CHAPTER 3

High-Level Requirements

This chapter presents system use case diagrams and supplementary specifications. It explains how the system use cases are connected to the business use cases, and which business actors become system actors. The chapter explains how high-level use cases represent the highest-level requirements of a system (functional and nonfunctional), and how the supplementary specifications may be obtained based on the quality model of the recently consolidated ISO 25010 *SQuaRE Model*. The chapter also explains how to choose the granularity of a use case in order to avoid diagrams so detailed that the number of elements hinders understanding, and how to avoid diagrams that miss important use cases so that the analysis could be deluded by the apparent simplicity of an actually more complex system. At the end of the chapter, we present the preliminary conceptual model, which is a class diagram used to represent the concepts that are relevant to the system. This model is detailed in **Chapters 6** and 7, but here it is briefly introduced just to stress its importance in order to find the correct and complete set of requirements that a system must implement.

Keywords

Use cases; actors; requirements; ISO 25010; SQuaRE; supplementary specifications; preliminary conceptual model

Key Topics in this Chapter

- System Actors
- System Use Cases

- How to Find System Use Cases in the Business Model
- Requirements
- Preliminary Conceptual Model

3.1 Introduction to high-level requirements

Once the business has been reasonably understood and modeled, the requirements analysis may begin. Although there are several approaches to represent requirements, this book presents a technique based on use cases.

The *system use case* is an individual process that is identified from the business activities (**Section 2.4**). *High-level* (or *brief*) system use cases represent the more general *functional requirements* of a system. The *annotations* on these use cases represent the *nonfunctional requirements*, that is, the constraints and qualities related to those functions. The *supplementary requirements* are nonfunctional requirements that apply to the system as a whole – not only to individual functions. They may be represented in the *supplementary specification* document (**Section 3.5.8**).

System use cases are useful for many activities related to systems development, such as:

- *Definition and validation of the system architecture*: In general, classes, associations, and attributes that form part of the system architecture are obtained from the use case[1] texts (**Section 3.6**).
- *Creation of test cases*: Use cases can be seen as a basis for system and acceptance tests (**Section 11.7**), where the functionality is tested from the point of view of the user.
- *Planning iterations*: Each use case receives a priority (**Section 4.3**), and the effort to develop it is estimated (**Section 4.2**), so that the team can decide

on which use cases to develop at each iteration.

- *Basis for user documentation*: Use cases are descriptions of the system's normal operation flows, as well as alternate flows that represent how to deal with eventual exceptional conditions (**Chapter 5**). These descriptions are an excellent basis for starting the user manual, because all possible functionalities are described there in a structured and complete way.

Each use case represents a coherent set of functional system requirements. Usually, more than one function is related to a single use case, especially if it is a complex one. Some functions, on the other hand, may be associated to more than one use case. In some situations it may also happen that a function corresponds to a single use case and vice versa. This usually occurs with very simple use cases such as reports and entity management.

There are at least three approaches for requirements and use cases:

- Create a list of functional requirements, and then identify use cases associated with them.

- Create a set of use cases, and then extract the functional requirements from them.

- Consider that use cases *are* requirements. The brief version of the use cases corresponds to high-level requirements and the expanded use cases correspond to the complete set of requirements.

In this book, the third approach is taken, because it is simpler, straighter, and avoids generating different documents with the same goal. Requirements elicitation and analysis corresponds, then, to the discovery of system use cases and their properties.

3.2 System actors

A *system actor* is an entity of the real world that interacts with the system through the use case. Actors may be roles performed by people such as customers, publishers, sellers, operators, etc. Actors can also be external systems, that is, systems that are outside the scope of the project being developed.

Human actors or external systems interact with the target system by sending and receiving information through an interface. In the case of human actors, the information is usually exchanged through data entry devices such as keyboards, mice, or other special devices. Those actors receive information from the system through interfaces such as screens, printers, or other special devices.

Communication with actors that are external systems usually happens through a computer network. In this case, the communication interface consists of the network and its protocols.

The idea of external systems as actors must not be confused with internal ones that are components of the system under development. For instance, a database management system (DBMS), used to implement data persistence for the system under development, is not an actor but a component of the inner architecture of the system. The following rules may help to appropriately identify external systems that could be actors:

- System actors are *complete information systems*, and not only libraries of classes or components. These systems store their own information, which can be exchanged with the system being developed. That information may change independently from the system being implemented.

- System actors are *out of the scope of development*, that is, the team will not necessarily have access to the internal design of these systems, or the ability to change it. The team must communicate with an system actor using the system actor's own definitions, because they cannot be modified.

Some approaches consider that there are two kinds of actors: *primary*

and *secondary*. Primary actors are the ones whose goal the use case is trying to satisfy, while secondary actors are actors that just provide some service to the process under discussion. Examples of secondary actors are printers, web services, or people that must provide information or confirmation, as for example, a clerk that confirms that the client has paid in cash. In this book we do not follow that distinction: we distinguish use case actors into *human* or *system* actors only.

3.3 System use cases

System use cases differ from business use cases (**Section 2.3**) in some aspects. For example, business actors may spend days or even weeks performing a business use case, while system use cases are often executed in a short period of time, usually minutes, with one or a few actors interacting with a system and obtaining a consistent and complete result for at least one of their goals. System use cases also must be performed without interruptions while business use cases are not restricted in that respect.

Another fundamental difference between a business use case and a system use case is that the business use case is usually performed by many human actors, while the system use case is normally performed by a few (sometimes just one) human actors. The fact is that if a system use case is going to be performed by more than one human actor, then they should be interacting at the same time with the system, and that is not a common situation. Usually, each human actor accesses the system at a time of her convenience, accessing necessary data and performing the necessary actions. This leads to a sequence of system use cases and not to a single use case, as explained later. However, sometimes more than one human actor may be available online; for example, at a supermarket when a customer is buying goods, a supervisor may be summoned to perform actions that the cashier is not allowed to do (e.g., cancel a sale). The point is that the supervisor must be available online and the other actors must wait for her to show up before proceeding.

On the other hand, external computational systems may be considered *online* actors, because they are available continuously. For example, it can be assumed that a credit card operator is available online continuously. At the moment a customer decides to make a payment with her credit card, the system actor *Credit card operator* will be available.

A high-level system use case is represented only by a name inside an ellipse. It is usually associated to one or more actors as seen in **Figure 3.1**.

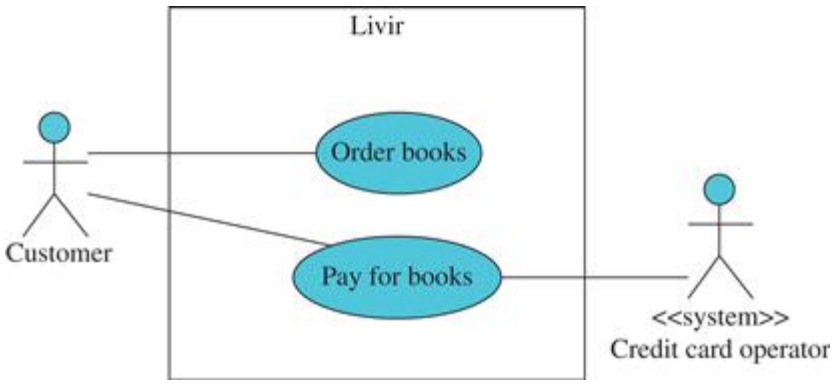

FIGURE 3.1 System use case diagram.

In the diagram in **Figure 3.1**, the ellipses represent system use cases. By default, an actor is a role performed by a human. Other kinds of actors may be represented by the use of stereotypes, as shown in **Figure 3.1**, where the stereotype ≪*system*≫ indicates that *Credit card operator* is an external system, not a human.

The use case diagram is a very popular UML diagram, but it is also frequently misunderstood. It is usual to see these diagrams with dozens of use cases and also some of their fragments attached. However, during Inception, the important thing is to know what the major processes of the system are, and not to detail them. Thus, the presence of fragments in the diagram, and the use of the *include* and *extend* relationships between use cases (which sometimes reveal part of their internal structure) is not advisable. Concerns about fragments and use cases including or extending others may be left for the moment when those high-level processes are detailed by system sequence diagrams (**Section 5.8**).

Usually there is not enough information to discover all of the fragments at this time. Why should only some of the use case fragments be shown and others not shown? Simply show none at all in the diagram during Inception! This prevents the diagram from having a large number of ellipses, which can hinder its comprehension. Thus, there is a need for strong criteria to decide which use cases should be maintained in the diagram to avoid, on one hand, a large number of excessively detailed processes, and on the other hand, too few processes, which could lack important features of the system.

The rule is to consider as a use case only those processes that can be performed in isolation. Partial processes that must *necessarily* be performed during other processes must not be represented in the system use case diagram.

However, even following this guide the number of use cases in a real-world system may still be too high, so that dealing with them may become difficult. To reduce the number without losing information and precision a second rule may be used. It consists of grouping use cases that are somewhat related, especially if that may be done more than once, like a pattern. For example, there could be four use cases such as *Create book*, *Retrieve book*, *Update book*, and *Delete book*, or just one use case called *Manage book* that includes the four single processes. This is a pattern, because it may be repeated for other concepts: *Manage publisher* may be used instead of *Create publisher*, *Retrieve publisher*, *Update publisher*, and *Delete publisher*. This pattern is known as *CRUD*, which is an acronym for *Create*, *Retrieve*, *Update*, and *Delete*.

