within ()

Object diagrams

Object diagrams shows the snapshot of the objects in a system at a point in time. Since it shows instances of classes, rather than the classes themselves, it is often called as an instance diagram. The objects are drawn using rounded rectangles (see Figure 7.28).

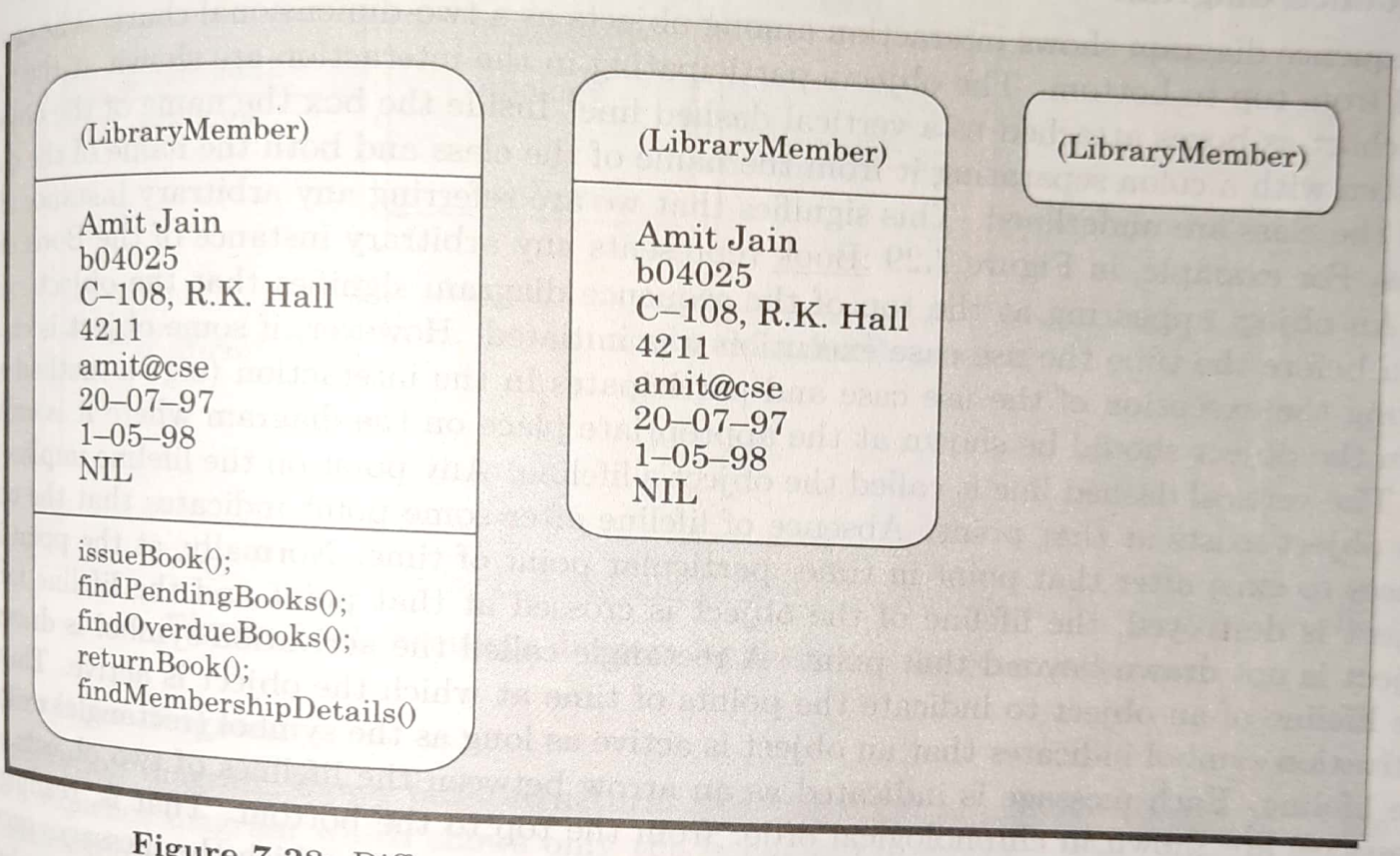

Figure 7.28: Different representations of a LibraryMember object.

An object diagram may undergo continuous change as execution proceeds. For example, links may get formed between objects and get broken. Objects may get created and destroyed, and so on. Object diagrams are useful to explain the working of a system.

7.4 USE CASE MODEL

The use case model for any system consists of a set of use cases.

Intuitively, the use cases represent the different ways in which a system can be used by the users.

