Chapter 6
Analysing a System

In Chaps. 6–8, we examine the essential steps in object-oriented software development: analysis, design, and implementation. To illustrate the process, we study a relatively simple example—a piece of software to manage a small library—whose function is limited to that of lending books to its members, receiving them back, doing the associated operations such as querying, registering members, etc., and keeping track of these transactions. In the course of these chapters, we go through the entire process of analysing, designing and implementing this system.

The software construction process begins with an analysis that determines the requirements of the system, which is what we introduce in this chapter. At this stage the focus is on determining what the system must perform without regard to the methodology to be employed. This process is carried out by a team of analysts, perhaps familiar with the specific type of application. The requirements are spelled out in a document known variously as the 'Requirements Specification', 'System Requirements', etc. Using these, the system analyst creates a model of the system, enabling the identification of some of the components of the system and the relationships between them. The end product of this phase is a *conceptual model* for the system which describes the functionality of the system, identifies its conceptual entities and records the nature of the associations between these entities.

Once the analysis has been satisfactorily completed, we move on to the design phase, which is addressed in the first part of Chap. 7. The design starts with a detailed breakdown of how the system will emulate the behaviour outlined in the model. In the course of this breakdown, all the parts of the system and their responsibilities are clearly identified. This step is followed by determining the software and hardware structures needed to implement the functionality discovered in the analysis stage. In the object-oriented world, this would mean deciding on the language or languages to be used, the packages, the platform, etc. The second part of Chap. 7 looks at implementation, wherein we discuss the lower-level issues, language features employed, etc.

A question that a conscientious beginner often ponders is: *Did I do a good job of the design*? or *Is my design really object-oriented*? Indeed, in the real world, it is often the case that designs conform to object-oriented principles to varying degrees.

B. Dathan and S. Ramnath, *Object-Oriented Analysis, Design and Implementation*,
Undergraduate Topics in Computer Science, DOI 10.1007/978-3-319-24280-4_6

Fortunately, in addition to the broad guidelines for what constitutes a good object-oriented design, there are some more specific rules that can be applied to look for common mistakes and correct them. These rules, known as *refactoring rules*, are more commonly presented as a means for improving the design of the existing code. They are, however, just as useful to check the design of a system before it is fully implemented. In Chap. 8, we introduce the concept of refactoring and apply these rules to our small system.

As our main focus in this book is the elaboration of the process of analysis, design, and implementation, we will bypass many software engineering and project management issues. We will not dwell on conceptual frameworks such as agile software development for managing the software development life cycle. We use UML notations in an appropriate manner that is sufficient to describe our design, but do not cover these exhaustively. For a detailed exposition on these topics, the reader is referred to the works cited at the end of each chapter.

6.1 Overview of the Analysis Phase

To put it in a simple sentence, the major goal of this phase is to address this basic question: what should the system do? A typical computer science student writes a number of programs by the time he/she graduates. Typically, the program requirements are written up by the instructor: the student does some design, writes the code, and submits the program for grading. To some extent, the process of understanding the requirements, doing the design, and implementing that design is relatively informal. Requirements are often simple and any clarifications can be had via questions in the classroom, e-mail messages, etc.

The above simple-minded approach does not quite suffice for 'real-life' projects for a number of reasons. For one reason, such systems are typically much bigger in scope and size. They also have complex and ambiguously-expressed requirements. Third, there is usually a large amount of money involved, which makes matters quite serious. For a fourth reason, hard as it may be for a student to appreciate it, project deadlines for these 'real-life' projects are more critical. (Users are fussier than instructors!)

However, as in the case of the classroom assignment, there are still two parties: the user community, which needs some system to be built and the development people, who are assigned to do the work. The process could be split into three activities:

1. Gather the requirements: this involves interviews of the user community, reading of any available documentation, etc.
2. Precisely document the functionality required of the system.
3. Develop a conceptual model of the system, listing the conceptual classes and their relationships.

It is not always the case that these activities occur in the order listed. In fact, as the analysts gather the requirements, they will analyse and document what they have collected. This may point to holes in the information, which may necessitate further requirements collection.

6.2 Stage 1: Gathering the Requirements

The purpose of *requirements analysis* is to define what the new system should do. The importance of doing this correctly cannot be overemphasized. Since the system will be built based on the information garnered in this step, any errors made in this stage will result in the implementation of a wrong system. Once the system is implemented, it is expensive to modify it to overcome the mistakes introduced in the analysis stage.

Imagine the scenario when you are asked to construct software for an application. The client may not always be clear in his/her mind as to what should be constructed. One reason for this is that it is difficult to imagine the workings of a system that is not yet built. Only when we actually use a specific application such as a word processor do we start realising the power and limitations of that system. Before actually dealing with it, one may have some general notions of what one would like to see, but may find it difficult to provide many details.

Incompleteness and errors in specifications can also occur because the client does not have the technical skills to fully realise what technology can and cannot deliver. Once again, the general concepts can be stated, but specifics are harder. A third reason for omissions is that it is all too common to have a client who knows the system very well and consequently either assumes a lot of knowledge on the part of the analyst or simply skips over the 'obvious details'.

Requirements for a new system are determined by a team of analysts by interacting with teams from the company paying for the development (clients) and the user community, who ultimately uses the system on a day-to-day basis. This interaction can be in the form of interviews, surveys, observations, study of existing manuals, etc.

Broadly speaking, the requirements can be classified into two categories:

- *Functional requirements* These describe the interaction between the system and its users, and between the system and any other systems, which may interact with the system by supplying or receiving data.
- *Non-functional requirements* Any requirement that does not fall in the above category is a non-functional requirement. Such requirements include response time, usability and accuracy. Sometimes, there may be considerations that place restrictions on system development; these may include the use of specific hardware and software and budget and time constraints.

It should be mentioned that initiating the development cycle for a software system is usually preceded by a phase that includes the initial conception and planning. A developer would be approached by a client who wishes to have a certain product

developed for his/her business. There would be a *domain* associated with the business, which would have its own jargon. Before approaching the developer, one would assume that the client has determined that a need for a product exists. Once all these issues are sorted out, the developer(s) would meet with the client and, perhaps several would-be end-users, to determine what is expected of the system. Such a process would result in a list of requirements of the system.

As mentioned at the beginning of this chapter, we study the development process by analysing, designing, and implementing a simple library system; this is introduced next.

6.2.1 Case Study Introduction

Let us proceed under the assumption that developers of our library system have available to them a document that describes how the business is conducted. This functionality is described as a list of what are commonly called *business processes.*

The business processes of the library system are listed below.