The next subsections detail those rules and present other criteria for achieving the best use case granularity.

3.3.1 Single session

A good system use case must be performed in a single session.[2] This means that it should begin and finish without interruption. For example,

the registration of an order is made during a single session of the system, involving the identification of the customer, book selection, price visualization, payment, address selection, etc. Each of these aspects is a functional requirement of the system.

Use cases must be performed as single processes. Processes that can only happen in the context of other processes are just fragments, not use cases.

In the case of the Livir system, calculating taxes is something that will happen only during the process of ordering books. Let's suppose that the requirements of the system do not determine that it must be considered an independent process (although it could be, if requirements were different). In this case, *Calculating taxes* is not a use case, and because of that fact, it should not be included in the diagram. **Figure 3.2** illustrates the situation that must be avoided – using fragments in the diagram.

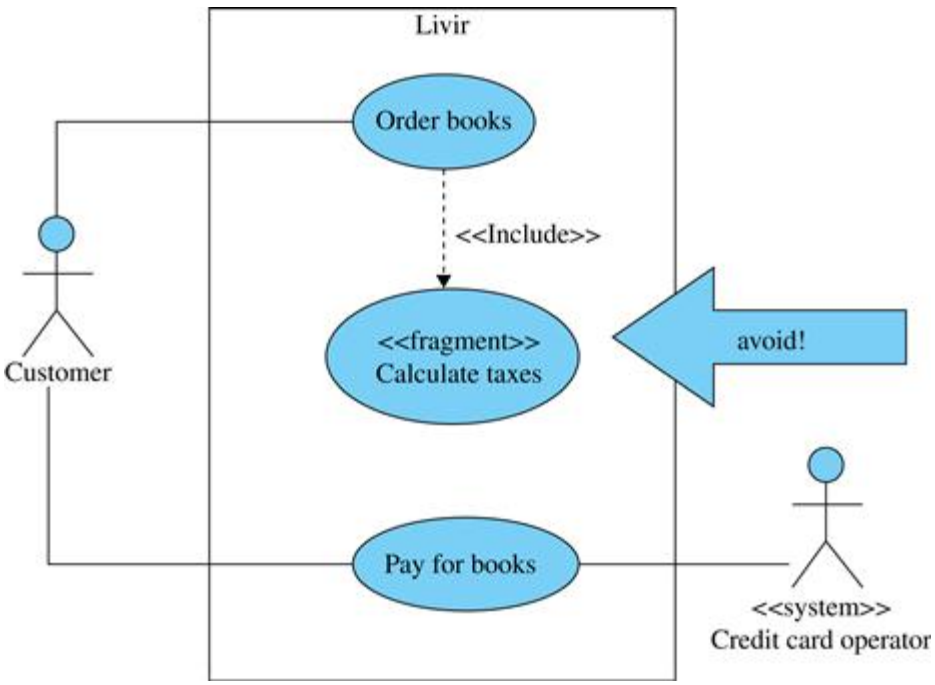

FIGURE 3.2 Example of a use case fragment.

On the other hand, *Order books* can be considered a use case because it is a process with a well-defined start and finish, it occurs in a contiguous

time interval (without interruptions), and it produces a sound result (the order is registered).

In the example represented in **Figure 3.1**, there are two possible scenarios that could be investigated:

- The system will only confirm the order if the payment is made online (it is not possible to save the shopping cart). In this case, the process of ordering books and paying are just fragments of a single use case that could be called *Buy books*. If this is the real requirement, then the diagram must be changed and the two use cases replaced by a single use case named *Buy books*.

- The system will register the order, but not necessarily proceed to checkout, that is, the shopping cart may be stored until the customer decides to finish the order (which can be done immediately or at some other time). This is the case represented in **Figure 3.1**. In this case, *Order books* is one use case and *Pay for books* is another use case. They can be performed immediately one after the other, or with an interval of some days, and because of that, they should be considered independent use cases.

On the other hand, books are not automatically delivered when the order is paid: someone is responsible for verifying, from time to time, which orders have been issued; that person collects books in stock, and sends the package. The delivery process may happen just after the order is paid, but it can also happen on the next day (or even six to eight weeks later). This fact characterizes *Delivery* as a different use case. The *Delivery* use case has buying books as a precondition, but buying is not part of delivery: delivery just happens after it. Thus, buying and delivering, although sequential and even part of the same business use case, must be considered as different system use cases, as shown in **Figure 3.3** where the *Send books* use case is introduced.

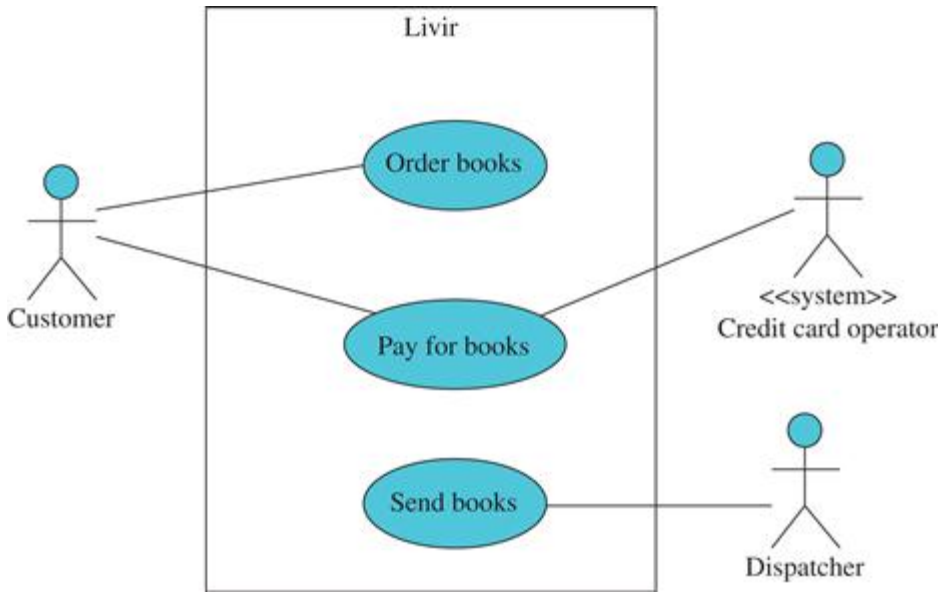

FIGURE 3.3 Use cases related to the business process of selling books.

3.3.2 Interactive

A use case must also be interactive, meaning that an actor must exist to interact with the system. The internal processes of the system are not use cases no matter how complex they are. On the other side, a simple query on some information may be a use case if there is an actor that starts it.

The aforementioned internal processes are either part of a use case (as, for instance, *Calculate taxes* is part of *Buy books*), or supplementary requirements (as, for instance, *Store data at a relational database*). In the case of supplementary requirements, they are usually implemented by internal mechanisms that do not depend on user interaction. During the Inception phase they are recorded as supplementary specifications (Section 3.4.8) or nonfunctional requirements (annotations on use cases – Section 3.4.5), so that the risk and effort that they imply is not underestimated.

3.3.3 Consistent result

A use case must produce a consistent result, be it a complete entry or transformation on a piece of information, or simply a query where rele-

vant information is passed to a user. A use case cannot leave the information at an inconsistent state at its end. For instance, the registering of an order cannot be concluded without the identification of the customer and the books she is ordering, or else the information about the order would be incomplete regarding the business rules; the bookstore could not resume the order or charge it without knowing who the customer is and which books are ordered.

To decide if a use case has a consistent result, one could think like this: only a complete process is a system use case, in the sense that a user could go to the computer, turn it on, perform the process, and at its end, turn off the computer, because the process is complete, and some business goal was obtained (some relevant information was received or updated by the user).

This excludes from the use case definition fragments such as *Calculate taxes*, in the case of the bookstore example, because those taxes are calculated only inside the ordering process, and not as an isolated process. That also excludes operations such as *Login*, because performing a login and then turning off the computer cannot be seen as a complete process. It may be just part of one or more use cases (**Figure 3.4**).

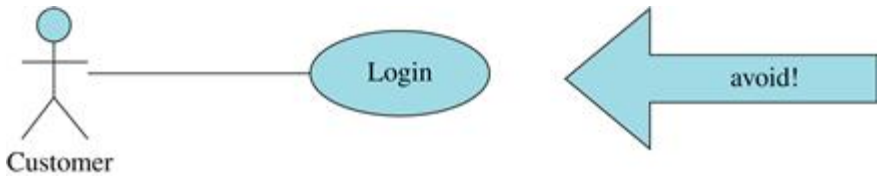

FIGURE 3.4 Login must not be considered a complete use case.

To avoid representing *Login* as a use case or even as part of one it may be simply assumed that every system operation could only be performed by a user that is regularly identified and authorized to perform it. This mechanism is part of the technological design of the system and does not need to be considered in the high-level use case model.

