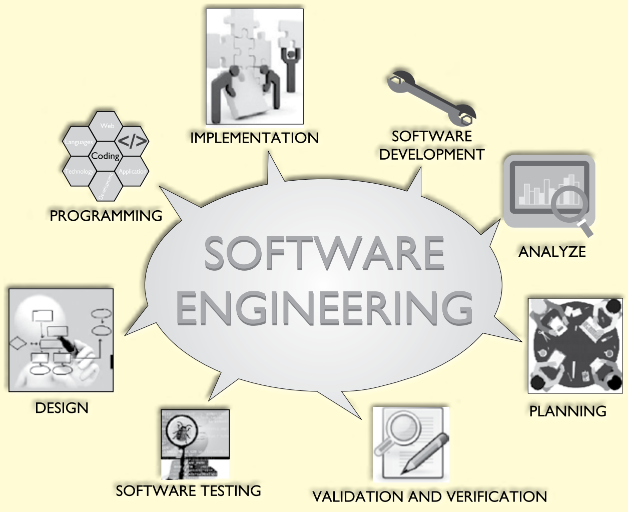
CHAPTER

7

OBJECT MODELLING USING UML



CHAPTER OUTLINE

- Basic Object-Orientation Concepts
- Unified Modelling Language (UML)
- UML Diagrams
- Use Case Model
- Class Diagrams
- Interaction Diagrams
- Activity Diagram
- State Chart Diagram

There are two ways of constructing a software design: One way is to make it so simple that there are obviously no deficiencies, and the other way is to make it so complicated that there are no obvious deficiencies. The first method is far more difficult.

—C.A.R. Hoare

LEARNING OBJECTIVES

- Basic object-orientation concepts
- Overview of modeling using unified modeling language (UML)
- Requirements modeling using use case diagram
- Structure modeling using class and object diagrams
- Behaviour modeling using sequence, collaboration, and state machine diagrams

In recent years, the object-oriented software development style has become very popular and is at present being widely used in industry as well as in academic circles. Since its inception in the early eighties, the object technology has made rapid progress. From a modest beginning in the early eighties, the advancements to the object technology gathered momentum in the nineties and the technology is now nearing maturity. Considering the widespread use and popularity of the object technology in both industry and academia, it is important to learn this technology well.

It is well known that mastering an object-oriented programming language such as Java or C++ rarely equips one with the skills necessary to develop good quality object-oriented software—it is important to learn the object-oriented design skills well. Once a good design has been arrived at, it is easy to code it using an object-oriented language. It has now even become possible to automatically generate much of the code from the design by using a CASE tool. In order to arrive at a satisfactory object-oriented design (OOD) solution to a problem, it is necessary to create several types of models. But, one may ask: "What has modelling got anything to do with designing?" Let us answer this question in the following:

A model is constructed by focusing only on a few aspects of the problem and ignoring the rest. A model of a given problem is called its *analysis model*. On the other hand, a model of the solution (code) is called *design model*. The design model is usually obtained by carrying out iterative refinements to the analysis model using a design methodology.

Let us distinguish between the model of a problem and the model of its solution. A design is a model of the solution, whereas any model of the problem is an analysis model. In this chapter, we shall discuss how to document a model using a modelling language. In the subsequent chapter, we shall discuss a design process that can be used to iteratively refine an analysis model into a design model.

In the context of model construction, we need to carefully understand the distinction between a modelling language and a design process, since we shall use these two terms frequently in our discussions.

Modelling language: A modelling language consists of a set of notations using which design and analysis models are documented.

Design process: A design process addresses the following issue: "Given a problem description, how to systematically work out the design solution to the problem?" In other words, a design process is a step by step procedure (or recipe) using which a problem description can be converted into a design solution. A design process is, at times, also referred to as a *design methodology*. In this text, we shall use the terms *design process* and *design methodology* interchangeably.

A model can be documented using a modelling language such as *unified modelling language* (UML). Over the last decade, UML has become immensely popular. UML has also been accepted by ISO as a standard for modelling object-oriented systems. In this Chapter, we primarily discuss the syntax and semantics of UML. However, before discussing the nitty-gritty of the syntax and semantics of UML, we review a few basic concepts and terminologies that have come to be associated with object-orientation.

7.1 BASIC OBJECT-ORIENTATION CONCEPTS

The principles of object-orientation have been founded on a few simple concepts. Some of these concepts are pictorially shown in Figure 7.1. After discussing these basic concepts, we examine a few related technical terms.

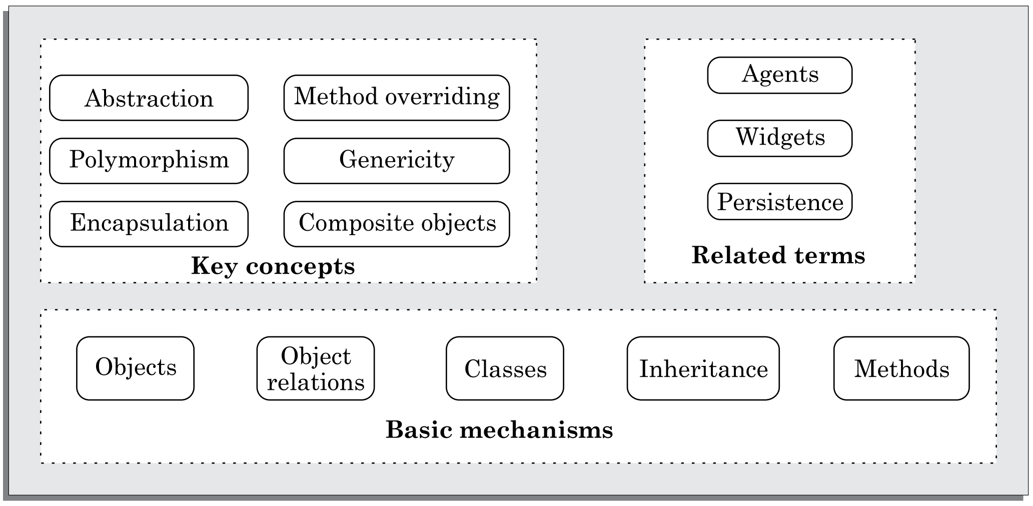


FIGURE 7.1 Important concepts used in the object-oriented approach.

7.1.1 Basic Concepts

A few important concepts that form the corner stones of the object-oriented paradigm have pictorially been shown in Figure 7.1. In the following sections and subsections, we discuss these concepts in detail.

Objects

In the object-oriented approach, it is convenient to imagine the working of a software in terms of a set of interacting objects. This is analogous to the way objects interact in a real-world system for getting some work done. For example, consider a manually operated library system. For issuing a book, an entry needs to be made in the *issue register* and then the return date needs to be stamped on the book. In an object-oriented library automation software, analogous activities involving the book object and the issue register object take place.

Each object in an object-oriented program usually represents a tangible real-world entity such as a library member, a book, an issue register, etc. However while solving a problem, in addition to considering the tangible real-world entities as objects, it becomes advantageous at times to consider certain conceptual entities (e.g., a scheduler, a controller, etc.) as objects as well. This simplifies the solution and helps to arrive at a good design.

When a system is analysed, developed, and implemented in terms of objects, it becomes easy to understand the design and the implementation of the system, since objects provide an excellent decomposition of a large problem into small parts.

A key advantage of considering a system as a set of objects is an excellent decomposition of the system into parts that have low coupling and high cohesion.

Each object essentially consists of some data that is private to the object and a set of functions (termed as *operations* or *methods*) that operate on those data. This aspect has pictorially been illustrated in Figure 7.2. Observe that the data of the object can only be accessed by the methods of the object. Consequently, external objects can only access data of an object through invocation of its methods. In fact, the methods of an object have the sole authority to operate on the data that is private to the object. In other words, no object can directly access the data of any other object. Therefore, each object can be thought of as hiding its internal data from other objects. However, an object can access the private data of another object by invoking the methods supported by that object. This mechanism of hiding data from other objects is popularly known as the principle of *data hiding* or *data abstraction*. Data hiding promotes high cohesion and low coupling among objects, and therefore is considered to be an important principle that can help one to arrive at a reasonably good design.

As already mentioned, each object stores some data and supports certain operations on the stored data. As an example, consider the `libraryMember` object of a library automation application. The private data of a `libraryMember` object can be the following:

- name of the member
- membership number
- address
- phone number
- e-mail address
- date when admitted as a member
- membership expiry date
- books outstanding

The operations supported by a `libraryMember` object can be the following:

- issue-book
- find-books-outstanding
- find-books-overdue
- return-book
- find-membership-details

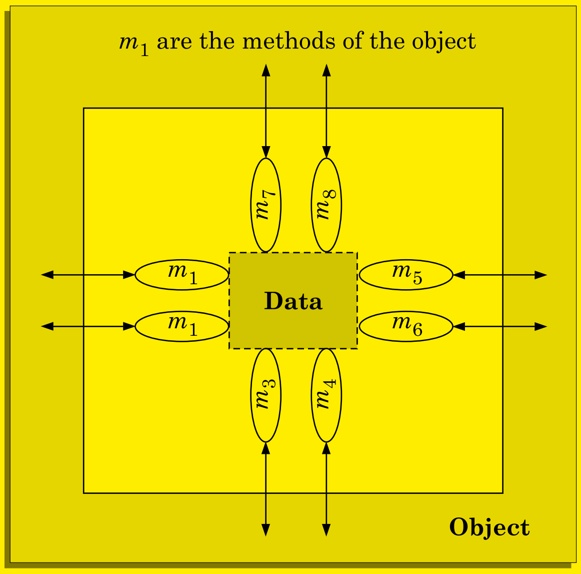


FIGURE 7.2 A model of an object

The data stored internally in an object are called its *attributes*, and the operations supported by an object are called its *methods*.

Though the terminologies associated with object-orientation are simple and well-defined, a word of caution here: the term 'object' is often used rather loosely in the literature and also in this text. Often an 'object' would mean a single entity. However, at other times, we shall use it to refer to a group of similar objects (class). In this text, usually the context of use would resolve the ambiguity, if any.

Class

Similar objects constitute a class. That is, all the objects possessing similar attributes and methods constitute a class. For example, the set of all library members would constitute the class LibraryMember in a library automation application. In this case, each library member object has the same set of attributes such as `member name`, `membership number`, `member address`, etc. and also each has methods such as `issue-book`, `return-book`, etc. Once we define a class, it can be used as a template for object creation.

Let us now investigate the important question: Whether a class is an abstract data type (ADT)? To be able to answer this question, we must first be aware of the basic definition of an ADT. We first discuss the same in a nutshell. There are two things that are inherent to an ADT—*abstract data* and *data type*. In programming language theory, a data type identifies a group of variables having a particular behaviour. In particular, a data type identifies the following:

Representation: The way values of variables of that type can be stored.

Value space: The possible values which a variable of that type can be assumed.

Behaviour: The operations that can be applied on variables of that type and the result that would be obtained.

A data type can be instantiated to create a variable. For example, `int` is a type in C++ language. When we write an instruction `int i;` we are actually creating an instance of `int` that is identified by the name `i`. From this, we can infer the following—An ADT is a type where the data contained in each instantiated entity is abstracted (hidden) from other entities. Let us now examine whether a class supports the two mechanisms of an ADT.

Abstract data: The data of an object can be accessed only through its methods. In other words, the exact way data is stored internally (stack, array, queue, etc.) in the object is abstracted out (not known to the other objects).

Data type: In programming language terminology, a data type defines a collection of data values and a set of predefined operations on those values. Thus, a data type can be instantiated to create a variable of that type. We can instantiate a class into objects. Therefore, a class is a data type.

It is easy to see that programming using ADT allows implementation flexibility. That is, the internal storage of a data can be changed even after the program is operational, without affecting the working of the program. It can be inferred from the above discussions that a class is an ADT. But, an ADT need not be a class, since to be a class it needs to support the inheritance and other object-orientation properties.

Every class is an ADT, but not all ADTs are classes.

Methods

The operations (such as `create`, `issue`, `return`, etc.) supported by an object are implemented in the form of *methods*. Notice that we are distinguishing between the terms *operation* and *method*. Though the terms 'operation' and 'method' are sometimes used interchangeably, there is a technical difference between these two terms which we explain in the following.

An operation is a specific responsibility of a class, and the responsibility is implemented in the form of a method. However, it is at times useful to have multiple methods to implement a single responsibility. In this case, all the methods share the same name (that is, the name of the operation), but parameter list of each method is required to be different for enabling the compiler to determine the exact method to be bound on a method call. We, therefore, say that in this case, the operation name is overloaded with multiple implementations of the operation. As an example, consider the following. Suppose one of the responsibilities of a class named `Circle` is to create instances of itself. Assume that the class provides three definitions for the create operation—`int  reate()`, `int create(int radius)` and `int create(float x, float y, int radius);`. In this case, we say that `create` is an overloaded method. When an operation has a single implementation, there is no difference between the terms operation and method.

The implementation of a responsibility of a class through multiple methods that have the same name is called method overloading.

Methods are the only means available to other objects in a software for accessing and manipulating the data of another object. Let us now try to understand the distinction between a message and a method.

In Smalltalk, an object could request the services of other objects by sending messages to them. The idea was that the mechanism of message passing would lead to weak coupling among objects. Though this was an important feature of Smalltalk, yet the programmers who were trying to migrate from procedural programming to object-oriented programming, found it to be a paradigm shift and therefore difficult to accept. Subsequently, when the C++ language was designed, message passing was implemented by method invocation (similar to a function call). This was rapidly accepted by the programmers. Later object-oriented languages such as Java have followed the same trait of retaining the method invocation feature, that is, normally associated with the procedural languages.

7.1.2 Class Relationships

Classes in a program can be related to each other in the following four ways:

- Inheritance
- Association and link
- Aggregation and composition
- Dependency

In the following subsection, we discuss these different types of relationships that can exist among classes.

Inheritance

The inheritance feature allows one to define a new class by extending the features of an existing class. The original class is called the *base class* (also called *superclass* or *parent class*) and the new class obtained through inheritance is called the *derived class* (also called a *subclass* or a *child class*). The derived class is said to inherit the features of the base class. An example of inheritance is shown in Figure 7.3. In Figure 7.3, observe that the classes `Faculty, Students, and Staff` have been derived from the base class `LibraryMember` through an inheritance relationship (note the special type of arrow that has been used to draw it). The inheritance relation between library member and faculty can alternatively be expressed as the following—A faculty member is a special type of library member. The inheritance relationship is also at times called *is a* relation.

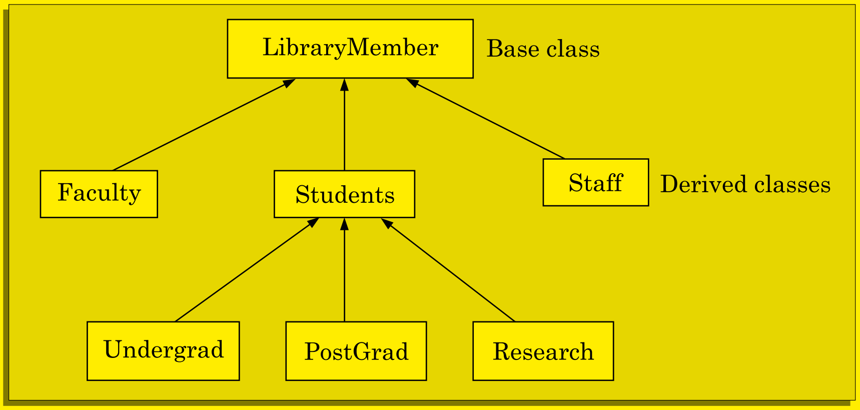


FIGURE 7.3 Library information system example.

A base class is said to be a generalisation of its derived classes. This means that the base class should contain only those properties (i.e., data and methods) that are common to all its derived classes. For example, in Figure 7.3 each derived class supports the `issue-book` method, and this method is supported by the base class as well. In other words, each derived class is a specialised base class that extends the base class functionalities in certain ways.

Each derived class can be considered as a specialisation of its base class because it modifies or extends the basic properties of the base class in certain ways. Therefore, the inheritance relationship can be viewed as a generalisation-specialisation relationship.

Observe that in Figure 7.3 the classes `Faculty`, `Student`, and `Staff` are all special types of library members. Several things are common among these members. These include attributes such as membership id, member name and address, date on which books issued out, etc. However, for the different categories of members, the issue procedure may differ since for example, different types members may issue books for different durations. We can convey as much by saying that the classes `Faculty`, `Staff`, and `Students` are special types of `LibraryMember` classes. Using the inheritance relationship, different classes can be arranged in a class hierarchy as shown in Figure 7.3.

