Module 7

Software Engineering Issues

Lesson 37

Software Design – Part 2

Specific Instructional Objectives

At the end of this lesson, the student would be able to:

- Define classes, objects, attributes, and methods
- Explain data abstraction and identify its advantages
- Explain inheritance, and identify the different types of inheritance and its advantages
- Explain encapsulation and identify its advantages
- Explain polymorphism, static and dynamic binding
- Identify the advantages of object-oriented design
- Explain what a model is
- Understand the different views of the system that are captured by UML diagrams
- Explain the use case model of a system
- Factorize use cases into different component use cases
- Explain the relationships among classes by means of association, aggregation, and composition
- Draw interaction diagrams for a given problem
- Draw activity diagrams for a given problem
- Develop the state chart diagram for a given class
- Explain design patterns
- Identify pattern solution for a particular problem in terms of class and interaction diagrams
- Explain expert pattern, creator pattern, controller pattern, and model-view-separation pattern
- Explain domain modelling
- Identify the types of objects identified during domain analysis and explain their interaction
- Identify the different approaches for identifying objects in the context of OOD methodology
- Develop sequence diagram for a use case

1. Object-Oriented Concepts – An Introduction

In this section, first we will discuss the basic mechanisms in object-oriented paradigm. We will then discuss some key concepts and a few related technical terms.

1.1. Basic Entities

Classes, Objects, Attributes, and Methods: Similar objects constitute a class. This means, objects possessing similar attributes and displaying similar behaviour constitute a class. For example, all employee objects have similar attributes such as his name, code number, salary, address, etc. and exhibits similar behaviour as other employee objects. Once the class is defined, it serves as a template for object creation. Since each object is created as an instance of some class, classes can be considered as abstract data types (ADTs).

In the object-oriented approach, a system is designed as a set of interfacing objects. Normally, each object represents a tangible real-world entity such as a library member, an employee, a book, etc. Objects are basically class variables. However, at times some conceptual entities can be considered as objects (e.g. a scheduler, a controller, etc.) to simplify solutions to certain problems. When a system is analysed, developed, and implemented in terms of the natural objects occurring in it, it becomes easier to understand the design and implementation of the system. Each object essentially consists of some data that are private to the object and a set of functions that operate on those data as shown in fig.37.1. In fact, the functions of an object have the sole authority to operate on the private data of that object. Therefore, an object can not directly access the data internal to another object. However, an object can indirectly access the internal data of the other objects by invoking the operations (i.e. methods) supported by those objects.