On the other hand, it is possible that complete use cases occur inside other use cases. For example, the process of registering a customer can be considered a complete use case. But that process can also occur during book ordering, especially if it is the first time that the customer is using the system, or when she needs to update data (**Figure 3.5**).

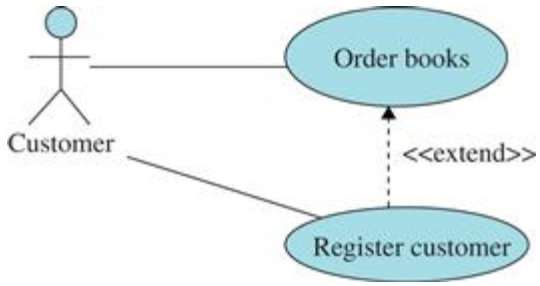

FIGURE 3.5 Independent use cases that may be related.

The dependency association between use cases is stereotyped with ≪*extend*≫ to indicate that the process of ordering books may occasionally be extended by the process of registering the customer. Both processes may occur independently from each other, but the customer registration can also occur during book ordering.

To avoid abusing ≪*extend*≫ in the use case diagram, it is important to keep in mind that only complete use cases must be maintained in the diagram. Fragments like those shown in **Figures 3.2** and **3.4** must be avoided, even if they are extension points of a use case. These fragments will be addressed appropriately later, when the high-level use cases are detailed (**Chapter 5**). In practice, at this point, any association between use cases such as *extend* or *include* should be avoided in the diagram because they usually add no useful information for the Inception phase.

3.3.4 Essential

Two styles for writing use cases may be identified:

- *Essential use cases*, which do not mention interface technology.

- *Concrete* (or *real*) *use cases*, which are specifically written for a given interface technology.

During requirements elicitation and analysis, system use cases are considered requirements, not design. It is a common mistake to include among these use case actions that are purely related to an interface technology (such as *Open main window*, *Print report*, and *Login*). People prepared to handle that facet of design will decide on those actions later, after the requirements are discovered.

Ambler (2000) points out that essential models are more flexible, leaving more options open and more readily accommodating changes in technology. He also says that essential models are more robust than concrete representations because they are more likely to remain valid in the face of changing implementation technology.

Thus, essential use cases must be considered as the correct option during requirements elicitation, although technology details or options may be annotated to allow risk management and effort calculation. Later in this book (**Section 5.4.1**), the difference between essential and concrete use cases will be explained in more detail.

3.3.5 Brief

During Inception use cases usually are brief, meaning that they are described just by their name or, in some cases, by one or two sentences. However, this is not the only way a use case may be described. Later they will be extended and contain more details about the requirements (**Chapter 5**). **Cockburn (2001)** identifies three types of use cases with regard to the level of detail:

- *Brief*: A one-paragraph synopsis of the use case.

- *Casual*: Written in simple, paragraph, prose style. It is likely to be missing project information associated with the use case, and is likely to be less

rigorous in its description than a fully dressed use case.

- *Fully Dressed*: Expanded to include a main flow and alternate flows, as well as other sections such as postconditions, preconditions, stakeholders, and technological variations.

In this book we expect that use cases considered during the Inception phase will be brief. This means that usually their name is sufficient to explain their meaning. However, additional explanation is allowed for the sake of requirements comprehension. Also, if some key use cases have to be expanded in order to identify risks regarding their inherent complexity, this is also acceptable. Usually, fully dressed or expanded use cases will be useful only after Inception, when requirements must be detailed adequately.

3.3.6 System boundary

One of the decisions a team must make when designing system use cases is where to place the system boundary. Graphically, the boundary is only a rectangle that is placed on the diagram. Inside it are the use cases and outside are the actors. In business use case diagrams, the system boundary represents the limits of the organization (company, department, etc.). Here, it represents the limits of a computational system.

A decision was made earlier on which workers to include in the automation border. One example is the clerk that helps the customer buy books. In that case, when modeling system use cases the *Clerk* disappears, and the use case is performed by the customer itself.

However, if it was not a virtual bookstore, but a regular one where customers buy at a store, should the clerk be kept in the use case diagram? For a use case to be as essential as possible, it is recommended that only actors that are really interested in the use case be kept in the diagram. The clerk of the example is just a proxy of the customer; the clerk has no personal goals in the process of buying books. It acts simply as a system

interface. Whether the bookstore is virtual or not, the essential use case should be the same, because the same information would be exchanged between the customer and the system; the clerk would only transmit information as she receives it. In this case, the only actor would be the customer and the clerk should not even appear in the diagram. In this way, the analysis will produce use cases that are technology independent.

Someone could ask the following: What if the clerk must indicate that she assisted in the sale so that a percentage would be paid to her? In that case, what we have is a use case that is different from the one mentioned in the last paragraph. Here, both the customer and the clerk *should* be actors: the customer is interested in the books and the clerk is interested in the percentage. Also, if the clerk may perform actions that are not allowed for the customer (e.g., overriding a product price) she must be considered an actor and may not be removed from the diagram.

3.4 How to find system use cases in the business model

In order to discover system use cases and actors, the team may examine the business use case diagram with the defined automation boundary (scope of automation).

First, the actors that are really interested in the process to be automated should be identified. These are the business actors that interact with the business use cases inside the automation scope, and the business workers that interact with such use cases but that are not themselves inside the automation scope. In the case of **Figure 2.11**, the following system actors could be identified:

- *Customer*, *Publisher* and *Credit card operator*, because they are business actors that interact with use cases that will be automated.

- *Acquisition manager*, because it is a business worker that will not be auto-

mated, but that interacts with a use case that will be.

On the other hand, the following will not be considered system actors:

- *Clerk*, because it is a business worker that will be automated.
- *Marketing Department*, because it is a business worker that does not interact with any use case inside the automation scope.

The business workers whose functions will be partially or totally supported by the system may be important sources of information, because knowing how they perform their duties currently may be crucial to understand how the system must work. Moreover, knowing how the duties are really performed may be valuable because sometimes what happens in practice is not what is written in the procedure books.

Another source of requirements is the business actors that will become system actors. However, they will not always be available for requirements elicitation, and, in that case, they should be replaced by domain specialists. For example, it may not be possible to interview the customer of a bookstore that does not exist yet. Thus, someone that knows the business and can explain how it is supposed to work will be a key person for obtaining the right requirements. If the publishers and credit card operators are also not available, it still would be possible to examine existing documentation on the communication protocols and services offered, so that the interfacing requirements with them can be known. In the case of the customer, however, such a possibility would not be feasible.

Once system actors have been identified from the business use case diagrams, it will be possible to observe the activity and state diagrams that were produced during business analysis in order to verify which activities performed by the system actors can be considered system use cases.

By analyzing the activities of **Figure 2.10**, we can see that the activities performed by the *Customer* are:

• *Order books*.

• *Pay for books*.

The only activity of the *Publisher* is *Send books*, and the only activity of the *Credit card operator* is *Confirm payment*.

Not every activity of the original *Clerk* becomes a system use case. As seen in **Figure 2.7**, the clerk was split into two actors: *Acquisition manager*, which will not be automated, and *Clerk*, which will be automated. The activities performed by the *Acquisition manager* role may become system use cases. However, the activities performed by the business worker *Clerk*, which is being automated, become internal actions of the system. For example, the activities *Register order* and *Inform availability and total* in **Figure 2.10** do not become system use cases; they are just part of the use case *Order books* initiated by the customer. However, *Order books from supplier* is a system use case whose actor is the *Acquisition manager*.

Regarding processes that will be identified as use cases, one must still consider two concerns:

• Do the activities connected by a flow necessarily occur immediately or can they occur in different moments? In the example, are *Order books* and *Pay for books* activities that must occur in a single session of use of the system, or can they be performed in different moments? The answer will depend on the way the business is organized, that is, it will depend on the business rules. If the company decides that the order is received only after the payment is made, then those two activities form a single use case. However, if the company decides that it is possible to register an order (save a shopping cart) and pay for it some other day, then there are two different use cases.

• Are the activities selected complete and sound? For example, is *Send books* a good description of the activity performed by the publisher? By the sim-

ilarity to the process performed by the customer, it can be inferred that the publisher will first receive an order, and then it will send the books it has in stock. However, the payment in the business-to-business case is usually not made immediately – an invoice is generated and paid by the bookstore with a predefined deadline (for example, four weeks after purchasing).

At this point, the diagram of **Figure 2.10** should be revised. The publisher does not simply *Send books*. It is going to receive the order, send an invoice for payment, and finally send the books that are available. The books must be received and registered by some worker at the bookstore. That worker will be a new system actor and could be called *Deposit worker*.

A recurrent question is if actors correspond to security profiles. This is not necessarily the case. The goal behind the identification and modeling of actors is more related to the process of finding and organizing requirements than to the process of granting access to the system. A good software design will treat access permission in a dynamic way, allowing user profiles to be created, and permissions to be associated to the different profiles of individual users. As that aspect of the system is not domain dependent it is not adequate to model it within the domain. In other words, use case actors should not be considered as security profiles. Those profiles will be created dynamically with the use of a generic domain-independent mechanism.