In addition to inheriting all the data and methods of the base class, a derived class usually defines some new data and methods. A derived class may even redefine some methods which already exist in the base class. Redefinition in a derived class of a method that already exists in its base class is termed as *method overriding*.

When a new definition of a method that existed in the base class is provided in a derived class, the base class method is said to be overridden in the derived class.

The inheritance relationship existing among certain classes in a library automation system is shown in Figure 7.3. As shown, `LibraryMember` is the base class for the derived classes `Faculty`, `Student`, `and Staff`. Similarly, `Student` is the base class for the derived classes `Undergrad`, `Postgrad`, `and Research`. Each derived class inherits all the data and methods of the base class, and defines some additional data and methods or modifies some of the inherited data and methods. The inheritance relationship has been represented in Figure 7.3 using a directed arrow drawn from a derived class to its base class. We now illustrate how the method of a base class is overridden by the derived classes. In Figure 7.3, the base class `LibraryMember` might define the following data—`member name`, `address`, `and library membership number`. Though faculty, student, and staff classes inherit these data, they need to redefine their respective `issueBook` method because for the specific library that we are modelling, the number of books that can be borrowed and the duration of loan are different for different categories of library members.

Inheritance is a basic mechanism that every object-oriented programming language supports. If a language supports ADTs, but does not support inheritance, then it is called an *object-based* language and not object-oriented. An example of an object-based programming language is Ada.

Two important advantages of using the inheritance mechanism in programming are code reuse and simplicity of program design.

Now let us try to understand why we need the inheritance relationship in the first place. Can't we program as well without using the inheritance relationship?

Let us examine how code reuse and simplicity of design come about while using the inheritance mechanism.

Code reuse and consequent reduction in development effort: If certain methods or data are observed to be similar across several classes, then instead of defining these methods and data in each of the classes separately, these methods and their associated data are defined only once in the base class and then inherited by each of the subclasses. For example, in the Library Information System example of Figure 7.3, the classes corresponding to each category of member (that is, `Faculty`, `Student`, and `Staff`) need to store the following data: `member-name`, `member-address`, and `membership-number`. Therefore, these data are defined only once in the base class `LibraryMember` and are inherited by all its subclasses.

Simplification of code and design: An=advantage that accrues from the use of the inheritance mechanism is the conceptual simplification brought about through the reduction of the number of independent features of the different classes. Also incremental understanding of the different classes becomes possible. In fact, inheritance can be interpreted in terms of the abstraction mechanism we discussed in Chapter 1. The class at the root of an inheritance hierarchy (e.g. LibraryMember in Figure 7.3) is the simplest to understand—as it has the least number of data and method members compared to all other classes in the hierarchy. The classes at the leaf-level of the inheritance hierarchy have the maximum number of features (data and method members) because they inherit features of all their ancestors, and therefore turn out to be the toughest to understand in isolation. However, it is possible to incrementally understand the features of various classes in the inheritance hierarchy, which enhances code readability.

When a design includes a large class hierarchy, it is easier to first understand the root class and then subsequently understand the lower level classes in the hierarchy, and finally understand the leaf level classes.

Multiple inheritance

While constructing the model of a problem, at times it may so happen that some features of a class are similar to one class and some other features of the class are similar to those of another class. In this case, it would be useful if the class could be allowed to inherit features from both the classes. Using the multiple inheritance feature, it becomes possible for a class to inherit features from multiple base classes.

Multiple inheritance is a mechanism by which a subclass can inherit attributes and methods from more than one base class.

Consider the following example of a class that is derived from two base classes through the use of the multiple inheritance mechanism. In an academic institute, research students are also employed as staff of the institute. To model this situation, the classes `Student`, `Research`, and `Staff` may be created. In this case, some of the characteristics of the `Research` class are similar to the `Student` class (e.g. every student would have a roll number) while some other characteristics (e.g. having a basic salary and employee number, etc.) are similar to the `Staff` class. It is easy to model this situation using multiple inheritance. Using multiple inheritance, the class `Research` can inherit features from both the classes `Student` and `Staff`. In Figure 7.4, we have shown the class `Research` to be derived from both the `Student` and `Staff` classes by drawing inheritance arrows to both the parent classes of the `Research` class.

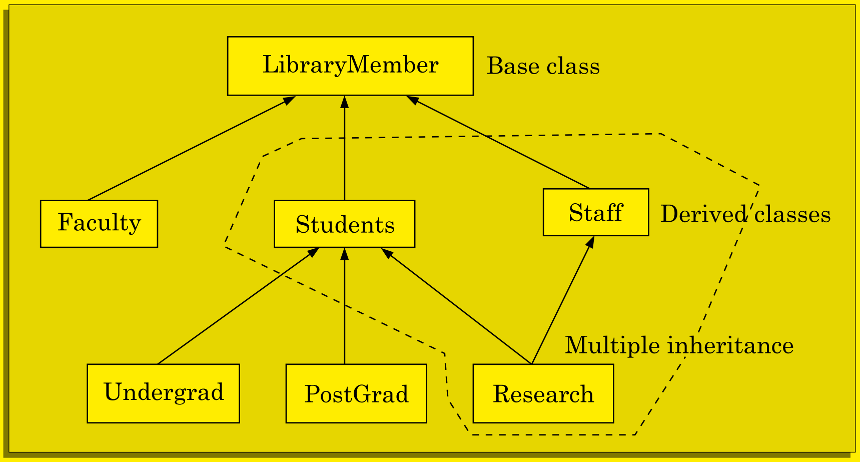


FIGURE 7.4 An example of multiple inheritance.

Association and link

Association is a common relation among classes and frequently occurs in design solutions. When two classes are associated, they can take each others' help (i.e., invoke each others' methods) to serve user requests. More technically, we can say that if one class is associated with another, then the corresponding objects of the two classes *know* each others' ids (also called object identities or object references). As a result, it becomes possible for the object of one class to invoke the methods of the corresponding object of the other class.

Consider the following example. `A Student can register in one Elective subject`. In this example, the class `Student` is associated with the class `ElectiveSubject`. Therefore, an `ElectiveSubject` object (consider for example, MachineLearning) would know the ids of all `Student` objects that have registered for that Subject and can invoke the methods of the registered students such as `printName`, `printRoll`, `enterGrade`, `etc`. This relationship has been represented in Figure 7.5(a). When an object knows some other object, it must internally store its id. For example, for an object e_1 of `ElectiveSubject` class to invoke the `printName()` method for one of the registered students s_1, it must execute the code s_1. `printName()`. Thus, the object e_1 should store the id s_1 of the registered student as an attribute. The association relationship can either be bidirectional or unidirectional. If the association is bidirectional, both the associated classes know each other (store each others' ids). We have graphically shown this association between `Student` class and `ElectiveSubject` in Figure 7.5(a).

Consider another example of a bidirectional association between two classes: `Library Member` *borrows* `Books`. Here, borrows is the association between the class `LibraryMember` and the class `Book`. The association relation would imply that given a book, it would be possible to determine the borrower and *vice versa*.

n-ary association

Binary association between classes is very commonly encountered in design problems. However, there are situations where three or more different classes are involved in an association. As an example of a ternary association, consider the following—`A person books a ticket for a certain show`. Here, an association exists among the classes `Person`, `Ticket`, and `Show`. This example of ternary association relationship has been represented in Figure 7.5(b).

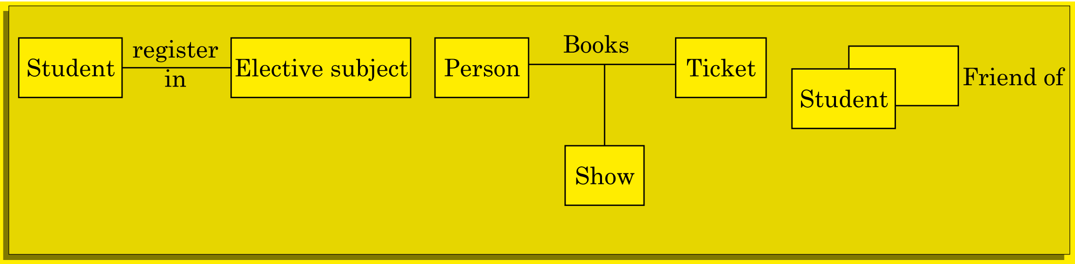


FIGURE 7.5 Example of (a) binary (b) ternary (c) unary association.

A class can have an association relationship with itself. This is called *recursive* association or *unary* association. As an example, consider the following—two students may be friends. Here, an association named friendship exists between pairs of objects of the `Student` class. This has been represented in Figure 7.5(c). It may be observed that even though unary association is defined on one class, the corresponding links exist between pairs of unique objects of the class.

In unary association, only a single class participates in the association relationship. However, different pairs of objects of the same class are linked by the association relationship.

When two classes are associated, the relationship between the objects of the corresponding classes is called a *link*. In the example of Figure 7.5(a), an object e_1 of the `ElectiveSubject` class can have links with the registered students such a: s_1, s_2, and s_3 after the three students register.

An association relationship between two classes represents a group of similar links to exist between pairs of objects belonging to the two classes. Alternatively, we can say that a *link* can be considered as an instance of an association relation.

Identify the association relation among classes and the corresponding association links among objects from an analysis of the following description. "`A person works for a company. Ram works for Infosys. Hari works for TCS.`"

In this example, an association relationship named *works for* exists between the classes `Person` and `Company`. However, the last two statements describe links between pairs of objects. `Ram` works for `Infosys`, this implies that a link exists between the object `Ram` and the object `Infosys`. Similarly, a *works for* link exists between the objects `Hari` and `TCS`.

During a run of the system, new links can get formed among the objects of the associated classes and some existing links may get dissolved. Note that some objects may not have any association link to any of the objects of the associated class. For example in course of time, Ram may resign from Infosys and join Wipro. In this case, the link between Ram and Infosys breaks and a link between Ram and Wipro gets formed. But, suppose Ram does not join any other job after leaving Infosys. In this case, Ram does not have *works for* link with any Company object, even though association relationship exists between the corresponding classes. We can observe that links are time varying (or dynamic) in nature. In contrast to the links among objects, association relationship between two classes is static in nature.

Existence of an association relation between two classes means that zero or more links may be present among the objects of the associated classes at any time during execution.

If two classes are associated, then the association relationship exists at all points of time. In contrast, links between objects are dynamic in nature. Links between the objects of the associated classes can get formed and dissolved as the program executes.

Mathematically, a link can be represented by a tuple. Consider the following example. "Amit has borrowed the book Graph Theory." Here, a link named *borrowed* gets established between the objects Amit and the Graph Theory book. This link can also be expressed as the ordered pair of object instances {Amit, Graph Theory}.

Composition and aggregation

Composition and aggregation represent part/whole relationships among objects. Objects which contain other objects are called *composite objects*. As an example, consider the following: `A Book can have upto ten Chapters`. In this case, a `Book` object is said to be an aggregate of one and upto ten `Chapter` objects. This class relationship has been shown in Figure 7.6. Observe the 1 written at the Book-end, and 1.10 shown at the Chapter-end. These are called association multiplicity and indicate that a single book object is an aggregate of anywhere from one to ten Chapter objects. The composition/aggregation relationship of Figure 7.6 can also be read as follows: A `Book` has at least one and up to ten `Chapter` objects. For this reason, the composition/aggregation relationship is also known as has a relationship. Aggregation/composition can occur in a hierarchy of levels. That is, an object contained in another object may itself contain some other object. Composition and aggregation relationships cannot be reflexive or symmetric. That is, an object cannot contain an object of the same type as itself.

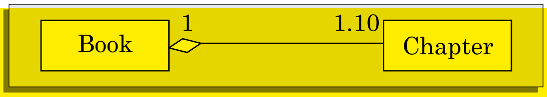


FIGURE 7.6 Example of aggregation relationship.

Dependency

A class is said to be *dependent* on another class, if any changes to the latter class necessitates a change to be made to the dependent class.

A dependency relation between two classes shows that any change made to the independent class would require the corresponding change to be made to the dependent class.

Dependencies among classes may arise due to various causes. Two important reasons for dependency to exist between two classes are the following:

- A method of a class takes an object of another class as an argument. Suppose, a method of an object of class C1 takes an object of class C2 as argument. In this case, class C1 is said to be dependent on class C2, as any change to class C2 would require a corresponding change to be made to class C1.

- A class implements an interface class (as in Java). In this case, dependency arises due to the following reason. If some properties of the interface class are changed, then a change becomes necessary to the class implementing the interface class as well.
- A class has an object of another class as its local variable.

Abstract class

Classes that are not intended to produce instances of themselves are called *abstract classes*. In other words, an abstract class cannot be instantiated into objects. If an abstract class cannot be instantiated to create objects, then what is the use of defining an abstract class? Abstract classes merely exist so that behaviour common to a variety of classes can be factored into one common location, where they can be defined once. Definition of an abstract class helps to push reusable code up in the class hierarchy, thereby enhancing code reuse.

By using abstract classes, code reuse can be enhanced and the effort required to develop software brought down.

Abstract classes usually support generic methods and even the method body for some of the methods may not have been defined. These methods for which only the method prototypes have been provided in the base class, help to standardize the method names and input and output parameters in the derived classes. The concrete subclasses of the abstract classes are expected to provide implementations for these methods. For example, in a Library Automation Software `Issuable` can be an abstract class (see Figure 7.7) and the concrete classes `Book`, `Journal`, and `CD` are derived from the abstract `Issuable` class. The `Issuable` class may define several generic methods such as `issue`. Since the issue procedures for books, journals, and CDs would be different, the `issue` method would have to be overridden in the `Book`, `Journal`, and `CD` classes. Though an abstract class with only method prototypes and no method body does not help in code reuse, but helps to have standardized prototype of the `issue` method across different concrete classes. Observe that `Issuable` is an abstract class and cannot be instantiated. On the other hand, `Book`, `Journal`, and `CD` are concrete classes and can be instantiated to create objects.

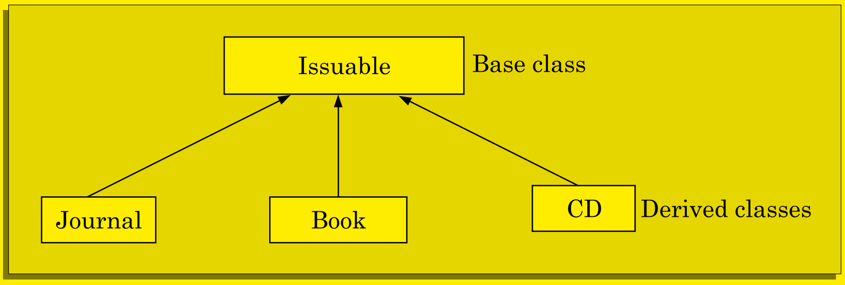


FIGURE 7.7 An example of an abstract class.

7.1.3 How to Identify Class Relationships?

Suppose we want to write object-oriented code for a simple programming problem. How do we identify what would be the classes and what would be their relationships from this description, so that we can write the necessary code? This can be done by a careful analysis of the sentences given in the problem description. The nouns in a sentence often denote the classes. On the other hand, the relationships among classes are usually indicated by the presence of certain key words. In the following, we give examples of a few key words (shown in italics) that indicate the existence of specific relationships among two classes A and B:

Composition

- B is a *permanent part of* A
- A is *made up of* Bs
- A is a *permanent collection of* Bs

Aggregation

- B is a *part of* A
- A *contains* B
- A is a *collection of* Bs

Inheritance

- A is a *kind of* B
- A is a *specialisation of* B
- A *behaves like* B

Association

- A *delegates* to B
- A *needs help* from B
- A *collaborates with* B. Here the phrase *collaborates with* can indicate any of a large variety of collaborations that are possible among classes such as employs, credits, precedes, succeeds, etc.

7.1.4 Other Key Concepts

We now discuss a few other key concepts used in the object-oriented program development approaches:

Abstraction

Let us first recapitulate how the abstraction mechanism works (we had already discussed this basic mechanism in Chapter 1). Abstraction is the selective consideration of certain aspects of a problem while ignoring all the remaining aspects of a problem. In other words, the main purpose of using the abstraction mechanism is to consider only those aspects of the problem that are relevant to a given purpose and to suppress all aspects of the problem that are not relevant.

The abstraction mechanism allows us to represent a problem in a simpler way by considering only those aspects that are relevant to some purpose and omitting all other details that are irrelevant.