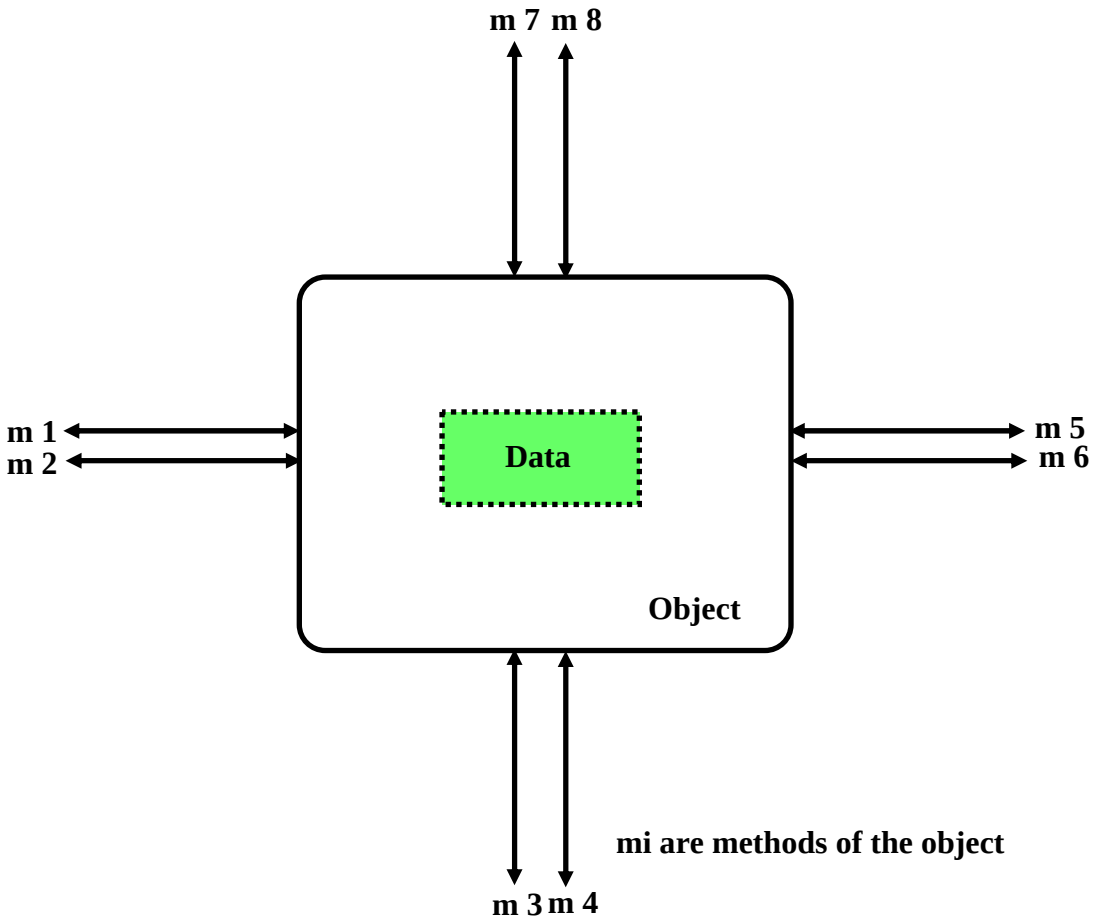

Fig. 37.1 A Model of an object

The data internal to an object are called the attributes of the object, and the functions supported by an object are called its methods. Fig. 37.2 shows LibraryMember class with eight attributes and five methods.

1.2. Data Abstraction

Data abstraction means that each object hides (abstracts away) from other objects the exact way in which its internal information is organized and manipulated. It only provides a set of methods, which other objects can use for accessing and manipulating this private information of the object. Other objects can not directly access the private data of an 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 and how it 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.

1.2.1. Advantages of Data Abstraction

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

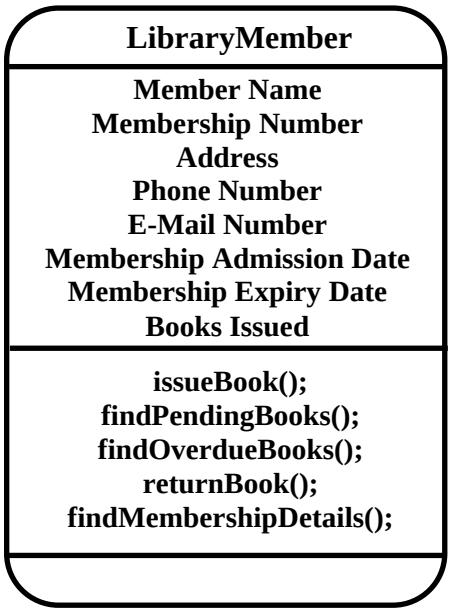

Fig. 37.2 A class model with attributes and methods

1.3. Inheritance

The inheritance feature allows us to define a new class by extending or modifying an existing class. The original class is called the base class (or super class) and the new class obtained through inheritance is called as the derived class (or sub class). A base class is a generalization of its derived classes. This means that the base class contains only those properties that are common to all the derived classes. Again each derived class is a specialization of its base class because it

modifies or extends the basic properties of the base class in certain ways. Thus, the inheritance relationship can be viewed as a *generalization-specialization* relationship.