The state machine diagram presented in **Figure 2.15** is also particularly helpful in terms of system use cases that must be analyzed. Each change of state of a book (main business object), must be done by someone, and that "someone" is an actor performing a use case. Thus, from the transitions in that diagram a new set of system use cases was discovered as shown in **Figure 3.6**.

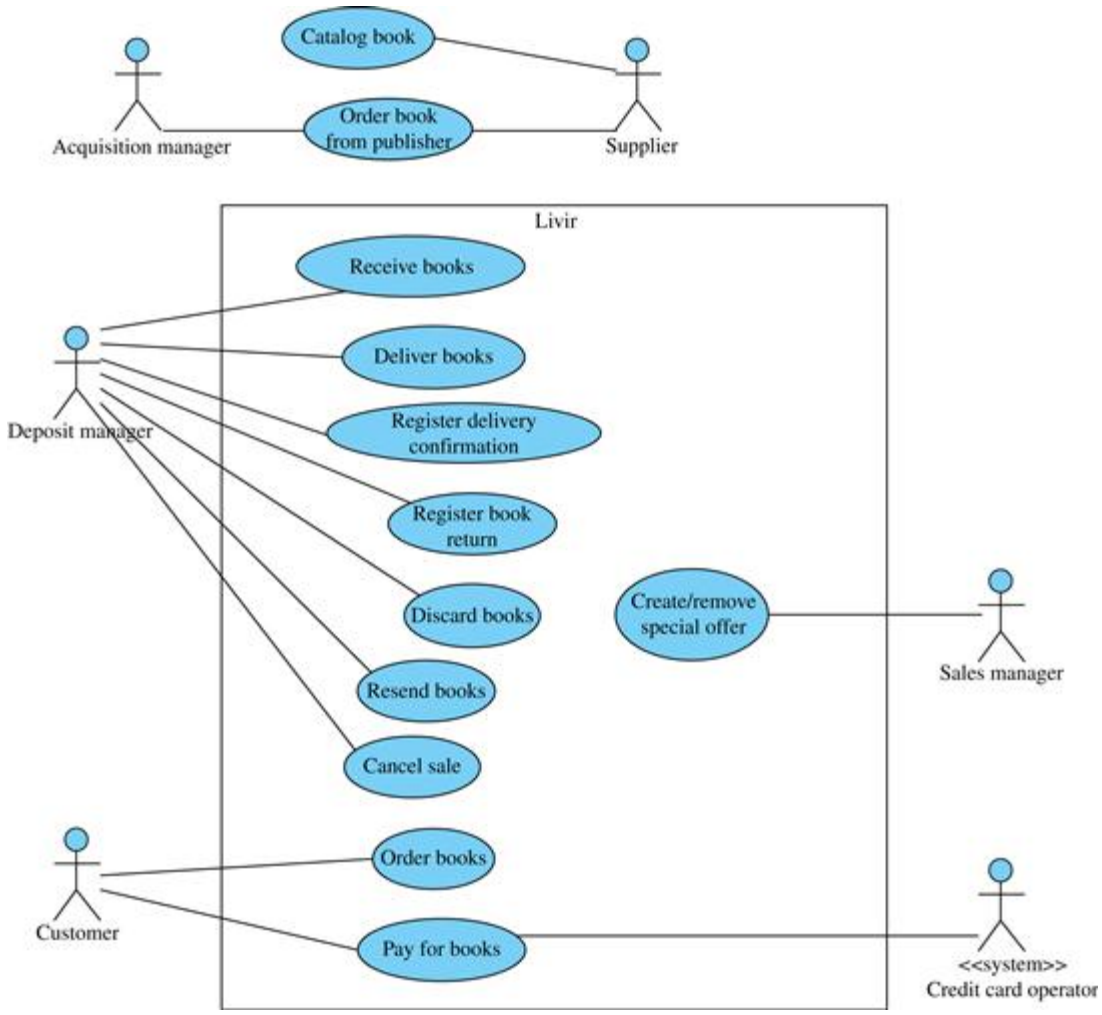

FIGURE 3.6 System use case model completed with information discovered on the state machine diagram of **Figure 2.15**.

In **Figure 3.6**, the use cases *Catalog book* and *Order book from publisher* are maintained outside the scope of the Livir system. This is due to the fact that it was discovered at a point that such use cases would be performed within the publisher system, not within the system under development. As they are not included in the bookstore system, they may be maintained in the diagram, but outside the system boundary, because they are out of scope and will not be implemented.

After making sure that it is understood that they will not be implemented, they may be removed from the diagram, in order to simplify it and avoid later confusion with people who may not understand that what is outside the boundary will not be implemented. In that case, their

names may be listed in an *out of scope list*, for recording purposes.

The use cases associated with the new actor *Deposit manager* are necessary to cover most of the transitions indicated in the state machine diagram of **Figure 2.15**. It is interesting to notice that some transitions are performed outside the scope of the company. For example, when a customer decides to return a book, this is not done through the bookstore system; she simply sends the book back, usually by mail. The information regarding the return only reaches the bookstore when it receives the book. Thus, the use case referred to here is *Register book return* and not *Return book*, because the registry is made by the deposit manager, and not by the customer. This is similar to the registry of the arrival of books, represented by the use case *Receive books*, because when the publisher sends the books it cannot be registered in the bookstore system directly (unless both systems are integrated). Only the arrival can be registered by the bookstore. This is a fundamental difference between a business use case and a system use case: in a business use case the process as it happens in the real world may be described as part of the use case, but when the focus is on the system, only processes that involve the system may be represented.

3.5 Requirements

Identifying system use cases is an activity that is part of the *Requirements* discipline of the Unified Process. Requirements elicitation and analysis is a significant part of the Inception phase. The team can and must use every available source to identify requirements (specialists, users, documents, interfaces, literature, etc.), and, for each source, a set of functions that the system must perform may be identified.

3.5.1 Requirements elicitation

Requirements elicitation corresponds to the search for information about the functions that the system must perform, and for the constraints under

which the system must operate. In the approach to requirements presented in this book, they will be recorded in the system use cases.

Another way to register requirements without use cases would be by using a document that consists of a list of functional requirements, possibly accompanied by a list of constraints (**Sommerville, 2006**). The advantage of using use cases instead of a list of functions is that a good use case has a well-defined granularity, which allows for the generation of a high-level requirements list that is much more comprehensible than a list of individual functions. Usually the number of high-level use cases is much lower than the number of individual functions. The individual functions will appear in the use case expansion, when the high-level use cases have their structure detailed.

In the case of the Livir example, requirements elicitation will allow the team to discover that the system must control buying and selling books, receiving payments, allow damaged books to be registered, generate sales reports, verify if books are available in stock, etc. These operations and many others will constitute the functionality of the system, and this is why these requirements are called functional requirements. These functions will be incorporated into one or more use cases.

On the other hand, during requirements elicitation the analyst may face business rules or constraints on how functions must be performed by the system. For example, a business rule could state that the bookstore only sends books after the payment is confirmed. That kind of rule is a nonfunctional requirement that can be recorded as an annotation or remark on the use case itself, in order to be recalled and verified when the use case has to be detailed (**Figure 3.7**). Alternatively it could be recorded separately as a numbered list or spreadsheet, with a reference to the use cases through a unique number.

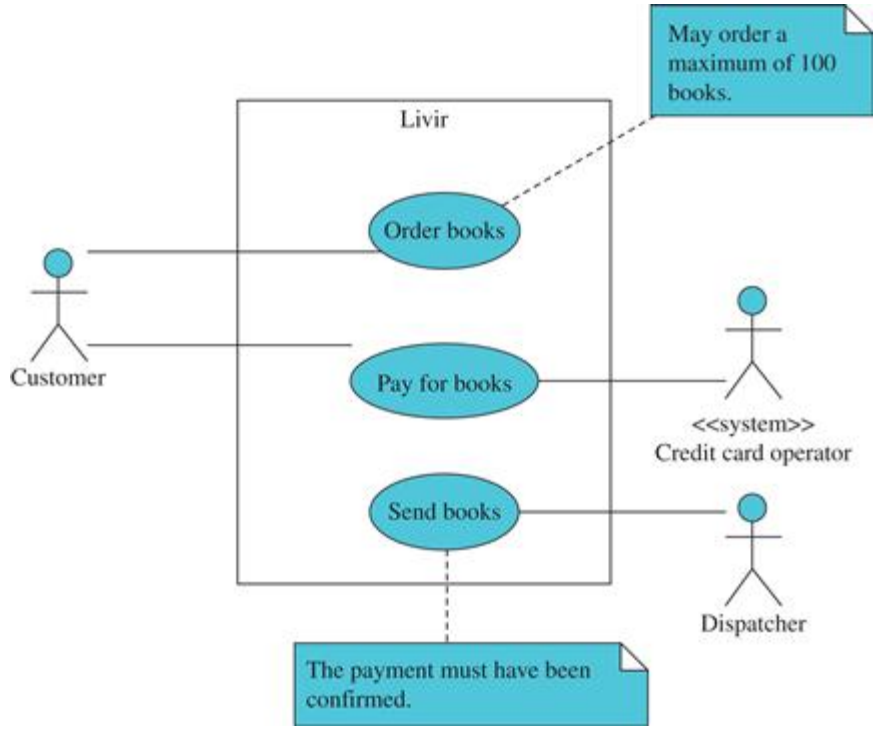

FIGURE 3.7 Nonfunctional requirements annotated in use cases.