Many different abstractions of the same problem can be constructed depending on the purpose for which the abstractions are made. The abstraction mechanism cannot only help the development engineers to understand and appreciate a problem better while working out a solution, but can also help better comprehension of a system design by the maintenance team. Abstraction is supported in two different ways in an *object-oriented designs* (OODs). These are the following:

Feature abstraction: A class hierarchy can be viewed as defining several levels (hierarchy) of abstraction, where each class is an abstraction of its subclasses. That is, every class is a simplified (abstract) representation of its derived classes and retains only those features that are common to all its children classes. Thus, the inheritance mechanism can be thought of as providing feature abstraction of its children classes.

Data abstraction: An object itself can be considered as a data abstraction entity, because it abstracts out the exact way in which it stores its private data items and it merely provides a set of methods to other objects to access and manipulate these data items. In other words, we can say that data abstraction implies that each object hides (abstracts away) from other objects the exact way in which it stores its internal information. This helps in developing good quality programs, as it causes objects to have low coupling with each other, since they do not directly access any data belonging to each other. Each object only provides a set of methods, which other objects can use for accessing and manipulating this private information of the object. For example, a stack object might store its internal data either in the form of an array of values or in the form of a linked list. Other objects would not know how exactly this object has stored its data (i.e. data is abstracted) and how it internally manipulates its data. What they would know is the set of methods such as push, pop, and top-of-stack that it provides to the other objects for accessing and manipulating the data.

An important advantage of the principle of data abstraction is that it reduces coupling among various objects. Therefore, it leads to a reduction of the overall complexity of a design, and helps in easy maintenance and code reuse.

Abstraction is a powerful mechanism for reducing the perceived complexity of software designs. Analysis of the data collected from several software development projects shows that software productivity is inversely proportional to the complexity of the developed software. Therefore, implicit use of abstraction, as it takes place in object-oriented development, is a promising way of increasing productivity of the software developers and also at the same time, bringing down maintenance costs.

Encapsulation

The data of an object is encapsulated within its methods. To access the data internal to an object, other objects have to invoke its methods, and cannot directly access the data. This concept is schematically shown in Figure 7.8. The figure shows that the data of an object are encapsulated within the methods (m_1, m_2, m_3, etc.). Observe from Figure 7.8 that there is no way for an object to access the data private to another object, other than by invoking its methods. Encapsulation offers the following three important advantages:

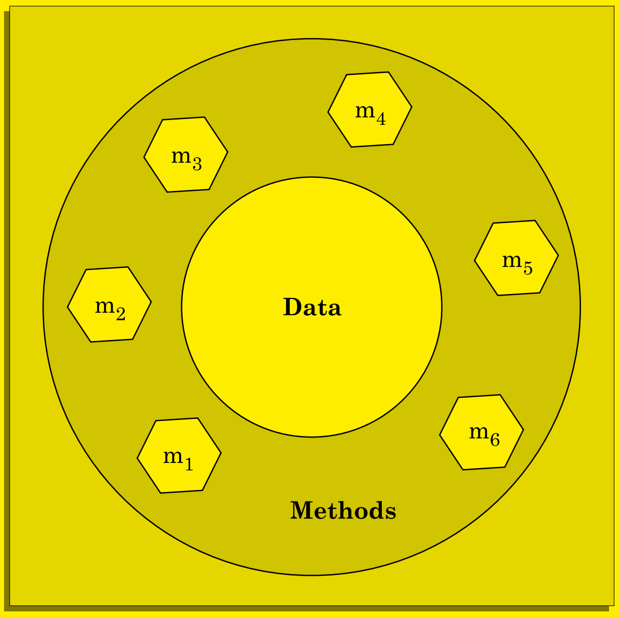


FIGURE 7.8 Schematic representation of the concept of encapsulation.

Protection from unauthorised data access: The encapsulation feature protects an object's variables from being accidentally corrupted by other objects. This protection includes protection from unauthorised access and also protection from the problems that arise from concurrent access to data such as deadlock and inconsistent values.

Data hiding: Encapsulation implies that the internal structure data of an object are hidden, so that all interactions of other objects with this object are standardized. This facilitates reuse of a class across different projects. Furthermore, if the internal data or the method body of a class are modified, other classes are not affected as long as the method prototype remains the same. This leads to easier maintenance and bug correction.

Weak coupling: Since objects do not directly change each other's internal data, they are weakly coupled. Weak coupling among objects enhances understandability of the design. This is so because each object can be studied and understood in isolation from other objects.

Polymorphism

Polymorphism literally means *poly* (many) *morphism* (forms). Recollect that in Chemistry, diamond, graphite, and coal are called *polymorphic* forms of carbon. The implication of this is that though diamond, coal, and graphite are essentially carbon, they behave very differently. In an analogous manner in the object-oriented paradigm, polymorphism denotes that an object may respond (behave) very differently even when the same operation is invoked, the specific method to which the call is bound depends on the exact polymorphic object to which the call gets made.

There are two main types of polymorphism in object-orientation:

Static polymorphism: Static polymorphism occurs when multiple methods in a class implement the same operation through multiple methods having the same method name but different parameter types. In this case, when the operation of the class is invoked, different behaviour (actions) would be observed depending which method implementing the operation gets executed. This type of polymorphism is also referred to as *static binding*, because the exact method to be bound on a method call is determined at compile-time (statically). Let us try to understand the concept of static binding through the following example. Suppose a class named `Circle` has three definitions for the create operation: `int create()`, `int create(int radius)`, and `int create(float x, float y, int radius)`. Recollect that when multiple methods of a class implement the same operation, the mechanism used is called *method overloading*. (Notice the difference between an operation and a method.) When the same operation (e.g. create) is implemented by multiple methods, the method name is said to be overloaded. One implementation of the create operation does not take any argument (`create()`) and creates a circle with default parameters. The second implementation takes the center point and the radius of the circle as its parameters (`create(float x, float y, float radius)`). Assume that in both the above method definitions, the fill style would be set to the default value "no fill". The third definition of the create operation takes the center point, the radius, and a fill style as its input. When the create method is invoked, depending on the parameters given in the invocation, the matching method can be easily determined during compilation by examining the parameter list of the call and the call would get statically bound. If `create` method is invoked with no parameters, then a default circle would be invoked. If only the center and radius are supplied, then an appropriate circle would be invoked with no fill type, and so on. The skeletal definition of the `Circle` class with the overloaded `create` method is shown in Figure 7.9.

Dynamic polymorphism: Dynamic polymorphism is also called dynamic binding. In dynamic binding, the exact method that would be invoked (bound) on a method call is determined at the run time (dynamically) and cannot be determined at compile time. That is, the exact behavior that would be produced on a method call cannot be predicted at compile time and can only be observed at run time, depending on how the objects get created in a program run and the specific object in the context of which the method call is made.

Let us now explain how dynamic binding works in object-oriented programs. Dynamic binding is based on two important concepts:

- Assignment of an object to a compatible object (explained below).
- Method overridissng in a class hierarchy.

```
class Circle{
        private float x, y, radius;
        private int fillType;

        public create();
        public create(float x, float y, float radius);
        public create(float x, float y, float radius, int fillType);
    }
```

FIGURE 7.9 Circle class with overloaded create method.

Assignment to compatible of objects

In object-orientation, objects of the derived classes are compatible with the objects of the base class. That is, an object of the derived class can be assigned to an object of the base class, but not *vice versa*. Also an object cannot be assigned to an object of a sibling class or an object of an unrelated class for obvious reasons. This is an important principle in object-orientation and is known as the *Liskov Substitution principle*. To understand this principle, recollect that a derived class usually defines a few additional attributes. Assignment of an object of the base class to an object of the derived class can leave those extra attributes of the derived class undefined.

Method overriding

We have already explained the method overriding principle in which a derived class provides a new definition to a method of the base class. In method overriding, a method of the derived class has the same signature (parameter list) as a method of the base class, but provides a new definition to the method. Method overriding plays an important role in dynamic binding.

Let us now understand through an example how dynamic binding works by making use of the above two mechanisms. Suppose we have defined a class hierarchy of different geometric shapes for a graphical drawing package as shown in Figure 7.10. As can be seen from the figure, `Shape` is an abstract class and the classes `Circle`, `Rectangle`, and `Line` are directly derived from it. Further, in the next level of the inheritance hierarchy, the classes `Ellipse`, `Square` and `Cube` have been derived. Now, suppose the `draw` method is declared in the `Shape` class and is overridden in each derived class. Further, suppose a set of different types of `Shape` objects have been created one by one. By Liskov's substitution principle, the created `Shape` objects can be stored in an array of type `Shape`. If the different types of geometric objects making up a drawing are stored in an array of type `Shape`, then a call to the `draw` method for each object would take care to display the appropriate object. That is, the same `draw` call to a `Shape` object would take care of displaying the appropriate object. Observe that due to dynamic binding, a call to the `draw` method of the shape class takes care of displaying the appropriate drawing object residing in the shape array. This is illustrated in the code segment shown in Figure 7.11.

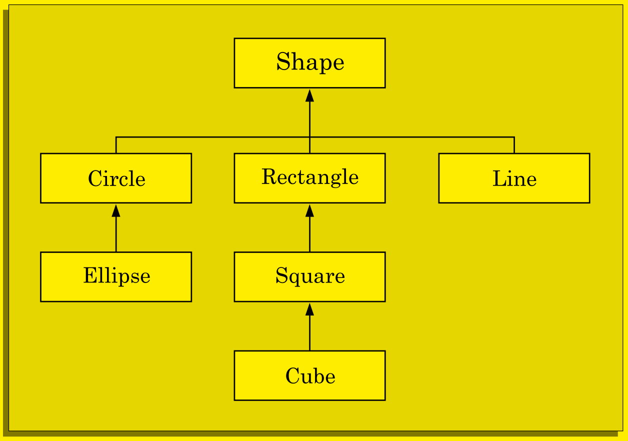


FIGURE 7.10 Class hierarchy of geometric objects.

We analyse the advantage of polymorphism by comparing the code segments of an object-oriented program and a traditional program for drawing various graphic objects on the screen (shown in Figure 7.11). Using dynamic binding, a programmer can invoke a generic method of an object and leave the exact way in which this method call would be handled would be decided depending on the object that is currently assigned to the object on which the method call is invoked. With dynamic binding, new derived objects can be added with minimal changes to a program.

It can be easily seen from Figure 7.11 that when dynamic binding, the object-oriented code is much more concise, understandable, and intellectually appealing as compared to equivalent procedural code. Further, judicious use of dynamic binding can help in maintenance. Suppose in the example program segment, it is later found necessary to support a new graphics drawing primitive, say ellipse. Then, the procedural code has to be changed by adding a new `if-then-else` clause. However, in case of an object-oriented program, the code need not change, only a new class called `Ellipse` has to be derived in the `Shape` hierarchy.

The principal advantage of dynamic binding is that it leads to elegant programming and facilitates code reuse and maintenance.

Traditional code	Object–oriented code

```
Shape s[100];

for(i=0;i<100;i++){
    if(s[i]==Circle) then
            draw_circle();
    else if(shape==Rectangle) then
            draw_rectangle();
        —
        —
        —
}
```
	```
	Shape s[100];

	for(i=0;i<100;i++)
	        shape.draw();
	```

FIGURE 7.11 Traditional code versus object-oriented code incorporating the dynamic binding feature.

We can now summarise the mechanism of dynamic binding as follows:

Even when the method of an object of the base class is invoked, an appropriate overridden method of a derived class would be invoked depending on the exact object on which the method invocation occurs.

Genericity

Genericity is the ability to parameterise class definitions. For example, while defining a class named stack, we may notice that we need stacks of different types of elements such as integer stack, character stack, floating-point stack, etc.; *genericity* permits us to effortlessly define a generic class of type stack and later instantiate it either as an integer stack, a character stack, or a floating-point stack as may be required. This can be achieved by assigning a suitable value to a parameter used in the generic class definition.

7.1.5 Related Technical Terms

In the following, we discuss a few terms related to object-orientation:

Persistence

All objects that get created during a run of a program usually get destroyed once the program completes its execution. Persistent objects are stored permanently. That is, they live across different executions. An object is usually made persistent by maintaining copies of the object in a secondary storage or in a database, and loading it during a run of the program.

Agents

A passive object is one that performs some action only when requested through invocation of some of its methods. An agent (also called an *active object*), on the other hand, monitors events such as key board press and takes actions autonomously. An example of application of an agent is monitoring inconsistencies occurring in a database transaction. For example, in a database application such as accounting, an agent may monitor the balance sheet and would alert the user whenever inconsistencies arise in a balance sheet due to some improper transaction taking place.

Widgets

The term widget stands for *window object*. A widget is a primitive object used for *graphical user interface* (GUI) design. More complex graphical user interface design primitives (widgets) can be derived from the basic widgets using the inheritance mechanism. A widget maintains internal data such as the geometry of the window, back ground and fore ground colors of the window, cursor shape and size, etc. The methods supported by a widget help to manipulate the stored data and carry out operations such as resize window, iconify window, destroy window, etc. Widgets are becoming the standard components of GUI design. This has given rise to the technique of component-based user interface development. We shall discuss more about widgets and component-based user interface development in Chapter 9 where we discuss GUI design.

7.1.6 Advantages and Disadvantages of OOD

As is true for any other technique, object-oriented development (OOD) has its own advantages and disadvantages. We briefly review these in the following.

Advantages of OOD

In the last couple of decades since OOD has come into existence, it has found widespread acceptance in industry as well as in academic circles. The main reason for the popularity of OOD is that it holds out the following promises:

- Code and design reuse
- Increased productivity
- Ease of testing and maintenance
- Better code and design understandability, which are especially important to the development of large programs

Out of all the above mentioned advantages, it is usually agreed that the chief advantage of OOD is *improved productivity*—which comes about due to a variety of factors, including the following:

- Code reuse is facilitated through easy use of predeveloped class libraries
- Code reuse due to inheritance
- Simpler and more intuitive abstraction, which support, better management of inherent problem and code complexities
- Better problem decomposition

Several research results indicate that when companies start to develop software using the object-oriented paradigm, the first few projects incur higher costs than the traditionally developed projects. This is possibly due to the initial overhead of getting used to a new technique and building up the class libraries that can be reused in the subsequent projects. After completion of a few projects, cost saving becomes possible. According to experience reports, a well-established object-oriented development environment can help to reduce development costs by as much as 20–50 per cent over a traditional development environment.

Disadvantages of OOD

The following are some of the prominent disadvantages inherent to the object paradigm:

- The principles of abstraction, data hiding, inheritance, etc. do incur run time overhead due to the additional code that gets generated on account of these features. This causes an object-oriented program to run a little slower than an equivalent procedural program.
- An important consequence of object-orientation is that the data that is centralised in a procedural implementation, gets scattered across various objects in an object-oriented implementation. For example, in a procedural program, the set of books may be implemented in the form of an array. In this case, the data pertaining to the books are stored at consecutive memory locations. On the other hand, in an object-oriented program, the data for a collection of book objects may not get stored consecutively. Therefore, the spatial locality of data becomes weak and this leads to higher cache miss ratios and consequently to larger memory access times. This finally shows up as increased program run time.

As we can see, increased run time is the principal disadvantage of object-orientation and higher productivity is the major advantage. In the present times, computers have become remarkably fast, and a small run time overhead is not any more considered to be a detractor. Consequently, we can say that the advantages of OOD overshadow its disadvantages. However, for development of small embedded applications, the procedural style of development may be advantageous.

7.2 UNIFIED MODELLING LANGUAGE (UML)

As the name itself implies, UML is a language for documenting models. As is the case with any other language, UML has its syntax (a set of basic symbols and sentence formation rules) and semantics (meanings of basic symbols and sentences). It provides a set of basic graphical notations (e.g. rectangles, lines, ellipses, etc.) that can be combined in certain ways to document the design and analysis results.

It is important to remember that UML is not a system design or development methodology by itself, neither is it designed to be tied to any specific methodology. UML is merely a language for documenting models. Before the advent of UML, every design methodology that existed, not only prescribed its unique set of design steps, but each was tied to some specific design modelling language. For example, OMT methodology had its own design methodology and had its own unique set of notations. So was the case with Booch's methodology, and so on. This situation made it hard for someone familiar with one methodology to understand and reuse the design solutions available from a project that used a different methodology. In general, reuse of design solutions across different methodologies was hard. UML was intended to address this problem that was inherent to the modelling techniques that existed.

UML can be used to document object-oriented analysis and design results that have been obtained using any methodology.

