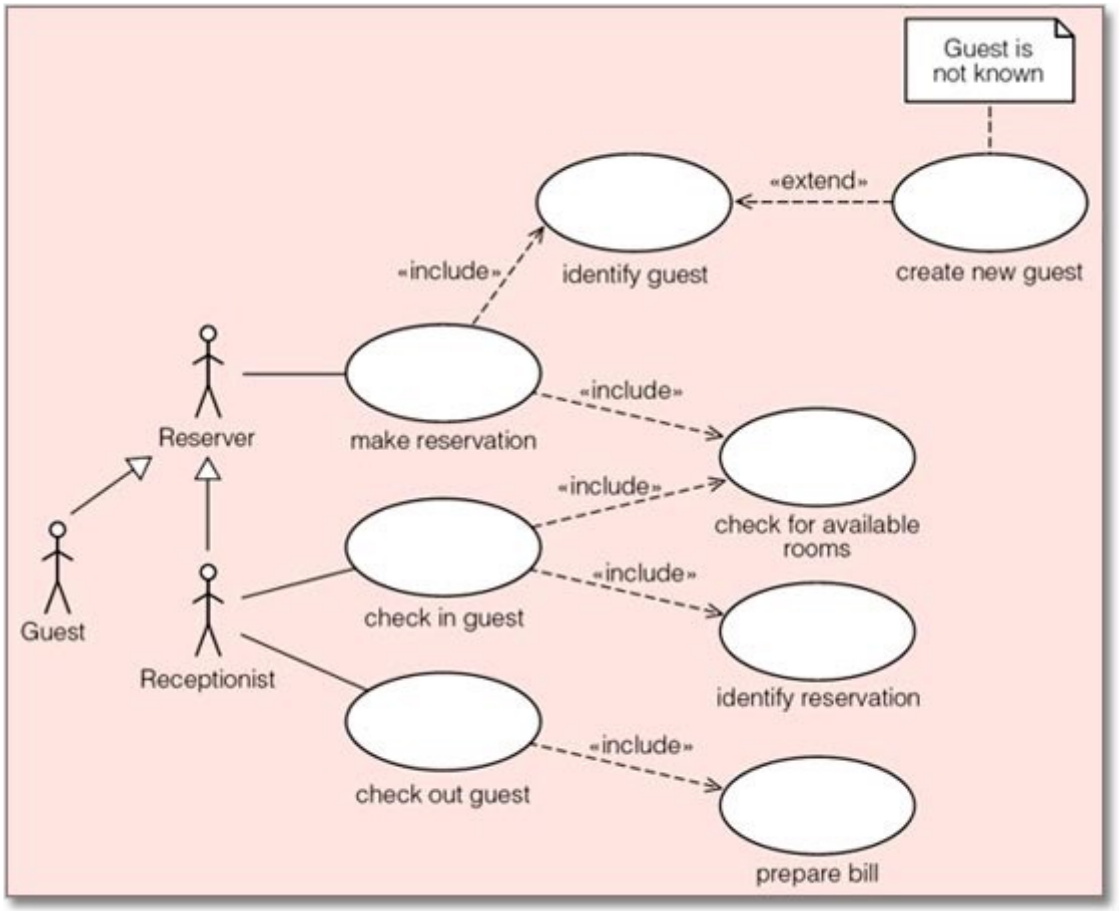

Figure 6 Alternative behaviour in a hotel system

Table 3 shows how the original description of the *make reservation* use case given in Table 2, has been changed to take into account the extension to deal with instances where the hotel has no unoccupied room available for the requested period, and the introduction of a new actor called *Reserver*. Each step in the main success scenario acts as a potential extension point, from which the relationship to a new use case can be defined. In Table 3, step 3 is the extension point that leads to the additional steps described in steps 3.1 and 3.2. As the second extension point at step 5 shows, some work can be avoided if the potential guest for the reservation has stayed somewhere in the hotel chain before. Where such choices arise, your main success scenario should reflect the more dominant or typical flow. Table 3 reflects an emphasis upon new guests for the hotel chain.