As can be seen in **Figure 3.7**, the use case diagram with notes will quickly become too complex to be useful. This is why it is recommended that such annotations should be recorded inside the use case specifications or in a separate document. Most CASE (Computer-Aided Software Engineering) tools allow that a diagram element such as a use case have a specification window as shown in **Figure 3.8**. One exception to that rule would be the case when the annotation is absolutely required to help understanding the diagram. However, this is seldom the case.

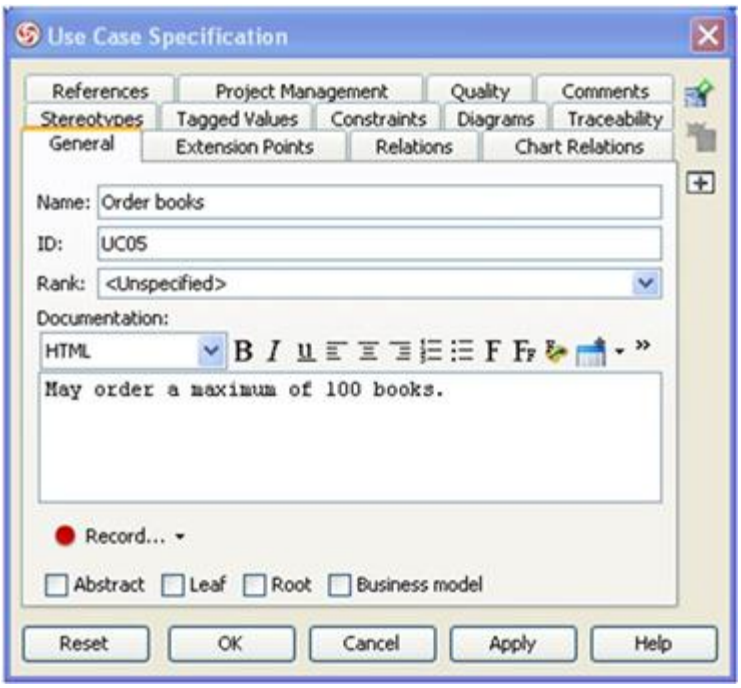

FIGURE 3.8 Example of specification window for a use case. Image produced with the CASE tool Visual Paradigm™ Community Edition.

3.5.2 Eliciting requirements is not design!

A system that is going to be analyzed is like a forest. To explore an unknown forest one cannot start by examining each plant and animal. There is a saying that states that some people cannot see the forest for the trees. The forest is the system and the trees are the requirements. Only at the end of the process could a team say that they acquired knowledge about the smallest parts. First, a view of the whole must be taken, and only after that the details may be studied.

Thus, the Inception phase must provide a view of the whole – so that what is more important can be seen first – and then the whole can be divided into parts so that the details can be analyzed and finally a solution designed. The organization of iterations in the Elaboration and Construction phases correspond to dividing the forest into sectors, in order to see each sector at a time, and in that way, being able to deal with the inherent complexity. Thus, one of the final goals for the Inception phase is to organize work division in terms of use cases, which will be ex-

plored during the upcoming iterations.

During Inception, requirements elicitation must be quick and generic. The right way to do this is to look at the extension of the requirements, not their details. The analyst must understand the extension of what the system must do, without detailing how it will do that. Only during the Elaboration iterations will the requirement analysis be deepened.

The time used for requirements elicitation time must be about discovery, not invention. During this period, the team of analysts, with the clients, users, and other stakeholders will try to list most of the capacities and restrictions with no concern about detailing them. Details about requirements will be conveniently accommodated during the next iterations.

It must also be clear that requirements are something that a client asks for, and not something that the team designs. Some analysts mix requirements gathering – that is, the memory of the demands of the client – with the beginning of the system's design. One example of this kind of confusion is a set of requirements involving relational database design. Unless there are legacy systems that must be compatible to the new system, how can it be justified that a set of relational tables is a client demand? That could eventually be possible with some sophisticated clients with a degree in Computer Science, but it is not the general rule. Database tables are part of the *solution* domain, not the *problem* domain. The analyst must seek the requirements that correspond to the client needs and goals in terms of information. Later, she may decide if that information would be stored in a relational database or in another structure.

3.5.3 Requirements challenges

The requirements document, which can be formed by the use case diagrams with annotations and by the supplementary specifications, must register the capacities of the system and the conditions under which it must operate. The challenges regarding requirements are, at a minimum,

the following (**Pressman, 2010**):

- How to *discover* requirements.
- How to *communicate* requirements to the other phases and teams of the project.
- How to *recall* requirements during development in order to verify if they have all been implemented.
- How to *manage* requirements change.

It would be useless to develop a nice use case diagram, and later not be able to know if the requirements incorporated there were included in the design. The existence of automatic mechanisms to conduct this verification is crucial. Therefore, it is important to maintain traceability relations between use cases and other parts of the design. The following chapters show how those relations among design artifacts are obtained.

It is necessary to keep in mind that requirements necessarily change during the development of a project. Thus, the change must be managed, not avoided, after Inception.

Sometimes, requirements change after the system is deployed. Context conditions, regulations, company policies, or work methods may change anytime. Although the analyst cannot foresee these changes, mechanisms to accommodate them may be created in order to ease the change process when it is necessary. There are specific design patterns to deal with these requirement instabilities (as, for instance, the *Strategy* pattern, presented in **Section 7.6**). The changes are totally unpredictable. If the system is not structured to accommodate the changes in requirements it would probably be hard to implement them.

This kind of situation makes *requirements-driven analysis and design* processes (**Alford, 1991**) inadequate for most systems. Using require-

ments as a base to support system architecture is like building a house over moving sand; when requirements change, the structure suffers. However, the Unified Process (UP) uses a much more stable basis for the architecture, which is based on classes and components that encapsulate information and behavior.

These classes implement functionalities that combined allow the implementation of the requirements. If requirements change, the combinations change, but not the basic structure. This kind of architecture follows the *open-close principle* (**Meyer, 1988**), in the sense that it is always closed for modification (it works), but open for extension (it may accommodate new functionalities).

It is important to trace the origin of each requirement (e.g., a business actor, a business worker, the client itself, or even a domain specialist), because it is necessary to validate requirements with those sources, verifying if they are well written, complete and sound.

Sometimes it may also occur that different people or departments present different specifications for the same requirement. In that case, it is necessary to produce an agreement among them, or identify who has the highest authority to determine the acceptable form for the requirement.

3.5.4 Evident and hidden functional requirements

Functional requirements can be optionally identified as *evident* or *hidden* (Gause and Weinberg, 1989):

- *Evident* functional requirements are functions that are performed with the user's knowledge. These requirements usually correspond to information exchange between the user and the system, such as queries and data entry, which flows through the system interface.

- *Hidden* functional requirements are functions performed by the system

without explicit knowledge of the user. Usually these functions are math operations and data updating performed by the system without explicit user knowledge, but as a consequence of other functions performed by the user.

Hidden requirements are performed internally by the system. Thus, although they do not appear explicitly as use cases, they must be adequately associated to them in order to be recalled at the time of design and implementation. Thus, they could also be added as annotations to a use case.

An example of an evident requirement is the production of a report on bestselling books, which is something a manager might require. An example of a hidden requirement is applying a discount policy to a sale. In that case, the user does not explicitly request that the system perform the operation. As it is an activity that the system performs automatically, it is a hidden requirement.

3.5.5 Nonfunctional requirements

Nonfunctional requirements are constraints or qualities that may be linked to specific functions of a system (for example, “an order cannot contain more than one hundred books,” “the order transaction must be preserved if communications are broken,” etc.), and therefore they can be treated as annotations on the use cases that incorporate the respective functions. Sometimes, however, nonfunctional requirements may be general, that is, not necessarily attached to a function (for example, “the system must be implemented in Java”), and in that case, they will appear in the supplementary specifications document. Constraints and qualities that are specifically attached to a function are called *nonfunctional requirements*, and general constraints and qualities are called *supplementary requirements*.

There are two kinds of nonfunctional requirements:

- *Logical issues*: Business rules attached to a function. For example, during sales registration, a series of constraints could be considered, such as not closing the sale until the credit card operator confirms the payment, or not closing the sale if the last delivery to the same address has been returned due to invalid address.

- *Technological issues*: Constraints and qualities related to the technology used to perform the function, such as, for example, the user interface, the kind of communication protocol, security constraints, fault tolerance, and so on.

For example, establishing that the user interface to perform an order must follow a design pattern based on a sequential flow of screens is a technological constraint (interface constraint) about the way that function must be performed. Another example of a nonfunctional requirement is "the payment confirmation must not take more than 5 seconds." This is a technological constraint related to system performance and it would affect the way a designer would think about the communication mechanism with the credit card operator. In this case, the system design would have to consider seriously the broadband connections to the operators.