One of the objectives of the developers of UML was to keep the notations of UML independent of any specific design methodology, so that it can be used along with any specific design methodology. In this respect, UML is different from its predecessors.

7.2.1 Origin of UML

In the late eighties and early nineties, there was a proliferation of object-oriented design techniques and notations. Many of these had become extremely popular and were widely used. However, the notations they used and the specific design paradigms that they advocated, differed from each other in major ways. With so many popular techniques to choose from, it was not very uncommon to find different project teams in the same organisation using different methodologies and documenting their object-oriented analysis and design results using different notations. These diverse notations used for documenting design solutions gave rise to a lot of confusion among the team members and made it extremely difficult to reuse designs across projects and also for communicating ideas across project teams.

UML was developed to standardise the large number of object-oriented modelling notations that existed in the early nineties. The principal ones in use those days include the following:

- OMT [Rumbaugh, 1991]
- Booch's methodology [Booch, 1991]
- OOSE [Jacobson, 1992]
- Odell's methodology [Odell, 1992]
- Shlaer and Mellor methodology [Shlaer, 1992]

Needless to say that UML has borrowed many concepts from these modeling techniques. Concepts and notations from especially the first three methodologies have heavily been drawn upon. The relative degrees of influence of various object modeling techniques on UML is shown schematically in Figure 7.12. As shown in Figure 7.12, OMT had the most profound influence on UML.

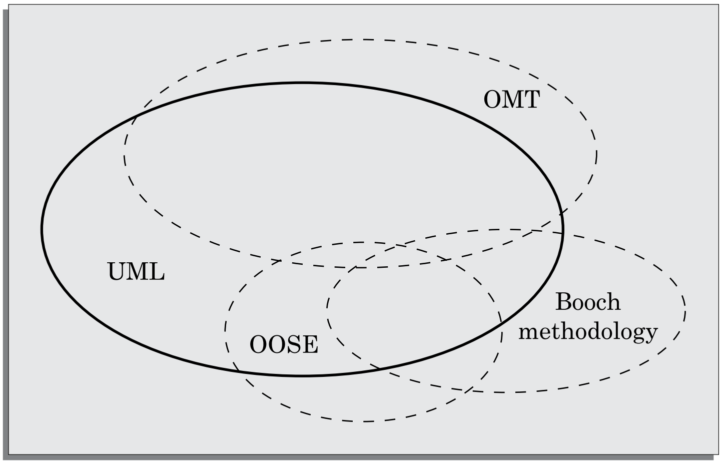


FIGURE 7.12 Schematic representation of the impact of different object modelling techniques on UML.

UML was developed by *object management group* (OMG) in 1997 and it has emerged as a *de facto* object modeling standard. Actually, OMG is not a standards formulating body, but it is an association of a large number of industries. It tries to facilitate early formulation of standards. OMG aims to promote consensus notations and techniques with the hope that if the usage becomes wide-spread, then these would automatically become standards. For more information on OMG, see *www.omg.org*. With widespread use of UML, ISO adopted UML a standard (ISO 19805) in 2005, and with this UML has become an official standard; this has further enhanced the use of UML.

UML is more complex than its antecedents. This is only natural and expected because it is intended to be more comprehensive and applicable to a wider gamut of problems and design methodologies than any of the modeling techniques that existed before UML. UML contains an extensive set of notations to help document several aspects (views) of a design solution through many types of diagrams. UML has successfully been used to model both large and small problems. The elegance of UML, its adoption by OMG, and subsequently by ISO as well as a strong industry backing have helped UML to find wide spread acceptance. UML is now being used in academic and research institutions as well as in large number of software development projects world-wide. It is interesting to note that the use of UML is not restricted to the software industry alone. As an example of UML's use outside the software development problems, some car manufacturers are planning to use UML for their "build-to-order" initiative, where the users would express their options using UML notations.

Many of the UML notations are difficult to draw by hand on a paper and are best drawn using a CASE tool such as Rational Rose (see *www.rational.com*) ArgoUML (argouml.

stage.tigris.org) or MagicDraw (*www.magicdraw.com*). Now several free UML CASE tools are available on the web. Most of the available CASE tools help to refine an initial object model to final design, and many of these also automatically generate code templates in a variety of languages such as Java, C++, and C, once the UML models have been constructed.

7.2.2 Evolution of UML

Since the release of UML 1.0 in 1997, UML as continued to evolve (see Figure 7.13) with feedback from practitioners and academicians to make it applicable to different system development situations. Almost every year several new releases (shown as UML 1.X in Figure 7.13) are being announced. A major milestone in the evolution of UML was the release of UML 2.0 in the year 2007. Since the use of embedded applications is increasing rapidly, there was popular demand to extend UML to support the special concepts and notations required to develop embedded applications. UML 2.0 was an attempt to make UML applicable to the development of concurrent and embedded systems. For this, many new features such as events, ports, and frames were introduced. In addition to the features supported in UML 1.0, we briefly discuss these developments in this chapter.

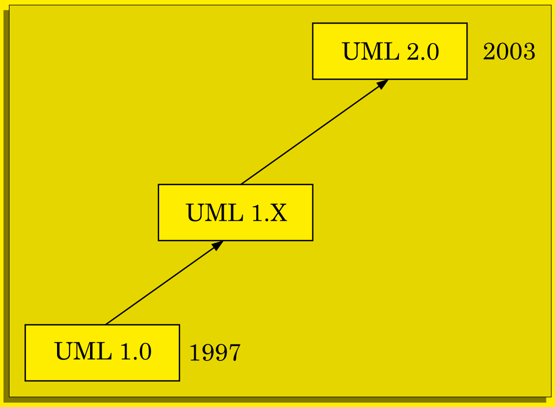


FIGURE 7.13 Evolution of UML.

What is a model?

Before we discuss the features of UML in detail, it is important to understand what exactly is meant by a model, and why is it necessary to construct a model during software development.

A *model* is an abstraction of a real problem (or situation), and is constructed by leaving out unnecessary details. This reduces the problem complexity and makes it easy to understand the problem (or situation).

A model is a simplified version of a real system. It is useful to think of a model as capturing aspects important for some application while omitting (or abstracting out) the rest. As we had already pointed out in Chapter 1, as the size of a problem increases, the perceived complexity of the problem increases exponentially due to human cognitive limitations. Therefore, to develop a good understanding of any non-trivial problem, it is necessary to construct a model of the problem. Modelling has turned out to be a very essential tool in software design and helps the developer to effectively handle the complexity in a problem. In almost every design methodology, the models of the problem are first constructed. These are called the *analysis models*. A design methodology essentially proceeds thereon to transforms these analysis models into a design model through iterative refinements.

Different types of models are obtained depending on the specific aspects of the actual system that are ignored while constructing the model. To understand this, let us consider the models constructed by an architect of a large building. While constructing the frontal view of a large building (elevation plan), the architect ignores aspects such as floor plan, strength of the walls, details of the inside architecture, etc. While constructing the floor plan, he completely ignores the frontal view (elevation plan), site plan, thermal and lighting characteristics, etc. of the building.

A model in the context of software development can be graphical, textual, mathematical, or program code-based. Graphical models are very popular among software developers because these are easy to understand and construct. UML is primarily a graphical modeling language. However, there are certain types of UML models (discussed later in this Chapter), for which separate textual explanations are required to accompany the graphical models.

Why construct a model?

An important reason behind constructing a model is that it helps to manage the complexity in a problem and facilitates arriving at good solutions and at the same time helps to reduce the design costs. The initial model of a problem is called an *analysis model*. The analysis model of a problem can be refined into a design model using a design methodology. Once the required models of a system have been constructed, these can be used for a variety of purposes during software development, including the following:

- Analysis
- Specification
- Design
- Coding
- Visualisation and understanding of the implementation
- Testing, etc.

Since a model can be used for a variety of purposes, it is reasonable to expect that the models would vary with respect to the details they capture and this would depend on the purpose for which these are being constructed. For example, a model developed for initial analysis and specification should be different from the one used for design. A model that is constructed for analysis and specification would not show any of the design decisions that would be made later on during the design stage. On the other hand, a model constructed for design purposes should capture all the design decisions. Therefore, it is a good idea to explicitly mention the purpose for which a model has been developed.

We now discuss the different types of UML diagrams and the notations used in constructing these diagrams.

7.3 UML DIAGRAMS

In this section, we discuss the diagrams supported by UML 1.0. Later in Section 7.9.2, we discuss the changes to UML 1.0 brought about by UML 2.0. UML 1.0 can be used to construct nine different types of diagrams to capture five different views of a system. Just as a building can be modelled from several views (or perspectives) such as ventilation perspective, electrical perspective, lighting perspective, heating perspective, etc.; the different UML diagrams provide different perspectives of a software system to be developed. A representation of the different views of a system facilitates a comprehensive understanding of the system. Each perspective focuses on some specific aspect and ignores the rest. Some may ask, why construct several models from different perspectives—why not just construct one model that captures all perspectives? The answer to this is the following:

If a single model is constructed to capture all the required perspectives of the problem, then it would be as complex as the original problem, and would be of little use.

Once a system has been modelled from all the required perspectives, the constructed models can be refined into a design model that can help to get the actual implementation of the system.

UML diagrams help to capture the following five views (models) of a system:

- User's view
- Structural view
- Behaviourial view
- Implementation view
- Environmental view

Figure 7.14 shows the different views that the UML diagrams can help document. Observe that the users' view is shown as the central view. This is because based on the users' view, all other views are developed and also all other views need to conform to the user's view. Most of the object oriented analysis and design methodologies, including the one we discuss in Chapter 8 require iteration among the different views a number of times to arrive at the final design. We first provide a brief overview of the different views of a system which can be documented using UML. In the subsequent sections, the diagrams that are used to construct these views have been discussed.

Users' view

This view defines the functionalities that would be made available by a system to its users.

The users' view captures the functionalities offered by the system to its users.

The users' view is a black-box view of the system where the internal structure, the dynamic behaviour of different system components, the implementation etc. are ignored. The users' view is very different from all other views in UML in the sense that it is a *functional model*[1] compared to all other views that are essentially *object models*.[2]

[1] A functional model captures the functions supported by the system.

[2] An object model captures the objects in the system and their interrelations.

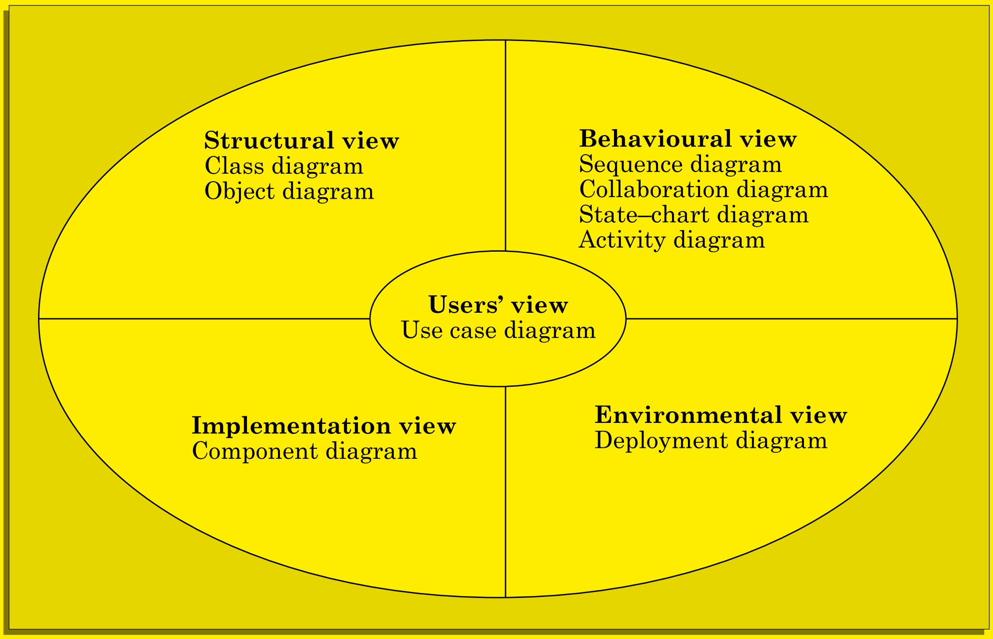


FIGURE 7.14 Different types of diagrams and views supported in UML.

As already mentioned, it is generally accepted that the users' view is considered as the central view and all other views are required to conform to this view. This thinking is in fact the crux of any *user centric development* style. It is indeed remarkable that even for object-oriented development, we need a functional view. We can justify this apparent anomaly by noting that after all, a user considers a system as providing a set of functionalities.

Structural view

The structural view defines the structure of the problem (or the solution) in terms of the kinds of objects (classes) important to the understanding of the working of a system and to its implementation. It also captures the relationships among the classes (objects).

The structural model is also called a *static model*, since the structure of a system does not change with time.

Behavioural view

The behavioural view captures how objects interact with each other in time to realise the system behaviour. The system behaviour captures the time-dependent (dynamic) behaviour of the system. It, therefore, constitutes the dynamic model of the system.

Implementation view

This view captures the important components of the system and their interdependencies. For example, the implementation view might show the GUI part, the middleware, and the database part as the different components and also would capture their interdependencies.

Environmental view

This view models how the different components are implemented on different pieces of hardware.

For any given problem, should one construct all the views using all the diagrams provided by UML? The answer is No. For a simple system, the use case model, class diagram, and one of the interaction diagrams may be sufficient. For a system in which the objects undergo many state changes, a state chart diagram may be necessary. For a system, which is implemented on a large number of hardware components, a deployment diagram may be necessary. So, the type of models to be constructed depends on the problem at hand. Rosenberg provides an analogy [Ros, 2000] saying that "Just like you do not use all the words listed in the dictionary while writing a prose, you normally do not use all the UML diagrams and modeling elements while modeling a system."

7.4 USE CASE MODEL

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

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

A simple way to find all the use cases of a system is to ask the question: "What all can the different categories of users achieve by using the system?" When we pose this question for the *library information system* (LIS), the use cases could be identified to be:

- issue-book
- query-book
- return-book
- create-member
- add-book, etc.

Roughly speaking, the use cases correspond to the high-level functional requirements that we 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 (or scenarios of execution). We elaborate this idea in the following:

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*) usually exist. Typically, a variation from the mainline sequence occurs when some specific conditions hold. For the bank ATM example, consider the following variations or alternate sequences:

- Password is invalid. In this case the normal next step of cash withdrawal does not occur, and the user is asked to reenter the password.
- The amount to be withdrawn exceeds the account balance. In this case, the normal next step of cash withdrawal does not occur, and the use case terminates.

The mainline sequence and each of the alternate sequences of a use case are each 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 corresponds to a single sequence 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 that may exist among use cases at the implementation level due to factors such as 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 an earlier 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 the 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. We wish to restate for emphasis that the "use case model" is not really an object-oriented model but a functional model of the system.

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

7.4.1 Representation of Use Cases

A use case model can be documented by drawing a use case diagram and writing an accompanying text to elaborate the diagram. In the use case diagram, each use case is represented by an ellipse with the name of the use case written inside the ellipse. All the ellipses (i.e., use cases) of a system are enclosed within a rectangle which represents the system boundary. The name of the system being modeled (e.g., *library information system*) appears inside the rectangle (see Figure 7.15).

The different users of the system are represented by using *stick person* icons. Each stick person icon is referred to as an actor.[3] An actor is actually a role played by a user.

[3] A more appropriate name for the stick person icon could have been 'role' rather than 'actor'. It appears that the apparent anomaly in term used in referring to the stick person icon was caused by a wrong translation from the original Swedish document of Jacobson.

It is possible that the same user may play multiple roles (actors). For example, a library may create a book in the role of the librarian and issue a book in the role of a library member. An actor can participate in one or more use cases. The line connecting an actor and the use case is called the *communication relationship*. It indicates that an actor makes use of the functionality provided by the use case.

Both human users and external systems can be represented by stick person icons. When a stick person icon represents an external system, it is annotated by the stereotype «external system».

At this point, it is necessary to explain the concept of a stereotype in UML. One of the main objectives of the creators of the UML was to restrict the number of primitive symbols in the language. It was clear to them that when a language has a large number of primitive symbols, it becomes very difficult to learn use. To convince yourself, consider that English with 26 alphabets is much easier to learn and use compared to the Chinese language that has thousands of symbols. In this context, the primary objective of stereotype is to reduce the number of different types of symbols that one needs to learn. By using stereotypes, the number of basic symbols of the language is reduced significantly.