Table 3 Extending the description of a use case in the hotel domain

Identifier and name	UC_1 Make reservation.
Initiator	Reserver (may be a Guest or a Receptionist).
Goal	Reserve a room at a hotel for a guest.
Pre-condition	None.
Post-condition	The guest will have been allocated to a room for the requested period and the room will be occupied for that period.
Assumptions	The expected initiator is a guest using an Internet browser to perform the use case. The guest is not already known to the hotel's software system (see main success scenario, step 5).

Main success scenario

1 The reserver requests a reservation on behalf of a potential guest.

2 The reserver selects the desired hotel, dates and type of room.

3 The receptionist provides the availability and price for the request. (An offer is made.)

4 The reserver agrees to proceed with the offer.

5 The reserver provides identification and contact details for the hotel's records.

6 The reserver provides payment details.

7 The receptionist creates a reservation and gives it an identifier.

8 The receptionist reveals the identifier to the reserver.

9 The receptionist creates a confirmation of the reservation and sends it to the guest identified by the reserver.

Extensions

3 A room matching the request is not available.

3.1 The receptionist offers alternative dates and types of room.

3.2 The guest selects from the alternatives.

5 The guest is already on record.

5.1 Resume at step 6.

1.11. To extend or include?

Whatever kind of system you intend to develop, you will need to consider its security. Usually, we allow only trustworthy people to use a new system. Therefore, in a software solution we can envisage a log-on use case, which describes how a user gains access through some authentication procedure. How should such a requirement be included in the example of the hotel chain?

By analogy with natural languages, the UML allows a number of 'grammatically correct' options each of which will make more or less sense depending on the context. For example, we could show the *log-on* use case as a component of every use case that is associated with an actor, as shown in Figure 7.

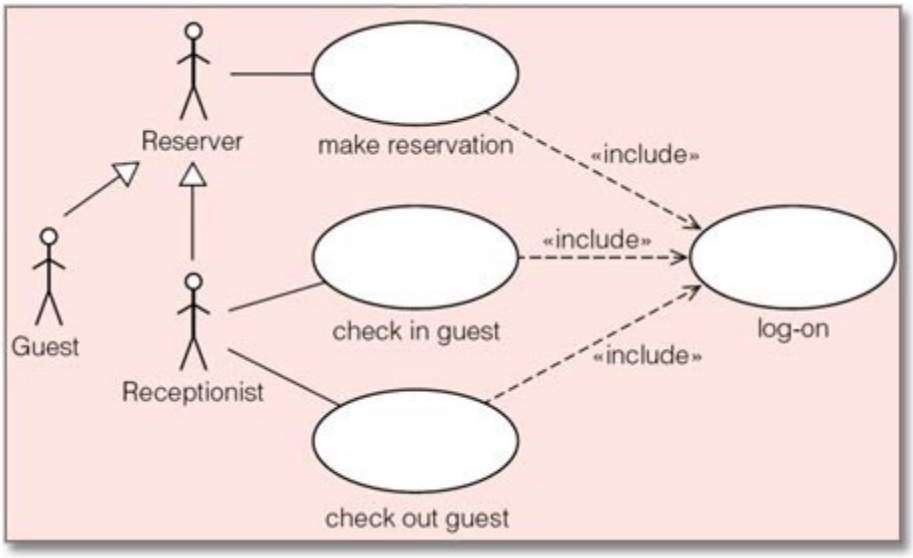

Figure 7 Including the log-on use case in the hotel domain

You can also redraw Figure 7, and produce Figure 8, showing the three original use cases as variations of the *log-on* use case. It would be 'grammatically correct' although it would be difficult for the reader to see the intended purpose of the system.