[2] A functional model is constructed from functions.
[3] An object model is constructed from objects.

A simple way to find all the use cases of a system is to ask the question "What all can the users do by using the system?" Thus, for the Library Information System (LIS), the use cases could be:

- Issue-book
- Query-book
- Return-book
- Create-member
- Add-book, etc.

Roughly speaking, the use cases correspond to the high-level functional requirements discussed in Chapter 4. We can also say that the use cases partition the system behaviour into transactions, such that each transaction performs some useful action from the user's point of view. Each transaction, to complete, may involve multiple message exchanges between the user and the system.

The purpose of a use case is to define a piece of coherent behaviour without revealing the internal structure of the system. The use cases do not mention any specific algorithm to be used nor the internal data representation, internal structure of the software. A use case typically involves a sequence of interactions between the user and the system. Even for the same use case, there can be several different sequences of interactions. A use case consists of one main line sequence and several alternate sequences. The main line sequence represents the interactions between a user and the system that normally take place. The mainline sequence is the most frequently occurring sequence of interaction. For example, in the mainline sequence of the withdraw cash use case supported by a bank ATM would be: the user inserts the ATM card, enters password, selects the amount withdraw option, enters the amount to be withdrawn, completes the transaction, and collects the amount. Several variations to the main line sequence called **alternate sequences** may also exist. Typically, a variation from the mainline sequence occurs when some specific conditions hold. For the bank ATM example, several variations or alternate sequences may occur. For example, consider the situation in which the password is invalid or the amount to be withdrawn exceeds the account balance. The mainline sequence and each of the alternate sequences corresponding to the invocation of a use case is called a **scenario** of the use case.

A use case can be viewed as a set of related scenarios tied together by a common goal. The main line sequence and each of the variations are called *scenarios* or instances of the use case. Each scenario is a single path of user events and system activity.

Normally, each use case is independent of the other use cases. However, implicit dependencies among use cases may exist because of dependencies existing among use cases at the implementation level due to shared resources, objects or functions. For example, in the Library Automation System example, `renew-book` and `reserve-book` are two independent use cases. But, in actual implementation of `renew-book`, a check is to be made to see if any book has been reserved by a previous execution of the `reserve-book` use case. Another example of dependence among use cases is the following. In the Bookshop Automation Software, `update-inventory` and `sale-book` are two independent use cases. But, during execution of

`sale-book` there is an implicit dependency on `update-inventory`. Since when sufficient quantity is unavailable in the inventory, `sale-book` cannot operate until inventory is replenished using `update-inventory`.

The use case model is an important analysis and design artifact. As already mentioned, other UML models must conform to this model in any use case-driven (also called as the user-centric) analysis and development approach. It should be remembered that the use case model is not really an object-oriented model according to a strict definition of the term.

In contrast to all other types of UML diagrams, the use case model represents a functional or process model of a system.

7.4.5 Factoring of Commonality among Use Cases

It is often desirable to factor use cases into component use cases. All use cases need not be factored. In fact, factoring of use cases are required under two situations as follows:

- Complex use cases need to be factored into simpler use cases. This would not only make the behaviour associated with the use case much more comprehensible, but also make the corresponding interaction diagrams more tractable. Without decomposition, the interaction diagrams for complex use cases may become too large to be accommodated on a single standard-sized (A4) paper.
- Use cases need to be factored whenever there is common behaviour across different use cases. Factoring would make it possible to define such behaviour only once and reuse it wherever required.

It is desirable to factor out common usage such as error handling from a set of use cases. This makes analysis of the class design much simpler and elegant. However, a word of caution here. Factoring of use cases should not be done except for achieving the above two objectives. From the design point of view, it is not advantageous to break up a use case into many smaller parts just for the sake of it. UML offers three factoring mechanisms as discussed further.

Generalization

Use case generalization can be used when you have one use case that is similar to another, but does something slightly differently or something more. Generalization works the same way with use cases as it does with classes. The child use case inherits the behavior and meaning of the present use case. The notation is the same too (see Figure 7.16). It is important to remember that the base and the derived use cases are separate use cases and should have separate text descriptions.