The *stereotype* construct when used to annotate a basic symbol, can give slightly different meaning to the basic symbol—thereby eliminating the need to have several symbols whose meanings differ slightly from each other.

Just as you stereotype your friends as studious, jovial, serious, etc. stereotyping can be used to give special meaning to any basic UML construct. We shall, later on, see how other UML constructs can be stereotyped. We can stereotype the stick person icon symbol with the stereotype <<external>> to denote an external system. If the developers of UML had assigned a separate symbol to denote things such as an external system, then the number of basic symbols one would have to learn and remember for using UML would have increased significantly. This would have certainly made learning and using UML much more difficult.

You can draw a rectangle around the use cases, called the *system boundary box*, to indicate the scope of your system. Anything within the box represents functionality that is in scope and anything outside the box is not. However, drawing the system boundary is optional.

We now give a few examples to illustrate how use cases of a system can be documented.

EXAMPLE 7.2 The use case model for the Tic-tac-toe game software is shown in Figure 7.15. This software has only one use case, namely, "play move". Note that we did not name the use case "get-user-move", which would be inappropriate because this would represent the developer's perspective of the use case. The use cases should be named from the users' perspective.

Text description

Each ellipse in a use case diagram, by itself conveys very little information, other than giving a hazy idea about the use case. Therefore, every use case diagram should be accompanied by a text description. The text description should define the details of the interaction between the user and the computer as well as other relevant aspects of the

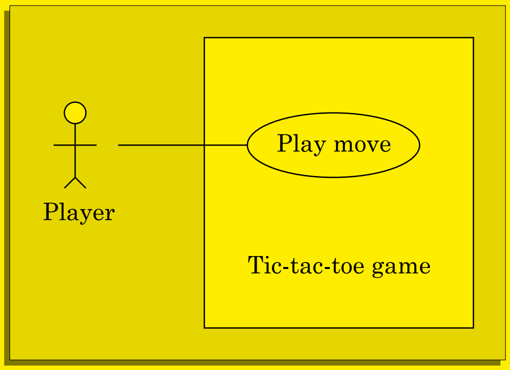


FIGURE 7.15 Use case model for Example 7.2.

use case. It should include all the behaviour associated with the use case in terms of the mainline sequence, various alternate sequences, the system responses associated with the use case, the exceptional conditions that may occur in the behaviour, etc. The behaviour description is often written in a conversational style describing the interactions between the actor and the system. The text description may be informal, but some structuring is helpful. The following are some of the information which may be included in a use case text description in addition to the mainline sequence, and the alternate scenarios.

Contact persons: This section lists of personnel of the client organisation with whom the use case was discussed, date and time of the meeting, etc.

Actors: In addition to identifying the actors, any information about the actors of a use case may be recorded, if the information may in some way be useful during implementation of the use case.

Pre-condition: The preconditions would describe the state of the system before the use case execution starts.

Post-condition: This captures the state of the system after the use case has successfully completed.

Non-functional requirements: This could contain the important constraints for the design and implementation, such as platform and environment conditions, qualitative statements, response time requirements, etc.

Exceptions, error situations: This contains response to an error situation, such as an invalid value entered in a field.

Sample dialogs: These serve as examples illustrating the use case.

Specific user interface requirements: These contain specific requirements for the user interface of the use case. For example, it may contain forms to be used, screen shots, interaction style, etc.

Document references: This part contains references to specific domain-related documents which may be useful to understand the system operation.

EXAMPLE 7.3 The use case diagram of the Super market prize scheme described in Example is shown in Figure 7.16.

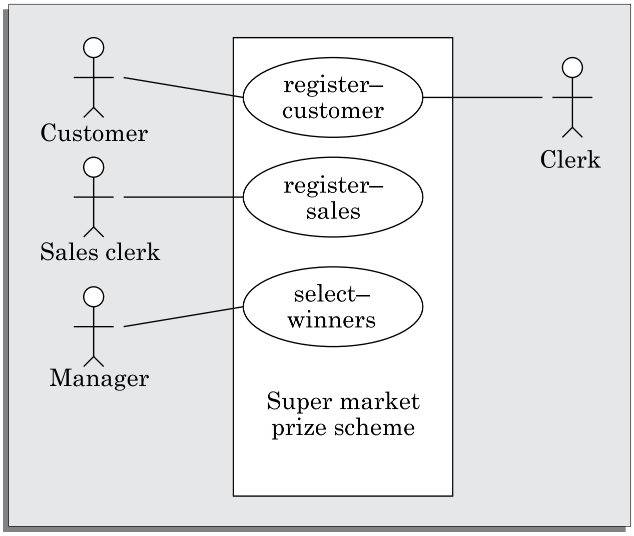


FIGURE 7.16 Use case model for Example 7.3.

Text description

U1: register-customer: Using this use case, the customer can register himself by providing the necessary details.

Scenario 1: Mainline sequence

```
1. Customer: select register customer option
2. System: display prompt to enter name, address, and telephone number.
3. Customer: enter the necessary values
4. System: display the generated id and the message that the customer
   has successfully been registered.
```

Scenario 2: At step 4 of mainline sequence

```
4: System: displays the message that the customer has already
   registered.
```

Scenario 3: At step 4 of mainline sequence

```
4: System: displays message that some input information have
   not been entered. The system displays a prompt to enter
   the missing values.
```

U2: register-sales: Using this use case, the clerk can register the details of the purchase made by a customer.

Scenario 1: Mainline sequence

```
1. Clerk: selects the register sales option.
2. System: displays prompt to enter the purchase details and the id
   of the customer.
3. Clerk: enters the required details.
```

```
    4: System: displays a message of having successfully registered the
       sale.
```

U3: select-winners: Using this use case, the manager can generate the winner list.

Scenario 2: Mainline sequence

```
    1. Manager: selects the select-winner option.
    2. System: displays the gold coin and the surprise gift winner list.
```

7.4.2 Why Develop the Use Case Diagram?

The utility of the use cases represented by the ellipses is obvious. These provide a succinct representation of the functional requirements. Also these along with the accompanying text description serve as a type of requirements specification of the system. Based on the use case model, all other models are developed. In other words, the use case model forms the core model to which all other models must conform. But, what about the actors (stick person icons)? What way are they useful to system development? One possible use of identifying the different types of users (actors) is in implementing a security mechanism through a login system, so that each actor can invoke only those functionalities to which he is entitled to. Another important use is in designing the user interface in the implementation of the use case targeted for each specific category of users who would use the use case. Another possible use is in preparing the documentation (e.g., users' manual) targeted at each category of user. Further, actors help in identifying the use cases and understanding the exact functioning of the system.

7.4.3 How to Identify the Use Cases of a System?

Identification of the use cases involves brain storming and reviewing the SRS document. Typically, the high-level requirements specified in the SRS document correspond to the use cases. In the absence of a well-formulated SRS document, a popular method of identifying the use cases is actor-based. This involves first identifying the different types of actors and their usage of the system. Subsequently, for each actor the different functions that they might initiate or participate are identified. For example, for a Library Automation System, the categories of users can be members, librarian, and the accountant. Each user typically focuses on a set of functionalities. For example, the member typically concerns himself with book issue, return, and renewal aspects. The librarian concerns himself with creation and deletion of the member and book records. The accountant concerns itself with the amount collected from membership fees and the expenses aspects.

7.4.4 Essential Use Case *versus* Real Use Case

Essential use cases are created during early requirements elicitation. These are also early problem analysis artifacts. These are independent of the design decisions and tend to be correct over long periods of time.

Real use cases describe the functionality of the system in terms of the actual design targeted for specific input/output technologies. Therefore, the real use cases can be developed only after the design decisions have been made. Real use cases are a design artifact. However, sometimes organisations commit to development contracts that include the detailed user interface specifications. In such cases, there is no distinction between the essential use case and the real use case.

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 in the following two situations:

- **Decompose complex use cases:** 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.
- **Identify commonality:** 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 the later deign of class diagrams 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—it would only complicate the design. UML offers three factoring mechanisms as discussed further.

Generalisation

Use case generalisation can be used when one use case is quite similar to another, but does something slightly different or something more. That is one use case is a special case of another use case. In this sense, generalisation works the same way with use cases as it does with classes. The child use case inherits the behavior of the parent use case. The notation is the same too (See Figure 7.17). It is important to remember that the base and the derived use cases are separate use cases and should have separate text descriptions.

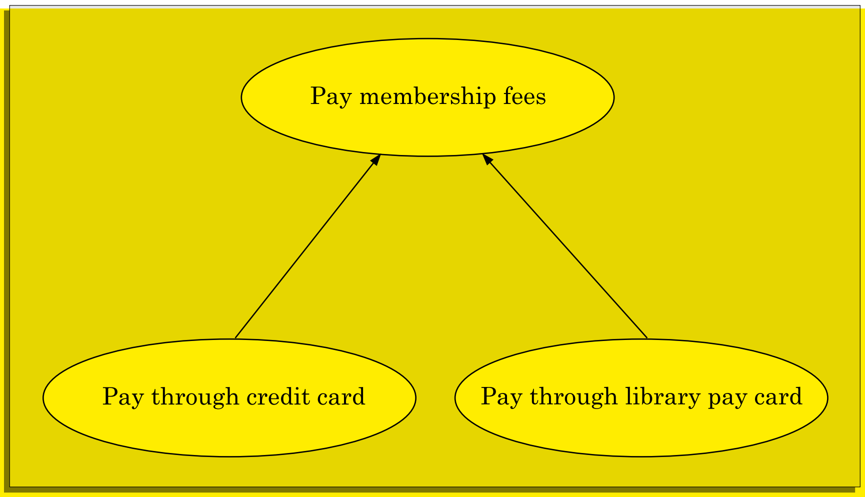


FIGURE 7.17 Representation of use case generalisation.

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 7.18, the includes relationship is represented using a predefined 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 7.19, the use cases `issue-book` and `renew-book` both include `check-reservation` use case. The implication is that during execution of either of the use cases `issue-book` and `renew-book`, the use case `check-reservation` is executed. The base use case may include several use cases. In such cases, it may interleave associated common use cases. The common use case becomes a separate use case and independent text description should be provided for it.

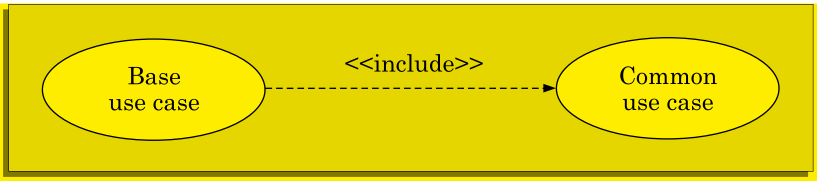


FIGURE 7.18 Representation of use case inclusion.

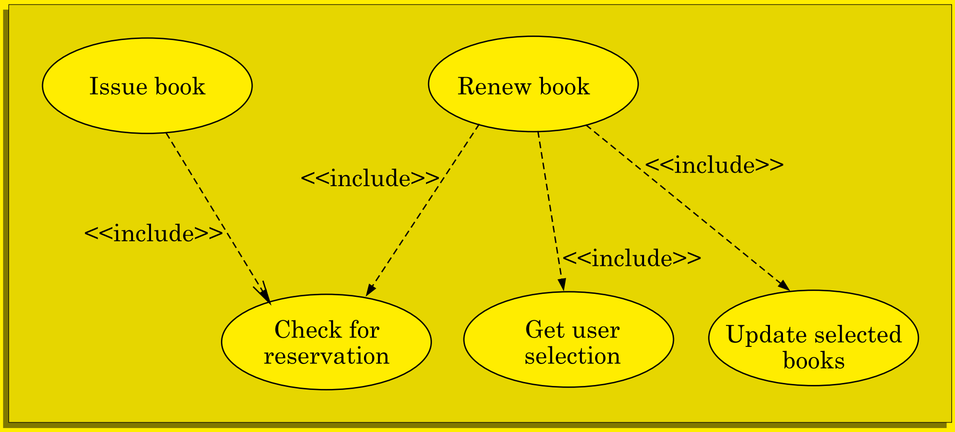


FIGURE 7.19 Example of use case inclusion.

Extends

The main idea behind the *extends* relationship among use cases is that it allows one to show optional behaviour that may occur during the execution of the use case. An optional system behaviour is executed only if certain conditions hold, otherwise the optional behaviour is not executed. This relationship among use cases is also predefined as a stereotype as shown in Figure 7.20.

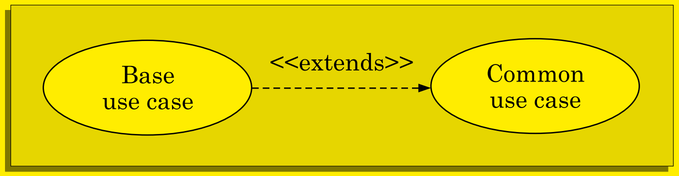


FIGURE 7.20 Example of use case extension.

The *extends* relationship is similar to generalisation. But unlike generalisation, 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.

Organisation

When the use cases are factored, they get organised hierarchically. The high-level use cases are factored into a set of smaller use cases as shown in Figure 7.21. Observe that one use case at a certain level, can get decomposed into a set of use cases at a lower level. Top-level use cases are super-ordinate to the factored use cases. Note that only the complex use cases should be decomposed and organized in a hierarchy. It is not necessary to decompose the simple use cases.

The functionality of a super-ordinate use case is traceable to its subordinate use cases. Thus, the functionality provided by the super-ordinate use cases is composite of the functionality of the sub-ordinate use cases.

At the highest level of the use case model, only the fundamental use cases are shown. The focus at the highest level is on the application context. Therefore, this level is also referred to as the *context diagram.* In the context diagram, the system limits are identified and only those use cases with which external users interact are shown. The topmost use cases specify the complete services offered by the system to the external users of the system. The subsystem-level use cases specify the services offered by the subsystems.

7.4.6 Use Case Packaging

Packaging is the mechanism provided by UML to handle complexity. When we have too many use cases in the top-level diagram, we can package the related use cases in packages, so that at most 6 or 7 packages are present at the top level diagram. Packaging is not restricted to the use case diagrams alone. Any modeling element that becomes large and complex can be broken up and made into packages. Please note that you can put any element of UML (including another package) in a package diagram. The symbol for a

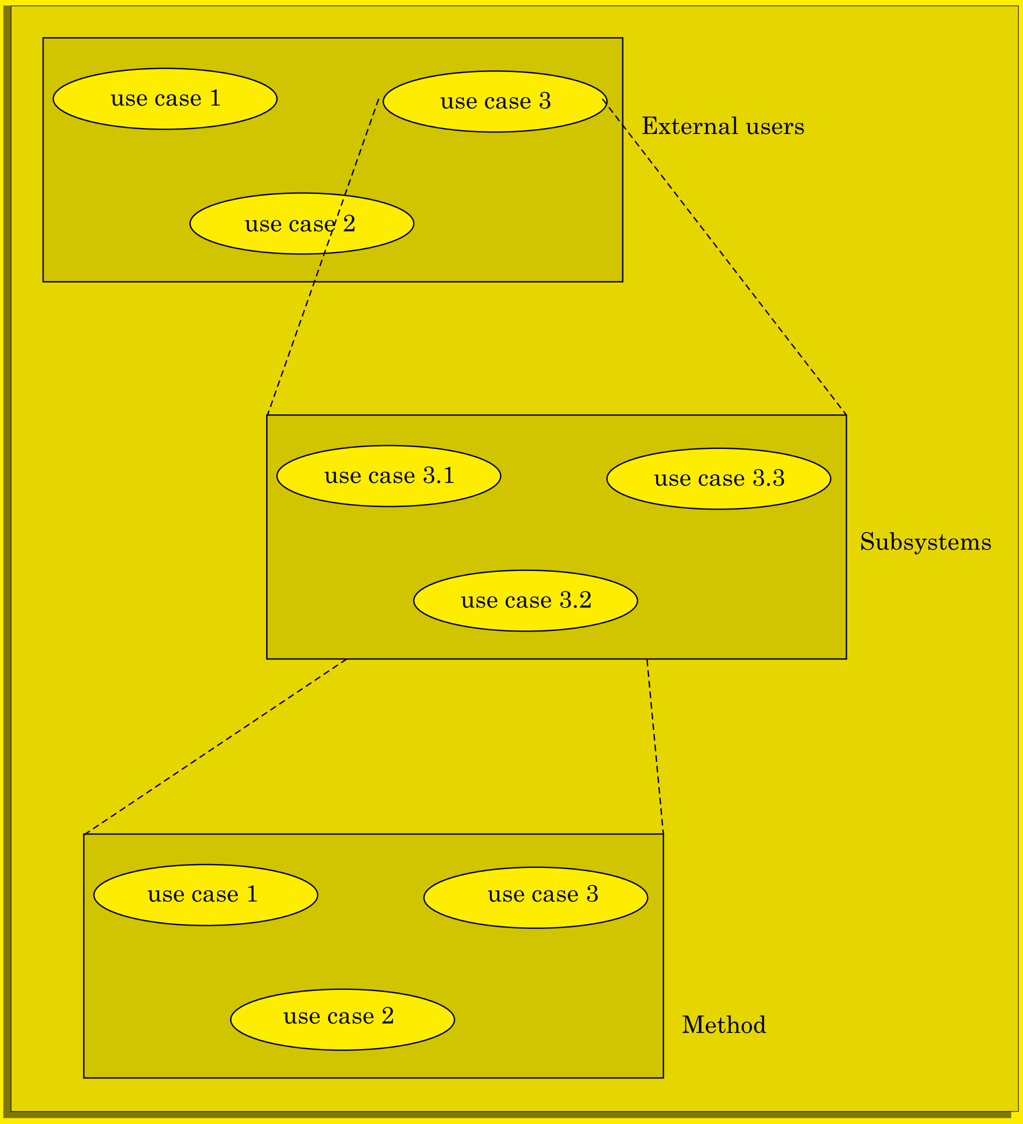


FIGURE 7.21 Hierarchical organisation of use cases.

package is a folder. Just as you organize a large collection of documents in a folder, you organize UML elements into packages. An example of packaging use cases is shown in Figure 7.22. Observe in Figure 7.22 that all accounts related use cases have been grouped into a package name `Accounts`. Similarly, all use cases pertaining to the academic functionalities have been put in a package named `Academics`, and so on.