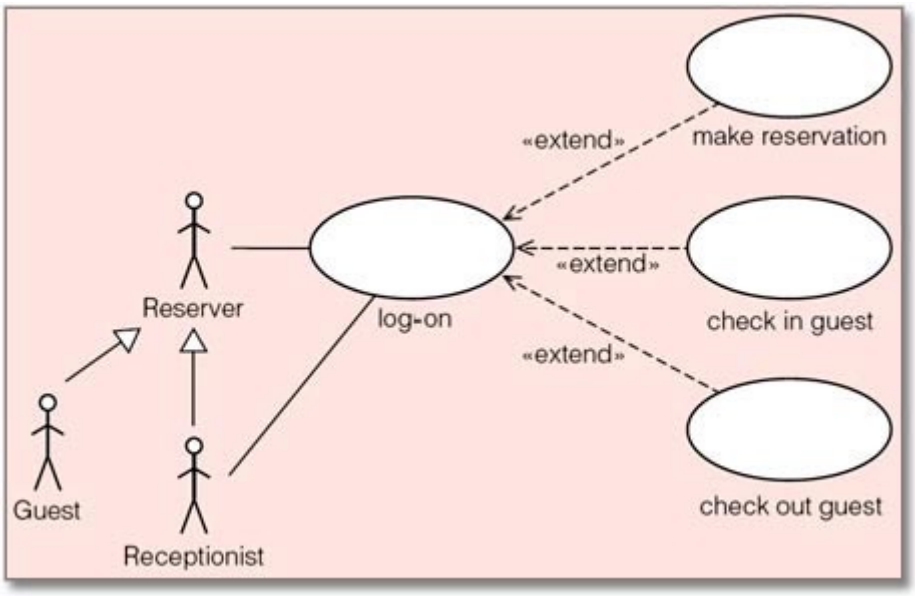

Figure 8 Making log-on the main use case

1.12. Issues with use cases

There can be a tendency to make diagrams too complex. You can reduce the complexity of your use case diagram by:

- redrawing it at a higher level of abstraction;
- splitting it up into smaller modules, which the UML calls *packages*.

In the case of the hotel chain, we might partition our model into the following three packages:

- reservations;
- checking guests in and out of their rooms;
- system access.

Each package may then be assigned to a separate developer for implementation. However, the project team must then deal with the dependencies between the three packages as they work towards a solution that incorporates all three packages.

The above example about access to the hotel system illustrates a more general modelling problem. It is often difficult to separate a problem from its solution. For example, it may seem obvious that, to gain access to the system, an authorised person would enter their name and password. However, this might not be the most appropriate method of authentication and it would be better to simply state that access to the system should be by authorised personnel only using an appropriate authorisation process. In practical terms, you should ask yourself the question, ‘Am I analysing the problem or designing a solution?’

In software development, this question can be hard to answer. You may find it easier to think of analysis as a way of investigating a problem and opening up choices, whereas design is a way of taking decisions and narrowing down the number of choices to arrive at a solution.

It is easy to forget that a use case diagram is part of the structural view of a system. It defines what tasks are to be supported, *not the order in which they might occur*. Although you record a workflow in the steps of each scenario of a use case, there will have been some initial analysis of the best or preferred way to achieve the goal of that use case. We shall look at how you can explore different scenarios in the next section, where we consider how the UML allows you to represent a workflow as it unfolds over time.

Use case modelling has led to most disagreement among experts and practitioners when they discuss the definition and the use of the UML. Space does not permit a great deal of elaboration of the arguments, but it is worth considering the kinds of problem that developers can have with use case modelling.

Having decided to model a system in one or more use case models, the most important thing to consider is their intended audience. You need use cases that can be read and understood by the domain experts as well as the team of developers. The domain experts usually come from the customer's area. If you cannot demonstrate the benefits of your proposed system to them, there is little chance of it being acceptable to the customer. All technical projects of any kind are vulnerable to this risk. Your only defence comes from your skills, experience and professional ability.

In the same way, your use case models must be useful to the rest of your team. For example, those who will be testing the new (software) system must be able to generate their tests from your use cases and the subsequent design artefacts.

In terms of the content of each use case diagram, you should avoid the use of the «include» and «extend» stereotypes for an audience that is less familiar with the UML than your team members. The simple notation for actors and their associations with use cases has been a factor in their favour.