- **Register new members** The library receives applications from people who want to become library members, whom we alternatively refer to as **users**. While applying for membership, a person supplies his/her name, phone number and address to the library. The library assigns each member a unique identifier (ID), which is needed for transactions such as issuing books.
- **Add books to the collection** We will make the assumption that the collection includes just books. For each book the library stores the title, the author's name, and a unique ID. (For simplicity, let us assume that there is only one author per book. If there are multiple authors, let us say that the names will have to be concatenated to get a pretty huge name such as 'Brahma Dathan and Sarnath Ramnath'. As a result, to the system, it appears that there is just one author.) When it is added to the collection, a book is given a unique identifier by the clerk. This ID is based on some standard system of classification.
- **Issue a book to a member (or user)** To check out books, a user (or member) must identify himself to a clerk and hand over the books. The library remembers that the books have been checked out to the member. Any number of books may be checked out in a single transaction.
- **Record the return of a book** To return a book, the member gives the book to a clerk, who submits the information to the system, which marks the book as 'not checked out'. If there is a hold on the book, the system should remind the clerk to set the book aside so that the hold can be processed.
- **Remove books from the collection** From time to time, the library may remove books from its collection. This could be because the books are worn-out, are no longer of interest to the users, or other sundry reasons.
- **Print out a user's transactions** Print out the interactions (book checkouts, returns, etc.) between a specific user and the library on a certain date.

- **Place/remove a hold on a book** When a user wants to put a hold, he/she supplies the clerk with the book's ID, the user's ID, and the number of days after which the book is not needed. The clerk then adds the user to a list of users who wish to borrow the book. If the book is not checked out, a hold cannot be placed. To remove a hold, the user provides the book's ID and the user's ID.
- **Renew books issued to a member** Customers may walk in and request that several of the books they have checked out be renewed (re-issued). The system must display the relevant books, allow the user to make a selection, and inform the user of the result.
- **Notify member of book's availability** Customers who had placed a hold on a book are notified when the book is returned. This process is done once at the end of each day. The clerk enters the ID for each book that was set aside, and the system returns the name and phone number of the user who is next in line to get the book.

In addition, the system must support three other requirements that are not directly related to the workings of a library, but, nonetheless, are essential.

- A command to save the data on a long-term basis.
- A command to load data from a long-term storage device.
- A command to quit the application. At this time, the system must ask the user if data is to be saved before termination.

To keep the process simple, we restrict our attention for the time being to the above operations. A real library would have to perform additional operations like generating reports of various kinds, impose fines for late returns, etc. Many libraries also allow users to check out books themselves without approaching a clerk. Whatever the case may be, the analysts need to learn the existing system and the requirements. As mentioned earlier, they achieve this through interviews, surveys, and study.

Our goal here is to present the reader with the big picture of the entire process so that the beginner is not overwhelmed by the complexity or bogged down in minutiae. Keeping this in mind, we will be designing a system that the reader may find somewhat simplistic, particularly if one compares this with the kinds of features that a 'real' system in today's market can provide. While there is some truth to this observation, it should be noted that the simplification of the system has been done with a view to reducing unnecessary detail so that we can focus instead on the development process, elaborate on the use of tools described previously, and explain through an example how good design principles are applied. In the course of applying the above, we come with a somewhat simplified *sample development process* that may be used as a template by someone who is getting started on this subject.

Assuming that we have a good grasp of the requirements, we need to document the functional requirements of the application and determine the system's major entities and their relationships. As mentioned earlier, the steps may be, and are often, carried out as an iterative, overlapping process; for pedagogical reasons, we discuss them as a sequence of distinct activities.

6.3 Functional Requirements Specification

It is important that the requirements be precisely documented. The requirements specification document serves as a contract between the users and the developers. When it is time to deliver the system, there should be no confusion as to what the expectations are. Equally or perhaps even more important, it also tells the designers the expected functionality of the system. Moreover, as we attempt to create a precise documentation of the requirements, we will discover errors and omissions.

An accepted way of accomplishing this task is the *use case analysis*, which we study now.

6.3.1 Use Case Analysis

Use case analysis is a case-based way of describing the uses of the system with the goal of defining and documenting the system requirements. It is essentially a narrative describing the sequence of events (actions) of an external agent (actor) using the system to complete a process. It is a powerful technique that describes the kind of functionality that a user expects from the system. Use cases have two or more parties: *agents* who interact with the system and the *system* itself. In our simple library system, the members do not use the system directly. Instead, they get services via the library staff.

To initiate this process, we need to get a feel for how the system will interact with the end-user. We assume that some kind of a user-interface is required, so that when the system is started, it provides a menu with the following choices:

1. Add a member
2. Add books
3. Issue books
4. Return books
5. Remove books
6. Place a hold on a book
7. Remove a hold on a book
8. Process Holds: Find the first member who has a hold on a book
9. Renew books
10. Print out a member's transactions
11. Store data on disk
12. Retrieve data from disk
13. Exit

The above menu gives us the list of ways in which the system is going to be used. There are some implicit requirements associated with these operations. For instance,

when a book is checked out, the system must output a due-date so that the clerk can stamp the book. This and other such details will be spelled out when we elaborate on the use cases.

The actors in our system are members of the library staff who manage the daily operations. This idea is depicted in the use case diagram in Fig. 6.1, which gives an overview of the system's usage requirements. Notice that even in the case of issuing books, the functionality is invoked by a library staff member, who performs the actions on behalf of a member.

We are about to take up the task of specifying the individual use cases. In order to keep the discussion within manageable size and not lose focus, we make the following assumption: While the use cases will state the need for the system to display different messages prompting the user for data and informing the results of operations, the user community is not fussy about the minute details of what the messages should be; any meaningful message is acceptable. For example, we may specify in a use case that the system 'informs the clerk if the member was added'. The actual message could be any one of a number of possibilities such as 'Member added', or 'Member registered', etc.

Use case for registering a user Our first use case is for registering a new user and is given in Table 6.1. Recall from our discussion in Chap. 2 that use cases are specified

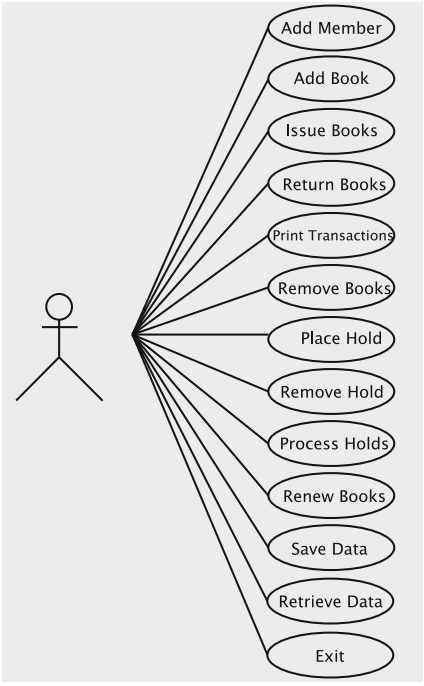

Fig. 6.1 Use case diagram for the library system

Table 6.1 Use case Register New Member

Actions performed by the actor	Responses from the system
1. The customer fills out an application form containing the customer's name, address, and phone number and gives this to the clerk	
2. The clerk issues a request to add a new member	
	3. The system asks for data about the new member
4. The clerk enters the data into the system	
	5. Reads in data, and if the member can be added, generates an identification number (which is not necessarily a number in the literal sense just as social security numbers and phone numbers are not actually numbers) for the member and remembers information about the member. Informs the clerk if the member was added and outputs the member's name, address, phone and id
6. The clerk gives the user his identification number	

in a two-column format, where the left-column states the actions of the actor and the right-column shows what the system does.