7.5 CLASS DIAGRAMS

A class diagram can describe the static structure of a simple system. It shows how a system is structured rather than how it behaves. The static structure of a more complex system usually consists of a number of class diagrams. Each class diagram represents classes and their inter-relationships. Classes may be related to each other in four ways: generalisation, aggregation, association, and various kinds of dependencies. We now discuss the UML syntax for representation of the classes and their inter-relationships.

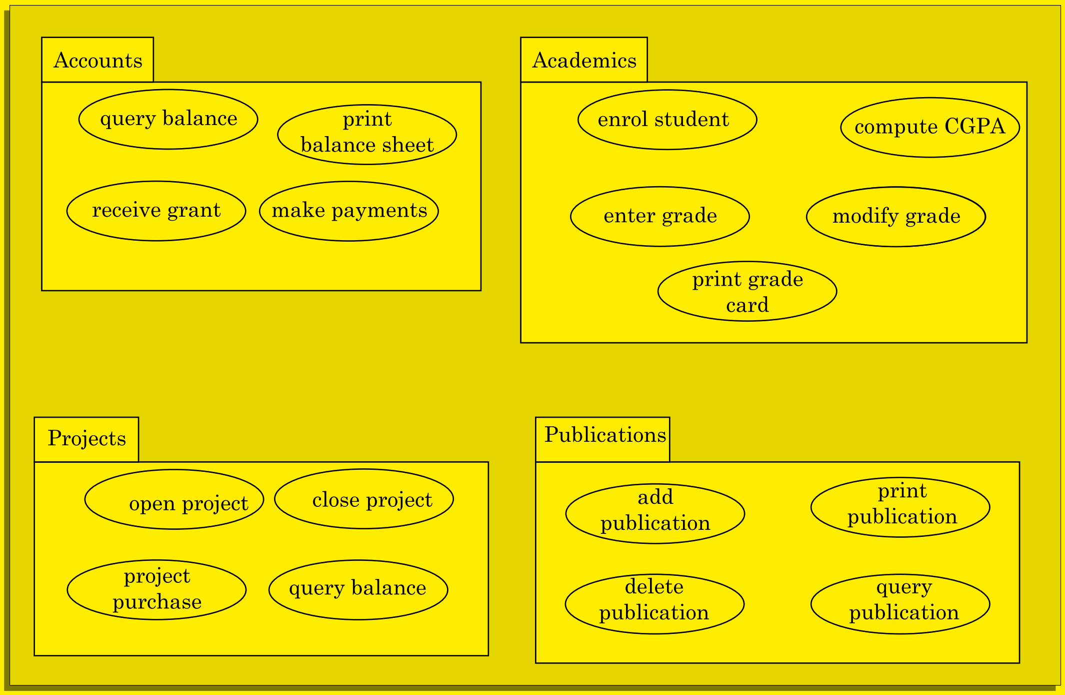


FIGURE 7.22 Use case packaging.

Classes

A class represents entities (objects) with common features. That is, objects having similar attributes and operations constitute a class. A class is represented by a solid outlined rectangle with compartments. Classes have a mandatory name compartment where the name is written centered in boldface. The class name is usually written using mixed case convention and begins with an uppercase (e.g., `LibraryMember`). Object names on the other hand, are written using a mixed case convention, but starts with a small case letter (e.g., `studentMember`). Class names are usually chosen to be singular nouns.

An example of various representations of a class supported in UML are shown in Figure 7.23. It can be seen in Figure 7.23 that classes can optionally have attributes and operations compartments. When a class appears in several diagrams, its attributes and operations are suppressed on all but one diagram. One may wonder why so many different representations for a class are supported! The answer is that these different notations are used depending on the amount of information about a class that is available at a given point of time during the design process. At the start of the design process, only the names of the classes is identified. This is the most abstract representation for the class. Later in the design process the methods for the class and the attributes are identified and the more elaborate represents are used.

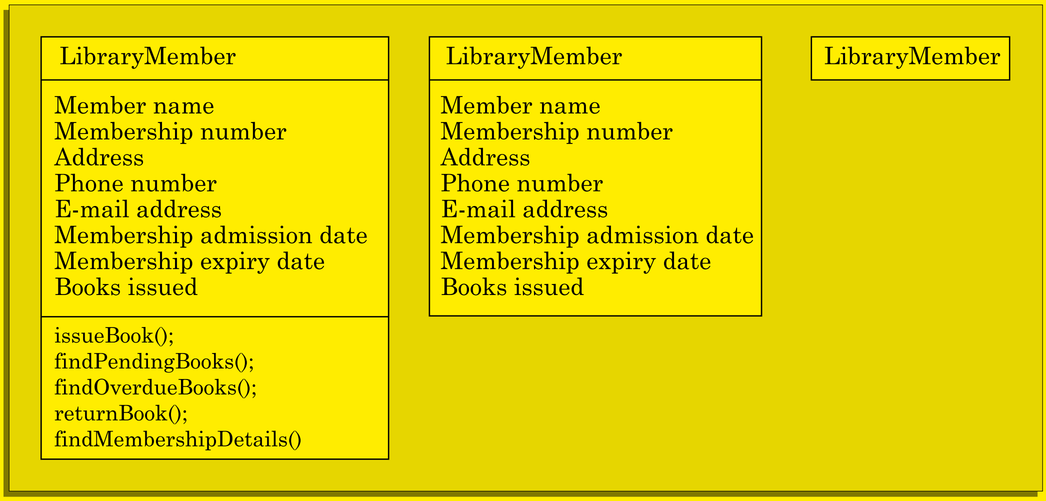


FIGURE 7.23 Different representations of the LibraryMember class.

Attributes

An attribute is a named property of a class. It represents the kind of data that an object might store. Attributes are listed by their names, and optionally their types (that is, their class, e.g., Int, Book, Employee, etc.), an initial value, and some constraints may be specified. Attribute names begin with a lower case letter, and are written left-justified using plain type letters.

An attribute name may be followed by square brackets containing a multiplicity expression, e.g., sensorStatus[10]. The multiplicity expression indicates the number of attributes that would be present per instance of the class. An attribute without square brackets must hold exactly one value. The type of an attribute is written by following the attribute name with a colon and the type name, (e.g. sensorStatus:Int).

The attribute name may be followed by an initialization expression. The initialization expression can consist of an equal sign and an initial value that is used to initialize the attributes of newly created objects, e.g., sensorStatus:Int=0.

Operation: The operation names always begin with a lower case letter are written using plain font type, and are typically left justified. Abstract operations are written in italics.[4] (Recollect that abstract operations are those for which the implementation is not provided during the class definition.) The parameters of a function may have a *type* or *kind* specified. The kind may be "in" indicating that the parameter is passed into the operation; or "out" indicating that the parameter is only returned from the operation; or "inout" indicating that the parameter is used for passing data into the operation and getting result from the operation. The default is "in".

[4] Many UML symbols are suitable for drawing using a CASE tool and are difficult to draw manually by hand. For example, italic names are very difficult to write by hand. When the UML diagram is to be drawn by hand, stereotypes such as ≪abstract≫ or constraints such as {abstract} can be used for difficult to draw symbols.

An operation may have a *return type* consisting of a single return type expression, e.g., issueBook(in bookName):Boolean. An operation may have a class scope (i.e., shared among all the objects of the class) and is denoted by underlining the operation name.

Often a distinction is made between the terms *operation* and *method*. An operation is something that is supported by a class and invoked by objects of other classes. There can be multiple methods implementing the same operation. We have pointed out earlier that this is called *static polymorphism*. The method names can be the same; however, it should be possible to distinguish among the methods by examining their parameters. Thus, the terms *operation* and *method* are distinguishable only when there is polymorphism. When there is only a single method implementing an operation, the terms method and operation are indistinguishable and can be used interchangeably.

Association

Association between two classes is represented by drawing a straight line between the concerned classes. Figure 7.24 illustrates the graphical representation of the association relation. The name of the association is written alongside the association line. An arrowhead may be placed on the association line to indicate the reading direction of the annotated association name. The arrowhead should not be misunderstood to be indicating the direction of a pointer implementing an association. On each side of the association relation, the multiplicity is either noted as a single number or as a value range. The multiplicity indicates how many instances of one class are associated with one instance of the other class. Value ranges of multiplicity are noted by specifying the minimum and maximum value, separated by two dots, e.g., 1..5. An asterisk is used as a wild card and means many (zero or more). The association of Figure 7.24 should be read as "Many books may be borrowed by a LibraryMember and a book may be borrowed by exactly one member". Usually, association relations among classes appear as verbs in the problem statement.

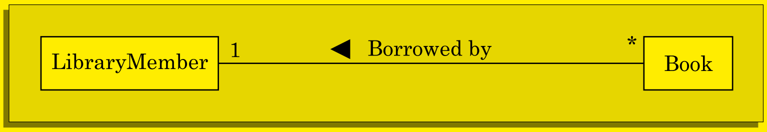


FIGURE 7.24 Association between two classes.

When two classes are associated with each other, it implies that in the implementation, the object instances of each class stores the id of the objects of the other class with which it is linked. In other words, bidirectional association relation between two classes is implemented by having the code of the two classes by maintaining the reference to the object of the object of the other class as an instance variable. Some CASE tools use the role names of the association relation for naming the corresponding automatically generated attribute.

Aggregation

Aggregation is a special type of association relation where the involved classes are not only associated to each other, but a whole-part relationship exists between them. That is, an aggregate object not only "knows" the ids of its parts (and therefore can invoke the methods of its parts), but also takes the responsibility of creating and destroying its parts. An example of an aggregation relation is a book register that aggregates book objects. Book objects can be added to the register and deleted as and when required.

Aggregation is represented by an empty diamond symbol at the aggregate end of a relationship. An example of the aggregation relationship has been shown in Figure 7.25. The figure represents the fact that a document can be considered as an aggregation of paragraphs. Each paragraph can in turn be considered as an aggregation of sentences. Observe that the number 1 is annotated at the diamond end, and the symbol * is annotated at the other end. This means that one document can have many paragraphs. On the other hand, if we wanted to indicate that a document consists of exactly 10 paragraphs, then we would have written number 10 in place of the symbol *.

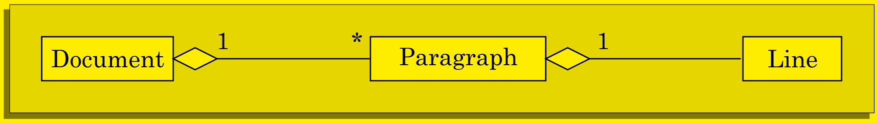


FIGURE 7.25 Representation of aggregation.

Composition

Composition is a stricter form of aggregation, in which the parts are existence-dependent on the whole. This means that the lifeline of the whole and the part are identical. When the whole is created, the parts are created and when the whole is destroyed, the parts are destroyed.

An example of composition is an order object where after placing the order, no item in the order can be changed. If any changes to any of the order items are required after the order has been placed, then the entire order has to be cancelled and a new order has to be placed with the changed items. In this case, as soon as an order object is created, all the order items in it are created and as soon as the order object is destroyed, all order items in it are also destroyed. That is, the life of the components (order items) is identical to that of the aggregate (order). The composition relationship is represented as a filled diamond drawn at the composite-end. An example of the composition relationship to model that an order consists of many items is shown in Figure 7.26.

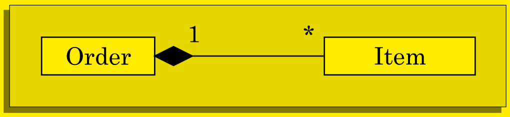


FIGURE 7.26 Representation of composition.

Aggregation *versus* Composition: Both aggregation and composition represent part/ whole relationships. If components can dynamically be added to and removed from the aggregate, then the relationship is expressed as *aggregation*. On the other hand, if the components are not required to be dynamically added/delete, then the components have the same life time as the composite. In this case, the relationship should be represented by *composition*. In the implementation of a composite relationship, the component objects are created by the constructor of the composite object. Therefore, when a composite object is created, the components are automatically created. No methods in the composite class for adding, deleting, or modifying the component is implemented.

We illustrate the subtle difference between the aggregation and composition relation between classes using the example of order and item objects. Consider that an order consists of many order items. If the order once placed, the items cannot be changed at all. In this case, the order is a composition of order items. However, if order items can be changed (added, delete, and modified) after the order has been placed, then aggregation relation can be used to model the relationship between the order and the item classes.

Inheritance

The inheritance relationship is represented by means of an empty arrow pointing from the subclass to the superclass. The arrow may be directly drawn from the subclass to the superclass. Alternatively, when there are many subclasses of a base class, the inheritance arrow from the subclasses may be combined to form a single line (see Figure 7.27) and is labelled with the specific aspect of the derived classes that is abstracted by the base class.

The direct arrows allow flexibility in laying out the diagram and can easily be drawn by hand. The combined arrows emphasise the collectivity of the subclasses. This is useful when specialisation for different groups derived classes has been done on the basis of some *discriminators*. In the example of Figure 7.27, issuable and reference are the discriminators. This highlights the facts that some library book types are issuable, whereas other types are for reference. In a more general case, various subclasses of a superclass can then be differentiated by means of the discriminators. The set of subclasses of a class having the same discriminator is called a partition of the inheritance relationship. It is often helpful to mention the discriminator during modelling, as these become documented design decisions, and it also enhances the readability of the design.

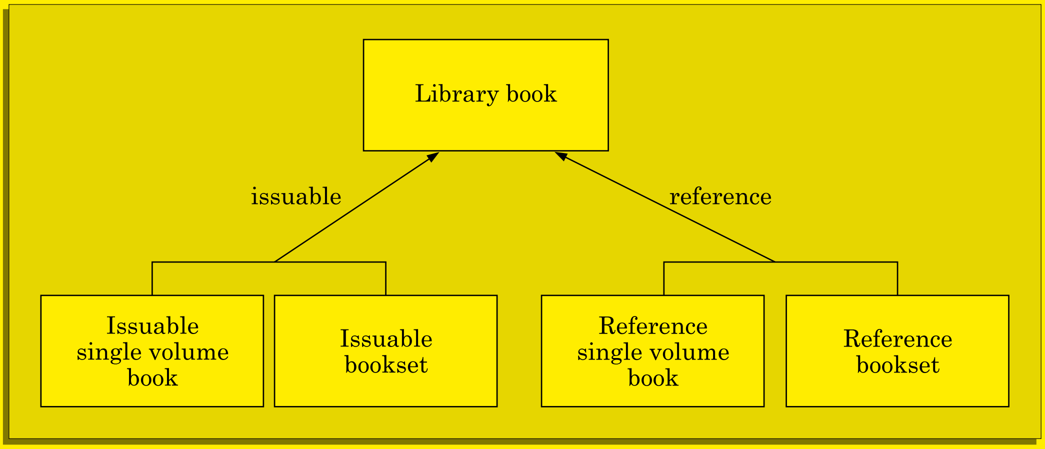


FIGURE 7.27 Representation of the inheritance relationship.