Using the inheritance relationship, different classes can be arranged in a class hierarchy (or class tree). In addition to inheriting all properties of the base class, a derived class can define new properties. That is, it can define new data and methods. It can even give new definitions to methods which already exist in the base class. Redefinition of methods which existed in the base class is called as **method overriding**. In fig. 37.3, LibraryMember is the base class for the derived classes Faculty, Student, and Staff. Similarly, Student is the base class for the derived classes Undergraduate, Postgraduate, and Research. Each derived class inherits all the data and methods of the base class. It can also define additional data and methods or modify some of the inherited data and methods. The different classes in a library automation system and the inheritance relationship among them are shown in the fig. 37.3. The inheritance relationship has been represented in fig. 37.3 using a directed arrow drawn from a derived class to its base class. In fig. 37.3, the LibraryMember base class might define the data for name, address, and library membership number for each member. Though Faculty, Student, and Staff classes inherit these data, they might have to redefine the respective issue-book methods because the number of books that can be borrowed and the duration of loan may be different for the different category of library members. Thus, the issue-book method is overridden by each of the derived classes and the derived classes might define additional data max-number-books and max-duration-of-issue which may vary for the different member categories.

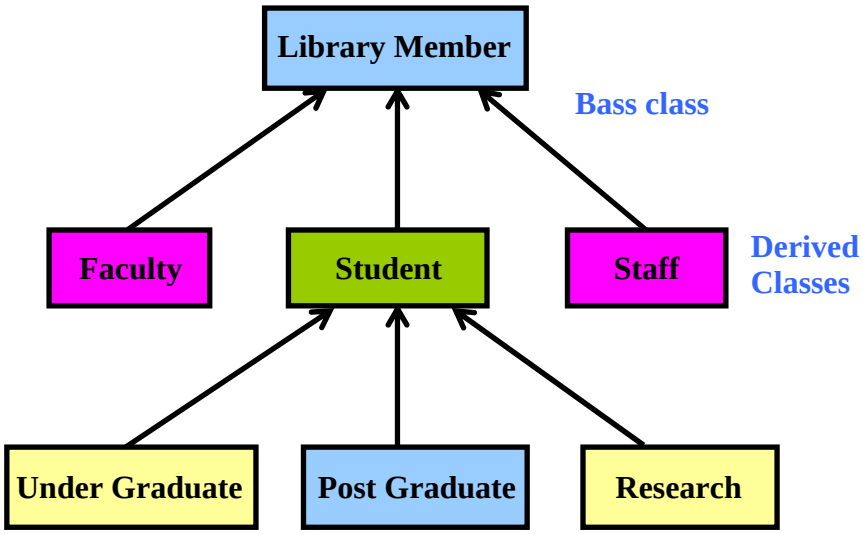

Fig. 37.3 Library Information System Example

1.3.1. Object-Oriented Vs. Object-Based Languages

Languages that support classes (Abstract Data Types) but do not support inheritance are called object-based languages. On the other hand, languages that support both classes as well as inheritance are called object-oriented languages.

1.3.2. Advantages of Inheritance

An important advantage of the inheritance mechanism is code reuse. If certain methods or data are similar in asset of classes, then instead of defining these methods and data each of these

classes separately, these methods and data are defined only once in the base class and are inherited by each of its subclasses. For example, in the Library Information System example of fig. 37.3, each category of member objects Faculty, Student, and Staff need the data member-name, member-address, and membership-number and therefore these data are defined in the base class LibraryMember and inherited by its subclasses.

Another advantage of the inheritance mechanism is the conceptual simplification that comes from reducing the number of independent features of the classes.

1.3.3. Multiple Inheritance

Multiple inheritance is a mechanism by which a sub class can inherit attributes and methods from more than one base class. Suppose research students are also allowed to be staff of the institute, then some of the characteristics of the Research class might be similar to the Student class and some other characteristics might be similar to the Staff class. Such a class hierarchy can be represented as in fig. 37.4. Multiple inheritance is represented by arrows drawn from the subclass to each of the base classes. The class Research inherits features from both the classes Student and Staff.