The above example illustrates several aspects of use cases.

1. Every use case has to be identified by a name. We have given the name Register New Member to this use case.
2. It should represent a reasonably-sized activity in the organisation. It is important to note that not all actions and operations should be identified as use cases. As an extreme example, stamping a due-date on the book should not be a use case. A use case is a relatively large end-to-end process description that captures some business process that a client purchasing the software needs to perform. In some instances, a business process may be decomposed into more than one use case, particularly when there is some intervening real-world event(s) for which the agent has to wait for an unspecified length of time. An example of such a situation is presented later in this chapter.
3. The first step of the use case specifies a 'real-world' action that triggers the exchange described in the use case. This is provided mainly for the sake of completeness and does not have much bearing on the actual design of the system. It does, however, serve a useful purpose: by looking at the first steps of all the use cases, we can verify that all external events that the system needs to respond to have been taken care of.
4. The use case does not specify how the functionality is to be implemented. For example, the details of how the clerk enters the required information into the

system are left unspecified. Although we assume that the user interacts with the system through the menu, which was briefly described earlier, we do not specify the details of this mechanism. The use case also does not state how the system accomplishes the task of registering a user: what software components form the system, how they may interact, etc.

5. The use case is not expected to cover all possible situations. While we would expect that the sequence of events that are specified in the above use case is what would actually happen in a library when a person wants to be registered, the use case does not specify what the system should do if there are errors. In other words, the use case explains only the most commonly-occurring scenario, which is referred to as the *main flow*. Deviations from the main flow due to occurrences of errors and exceptions are not detailed in the above use case.

Use case for adding books Next, we look at the use case for adding new books in Table 6.2. Notice that we add more than one book in this use case, which involves a repetitive process captured by a *go-to* statement in the last step. Notice that details of how the identifier is generated are not specified. From the point of view of the system analyst, this is something that the actor is expected to take care of independently.

Use case for issuing books Consider the use case where a member comes to the check-out counter to issue a book. The user identifies himself/herself to a clerk, who checks out the books for the user. It proceeds as in Table 6.3.

There are some drawbacks to the way this use case is written. One is that it does not specify how due-dates are computed. We may have a simple rule (example: due-dates are one month from the date of issue) or something quite complicated

Table 6.2 Use case Adding New Books

Actions performed by the actor	Responses from the system
1. Library receives a shipment of books from the publisher	
2. The clerk issues a request to add a new book	
	3. The system asks for the identifier, title, and author name of the book
4. The clerk generates the unique identifier, enters the identifier, title, and author name of a book	
	5. The system attempts to enter the information in the catalog and echoes to the clerk the title, author name, and id of the book. It then asks if the clerk wants to enter information about another book
6. The clerk answers in the affirmative or in the negative	
	7. If the answer is in the affirmative, the system goes to Step 3. Otherwise, it exits

Table 6.3 Use case Book Checkout

Actions performed by the actor	Responses from the system
1. The member arrives at the check-out counter with a set of books and supplies the clerk with his/her identification number	
2. The clerk issues a request to check out books	
	3. The system asks for the user ID
4. The clerk inputs the user ID to the system	
	5. The system asks for the ID of the book
6. The clerk inputs the ID of a book that the user wants to check out	
	7. The system records the book as having been issued to the member; it also records the member as having possession of the book. It generates a due-date. The system displays the book title and due-date and asks if there are any more books
8. The clerk stamps the due-date on the book and replies in the affirmative or negative	
	9. If there are more books, the system moves to Step 5; otherwise it exits
10. The customer collects the books and leaves the counter	

(example: due-date is dependent on the member's history, how many books have been checked out, etc.). Putting all these details in the use case would make the use case quite messy and harder to understand. Rules such as these are better expressed as **Business Rules**. A business rule may be applicable to one or more use cases.

The business rule for due-date generation is simple in our case. It is Rule 1 given in Table 6.4 along with all other rules for the system.

Table 6.4 Rules for the library system

Rule number	Rule
Rule 1	Due-date for a book is one month from the date of issue
Rule 2	All books are issuable
Rule 3	A book is removable if it is not checked out and if it has no holds
Rule 4	A book is renewable if it has no holds on it
Rule 5	When a book with a hold is returned, the appropriate member will be notified
Rule 6	Holds can be placed only on books that are currently checked out

A second problem with the use case is that as written above, it does not state what to do in case things go wrong. For instance,

1. The person may not be a member at all. How should the use case handle this situation? We could abandon the whole show or ask the person to register.
2. The clerk may have entered an invalid book id.

To take care of these additional situations, we modify the use case as given in Table 6.5. We have resolved these issues in Step 7 by having the system check whether the book is issuable, which can be expressed as a business rule. This could check one (or more) of several conditions: *Is the member in good standing with the library? Is there some reason the book should not be checked out? Has the member checked out more books than permitted (if such limits were to be imposed)?* The message displayed by the system in Step 7 informs the clerk about the result of the transaction. In a real-life situation, the client will probably want specific details of

Table 6.5 Use case Book Checkout revised

Actions performed by the actor	Responses from the system
1. The member arrives at the check-out counter with a set of books and supplies the clerk with his/her identification number	
2. Clerk issues a request to check out books	
	3. The system asks for the user ID
4. Clerk inputs the user ID to the system	
	5. If the ID is valid, the system asks for the ID of the book; otherwise it prints an appropriate message and exits the use case
6. The clerk inputs the identifier of a book that the user wants to check out	
	7. If the ID is valid and the book is issuable to the member, the system records the book as having been issued to the member; It records the member as having possession of the book and generates a due-date as in Rule 1. It then displays the book's title and due-date. If the book is not issuable as per Rule 2, the system displays a suitable error message. The system asks if there are more books
8. The clerk stamps the due-date, prints out the transaction (if needed) and replies positively or negatively	
	9. If there are more books for checking out, the system goes back to Step 5; otherwise it exits
10. The clerk stamps the due date and gives the user the books checked out. The customer leaves the counter	

what went wrong; if they are important to the client, these details should be expressed in the use case. Since our goal is to cover the basics of requirements analysis, we sidestep the issue.