A requirement does not establish how the constraint will be implemented, it just demands it. The design and implementation of the system must meet the requirement in one way or another, or otherwise the analyst should negotiate with the client for some requirement flexibility.

3.5.6 Permanence and transience of nonfunctional requirements

One of the most fundamental features of a requirement is whether a given nonfunctional or supplementary requirement is *permanent* or *transient*.

Nonfunctional and supplementary requirements can be considered

permanent (they are not expected to change) or transient (they are expected to change) depending on a decision made by the client. Permanence or transience is not an intrinsic feature of a requirement: it is decided depending on the convenience. The same given requirements can be considered permanent or transient depending on what is desired regarding the time and cost of software development and maintenance.

If one invests in flexible design, so that most constraints are transient, less effort will be spent during maintenance in order to accommodate changes. However, the cost of that flexible design during development may become prohibitive for some projects. It will always be a good idea to ponder which constraints should really be treated as transient requirements.

For example, a supplementary requirement could establish that the Livir system must deal with a single currency: the dollar. If that requirement is considered permanent, then the system will be designed for a single currency ("dollar" could even be a data type used to define variables and attributes). However, if the requirement is considered transient, then even if there is no other currency being used today, the whole system must be prepared to accommodate future currency, or even more than one currency at a time.

The consequences of deciding that a requirement is *permanent* are the following:

- It is cheaper and quicker to develop the system.
- It is more expensive and difficult to change the system if, by any chance, the requirements change in the future.

On the other hand, deciding that the requirement is *transient* has the following consequences:

- It is more expensive and complex to develop the system (it should accom-

modate functionalities for changing the currency, for example).

- It is easier and quicker to maintain the system (if the currency changes, the system is already prepared to accommodate that with a simple reconfiguration).

Thus, it is not the nature of the nonfunctional requirement that will decide whether it is permanent or transient. It is the client, with the help of the team, that must make the decision. The ideal situation would be to list the requirements of greater importance (those that are really expected to change in the near future with a bigger impact on the system) and consider them transient, while leaving the others as permanent.

3.5.7 Mandatory and desired requirements

Requirements may also be considered *mandatory* or *desired*,[3] that is, those that must be obtained by any means and those that could be obtained if no bigger issue hinders the development process.

In the case of functional requirements, that classification indicates development priorities. If there is flexibility in the contract such that only the most important use cases are implemented if there is no time to implement them all, then the team must know which ones are mandatory.

However, if the team is able to make a good estimate of the effort needed to develop the system, and if they have a good history of accurate estimation, there will be less motivation for such distinction regarding functional requirements, because all requirements are expected to be implemented in time.

On the other hand, nonfunctional and supplementary requirements are much more unpredictable than functional ones with regard to effort estimation. Thus, in some cases, it may be necessary to consider those requirements with some flexibility.

In this case, some restrictions are defined such that they must be obtained by any means and others may be considered simply desirable, and some time is allocated to pursue them.

For example, in the case of the Livir system, the requirement for using a web interface may be considered a mandatory supplementary requirement. In that case, other solutions are not acceptable. However, additional access through a cell phone could be considered as desirable requirement, because such access is not absolutely necessary for the effective success of the system.

Nowadays, with the formalization of software development contracts, there is less flexibility regarding desirable requirements. In most cases, the developer must state which requirements will be implemented, how much time it will take, and how much it is going to cost. Stepping outside that line may require the payment of fines or even canceling the project.

3.5.8 Supplementary requirements

Supplementary requirements are all types of constraints and qualities related to the system as a whole and not only to individual functions. For example, a supplementary requirement may establish that the system must be compatible with a given legacy database, or be implemented with a given programming language, or even follow a given *look and feel*.

Care must be taken when supplementary requirements are defined. A requirement such as “the system must be easy to use” is not sufficiently clear. It would be better to say something like “novice users must be able to complete tasks without errors on their first attempt.” That gives a more precise idea of what must be designed to accomplish the requirement.

Nonfunctional and supplementary requirements may also be identified with different groups such as, for example, interface, implementation, performance, fault tolerance, etc. The goal of making such distinctions is to allow for better organization.

Although most UP practitioners would choose to use the *FURPS+* classification system (**Grady, 1992**) for organizing supplementary requirements, the newest source to decide on supplementary requirements classification is ISO/IEC 25010,[4] as shown in **Table 3.1**.

Table 3.1

Supplementary Requirements Classification Based on ISO/IEC 25010:2011 and Requirements Generating Questions

25010 Feature	25010 Subfeature	Requirements Generating Questions
Functional suitability	Functional completeness	Are these all necessary functions that must be implemented? May some of them be left outside the system scope? May some functions be implemented by a further project?
	Functional appropriateness	To what degree must the user activities be made easier by the system?
	Functional correctness (accuracy)	Are there accuracy specifications, that is, a desirable precision for data? Are there limits for tolerating imprecision?
Reliability	Maturity	Must special care be taken to prevent the system from presenting defects during use? Must critical parts be defined by formal specifications? What is the intensity and type of tests that must be performed to assure the system is free from bugs?
	Availability	What is the degree of availability required for the system? How many simultaneous accesses must be supported? How many hours per day? How many days per year?
	Fault tolerance	How should the system react in the case of externally provoked anomalies, such as communication interruptions?
	Recoverability	Must the system recover automatically in the case of a disaster? Under which circumstances must lost data and aborted processes be recovered?
Usability	Appropriateness recognizability	In what way should the software present itself to a potential user so that she may recognize its applicability? How should the software be packaged?
	Learnability	How are the concepts inherent to the software being presented to the user so that she can become competent in its use?
	Operability	To what degree must the product be easy to use and control? What kind of help must the system provide? What forms of documentation and manuals should be available? How are they going to be produced? What kind of information should they present?
	User error protection	What kind of protection against user error is necessary?
	User interface aesthetics	Which design patterns will be used for the interface to provide visual pleasure and satisfactory interaction?
	Accessibility	To what degree must the product be designed to attend to people with special needs?
Performance efficiency	Time behavior	Which time restrictions exist related to the software processes and functions?
	Resource utilization	Are there data storage space restrictions? Energy limitations? Communication network limitations?
	Capacity	Regarding processing capacity, what are the nominal values expected and what are the critical values? For example, the software may be designed to support up to 2,000 simultaneous

(*Continued*)

25010 Feature	25010 Subfeature	Requirements Generating Questions
		accesses, but assured to keep working when up to 10,000 simultaneous accesses occur.
Security	Confidentiality	To what degree must the information and functions of the system be available only to those authorized to access them?
	Integrity	To what degree must data and functions be protected against unauthorized modification by people and systems?
	Nonrepudiation	To what must the system guarantee that the records kept by it are effectively true so that their authors cannot deny them?
	Accountability	To what degree are user actions recorded by the system? What information is kept?
	Authenticity	To what degree must it be guaranteed that a logged-in user is really who she is supposed to be?
Compatibility	Coexistence	With which products must the software potentially coexist? Which tools and programming languages must be used? Is it necessary to run the system with legacy systems?
	Interoperability	Which other products must communicate with the system? To which systems must it send data? From which systems must it receive data?
Maintainability	Modularity	What kind of architecture will be used? Layers? Partitions? Components? Web services?
	Reusability	Will the system be produced from parts of other systems? Must the system be designed so that its parts could be reused in future projects?
	Analyzability	Will special techniques or tools be used to ease debugging of the code?
	Modifiability	Will special techniques or tools be used to guarantee that changes introduced in the system will not produce new defects?
	Testability	Will special techniques and tools be used to ease regression tests? Will automated test tools be used? Will TDD (test-driven development) be used? How are test assets produced during development going to be kept?
Portability	Adaptability	To what degree must the software adapt itself to other contexts besides the ones it was originally designed for? Must it be recompiled or just reconfigured? What can be configured in the system? Examples of configurable items are printers, currency, company policies, interface fonts and colors, language, etc. Transient requirements would probably be related to configurable items.
	Installability	What installation resources will be provided? Should installation be automatic? Is data migration automatic?
	Replaceability	What resources must the system provide when it replaces other systems with the same goals? What resources must the system provide when it is going to be replaced by other systems with the same goals? Should it generate data and configuration in formats widely understood such as XML?

(*Continued*)

25010 Feature	25010 Subfeature	Requirements Generating Questions
Effectiveness	Effectiveness	Which business goals on the real use environment must the system help to obtain? To what degree must the system be responsible for obtaining those goals in a complete and correct way?
Efficiency	Efficiency	What kind of return of investment (ROI) must the system produce to the client?
Satisfaction	Utility	In which way must the software be designed to help the user perceive the consequences of its use?
	Pleasure	To what degree must the system provide pleasure to its users?
	Comfort	To what degree must the system preserve or improve physical and mental user comfort?
	Trust	In which way must the system be designed so that the stakeholders trust it will do the work?
Freedom from risk	Economic risk mitigation	What kind of financial risk (including property and moral damage) must the system reduce?
	Health and safety risk mitigation	What kind of physical risk to people must the system reduce?
	Environmental risk mitigation	What kind of environmental and property risks must the system reduce?
Context coverage	Context completeness	To what degree must the system be used effectively, efficiently, without risks, and with user satisfaction in its use contexts? What are those use contexts? Are there legal requirements related to the use of the software?
	Flexibility	To what degree must the system be used effectively, efficiently, without risks, and with user satisfaction in use contexts other than the ones it was originally designed for? What are those use contexts?