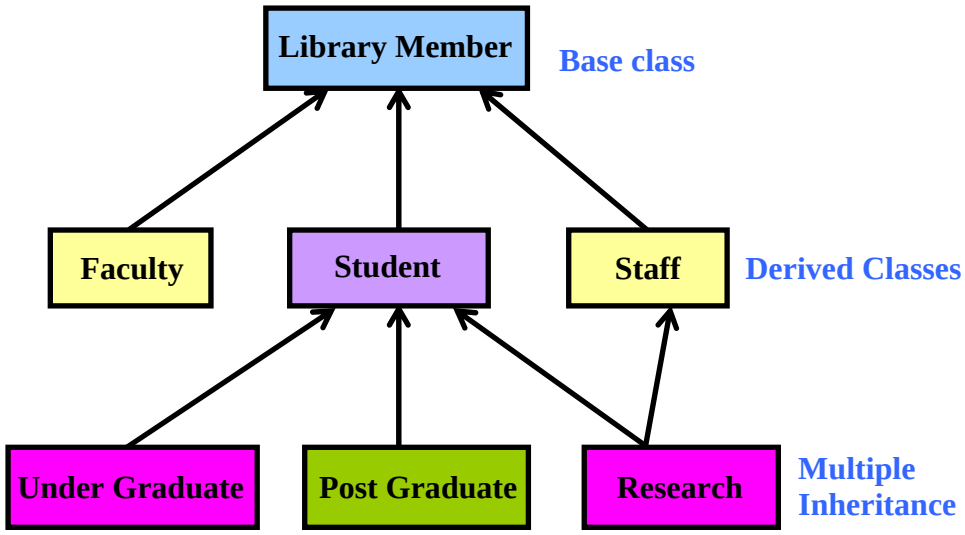

Fig. 37.4 Library Information System example with multiple inheritance.

1.4. Encapsulation

The property of an object by which it interfaces with the outside world only through messages is referred to as encapsulation. The data of an object are encapsulated within its methods and are available only through message-based communication. This concept is schematically represented in fig. 37.5.

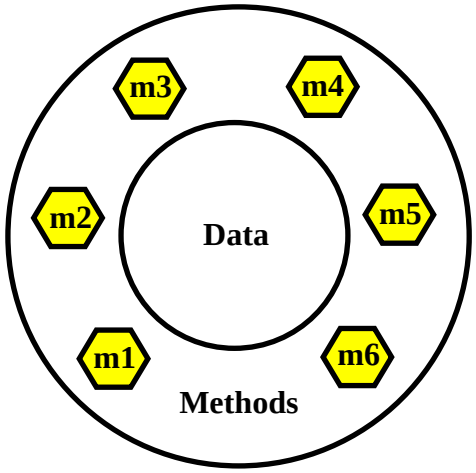

Fig. 37.5 Schematic representation of the concept of encapsulation

1.4.1. Advantages Of Encapsulation

Encapsulation offers three important advantages:

- It protects an object's internal data from corruption by other objects. This protection includes protection from unauthorized access and protection from different types of problems that arise from concurrent access of data such as deadlock and inconsistent values.
- Encapsulation hides the internal structure of an object so that interaction with the object is simple and standardized. This facilitates reuse of objects across different projects. Furthermore, if the internal structure or procedures of an object are modified, other objects are not affected. This results in easy maintenance.
- Since objects communicate among each other using messages only, they are weakly coupled. The fact that objects are inherently weakly coupled enhances understanding of design since each object can be studied and understood almost in isolation from other objects.

1.5. Polymorphism

Polymorphism literally means poly (many) morphs (forms). Broadly speaking, polymorphism denotes the following:

- The same message can result in different actions when received by different objects. This is also referred to as **static binding**. This occurs when multiple methods with the same operation name exist.
- When we have an inheritance hierarchy, an object can be assigned to another object of its ancestor class. When such an assignment occurs, a method call to the ancestor object would result in the invocation of the appropriate method of the object of the derived class. The exact method to which a method call would be bound cannot be known at compile time, and is dynamically decided at the runtime. This is also known as **dynamic binding**.