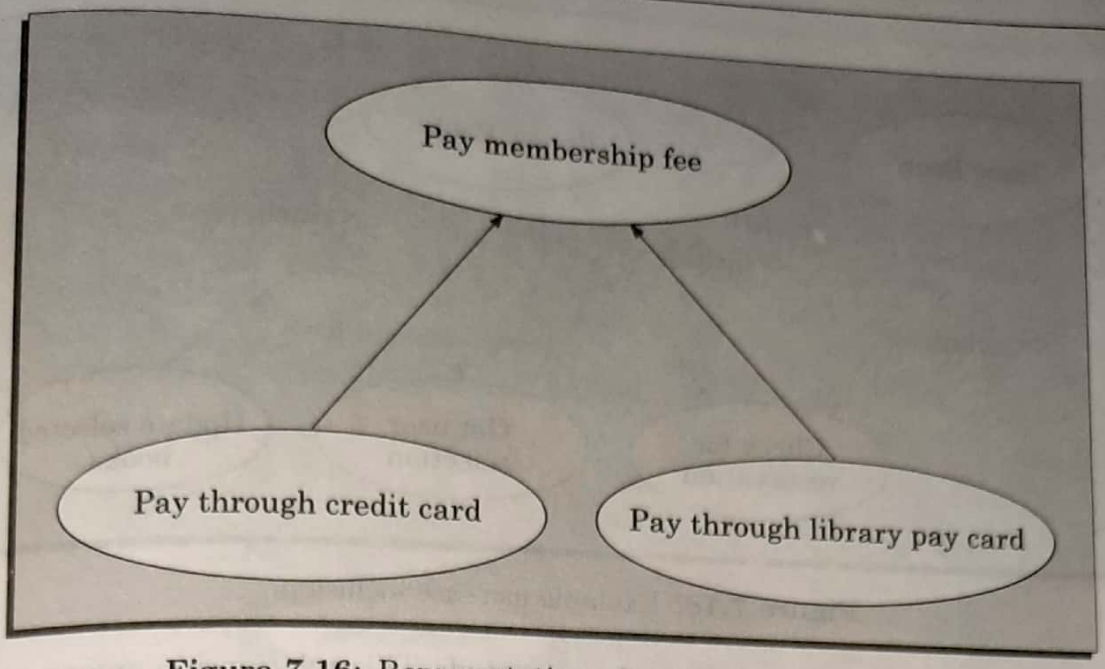

Figure 7.16: Representation of use case generalization.

Includes

The includes relationship in the older versions of UML (prior to UML 1.1) was known as the uses relationship. The includes relationship implies one use case includes the behaviour of another use case in its sequence of events and actions. The includes relationship is appropriate when you have a chunk of behaviour that is similar across a number of use cases. The factoring of such behaviour will help in not repeating the specification and implementation across different use cases. Thus, the includes relationship explores the issue of reuse by factoring out the commonality across use cases. It can also be gainfully employed to decompose a large and complex use case into more manageable parts.

As shown in Figure 7.17, the includes relationship is represented using a pre-defined stereotype ≪include≫. In the includes relationship, a base use case compulsorily and automatically includes the behaviour of the common use case. As shown in example Figure 7.18, the use cases `issue-book` and `renew-book` both include `check-reservation` use case. The base use case may include several use cases. In such cases, it may interleave their associated common use cases together. The common use case becomes a separate use case and independent text description should be provided for it.

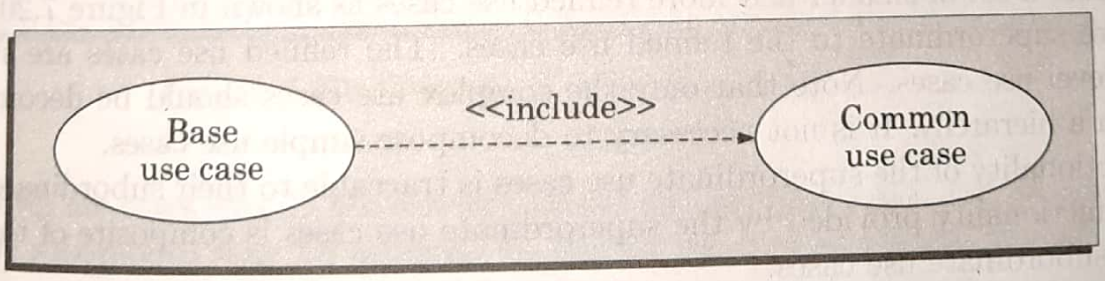

Figure 7.17: Representation of use case inclusion.