Dependency

A dependency relationship is represented by a dotted arrow (see Figure 7.28) that is drawn from the dependent class to the independent class.

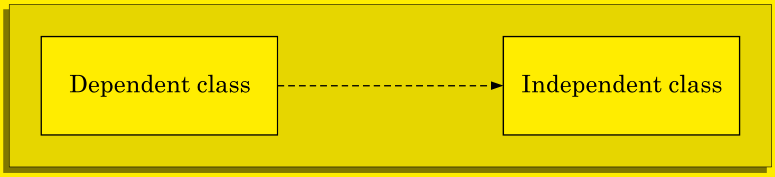


FIGURE 7.28 Representation of dependence between classes.

Constraints

A constraint describes either a condition that needs to be satisfied or an integrity rule for some entity. Constraints are typically used to describe aspects such as: permissible set of values of an attribute, specification of the pre- and post-conditions for operations, and definition of certain ordering of items. For example, to denote that the books in a library are maintained sorted on ISBN number, we can annotate the book class with the constraint {sorted}. UML allows the flexibility to use any set of words to describe the constraints. The only rule is that they are to be enclosed within braces. However, UML also provides *object constraint language* (OCL) to specify constraints more formally. In OCL the constraints are specified a semi-formal language, and therefore it is more amenable to automatic processing as compared to the informal constraints enclosed within {}. The interested reader is referred to [Rumbaugh, 1999] for more details on OCL.

Object diagrams

During the execution of a program, objects may dynamically get created and also destroyed. An object diagrams shows a snapshot of the objects in a system at a point in time. Since an object diagram shows instances of classes, rather than the classes themselves, it is often called as *instance diagram*. The objects are drawn using rounded rectangles (see Figure 7.29). Please observe that just like the class diagrams, the object diagrams may just indicate the name of the object or more details.

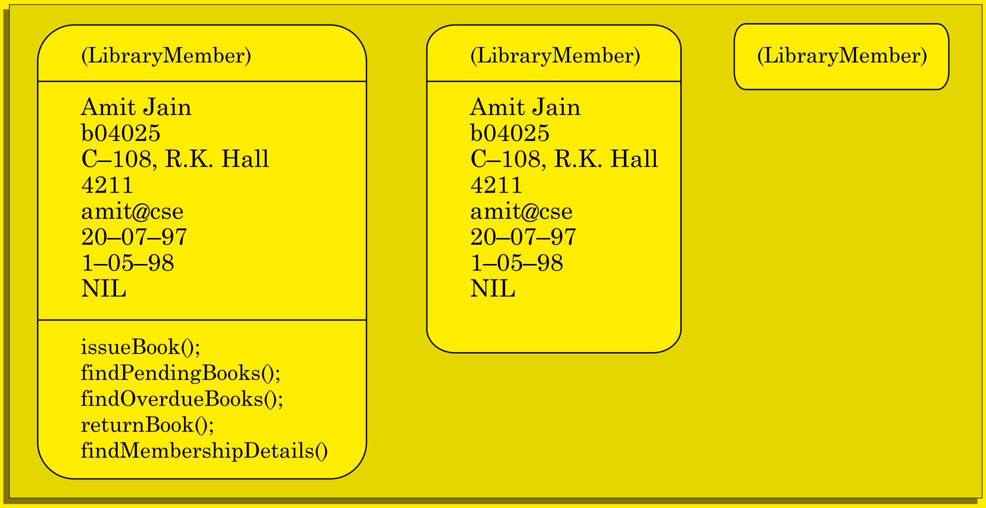


FIGURE 7.29 Different representations of a LibraryMember object.

An object model of a system usually undergoes continuous changes 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.

7.6 INTERACTION DIAGRAMS

Typically, each interaction diagram realises the behaviour of a single use case.

When a user invokes any one of the use cases of a system, the required behaviour is realized through the interaction of several objects in the system. An interaction diagram, as the name itself implies, is model that describes how groups of objects interact among themselves through message passing to realize some behaviour (execution of a use case).

Sometimes, especially for complex use cases, more than one interaction diagram may be necessary to capture the behavior.

There are two kinds of interaction diagrams—sequence diagrams and collaboration diagrams. These two diagrams are equivalent in the sense that any one diagram can be derived automatically from the other. In spite of apparent triviality of having different representations for essentially the same information, the fact is that both these representations are useful. These two representations actually portray different perspectives of behavior of a system and different types of inferences can be drawn from them. The interaction diagrams play a major role in any effective object-oriented design process. We discuss this issue in Chapter 8.

Sequence diagram

A sequence diagram shows the interactions among objects as a two dimensional chart. The chart is read from top to bottom. The objects participating in the interaction are drawn at the top of the chart as boxes attached to a vertical dashed line. Inside the box the name of the object is written with a colon separating it from the name of the class and both the name of the object and the class are underlined. When no name is specified, it indicates that we are referring any arbitrary instance of the class. For example, in Figure 7.30, *Book* represents any arbitrary instance of the Book class.

An object appearing at the top of a sequence diagram signifies that the object existed even before the time the use case execution was initiated. However, if some object is created during the execution of a use case and participates in the interaction (e.g., a method call), then the object should be shown at the appropriate place on the diagram where it is created.

The vertical dashed line attached to an object is called the *object's lifeline*. An object exists as long as its lifeline exists. Absence of lifeline after some point indicates that the object ceases to exist after that point in time. Normally, at a certain point if an object is destroyed, the lifeline of the object is crossed at that point and the lifeline for the object is not drawn beyond that point.

When an object receives a message, a rectangle called the *activation symbol* is drawn on the lifeline of an object to indicate the points of time at which the object is active. Thus, an activation symbol indicates that an object is active as long as the symbol (rectangle) exists on the lifeline. Each message is indicated as an arrow between the lifelines of two objects. The messages are drawn in chronological order from the top to the bottom. That is, reading the diagram from the top to the bottom would show the sequence in which the messages occur during the execution of the corresponding use case.

Each message is labelled with the message name, and optionally some control information can also be included along with the message name. Two important types of control information are:

- A condition is represented within square brackets. A condition prefixed to the message name, such as [invalid] indicates that a message is sent, only if the condition is true.
- An iteration marker shows that the message is sent many times to multiple receiver objects as would happen when you are iterating over a collection or the elements of an array. You can also indicate the basis of the iteration, e.g., [for every book object].

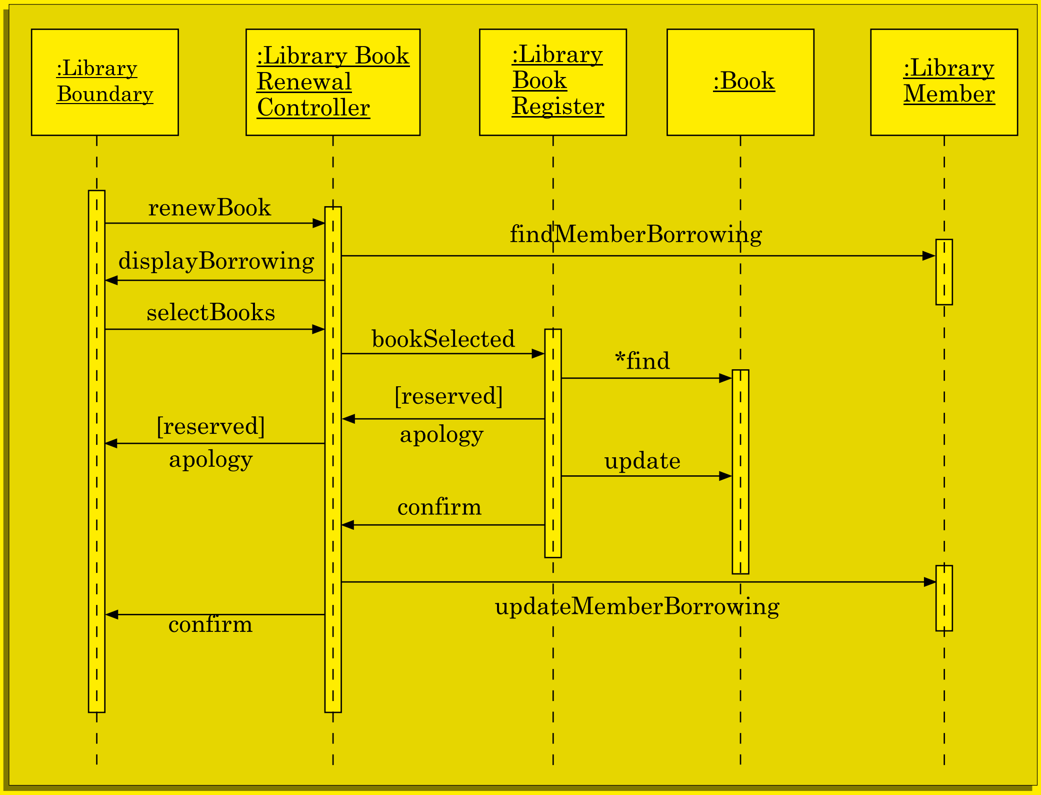


FIGURE 7.30 Sequence diagram for the renew book use case.

The sequence diagram for the book renewal use case (renew book) for the Library Automation Software is shown in Figure 7.30. Observe that the exact objects which participate to realise the behaviour of the renew book use case and the order in which they interact can be clearly inferred from the sequence diagram. The objects which interact to realise the behaviour of the use case are shown at the top of the diagram. The development of the sequence diagrams in the design methodology (discussed in Chapter 8) would help us to determine the responsibilities that must be assigned to the different classes. In other words, this indicates the methods that should be supported by a class.

Collaboration diagram

A collaboration diagram shows both structural and behaviourial aspects explicitly. This is unlike a sequence diagram which shows only the behaviourial aspects. The structural aspect of a collaboration diagram consists of objects and links among them indicating association between the corresponding classes. In this diagram, each object is also called a *collaborator.* The behaviourial aspect is described by the set of messages exchanged among the different collaborators. The messages are numbered to indicate the order in which they occur in the corresponding sequence diagram.

The link between objects is shown as a solid line and messages are annotated on the line. A message is drawn as a labelled arrow and is placed near the link. Messages are prefixed with sequence numbers because they are the only way to describe the relative sequencing of the messages in this diagram.

The collaboration diagram for the example of Figure 7.30 is shown in Figure 7.31. For a given sequence diagram, the collaboration diagram can be automatically generated by a CASE tool. A collaboration diagram explicitly shows which class is associated with other classes. Therefore, we can say that the collaboration diagram shows structural information more clearly than the sequence diagram.

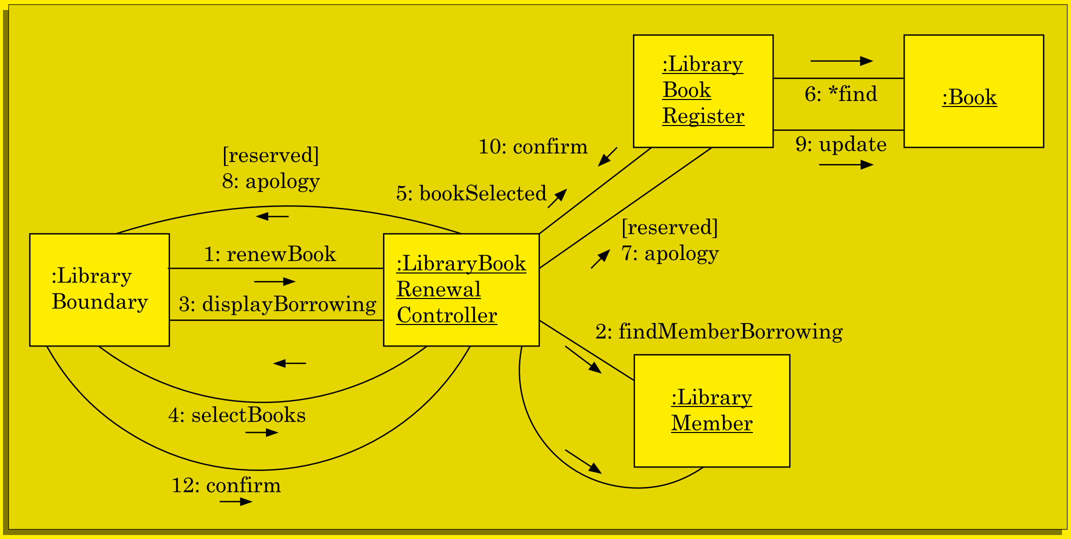


FIGURE 7.31 Collaboration diagram for the renew book use case.

7.7 ACTIVITY DIAGRAM

The activity diagram is possibly the one modelling element which was not present in any of the predecessors of UML. No such diagrams were present in the works of Booch, Jacobson, or Rumbaugh. It is possible that it has been based on the event diagram of Odell [1992], though the notations are very different from that used by Odell.

An activity diagram can be used to represent various activities (or chunks of processing) that occur during execution of the software and their sequence of activation. Please note that the activities, in general, may not correspond to the methods of classes, and represent higher granularity tasks. An activity is a state with an internal action and one or more outgoing transitions. On termination of the internal activity, the appropriate transition is taken. If an activity has more than one outgoing transition, then the exact conditions under which each is executed must be identified through use of appropriate conditions.

Activity diagrams are to some extent similar to flow charts. The main difference is that activity diagrams support description of parallel activities and synchronization aspects involved in different activities. Further activity diagrams incorporate swim lanes to indicate which components of the software are responsible for which activities.

Parallel activities are represented on an activity diagram by using *swim lanes*. Swim lanes make it possible to group activities based on who is performing them. For example, by observing the activity diagram of Figure 7.32, we can make out as to whether an activity is being performed by the academic department, hostel office, or any other entity. Thus, swim lanes subdivide activities based on the responsibilities of various components. Also, for each component participating in the use case execution, it becomes clear as to the specific activities for which it is responsible. For example, in Figure 7.32, the swim lane corresponding to the academic section, the activities that are carried out by the academic section (check student record and issue identity card) and the specific situation in which these are carried out are shown.

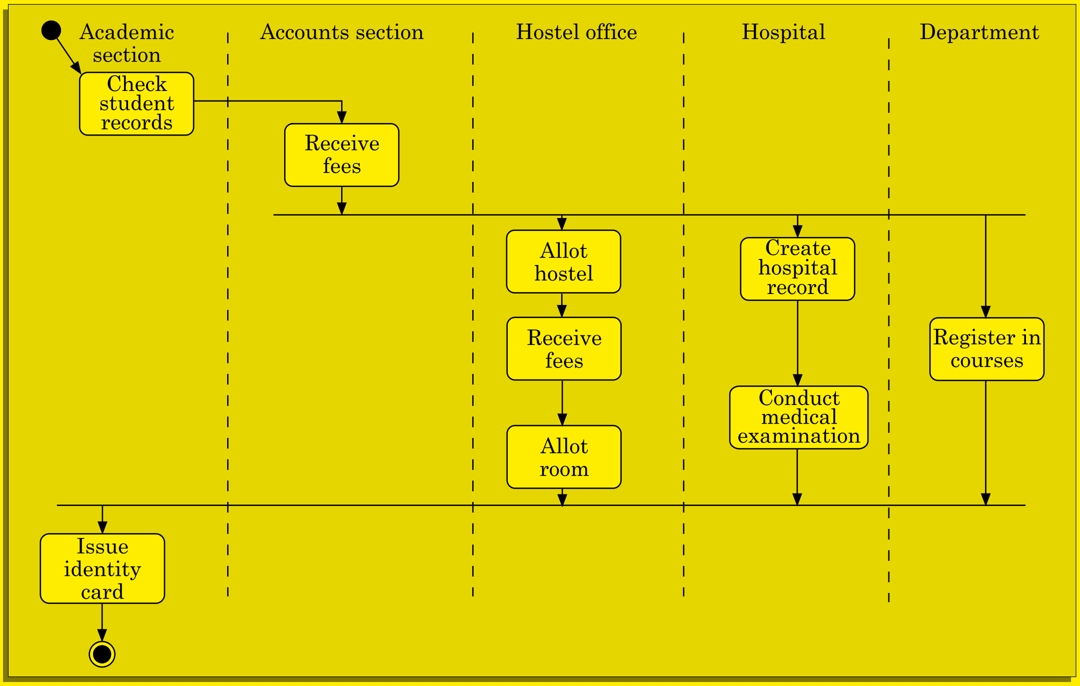


FIGURE 7.32 Activity diagram for student admission procedure at IIT.