1.5.1. Static Binding

An example of static binding is the following. Suppose a class named Circle has three definitions for the create operation. One definition does not take any argument and creates a circle with default parameters. The second definition takes the center point and radius as its parameters. In this case, the fill style values for the circle would be set to default "no fill". The third takes the centre point, the radius, and the fill style as its input. When the create method is invoked, depending on the parameters given in the invocation, the appropriate method will be called. If create is invoked with no parameters, then a default circle would be created. If only the centre and the radius are supplied, then an appropriate circle would be created with no fill type, and so on. A class definition of the Circle class with the overloaded create method is shown in fig. 37.6. When the same operation (e.g. *create*) is implemented by multiple methods, the method name is said to be overloaded.

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

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

Fig. 37.6 Circle class with overloaded create method

1.5.2. Dynamic Binding

Using dynamic binding a programmer can send a generic message to a set of objects which may be of different types (i.e., belonging to different classes) and leave the exact way in which the message would be handled to the receiving objects. Suppose we have a class hierarchy of different geometric objects in a drawing as shown in fig. 37.7. Now, suppose the display method is declared in the shape class and is overridden in each derived class. If the different types of geometric objects making up a drawing are stored in an array of type shape, then a single call to the display method for each object would take care to display the appropriate drawing element. That is, the same draw call to a shape object would take care of drawing the appropriate shape. This code segment is shown in fig. 37.8.

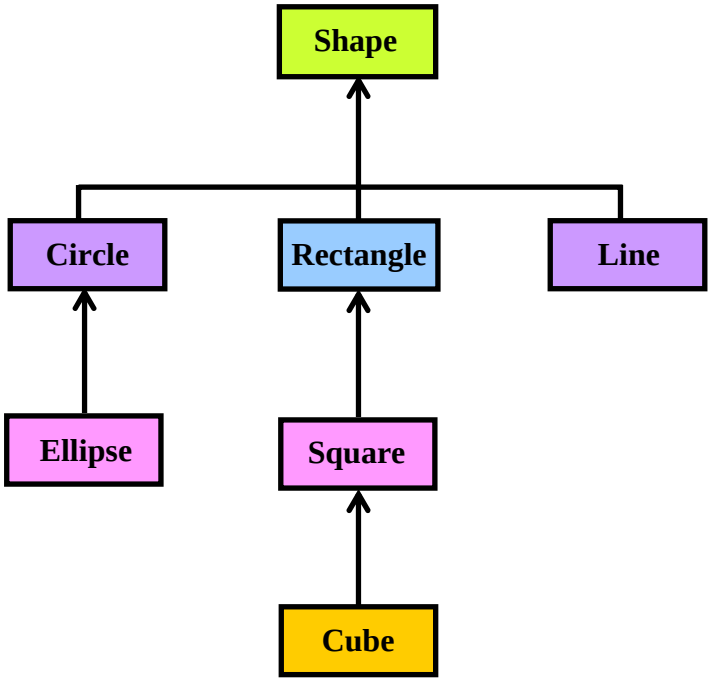

Fig. 37.7 Class hierarchy of geometric objects

Traditional Code	Object-oriented Code
`if(shape == Circle) then draw_circle(); else if(shape == Rectangle)then draw_rectangle(); ____ ____`	`shape.draw();`

Fig. 37.8 Traditional code and object-oriented code using dynamic binding

1.5.3. Advantages of Dynamic Binding

The main advantage of dynamic binding is that it leads to elegant programming and facilitates code reuse and maintenance. With dynamic binding, new derived objects can be added with minimal changes to existing objects. This advantage of polymorphism can be illustrated by comparing the code segments of an object-oriented program and a traditional program for drawing various graphic objects on the screen. It can be assumed that the shape is the base class, and the classes Circle, Rectangle, and Ellipse are derived from it. Now, shape can be assigned any objects of type Circle, Rectangle, Ellipse, etc. But, a draw method invocation of the shape object would invoke the appropriate method. It can be easily seen in fig. 37.8 that, because of dynamic binding, the object-oriented code is much more concise and intellectually appealing. Also, suppose in the example program segment, it is later found necessary to handle 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 the object-oriented program, the code need not change. Only a new class called Ellipse has to be defined.