Let us proceed to write more use cases. For the most part, these are quite elementary, and the reader may well choose to skip the details or try them out as an exercise.

Use case for returning books Users return books by leaving them on a library clerk's desk; the clerk enters the book ids one by one to return them. Table 6.6 gives the details of the use case. Here, as in the use case for issuing books, the clerk may enter incorrect information into the system, which the use case handles. Notice that if there is a hold on the book, that information is printed for use by the clerk at a later time.

Use cases for removing (deleting) books, printing member transactions, placing a hold, and removing a hold The next four use cases deal with the scenarios for removing books (Table 6.7), printing out member transactions (Table 6.8), placing a hold (Table 6.9), and removing a hold (Table 6.10). In the second of these, the system does not actually print out the transactions, but only displays them on the interface. We are assuming that the necessary facilities to print will be a part of the underlying platform.

In Step 5 in Table 6.7, we allow for the possibility that the deletion may fail. In this event, we assume that there will be some meaningful error message so that the clerk can take corrective action. We shall revisit this issue when we discuss the design and implementation in the next chapter.

Table 6.6 Use case Return Book

Actions performed by the actor	Responses from the system
1. The member arrives at the return counter with a set of books and leaves them on the clerk's desk	
2. The clerk issues a request to return books	
	3. The system asks for the identifier of the book
4. The clerk enters the book identifier	
	5. If the identifier is valid, the system marks that the book has been returned and informs the clerk if there is a hold placed on the book; otherwise it notifies the clerk that the identifier is not valid. It then asks if the clerk wants to process the return of another book
6. The clerk answers in the affirmative or in the negative and sets the book aside in case there is a hold on the book (see Rule 5)	
	7. If the answer is in the affirmative, the system goes to Step 3. Otherwise, it exits

Table 6.7 Use case Removing Books

Actions performed by the actor	Responses from the system
1. Librarian identifies the books to be deleted	
2. The clerk issues a request to delete books	
	3. The system asks for the identifier of the book
4. The clerk enters the ID for the book	
	5. The system checks if the book can be removed using Rule 3. If the book can be removed, the system marks the book as no longer in the library's catalog. The system informs the clerk about the success of the deletion operation. It then asks if the clerk wants to delete another book
6. The clerk answers in the affirmative or in the negative	
	7. If the answer is in the affirmative, the system goes to Step 3. Otherwise, it exits

Table 6.8 Use case Member Transactions

Actions performed by the actor	Responses from the system
1. The clerk issues a request to get member transactions	
	2. The system asks for the user ID of the member and the date for which the transactions are needed
3. The clerk enters the identity of the user and the date	
	4. If the ID is valid, the system outputs information about all transactions completed by the user on the given date. For each transaction, it shows the type of transaction (book borrowed, book returned or hold placed) and the title of the book
5. Clerk prints out the transactions and hands them to the user	

There may be some variations in the way these scenarios are played out. When placing or removing a hold, the library staff may actually want to see a message that the operation was successfully completed. These requirements would modify the manner in which the system responds in these use cases. While such information should be gleaned from the client as part of the requirements analysis, it is often necessary to go back to the client after the use cases are written, to ensure that the system interacts in the desired manner with the operator.

Table 6.9 Use case Place a Hold

Actions performed by the actor	Responses from the system
1. The clerk issues a request to place a hold	
	2. The system asks for the book's ID, the ID of the member, and the duration of the hold
3. The clerk enters the identity of the user, the identity of the book and the duration	
	4. The system checks that the user and book identifiers are valid and that Rule 6 is satisfied. If yes, it records that the user has a hold on the book and displays that; otherwise, it outputs an appropriate error message

Table 6.10 Use case Remove a Hold

Actions performed by the actor	Responses from the system
1. The clerk issues a request to remove a hold	
	2. The system asks for the book's ID and the ID of the member
3. The clerk enters the identity of the user and the identity of the book	
	4. The system removes the hold that the user has on the book (if any such hold exists), prints a confirmation and exits

Table 6.11 Use case Process Holds

Actions performed by the actor	Responses from the system
1. The clerk issues a request to process holds (so that Rule 5 can be satisfied)	
	2. The system asks for the book's ID
3. The clerk enters the ID of the book	
	4. The system returns the name and phone number of the first member with an unexpired hold on the book. If all holds have expired, the system responds that there is no hold. The system then asks if there are any more books to be processed
5. If there is no hold, the book is then shelved back to its designated location in the library. Otherwise, the clerk prints out the information, places it in the book and replies in the affirmative or negative	
	6. If the answer is yes, the system goes to Step 2; otherwise it exits

Use case for processing holds Given in Table 6.11, this use case deals with processing the holds at the end of each day. In this case, once the contact information for the member has been printed out, we assume that the library will contact the member. The member may not come to collect the book within the specified time, at which point the library will try to contact the next member in line. All this is not included in the use case. If we were to do so, the system would, in essence, be waiting on the user's response for a long period of time. We therefore leave out these steps and when the next user has to be contacted, we simply process holds on the book once again.

How do Business Rules Relate to Use Cases?

Business rules can be broadly defined as the details through which a business implements its strategy. Business analysts perform the task of gathering business rules, and these belong to one of four categories:

- **Definitional rules** which explain what is meant when a certain word is used in the context of the business operations. These may include special technical terms, or common words that have a particular significance for the business. For instance the term *Book* in the context of the library refers to a book owned by the library.
- **Factual rules** which explain basic things about the business's operations; they tell how the terms connect to each other. A library, for instance, would have rules such as 'Books are issued to Members,' and 'Members can place holds on Books'.
- **Constraints** which are specific conditions that govern the manner in which terms can be connected to each other. For instance, we have a constraint that says 'Holds can be placed only on Books that are currently checked out'.
- **Derivations** which are knowledge that can be derived from the facts and constraints. For instance, a bank may have the constraint, "The balance in an account cannot be less than zero," from which we can derive that if an amount requested for withdrawal is more than the balance, then the operation is not successful.

When writing use cases, we are mainly concerned with constraints and derivations. Typically, such business rules are in-lined with the logic of the use-case. The use-case may explicitly state the test that is being performed and cite the appropriate rule, or may simply mention that the system will respond in accordance with a specific rule.

In addition to the kinds of rules we have presented for this case study, there are always implicit rules that permeate the entire system. A common example of this is validation of input data; a zip code, for instance, can be validated against a database of zip-codes. Note that this rule does not deal with how entities are connected to one another, but specifies the required properties of a data element. Such constraints do not belong in use cases, but could be placed in classes that store the corresponding data elements.