A common problem with use case modelling is deciding the size and scope of each use case. There is no consensus on this issue because of the wide variety of contexts and viewpoints. However, we recommend that a use case should be smaller than a business process. In the hotel chain, for example, the handling of reservations would be treated as a separate business process to checking in and out. That is to say, *make reservation* is only one of the tasks in the process of handling reservations.

An associated issue is deciding whether or not you have identified *appropriate* use cases. You should always review your model and ask yourself, ‘Do the actors that have been associated with a use case actually gain value from the use case?’ If the answer is ‘no’, omit the use case! A useful technique for identifying appropriate use cases is to determine the life history of the objects in the system. For example, in the eTMA system, the central object is an assignment. In broad outline, its life history goes something like this. The student creates the assignment and then submits it to the university. Then a tutor downloads the assignment, marks and comments on

it and sends it back to the university. Finally, the student retrieves (downloads) the marked assignment and reads the tutor's comments. This leads to use cases that we might name as: *submit TMA, download TMA, mark TMA, return TMA*, and *retrieve TMA*.

The main problem with use cases, in general, is the risk of straying into a top-down, functional decomposition and away from the object-oriented viewpoint that is embedded within the UML. It is easy to decompose each use case into smaller use cases in your search for reuse through the «include» stereotype. Indeed, if you are making your project plans according to the use cases that you identify, the urge to find a use case of a size that you can easily estimate is understandable. A good project manager will make some assessment of this risk and review it upon each iteration of the life cycle.

It is worth reiterating that, in the process described in *MRP*, the purpose of use cases is to help with the understanding of the work that the product is to become a part of. There is always the danger that the use case diagram becomes a model of the product (a solution) rather than a model of the work (the problem), with the result that the product simply automates the current work and no attempt is made to identify the best product to help with the work.

No single technique can guarantee that you will collect and identify all the users' requirements. So, if you spend too much time modelling use cases, you can become distracted by the process of modelling and lose track of the main aim, which is to capture the functional requirements for a new system. Consequently, you should use more than one technique to produce a requirements specification.

You can find out more about use cases in Cockburn (2001) and about UML in Fowler (2003):

Cockburn, A. (2001) *Writing Effective Use Cases*, Harlow, UK, Addison-Wesley.

Fowler, M. (2003) *UML Distilled: A brief guide to the standard object modelling language* (3rd edn), Reading, MA, Addison-Wesley.

1.13. Self-assessment questions

1.13.1. Actors

(a) Explain why the actors in a use case diagram do not represent actual individuals.

(b) Suggest a guideline that will help you decide whether or not to include an interaction with an external system on your use case model.

1.13.2. Describing use cases

(a) From your own experience, write down a description of the use case *check out guest* shown in Figure 3.

(b) Suggest a pre-condition and a post-condition for the use case *check out guest*.

(c) If you were determining the requirements for a real hotel chain, what would you do next with your answers to parts (a) and (b)?

1.13.3. Scenarios

What is the relationship between a use case and a scenario?

1.13.4. To extend or include?

(a) What are the two stereotypes that are used to define relationships between use cases in the UML?

(b) What is the function of the «include» stereotype?

(c) What is the function of the «extend» stereotype?

(d) Is it necessary to place the «include» and «extend» stereotypes on all diagrams?

(e) How would you modify a use case model to show that you intend to employ a component that already exists? Would you show this change to a user?

1.14. Exercises

1.14.1. Exercise 1

Write down a textual description (using the format of Table 2, reproduced below) of the use case *check in guest*, shown in Figure 3, also below. As part of your deliberations, identify any exceptions to the main success scenario.

Table 2 Textual description of a use case in the hotel domain

Identifier and name	UC_1 Make reservation.
Initiator	Receptionist or Guest.
Goal	Reserve a room at a hotel for a guest.
Pre-condition	None (that is, there are no conditions to be satisfied prior to carrying out this use case).
Post-condition	A room of the desired type will have been reserved for the guest for the requested period and the room will be occupied for that period.
Assumptions	A guest can make a reservation via the Internet. The guest is not already known to the hotel's software system (see main success scenario, step 5).