Activity diagrams are normally employed in business process modelling. In business process modeling, an overall representation of the activities that occur in a business or an organization are represented. In software development, activity diagrams are often constructed during the initial stages of requirements analysis and specification. Activity diagrams can be very useful to understand the processing activities required in execution of complex use cases and the roles played by different components. Besides helping the developer to understand the complex use cases, these diagrams can also be helpful in developing interaction diagrams which help to allocate activities (responsibilities) to classes.

The student admission process (the business process of student admission) in IIT is shown as an activity diagram in Figure 7.32. This diagram shows the part played by different entities of the Institute in the admission procedure. After the fees are received at the account section, parallel activities start at the hostel office, hospital, and the Department. After all these activities complete (this is called synchronization and is represented by a horizontal line), the identity card can be issued to a student by the Academic section.

7.8 STATE CHART DIAGRAM

A state chart diagram is normally used to model how the state of an object may change over its life time. In particular, state chart diagrams are good at describing how the behaviour of an object changes across several use case executions. However, if we are interested in modelling some behaviour that involves several objects collaborating with each other to achieve the behaviour of a use case, state chart diagram is not appropriate. We have already seen that such behaviour is better modelled using sequence or collaboration diagrams.

State

State chart diagrams are based on the *finite state machine* (FSM) formalism. An FSM model consists of a finite number of states corresponding to those that the object being modeled can take. The object undergoes state changes when specific events occur. The FSM formalism existed long before the object-oriented technology and has been used for modelling a wide variety of applications. Apart from modelling, it has even been used in theoretical computer science as a generator and recogniser for regular languages.

Why state chart?

A major disadvantage of the FSM formalism is the state explosion problem. The number of states becomes too many and the model too complex when used to model practical systems. This problem is overcome in UML by using state charts rather than FSMs. The state chart formalism was proposed by David Harel [1990]. A state chart is a hierarchical model of a system and introduces the concept of a composite state (also called *nested state*).

Actions are associated with transitions and are considered to be processes that occur instantaneously and are not interruptible. Activities are associated with states and can take longer. An activity can be interrupted by an event.

Basic elements of a state chart

The basic elements of a state chart diagram are the following:

Initial state: This represented by a filled circle.

Final state: This is represented by a filled circle inside a larger circle.

State: These are represented by rectangles with rounded corners.

Transition: A transition is shown as an arrow between two states. Normally, the name of the event which causes the transition is placed alongside the arrow. You can also assign a guard to the transition. A guard is a Boolean logic condition. The transition can take place only if the guard evaluates to true. The syntax for the label of the transition is shown in 3 parts—[guard]event/action.

An example state chart model for the order object of the Trade House Automation software is shown in Figure 7.33. Observe that from the Rejected order state, there is an automatic and implicit transition to the end state. Such transitions which do not have any event or guard annotated with it are called *pseudo transitions*.

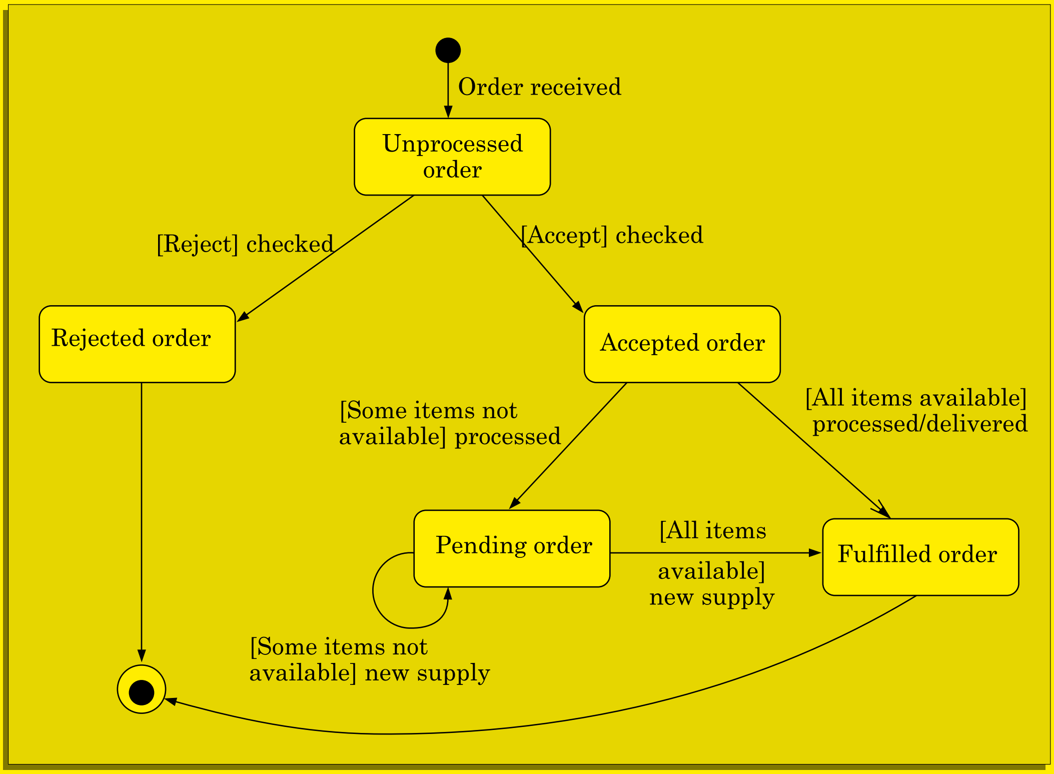

Order received
Unprocessed order
[Reject] checked
[Accept] checked
Rejected order
Accepted order
[Some items not available] processed
[All items available] processed/delivered
Pending order
[All items available] new supply
Fulfilled order
[Some items not available] new supply

FIGURE 7.33 State chart diagram for an order object.

7.9 POSTSCRIPT

UML has gained rapid acceptance among practitioners and academicians over a short time and has proved its utility in arriving at good design solutions to software development problems and facilitating standardised documentation of object-oriented designs.

In this text, we have kept our discussions on UML to a bare minimum and have concentrated only on those aspects that are necessary to solve moderate sized traditional software design problems.

Before concluding this chapter, we give an overview of some of the aspects that we had chosen to leave out. We first discuss the package, component, and deployment diagrams. Since UML has undergone a significant change with the release of UML 2.0 in 2003. We briefly mention the highlights of the improvements brought about by UML 2.0 over the UML 1.X—which was our focus so far. This significant revision into UML2.0 was necessitated to make UML applicable to the development of software for the emerging embedded and telecommunication domains.

7.9.1 Package, Component, and Deployment Diagrams

In the following subsections we provide a brief overview of the package, component, and deployment diagrams:

Package diagram

A package can be used to group several related classes. In fact, a package diagram can be used to group any UML artifacts. We have already discussed packaging of use cases in Section 7.4.6. Even before their use in UML, packages have been a popular way of organising source code files. However, the notation for packages were rather loosely used. Now with UML, these have become standardised. Package diagrams can show the different class groups (packages) and their inter-dependencies. These are very useful to document the organisation of the source files for projects having a large number of program files. An example of a package diagram has been shown in Figure 7.34.

Note, that a package may contain further packages.

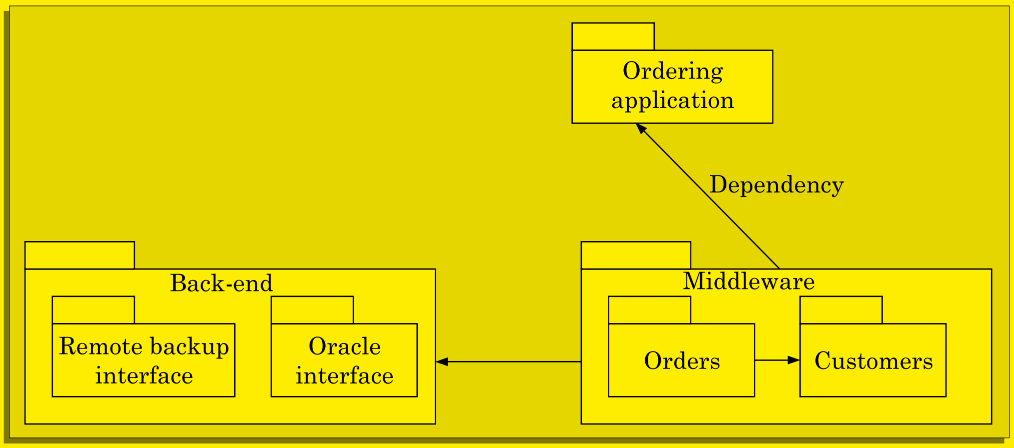


FIGURE 7.34 An example package diagram.

Component diagram

A component represents a piece of software that can be independently purchased, upgraded, and integrated into an existing software. A component diagram can be used to represent the physical structure of an implementation in terms of the various components of the system. A component diagram is typically used to achieve the following:

- Organise source code to be able to construct executable releases.
- Specify dependencies among different components.

A components diagram can be used to provide a high-level view of each component in terms the different classes it contains.

Deployment diagram

The deployment diagram shows the environmental view of a system. In other words, a deployment diagram shows how a software system will be physically deployed over the targeted hardware environment. It captures, which component will execute on which hardware component and how they will they communicate with each other. Since the diagram models the run time architecture of an application, this diagram can be very useful to the system's operation and maintenance staff.

The environmental view provided by the deployment diagram is important for complex and large software solutions that run on hardware systems comprising multiple components. In this case, deployment diagram provides an overview of how the different software components are distributed among the different hardware components of the system.

7.9.2 UML 2.0

UML 1.X lacked a few capabilities that made it difficult to use it in some non-traditional domains. Some of the features that prominently lacked in UML 1.X include lack of support for representing the following—concurrent execution of methods, asynchronous messages, events, ports, and active objects. In many applications, including the embedded and telecommunication software development, capability to model timing requirements using a timing diagram was urgently required to make UML applicable in these important segments of software development. Further, certain changes were required to support interoperability among UML-based CASE tools using XML *metadata interchange* (XMI).

UML 2.0 defines thirteen types of diagrams, divided into three categories as follows:

Structure diagrams: These include the class diagram, object diagram, component diagram, composite structure diagram, package diagram, and deployment diagram.

Behaviour diagrams: These diagrams include the use case diagram, activity diagram, and state machine diagram.

Interaction diagrams: These diagrams include the sequence diagram, communication diagram, timing diagram, and interaction overview diagram. The collaboration diagram of UML 1.X has been renamed in UML 2.0 as communication diagram. This renaming was necessary as the earlier name was somewhat misleading. The diagram highlights the communications among the classes during the execution of a use case rather than showing collaborative problem solving.

Though a large number of new features have been introduced in UML 2.0 as compared to 1.X, in the following subsections, we discuss only two of the important enhancements in UML2.0 combined fragments and composite structure diagram.

Combined fragments in sequence diagrams

A combined fragment is a construct that has been introduced in UML 2.0 to allow description of various control and logic structures in a more visually apparent and concise manner in a sequence diagram. It also allows representation of concurrent execution behaviour such as that takes place in a mutithreaded execution situation.

Let us now understand the anatomy of a combined fragment and its use. A combined fragment divides a sequence diagram into a number of areas or fragments that have different behaviour (see Figure 7.35). A combined fragment appears over an area of a sequence diagram to make certain control and logic aspects visually clear. As shown in Figure 7.35, a combined fragment consists of many fragments and an operator shown at the top left corner. Each fragment can be associated with a guard (a Boolean expression). We now discuss these components of a combined fragment in the following:

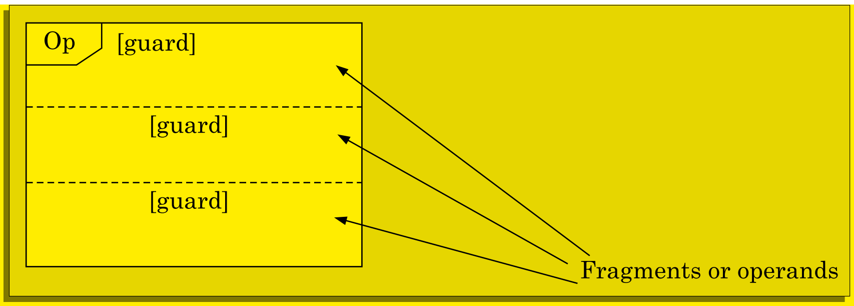


FIGURE 7.35 Anatomy of a combined fragment in UML 2.0.

Fragment: A fragment in a sequence diagram is represented by a box, and encloses a portion of the interactions within a sequence diagram. Each fragment is also known as an *interaction operand*. An interaction operand may contain an optional guard condition, which is also called an *interaction constraint*. The behaviour specified by the interaction operand is executed, only if its guard condition evaluates to true.

Operator: A combined fragment is associated with one operator called *interaction operator* that is shown at the top left corner of the fragment. The operator indicates the type of fragment. The type of logic operator specialised along with the guards in the fragment defines the behaviour of the combined fragment. A combined fragment can also contain nested combined fragments containing additional conditional structures to form more complex structures.

Some of the important operators of a combined fragment are the following:

alt : This operator indicates that among multiple fragments, only the one whose guard is true will execute.

opt : An optional fragment that will execute only if the guard is true.

par : This operator indicated that various fragments can execute at the same time.

loop : A loop operator indicates that the various fragments may execute multiple times. The guard indicates the basis of iteration, meaning that the execution would continue until the guard turns false.

region : It defines a critical region in which only one thread can execute. An example of a combined fragment has been shown in Figure 7.36.

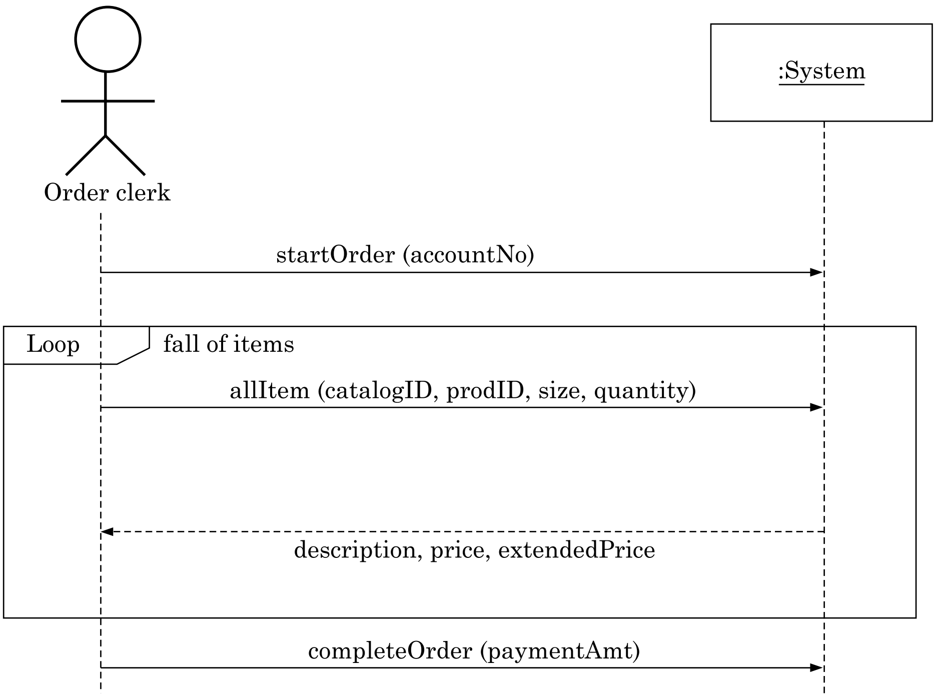

Order clerk
:System
startOrder (accountNo)
Loop
fall of items
allItem (catalogID, prodID, size, quantity)
description, price, extendedPrice
completeOrder (paymentAmt)

FIGURE 7.36 An example sequence diagram showing a combined fragment in UML 2.0.

Composite structure diagram

The composite structure diagram lets you define how a class is defined by a further structure of classes and the communication paths between these parts. Some new core constructs such as parts, ports and connectors are introduced.

Part: The concept of parts makes possible the description of the internal structure of a class.

Port: The concept of a port makes it possible to describe connection points formally. These are addressable, which means that signals can be sent to them.

Connector: Connectors can be used to specify the communication links between two or more parts.

SUMMARY

- In this chapter, we first reviewed some important concepts associated with object-orientation.
- One of the primary advantages of object-orientation is increased productivity of the software development team. The reason why object-oriented projects achieve