Use case for renewing books This use case (see Table 6.12) deals with situations where a user has several books checked out and would like to renew some of these. The user may not remember the details of all of them and would perhaps like the system to prompt him/her. We shall assume that users only know the titles of the books to be renewed (they do not bring the books or even the book ids to the library) and that most users would have borrowed only a small number of books. In this situation, it is entirely appropriate for the system to display the title of each book borrowed by the user and ask if that book should be renewed.

Table 6.12 Use case Renew Books

Actions performed by the actor	Responses from the system
1. Member makes a request to renew several of the books that he/she has currently checked out	
2. Clerk issues a request to renew books	
	3. System asks for the member's ID
4. The clerk enters the ID into the system	
	5. System checks the member's record to find out which books the member has checked out. If there are none, the system prints an appropriate message and exits; otherwise it moves to Step 6
	6. The system displays the title of the next book checked out to the member and asks whether the book should be renewed
7. The clerk replies yes or no	
	8. The system attempts to renew the book using Rule 4 and reports the result. If the system has displayed all checked-out books, it reports that and exits; otherwise the system goes to Step 6

It may be the case that a library has additional rules for renewability: if a book has a hold or a member has renewed a book twice, it might not be renewable. In the above interaction, the system displays all the books and determines the renewability only if the member wishes to renew the book. A different situation could arise if we require that the system display only the renewable books. (The system would have to have a way for checking renewability without actually renewing the book, which places additional requirements on the system's functionality.) For our simple library, we go with the scenario described in Table 6.5.

6.4 Defining Conceptual Classes and Relationships

As we discussed earlier, the last major step in the analysis phase involves the determination of the conceptual classes and the establishment of their relationships. For example, in the library system, some of the major conceptual classes include members and books. Members borrow books, which establishes a relationship between them.

We could justify the usefulness of this step in at several ways:

1. **Design facilitation** Via use case analysis, we determined the functionality required of the system. Obviously, the design stage must determine how to implement the functionality. For this, the designers should be in a position to determine the classes that need to be defined, the objects to be created, and how the objects interact. This is better facilitated if the analysis phase classifies the entities in the application and determines their relationships.
2. **Added knowledge** The use cases do not completely specify the system. Some of these missing details can be filled in by the class diagram.
3. **Error reduction** In carrying out this step, the analysts are forced to look at the system more carefully. The result can be shown to the client who can verify its correctness.
4. **Useful documentation** The classes and relationships provide a quick introduction to the system for someone who wants to learn it. Such people include personnel who join the project to carry out the design or implementation or subsequent maintenance of the system.

In practice, an analyst will probably use multiple methods to come up with the conceptual classes and their relationships. In this case study, however, we use a simple approach: we examine the use cases and pick out all the nouns in the description of the requirements. For example, from the text of the use case for registering new users, we can pick out the nouns.

Guidelines to Remember When Writing Use Cases

- A use case must provide something of value to an actor or to the business: when the scenario described in the use case has played out, the actor has accomplished some task. The system may have other functions that do not provide value; these will be just steps within a use case. This also implies that each use case has at least one actor.
- Use cases should be *functionally cohesive,* i.e., they encapsulate a single service that the system provides.
- Use cases should be *temporally cohesive.* This notion applies to the time frame over which the use case occurs. For instance, when a book with a hold is returned, the member who has the hold needs to be notified. The notification is done after some delay; due to this delay, we do not combine the two operations into one use case. Another example could be a university registration system—when a student registers for a class, he or she should be billed. Since the billing operation is not temporally cohesive with the registration, the two constitute separate use cases.
- If a system has multiple actors, each actor must be involved in at least one, and typically several use cases. If our library allowed members to check out books by themselves, "member" is another possible actor.
- The model that we construct is a *set* of use cases, i.e., there is no relationship between individual use cases.
- Exceptional exit conditions are not handled in use cases. For instance, if a system should crash in the middle of a use case, we do not describe what the system is supposed to do. It is assumed that some reasonable outcome will occur.
- Use cases are written from the point of view of the actor in the active voice.
- A use case describes a scenario, i.e., tells us what the visible outcome is and does not give details of any other requirements that are being imposed on the system.
- Use cases change over the course of system analysis. We are trying to construct a model and consequently the model is in a state of evolution during this process. Use cases may be merged, added or deleted from the model at any time.

Here is the text of that use case, once again, with all nouns bold-faced:

(1) The **customer** fills out an **application form** containing the **customer's name**, **address**, and **phone number** and gives this to the **clerk**. (2) The **clerk** issues a **request** to add a new **member**. (3) **The system** asks for **data** about the new **member**. (4) The **clerk** enters the **data** into the **system**. (5) Reads in **data**, and if the **member** can be added, generates an **identification number** for the **member** and remembers

information about the **member**. Informs the clerk if the member was added and outputs the **member's name**, **address**, **phone**, and **id**. (6) The **clerk** gives the **user** his **identification number**.

Let us examine the nouns. First, let us eliminate duplicates to get the following list: **customer**, **application form**, **customer's name**, **address**, **phone number**, **clerk**, **request**, **system**, **data**, **identification number**, **member**, **user**, **member information**, and **member's name**. Some of the nouns such as **member** are composite entities that qualify to be classes.

While using this approach, we must remember that natural languages are imprecise and that synonyms may be found. We can eliminate the others as follows:

1. **customer**: becomes a member, so it is effectively a synonym for member.
2. **user**: the library refers to members alternatively as users, so this is also a synonym.
3. **application form** and **request**: application form is an external construct for gathering information, and request is just a menu item, so neither actually becomes part of the data structures.
4. **customer's name, address**, and **phone number**: They are attributes of a customer, so the `Member` class will have them as fields.
5. **clerk**: is just an agent for facilitating the functioning of the library, so it has no software representation.
6. **identification number**: will become part of a member.
7. **data**: gets stored as a member.
8. **information**: same as data related to a member.
9. **system**: refers to the collection of all classes and software.

The noun **system** implies a conceptual class that represents all of the software; we call this class `Library`. Although we do not have as yet any specifics of this class, we note its existence and represent it in UML without any attributes and methods (Fig. 6.2). (Recall from Chap. 2 that a class is represented by a rectangle.)

A member is described by the attributes name, address, and phone number. Moreover, the system generates an identifier for each user, so that also serves as an attribute. The UML convention is to write the class name at the top with a line below it and the attributes listed just below that line. The UML diagram is shown in Fig. 6.3.