Extends

The main idea behind the extends relationship among use cases is that it allows you show optional system behaviour. An optional system behaviour is executed only if certain conditions

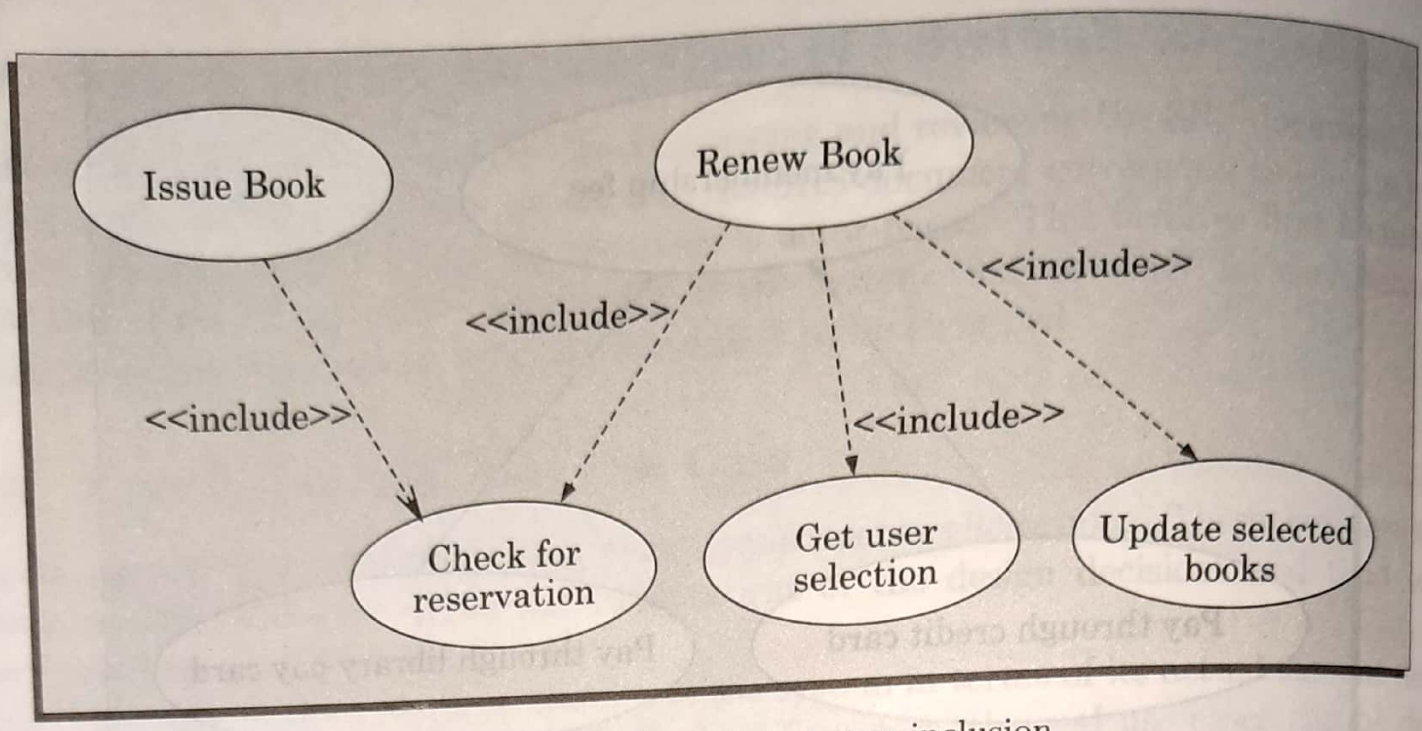

Figure 7.18: Example use case inclusion.

hold, otherwise the optional behaviour is not executed. This relationship among use cases is also predefined as a stereotype as shown in Figure 7.19.

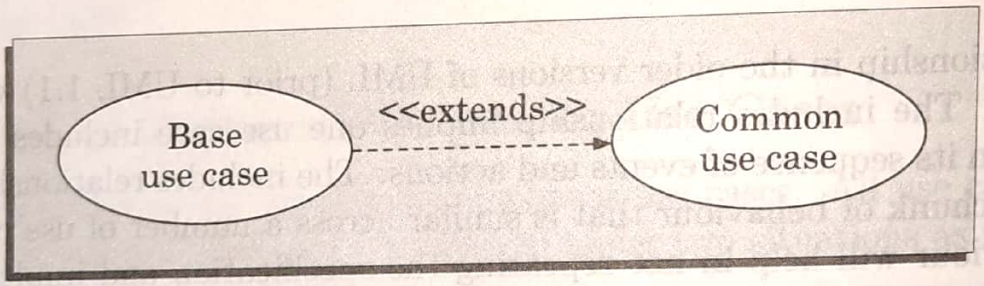

Figure 7.19: Example use case extension.

The extends relationship is similar to generalization. But unlike generalization, the extending use case can add additional behaviour only at an extension point only when certain conditions are satisfied. The extension points are points within the use case where variation to the mainline (normal) action sequence may occur. The extends relationship is normally used to capture alternate paths or scenarios.