Main success scenario

1. The guest requests a reservation.
2. The guest selects the desired hotel, dates and type of room.
3. The hotel receptionist provides the availability and price for the request (an offer is made).
4. The guest agrees to proceed with the offer.
5. The guest provides identification and contact details for the hotel's records.
6. The guest provides payment details.
7. The hotel receptionist creates a reservation and gives it an identifier.
8. The hotel receptionist reveals the identifier to the guest.
9. The hotel receptionist creates a confirmation of the reservation and sends it to the guest.

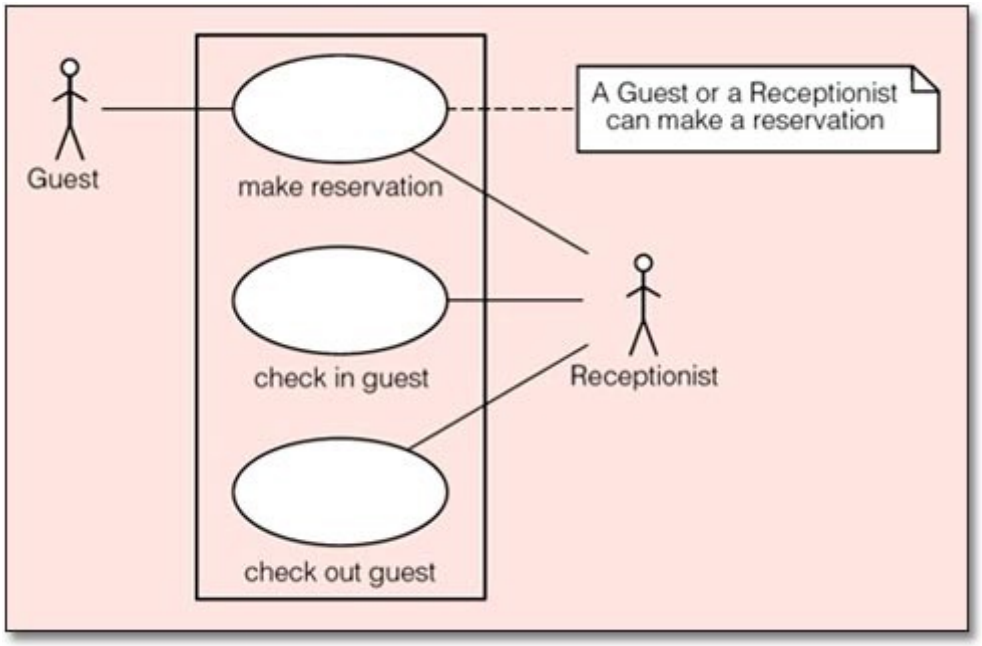

Figure 3 A use case model for a system for checking in and out of a hotel

Identifier and name	UC_2 Check in guest.
Initiator	Receptionist.
Goal	A guest takes up a reservation and occupies a room at the desired hotel.
Pre-condition	There is a reservation for the guest and there is, at least, one room available (of the desired type) and the guest can pay for the room.
Post-condition	The guest will have been allocated to a room for the period identified in the reservation and a bill will have been opened for the duration of the stay.
Assumptions	The guest is already known to the hotel's software system. The hotel is confident that the guest can pay. For example, the guest has a valid credit card.

Main success scenario

1 The guest provides a reservation reference number to the receptionist.

2 The receptionist uses the reference number to find the reservation.

3 The receptionist states the details of the room type and the duration of the stay recorded in the reservation.

4 The guest confirms the details of the room type and the duration of the stay.

5 The receptionist allocates a room to the guest.

6 The receptionist opens a bill for the guest. (It could be that there is a separate billing application, which must be notified upon check in.)

7 The receptionist issues a key to the guest.

1.14.2. Exercise 2

What are the tasks involved in preparing a use case diagram?