Fig. 6.2 UML diagram for the class `Library`

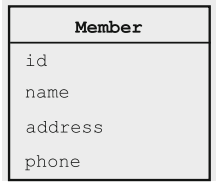

Fig. 6.3 UML diagram for the class `Member`

Recall the notion of association between classes, which we know from Chaps. 2 and 3 as a relationship between two or more classes. We note several examples of association in our case study. The use case Register New Member (Table 6.1) says that the system 'remembers information about the member'. This implies an association between the conceptual classes `Library` and `Member`. This idea is shown in Fig. 6.4; note the line between the two classes and the labels 1, *, and 'maintains a collection of' just above it. They mean that one instance of the `Library` maintains a collection of zero or more members.

Obviously, members and books are the most central entities in our system: the sole reason for the library's existence is to provide service to its members and that is effected by letting them borrow books. Just as we reasoned for the existence of a conceptual class named `Member`, we can argue for the need of a conceptual class called `Book` to represent a book. It has attributes `id`, `title`, and `author`. A UML description of the class is shown in Fig. 6.5. It should come as no surprise that an association between the classes `Library` and `Book`, shown in Fig. 6.6, is also needed. We show that a library has zero or more books. (Normally, you would expect a library to have at least one book and at least one member; But our design takes no chances!)

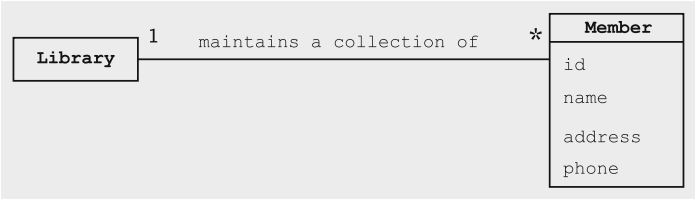

Fig. 6.4 UML diagram showing the association of Library and Member

Fig. 6.5 UML diagram for the class `Book`

Fig. 6.6 UML diagram showing the association of `Library` and `Book`

Some associations are *static*, i.e., permanent, whereas others are *dynamic*. Dynamic associations are those that change as a result of the transactions being recorded by the system. Such associations are typically associated with verbs.

As an example of a dynamic association, consider members borrowing books. This is an association between `Member` and `Book`, shown in Fig. 6.7. At any instant in time, a book can be borrowed by one member and a member may have borrowed any number of books. We say that the relationship `Borrows` is a one-to-many relationship between the conceptual classes `Member` and `Book` and indicate it by writing 1 by the side of the box that represents a user and the * near the box that stands for a book.

This diagram actually tells us more than what the Issue Book use case does. That use case does not say some of the considerations that come into play when a user borrows a book: for example, how many books a user may borrow. We might have forgotten to ask that question when we learned about the use case. But now that we are looking at the association and are forced to put labels at the two ends, we may end up capturing missing information. In the diagram of Fig. 6.7, we state that there is no limit. It also states that two users may not borrow the same book at the same time. Recollect from Chap. 3 that an association does not imply that the objects of the classes are always linked together; we may therefore have a situation where no book in the library has been checked out.

Another action that a member can undertake is to place a hold on a book. Several users can have holds placed on a book, and a user may place holds on an arbitrary number of books. In other words, this relationship is many-to-many between users and books. We represent this in Fig. 6.8 by putting a * at both ends of the line representing the association.

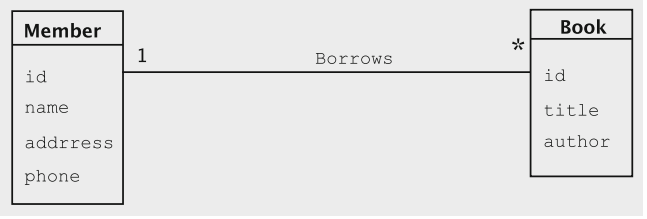

Fig. 6.7 UML diagram showing the association `Borrows` between `Member` and `Book`

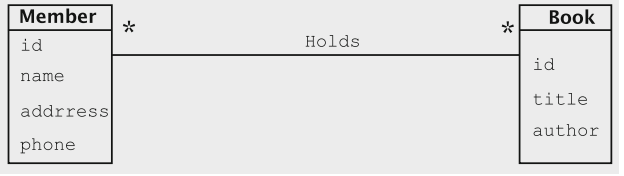

Fig. 6.8 UML diagram showing the association `Holds` between `Member` and `Book`

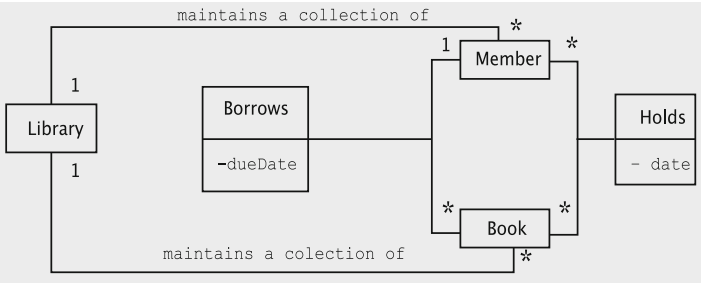

Fig. 6.9 Conceptual classes and their associations

We capture all of the conceptual classes and their associations into a single diagram in Fig. 6.9. To reduce complexity, we have omitted the attributes of `Library`, `Member`, and `Book`. As seen before, a relationship formed between two entities is sometimes accompanied by additional information. This additional information is relevant only in the context of the relationship. There are two such examples in the inter-class relationships we have seen so far: when a user borrows a book and when a user places a hold on a book. Borrowing a book introduces new information into the system, viz., the date on which the book is due to be returned. Likewise, placing a hold introduces some information, viz., the date after which the book is not needed. The lines representing the association are augmented to represent the information that must be stored as part of the association. For the association `Borrows` and the line connecting `Member` and `Book`, we come up with a conceptual class named `Borrows` having an attribute named `dueDate`. Similarly, we create a conceptual class named `Holds` with the attribute called `date` to store the information related to the association `Holds`. Both these conceptual classes are attached to the line representing the corresponding associations.

It is important to note that the above conceptual classes or their representation do not, in any way, tell us how the information is going to be stored or accessed. Those decisions will be deferred to the design and implementation phase. For instance, there may be additional classes to support the operations of the `Library` class. We may discover that while some of the conceptual classes have corresponding physical realisations, some may disappear and the necessary information may be stored as fields distributed over multiple classes. We may discover that while some of the conceptual classes have corresponding physical realisations, some may disappear and the necessary information may be stored as fields distributed over multiple classes. We may choose to move fields that belong to an association elsewhere. For instance, the field `dueDate` may be stored as a field of the book or as a separate object, which holds a reference to the book object and the user object involved. Upon making that choice, the designer decides how the conceptual relationship between User and Book is going to be physically realised. The conceptual class diagram is simply that: **conceptual**.