Although this list is extensive, the team must keep in mind that it is only a classification to improve the ability to identify which requirements are really important. There is no need to seek nonexistent requirements, for example establishing complicated packaging requirements for a client that does not care about the way the software will be packaged.

It is also advisable not to lose time discussing if a given requirement belongs to this or that type. More important than deciding on its type is to know that it exists: long discussions on requirements classification do not add significant knowledge to the project.

Table 3.2 presents an example of supplementary requirements that could be assigned to the Livir example.

Table 3.2

Supplementary Requirements for the Livir System

25010 Feature	25010 Subfeature	Supplementary Requirement
Functional suitability	Functional completeness	
	Functional appropriateness	
	Functional correctness (accuracy)	
Reliability	Maturity	
	Availability	The system must be available continuously (24 hours a day, 7 days a week).
	Fault tolerance	Transactions partially concluded by the user in the case of temporary faults must be preserved for recovery for at least 24 hours. After that deadline, they may suffer rollback.
	Recoverability	In the case of energy, communication, or data failure, the system must become operational again without operator intervention.
Usability	Appropriateness recognizability	All access functions that generate value for each kind of user must be easily recognizable in the initial page of the system. Primordial functions must be highlighted.
	Learnability	Names of functions (e.g., buttons and menus) must conform to usual Windows names in order to improve users' ability to learn about their use.
	Operability	The use of the system must be clear to novice users: there must not be the need for supplementary help or explanation.
	User error protection	The application must prevent the user from entering inconsistent information.
	User interface aesthetics	
	Accessibility	The system must be configurable to many human languages.
Performance efficiency	Time behavior	During peak hours it is important that the system's performance does not degrade.
	Resource utilization	
	Capacity	Peaks of 10,000 simultaneous accesses are expected.
Security	Confidentiality	Data about sales and customers are only accessible by the customer herself, the sales manager, and workers authorized by the sales manager.
	Integrity	A customer may change her own data and open orders only. All other data may be changed only by authorized personnel. Different levels of authority may be defined.
	Nonrepudiation	The system must store in a tamperproof record the following data about each order: its contents, the date and time it was issued and received, and the identity of the customer.
	Accountability	
	Authenticity	

(*Continued*)

25010 Feature	25010 Subfeature	Supplementary Requirement
		Users' passwords must not in any case be visible by other users or system managers.
Compatibility	Coexistence	
	Interoperability	The system must be able to communicate automatically with credit card operators to allow confirmation of payment.
Maintainability	Modularity	
	Reusability	
	Analyzability	
	Modifiability	
	Testability	
Portability	Adaptability	The server must be installed on a single platform. The client application must be accessible at least by Internet Explorer, Google Chrome, and Mozilla Firefox. The client application must be sensible to the configuration of the user computer in order to identify the language, currency, and other definitions.
	Installability	The customer may access the system via the Web without the need to install any additional component except those that are usually available for browsers.
	Replaceability	
Effectiveness	Effectiveness	
Efficiency	Efficiency	
Satisfaction	Utility	
	Pleasure	
	Comfort	
	Trust	
Freedom from risk	Economic risk mitigation	
	Health and safety risk mitigation	
	Environmental risk mitigation	
Context coverage	Context completeness	
	Flexibility	

Not every field was filled because, as said before, requirements must not be invented, they must be asked for by the client. Thus, usually there are not requirements in all categories. The questions that generate requirements mentioned in **Table 3.1** are a good basis for finding eventual needs. But it is not mandatory to have an answer to all of those questions.

3.6 Preliminary conceptual model

Although **Chapters 6** and 7 present conceptual modeling techniques in detail, it is necessary to mention here that there are interdependency relations between system use cases and the so-called *preliminary conceptual model* (**Larman, 2004**). The preliminary conceptual model is built during

Inception, and consists of a class diagram that represents the main information units of the system. It is still not necessary to represent attributes. Although associations must appear in that model, it is not necessary to detail their features.

By analyzing the system use case diagram, many important concepts may be discovered. These concepts are represented as classes in the preliminary conceptual model; they represent the structure of the information that will be managed by the system. At the same time, an analyst, by observing the conceptual model, could notice if the use case diagram is sufficiently complete. This verification usually occurs when new entities are identified in a business process, and it is necessary that they be somehow recorded by the system.

Figure 3.9 presents a possible preliminary conceptual model for the use case diagram presented in **Figure 3.6**. The process for discovering classes consists of thinking about the use cases and imagining which (high-level) information is exchanged between the actors and the system to allow the process to work. The associations between classes represent dependencies or relationships between the pieces of information represented by them.

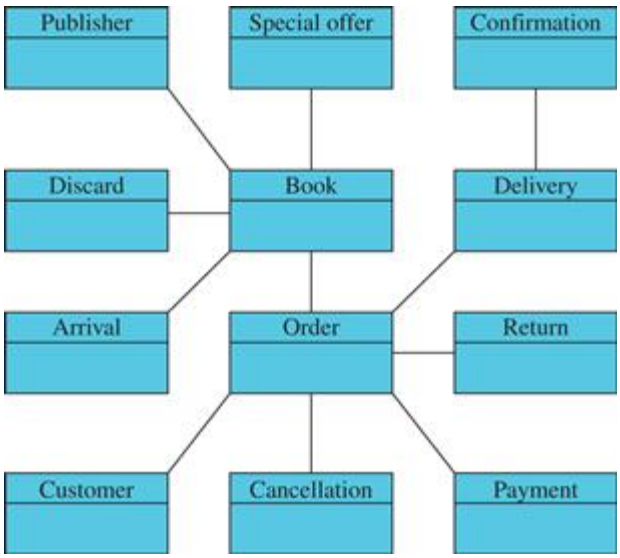

FIGURE 3.9 Preliminary conceptual model for the use cases of **Figure 3.6**.

Looking at **Figure 3.6** with the purpose of finding the concepts shown in **Figure 3.9**, it can be seen that:

- *Order books* is probably the most important use case in the system. It gives rise to two concepts, *Book* and *Order*, which are associated.

- *Pay for books* adds a new concept, *Payment*, which is associated with *Order*. At that point the team could decide to change the name of the use case to *Pay order*.

- The *Receive books* use case gives rise to the concept *Arrival*, which is associated to *Book*. Maybe at that point *EntryOrder* or *BuyOrder* could be introduced as a new concept.

- *Deliver books* produces a new concept, *Delivery*, which is associated to *Order*. Now *Deliver books* may/should be renamed to *Deliver order*.

- *Register delivery confirmation* produces the concept *Confirmation*, which is associated with *Delivery*.

- *Register book return* produces *Return*, which is associated to *Order*. Maybe at that point the name of the use case should be changed to *Register order return*.

- *Discard books* produces *Discard*, which is associated to *Book*.

- *Resend books* does not create a new concept at first glance because it may be considered the repetition of the use case *Deliver books*. However, that still must be analyzed further.

- *Cancel sale* produces the *Cancellation* concept, which is associated to an *Order*. Maybe the use case name should be changed to *Cancel order* at this point.

- *Create/remove special offer* creates the *SpecialOffer* concept, which is asso-

ciated to *Book*.

The identification of that preliminary conceptual model is especially useful to ease the visualization of the structure of the information that is going to be managed by the system; this helps in unifying the vocabulary among the team members and other stakeholders. The decisions about changing use case names in order to clean the vocabulary are already implemented in **Figure 3.10**.

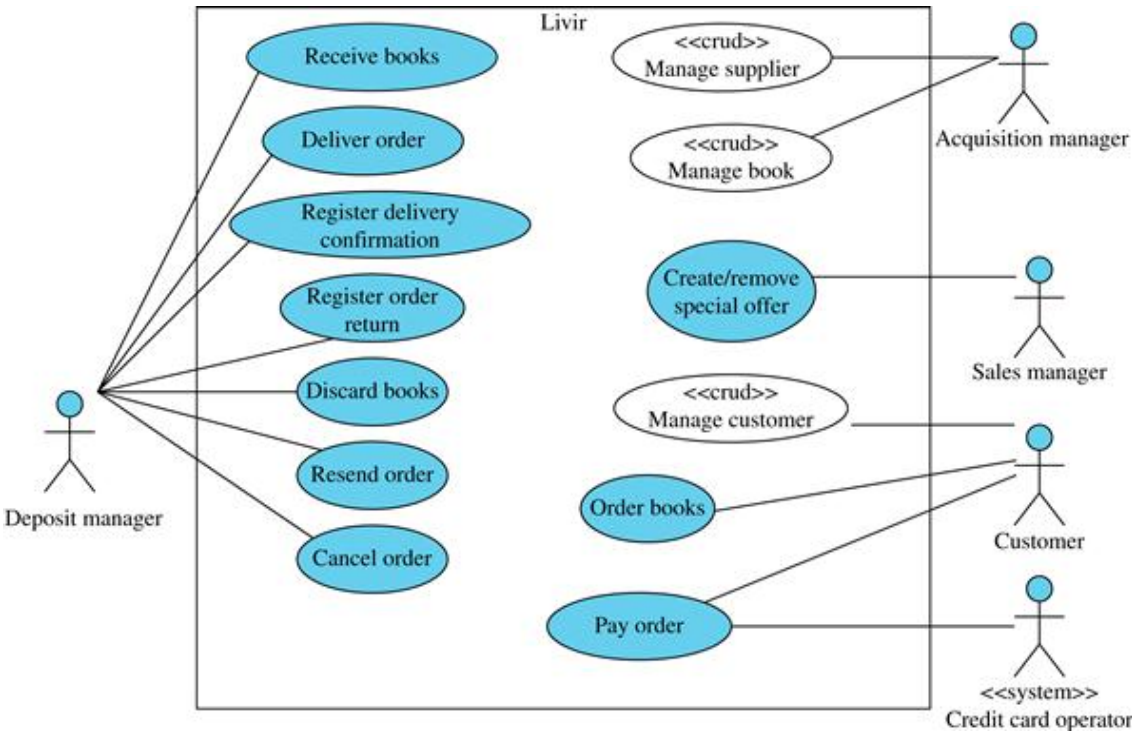

FIGURE 3.10 Use case model with names revised and CRUDs added.