1.14.4. Exercise 3

Redraw Figure 6, taking into account the information contained in Figures 7 or 8 (all figures reproduced below), to show common tasks and any extensions to the main success scenario.

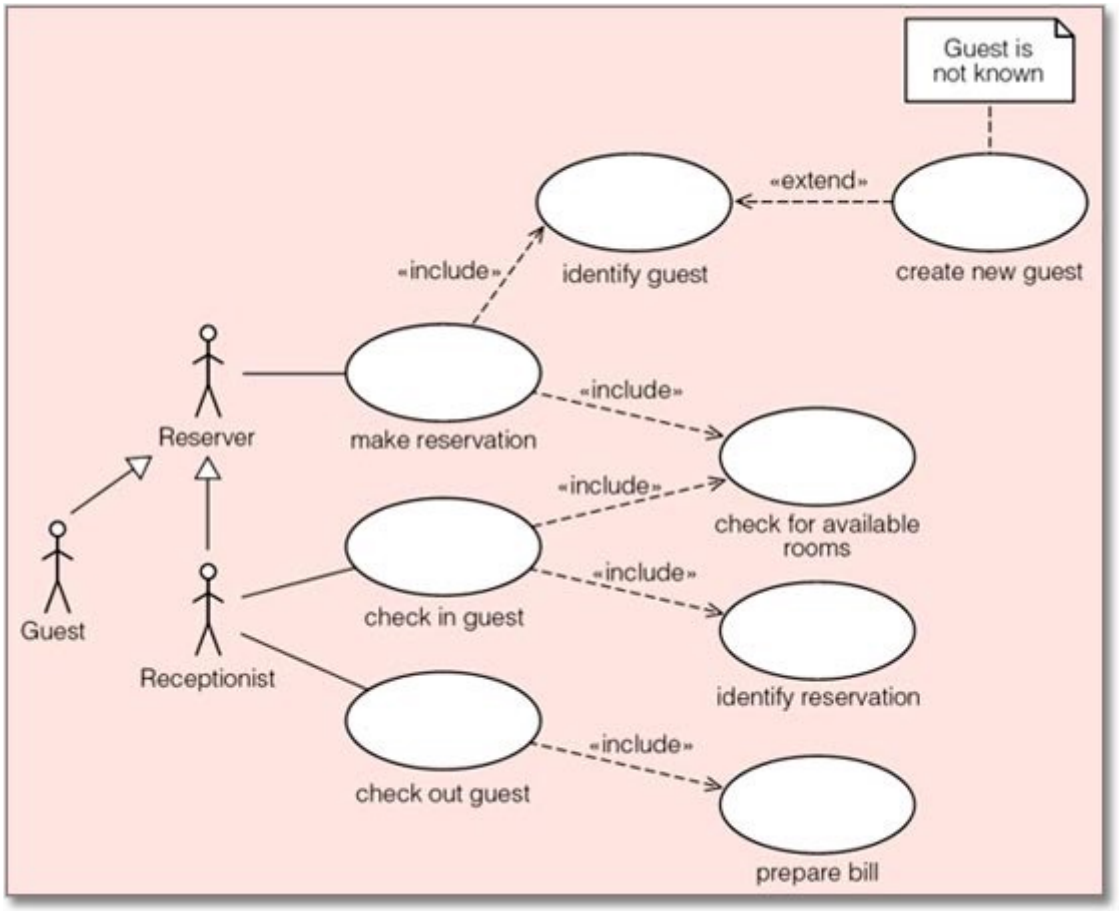

Figure 6 Alternative behaviour in a hotel system

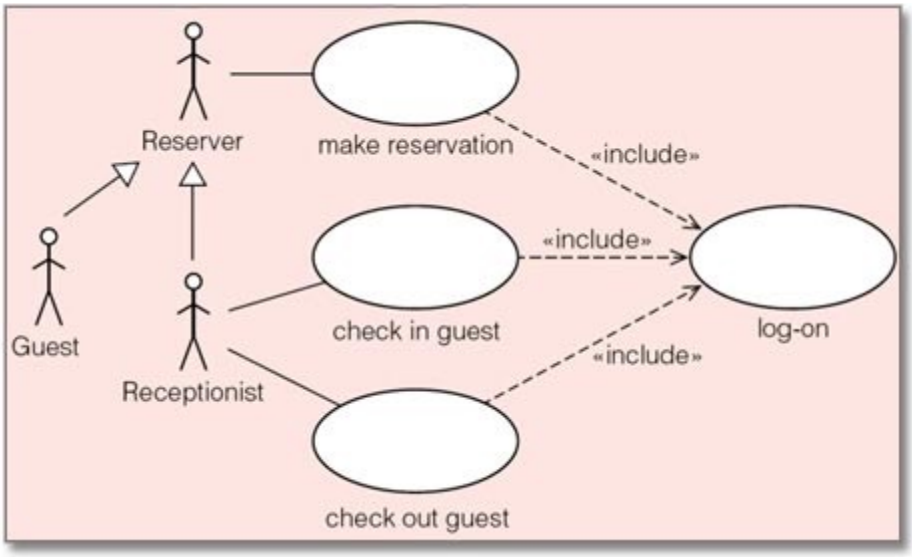

Figure 7 Including the log-on use case in the hotel domain

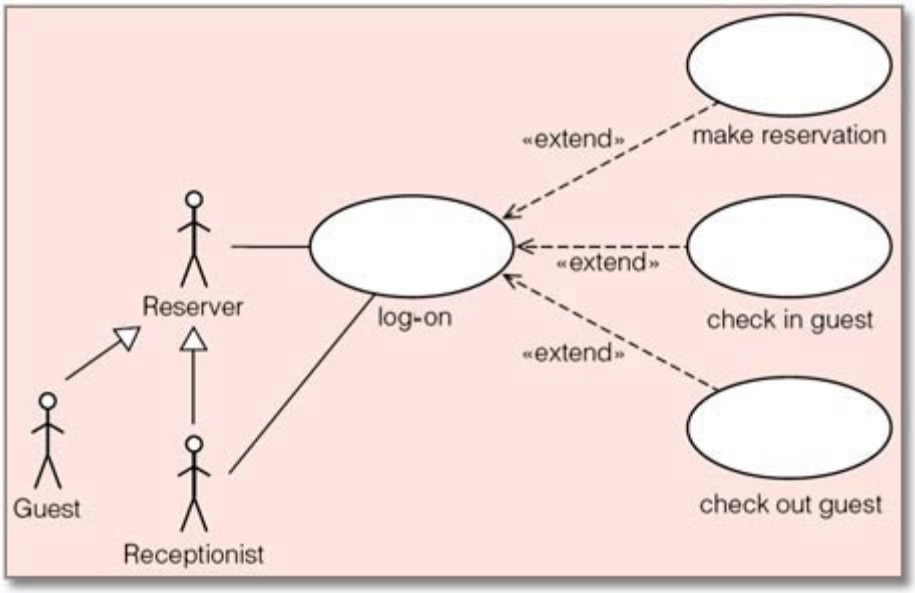

Figure 8 Making log-on the main use case

Do not show the *log-on* use case, described in Figure 7 or 8.

1.14.5. Exercise 4

A typical lending library keeps a stock of books for the use of its members. Each member can take out a number of books, up to a certain limit. After a given period of time, the library expects members to return the books that they have on loan.

When borrowing books members are expected to hand their chosen books to the librarian, who records each new loan before issuing the books to the member. When a book is on loan to a member, it is associated with that member: possession of the book passes from the library to the member for a defined time period. The normal loan period for each book is two weeks. If the member fails to bring the book back on or before the due date, the library imposes a fine.

In a proposed new system, anyone should be able to browse the stock of books held in the library, but only a member will be able to reserve a book.

Draw a simple use case diagram for the proposed system and identify the constraints or assumptions that you make. (For the moment, ignore the issue of fines because you would have to find out about the library's rules before including them.)