6.5 Using the Knowledge of the Domain

Domain analysis is the process of analysing related application systems in a domain so as to discover what features are common between them and what parts are variable. In other words, we identify and analyse common requirements from a specific application domain. In contrast to looking at a certain problem completely from scratch, we apply the knowledge we already have from our study of similar systems to speed up the creation of specifications, design, and code. Thus, one of the goals of this approach is reuse.

Any area in which we develop software systems qualifies to be a **domain**. Examples include library systems, hotel reservation systems, university registration systems, etc. We can sometimes divide a domain into several interrelated domains. For example, we could say that the domain of university applications includes the domain of course management, the domain of student admissions, the domain of payroll applications, and so on. Such a domain can be quite complex because of the interactions of the smaller domains that make up the bigger one.

Before we analyse and construct a specific system, we first need to perform an exhaustive analysis of the class of applications in that domain. In the domain of libraries, for example, there are things we need to know including the following.

1. The environment, including customers and users. Libraries have loanable items such as books, CDs, periodicals, etc. A library's customers are members. Libraries buy books from publishers.
2. Terminology that is unique to the domain. For example, the Dewey decimal classification (DDC) system for books.
3. Tasks and procedures currently performed. In a library system, for example:
   (a) Members may check out loanable items.
   (b) Some items are available only for reference; they cannot be checked out.
   (c) Members may put holds on loanable items.
   (d) Members will pay a fine if they return items after the due date.

Finding the Right Classes

In general, finding the right classes is non-trivial. It must be remembered that this process is iterative, i.e., we start with a set of classes and complete a conceptual design. In the process of walking through the use case implementations, we may find that some classes have to be dropped and some others have to be added. Familiarity with Design Patterns also helps in recognizing the classes. The following thumb rules and caveats come in handy:

- In general, do not build classes around functions. There are exceptions to this rule as we will see in Chap. 9. Write a class description. If it reads 'This class performs...' we most likely have a problem. If class name is imperative, e.g., print, parse, etc., it is likely that either the class is wrong or the name is wrong.
- Remember that a class usually has more than one method; otherwise it is probably a method that should be attached to some other class.
- Do not form an inheritance hierarchy too soon unless we have a pre-existing taxonomy. (Inheritance is supposed to be a relationship among well-understood abstractions.)
- Be wary of classes that have no methods, (or only query methods) because they are not frequent. Some situations in which they occur are:
  (i) representing objects from outside world, (ii) encapsulating facilities, constants or shared variables, (iii) applicative classes used to describe non-modifiable objects, e.g., integer class in Java generates new integers, but does not allow modification of integers.
- Check for the following properties of the ideal class: (i) a clearly associated abstraction, which should be a data abstraction (as opposed to a process abstraction), (ii) a descriptive noun/adjective for the class name, (iii) a non-empty set of runtime objects, (iv) queries and commands, (v) abstract properties that can be described as pre/post conditions and invariants.

One of the major activities of this analysis is discovering the business rules, the rules that any properly-functioning system in that domain must conform to.

Where does the knowledge of a specific domain come from? It could be from sources such as surveys, existing applications, technical reports, user manuals, and so on. As shown in Fig. 6.10, a domain analyst analyses this knowledge to come up with specifications, designs, and code that can be reused in multiple projects.

Clearly, a significant amount of effort has to be expended to domain analysis before undertaking the specific problem. The benefit is that after the initial investment of resources, the products (such as specifications, designs, code, test data, etc.) can be reused for the development of any number of applications in that domain. This reduces development time and cost.

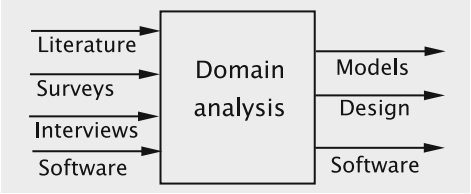

Fig. 6.10 Domain analysis

6.6 Discussion and Further Reading

A detailed treatment of object-oriented analysis methods can be found in [1]. The rules for finding the right classes are condensed from [2].

Obtaining the requirements specification is typically part of a larger '*plan and elaborate phase*' that would be an essential component of any large project. In addition to specification of requirements, this phase includes such activities as the *initial conception*, *investigation of alternatives, planning, budgeting* etc. The end product of this phase will include such documents as the *Plan* showing a schedule, resources, budget etc., a *preliminary investigation report* that lists the motivation, alternatives, and business needs, *requirements specification*, a *glossary* as an aid to understanding the vocabulary of the domain, and, perhaps, a *rough conceptual model*. Larger systems typically require more details before the analysis can proceed.

Use case modeling is one of the main techniques of a more general field of study called *usage modeling*. Usage modeling employs the following techniques: *essential use cases, system use cases*, *UML use case diagrams, user stories* and *features* [3]. What we have discussed here are essential use cases, which deal only with the fundamental business task without bringing technological issues into account. These are used to explore usage-based requirements.

Making sure that our use cases have covered all the business processes is in itself a non-trivial task. This area of study, called *business process modeling*, employs tools such as *data flow diagrams, flowcharts,* and *UML Activity Diagrams* [3] and is used to create process models for the business.

There are several UML tools available for analysis, and new variants are being constantly developed. What a practitioner chooses often depends on the development package being employed. A good, compact reference to the entire language can be found in [4]. The use case table and the class diagram with associations exemplify the very basic tools of object-oriented analysis.

There is no prescribed analysis or design technique that software designer must follow at all costs. There are several methodologies in vogue, and these ideas continue to evolve over time. In [5] it has been pointed out that while some researchers and developers are of the opinion that object-oriented methodologies are a revolutionary change from the conventional techniques, others have argued that object-oriented techniques are nothing but an elaboration of structured design. A comparative study of various object-oriented and conventional methodologies is also presented in that article.

Projects

1. **A database for a warehouse** A large warehousing corporation operates as follows:

   (a) The warehouse stocks several products, and there are several manufacturers for each product.

(b) The warehouse has a large number of registered clients. The clients place orders with the warehouse, which then ships the goods to the client. This process is as follows: the warehouse clerk examines the client's order and creates an invoice, depending on availability of the product. The invoice is then sent to the shop floor where the product is packed and shipped along with the invoice. The unfilled part of the order is placed in a waiting list queue.
(c) When the stock of any product runs low, the warehouse orders that product from one of the manufacturers, based on the price and terms of delivery.
(d) When a product shipment is received from a manufacturer, the orders in the waiting list are filled in first. The remainder is added to the inventory.

The business processes: The warehouse has three main operational business processes, namely,

(a) receiving and processing an order from a client,
(b) placing an order with the manufacturer,
(c) receiving a shipment,
(d) receiving payment from a client.