But there is also the following important practical utility for the preliminary conceptual model. Among the concepts shown, most are information elements created or changed in the context of the use cases already identified. However, some of them are neither created nor changed by those use cases, and that means that some use cases could still be missing. This is the case especially for the classes *Book*, *Publisher*, and *Customer*. Those concepts may be considered as a CRUD, because they allow the four classical operations: *Create*, *Retrieve*, *Update*, and *Delete*. If they are to be added to the diagram, instead of representing them individually it would

be better to represent the four operations by using a single CRUD use case, which is stereotyped with ≪*crud*≫, as seen in **Figure 3.10**.

A new actor was defined for managing the publishers and books: the *Acquisition manager*. On the other hand, it was decided that the customer would be responsible for its own record.

Why is there no CRUD for orders, canceling, returning, payment, etc.? Because those are already managed by the use cases on the diagram, and it is not necessary to create other use cases specifically for them. For example, an order is created by the use case *Order books*; it is changed by use cases such as *Cancel order* and *Pay order*; it is visualized (retrieved) in many use cases; and, finally, it is assumed that it cannot be removed from the system.

Another question that could be raised with the stakeholders is: What are the reports that the system must produce? Although they are the simplest use cases, as explained later, reports are an important source of information for identifying which information is necessary to fill the stakeholders' needs.

There is a difference between reports and the retrieve query of CRUD: retrieving is simply recovering stored data about an object; a report, however, usually involves a number of objects, sometimes from different classes, and it necessarily includes some kind of filtering or combination of data (sum, product, average, greater value, lower value, etc.). For example, a query for information about a book, given its ISBN, should not be considered as a report use case, because that query is already included in the CRUD use case.

For the running example, some reports that are interesting to certain actors could be identified, such as:

- *Deposit manager*: Report on upcoming orders by period of time, report on deliveries by period of time, report on the total number of books avail-

able for sale, report on returns received by period of time, and report on books discarded by period of time.

- *Customer*: Report on orders status, and report on past sales.

- *Sales manager*: Report on book sales by period of time.

The client could demand other reports, and the above are only examples. Those that are used exclusively inside one of the existing use cases must be excluded from the reports list. For example, the deposit manager could be interested in the list of orders that are arriving. If that list is going to be consulted only at the moment an order really arrives and must be registered, then the query must be considered as part of the use case *Receive books*, and must not be included in the diagram. Only reports that are not necessarily part of other use cases must be included in the diagram; otherwise the list could quickly grow far beyond the manageable, and would be redundant and incomplete (because it is the use case expansion with sequence diagrams that will really indicate which steps, including queries, are necessary for each use case). **Figure 3.11** shows the use case diagram updated with the reports (stereotyped with ≪*report*≫) as indicated above.

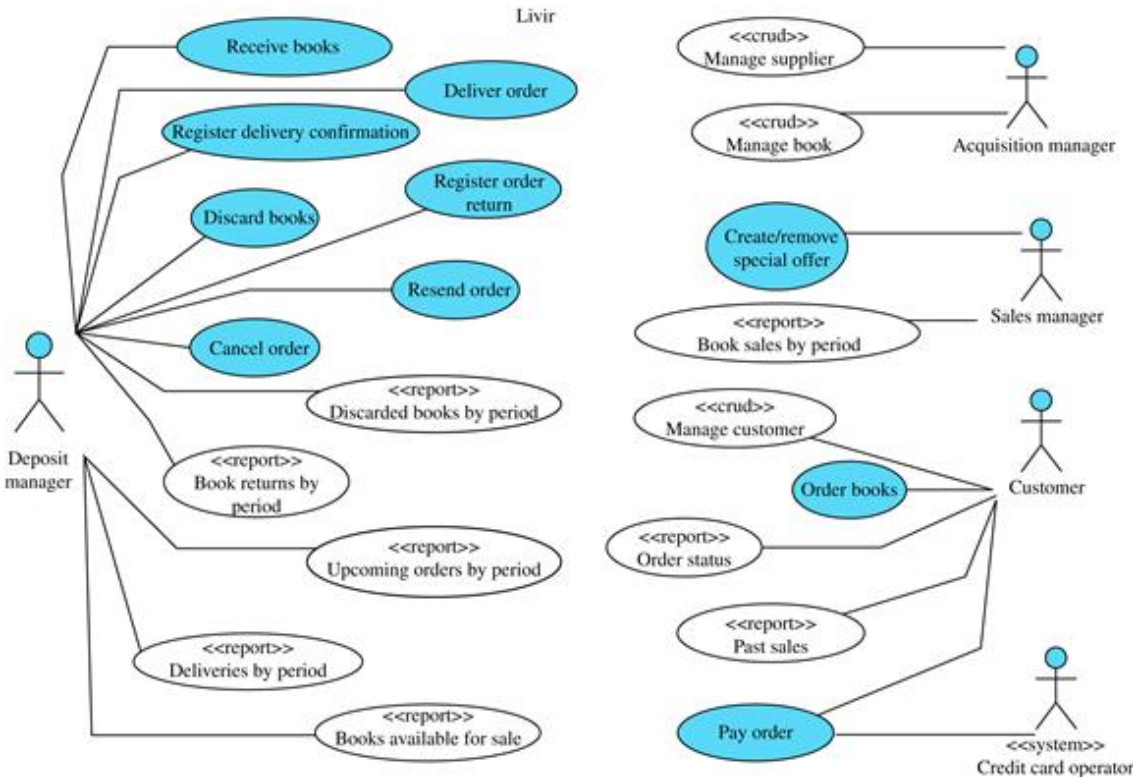

FIGURE 3.11 Use case diagram with reports.[5]

The quantity of ≪*report*≫ use cases will depend on the structure of the information they present. When parameterization is possible it must be used. It is not necessary, for example, to have a *Sales report by week* and *Sales report by month*. Unless they present different data structures, they are a single report. The reference to a week, month, or any other period of time is just a parameter.

However, it is not advisable to group reports with diverse natures, for example *Sales report by book* and *Deliveries report by period of time*: the inputs and outputs are different. Thus, those two reports must be considered two different use cases.

As seen in **Figure 3.11**, the number of use cases may become high and the diagram may quickly become hard to organize. This is one of the reasons to avoid including fragments in these diagrams. Each single use case will be detailed by other means. Also, the CRUD and report use cases may not be included in the main use case diagram if the system is at least of

mod-
er-
ate
size.
They
must
be
listed
some-
where
else
(a
spread-
sheet
or
a
sep-
a-
rated
use
case
di-
a-
gram,
for
in-
stance),
in
or-
der
not
to
pol-
lute

the main use case diagram. Another solution that may be used sometimes is to represent them with a different color (as done in **Figures 3.10**

and **3.11**), so that they do not hinder the visualization of the key use cases in the diagram.

3.7 The process so far

3.8 Questions

1. Explain the differences between a business use case and a system use case.

- What is the utility of a system use case through the software development process?
- Which business actors and business workers are converted

into system actors?

- What are functional, nonfunctional, and supplementary requirements and what features might they have?

- Why must a prelim-

i-
nary
con-
cep-
tual
model
be
made
dur-
ing
the
Inception
phase?

[1]From
now
on,
sys-
tem
use
cases
will
be
re-
ferred
to
sim-
ply
as
use
cases
for
sim-

plicity, except when they are compared to business use cases.

[2]That idea follows **Larman's (2004)** definition of an *EBP* (*Elementary Business Process*) *use case*. The

EBP definition comes from the business process engineering field: “a task performed by one person in one place at one time, in response to

a business event, which adds measurable business value and leaves the data in a consistent state."

[3] Another option is to use the MoSCoW scale: Must, Should,

Could,
and
Would.

[4]**http://www.iso.org/iso/catalogue_detail?csnumber=35733**

[5]Use cases stereotyped with

report

have their names abbreviated to avoid repeating "Generate report

for
..."
re-
peated
many
times.