Let us examine the first of these. When an order is received from a client, the following steps are involved:

(a) Clerk receives the order and enters the order into the system.
(b) The system generates an invoice based on the availability of the product(s).
(c) The clerk prints the invoice and sends it over to the storage area.
(d) A worker on the floor picks up the invoice, retrieves the product(s) from the shelves and packs them, and ships the goods and the invoice to the client.
(e) The worker requests the system to mark the order as having been shipped.
(f) The system updates itself by recording the information.

This is an interesting business process because of the fact that steps of printing the invoice and retrieving the product from the shelves are performed by different actors. This introduces an indefinite delay into the process. If we were to translate this into a single end-to-end use case, we have a situation where the system will be waiting for a long time to get a response from an actor. It is therefore appropriate to break this up into two use cases as follows:
1. Use case create-invoice.
2. Use case fill-invoice.

In addition to these operational business processes, the warehouse will have several other querying and accounting processes such as:

(a) Registering a new client.
(b) Adding a new manufacturer for a certain product.
(c) Adding a new product.
(d) Printing a list of clients who have defaulted on payments.
(e) Printing a list of manufacturers who are owed money by the warehouse, etc.

Write the use cases, and determine the conceptual classes and their relationships.

2. **Managing a university registration system**

   A small university would like to create a registration system for its students. The students will use this system to obtain information about courses, when and where the classes meet, register for classes, print transcripts, drop classes, etc. The faculty will be using this system to find out what classes they are assigned to teach, when and where these classes meet, get a list of students registered for each class, and assign grades to students in their classes. The university administrative staff will be using this database to add new faculty and students, remove faculty and students who have left, put in and update information about each course the university ofers, enter the schedules for classes that are being offered in each term, and any other housekeeping tasks that need to be performed.
   Your task is to analyse this system, extract and list the details of the various business processes, develop the use cases, and find the conceptual classes and their relationships.
   In finding the classes for this system, one of the issues that comes up is that of distinguishing a course from an offering of the course. For instance 'CS 430: Principles of Object-Oriented Software Construction' is a course listed in the university's course bulletin. The course is offered once during the fall term and once during the spring term. Each offering may be taught at a different time and place, and in all likelihood will have a different set of students. Therefore, all offerings have some information in common and some information that is unique to that offering. How will you choose a set of classes that models all these interactions?

3. **Creating an airline reservation and staff scheduling database**

   An airline has a weekly flight schedule. Associated with each flight is an aircraft, a list of crew, and a list of passengers. The airline would like to create and maintain a database that can perform the following functions:
   For passengers Add a passenger to the database, reserve a seat on a flight, print out an itinerary, request seating and meal preferences, and update frequent flier records.
   For crew Assign crew members to each flight, allow crew members to view their schedule, keep track of what kinds of aircraft the crew member has been trained to operate.
   For flights Keep track of crew list, passenger list, and aircraft to be used for that flight.
   For aircraft Maintain all records about the aircraft and a schedule of operation.
   Make an exhaustive list of queries that this system may be required to answer. Carry out a requirements analysis for the system and model it as a collection of use cases. Find the conceptual classes and their relationships.

6.7 Exercises

1. In the use case Issue Book, the system displays the transaction details with each book. Modify this so that there is only one display of transactions at the very end of the process.
2. (Discussion) In a real library, there would be several other kinds of query operations that would be performed. Carry out a brainstorming exercise to come up with a more complete list of use cases for a real library system.
3. A hotel reservation system supports the following functionality:

   (a) Room reservation
   (b) Changing the properties of a room (for example, from non-smoking to smoking)
   (c) Customer check-in
   (d) Customer check-out

   Come up with system use cases for the above functionality.
4. We are building a system to track personal finances. We plan an initial version with minimal functionality: tracking the expenditures. (Each expenditure has a description, date and amount.) We show below the use case for creating a new expenditure item and a new income item.

Actor	System
(1) Inputs a request to create a new expenditure item	
	(2) Asks for description, date, and amount
(3) Supplies the data	
	(4) Creates an expenditure item and notifies the user

Actor	System
(1) Inputs a request to create a new income item	
	(2) Asks for description, date, and amount
(3) Supplies the data	
	(4) Creates an income item and notifies the user

   (a) The use cases are quite weakly specified. In what ways? (Hint: Compare with the addition of a new member or book in the library system.)
   (b) What are the alternate flows in the use cases? Modify the two use cases to handle the alternate flows.
   (c) Identify the conceptual classes.

5. Consider the policies maintained by an automobile insurance company. A policy has a primary policy holder, a set of autos insured, and a list of people who are covered by the insurance. From your knowledge of insurance, come up with system use cases for

   (a) creating a new policy
   (b) adding a new person to a policy
   (c) adding a new automobile to a policy
   (d) recording a claim.

6. Consider an information system to be created for handling the business of a supermarket. For each of the following, state if it is a possible class. If not, explain why not. Otherwise, why would you consider it to be a class? What is its role in the system?

   (a) Customer
   (b) Vegetable
   (c) Milk
   (d) Stock
   (e) Canned food
   (f) Quantity on hand for a product

7. A company has several projects, and each employee works in a single project. The human resource system evaluates the personnel needs of each project and matches them against the personnel file to find the best possible employees to be assigned to the project. Come up with the conceptual classes by conducting use case analysis.
8. Explain why mistakes made in the requirements analysis stage are the costliest to correct.
9. Among the following requirements, which are functional and which are non-functional?

   (a) Paychecks should be printed every two weeks.
   (b) Database recovery should not take more than one hour.
   (c) The system should be implemented using the C++ language.
   (d) It should be possible to selectively print employee checks.
   (e) Employee list should be displayed in lists of size 10.

10. Suppose the library system has to be augmented so that it can support inter-library loans. That is, a customer can ask the clerk if a certain book, which is not locally available, is available in some other library. What changes are needed (classes and use cases) to incorporate this new functionality?
11. In Problem 6, assume that a customer may pay with cash, check, or credit/debit cards. Should this aspect be taken into consideration while developing the use case for purchasing grocery? Justify your answer.

12. Again, in Problem 6, suppose that a user may check out by interacting with a sales clerk or independently in an automated checkout counter. Should there be two versions of the grocery purchase use case? Explain.
13. What are the advantages of ignoring implementation-related aspects while performing analysis?

References

1. C. Larman, *Applying UML and Patterns* (Prentice Hall PTR, 1998)
2. B. Meyer, *Object-Oriented Software Construction* (Prentice Hall, 1997)
3. S. Ambler, *The Object Primer: Agile Model-Driven Development with UML 2.0* (Cambridge University Press, 2004)
4. M. Fowler, K. Scott, *UML Distilled* (Addison-Wesley Longman, 1997)
5. R. Fichman, C. Kemerer, *Object-Oriented and Conventional Analysis and Design Methodologies* (IEEE Computer Society Press, 1995)