1.6. Advantages of the Object-Oriented Design

In the last few years that OOD has come into existence, it has found widespread acceptance in industry as well as in academics. The main reason for the popularity of OOD is that it holds the following promises:

- Code and design reuse.
- Increased productivity.
- Ease of testing and maintenance.
- Better code and design understandability.

Out of all these advantages, the chief advantage of OOD is *improved productivity* – which comes about due to a variety of factors, such as

- Code reuse by the use of predefined class libraries.
- Code reuse due to inheritance.
- Simpler and more intuitive abstraction, i.e. better organization of inherent complexity.
- Better problem decomposition.

2. Object Modelling using UML

2.1. Model and its uses

A model captures aspects important for some application while omitting (or abstracting) the rest. A model in the context of software development can be graphical, textual, mathematical, or program code-based. Models are very useful in documenting design and analysis results. Models also facilitate the analysis and design procedures themselves. Graphical models are very popular because they are easy to understand and construct. UML is primarily a graphical modelling tool. However, it often requires text explanations to accompany the graphical models.

An important reason behind constructing a model is that it helps manage complexity. Once models of a system have been constructed, they can be used for a variety of purposes during software development, including the following:

- Code reuse by the use of predefined class libraries
- Analysis
- Specification
- Code generation
- Design
- Visualize and understand the problem and the working of a system
- Testing, etc.

In all these applications, the UML models can not only be used to document the results but also to arrive at the results themselves. Since a model can be used for a variety of purposes, it is reasonable to expect that the model would vary depending on the purpose for which it is being constructed. For example, a model developed for initial analysis and specification should be very different from the one used for design. A model that is being used 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 used 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, along with the model.

2.2. UML diagrams

UML 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 the software system to be developed and facilitate a comprehensive understanding of the system. Such models can be refined to get the actual implementation of the system.

The UML diagrams can capture the following five views of a system:

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

Fig. 37.9 shows the UML diagrams responsible for providing the different views.

User's view: This view defines the functionalities (facilities) made available by the system to its users. The users' view captures the external users' view of the system in terms of the functionalities offered by the system. The users' view is a black-box view of the system where the internal structure, the dynamic behavior of different system components, the implementation etc. are not visible. The users' view is very different from all other views in the sense that it is a functional model compared to the object model of all other views. The users' view can be considered as the central view and all other views are expected to conform to this view. This thinking is in fact the crux of any user centric development style.

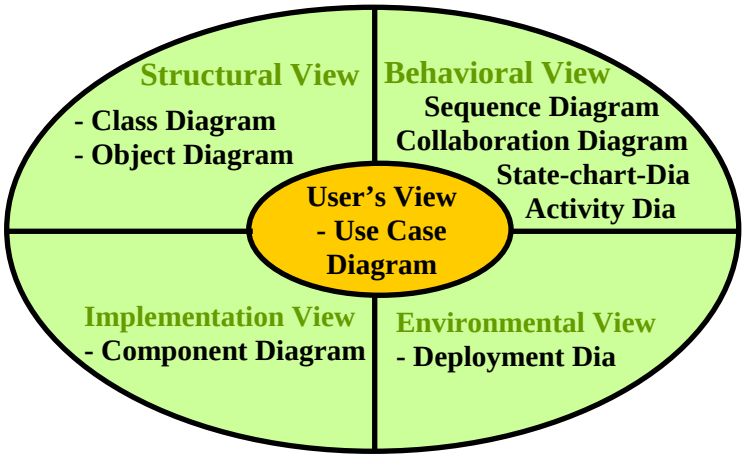

Fig. 37.9 Different types of diagrams and views supported in UML

Structural view: The structural view defines 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 the static model, since the structure of a system does not change with time.

Behavioral view: The behavioural view captures how objects interact with each other to realize the system behaviour. The system behaviour captures the time-dependent (dynamic) behaviour of the system.

Implementation view: This view captures the important components of the system and their dependencies.

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

2.3. Use Case Model

The use case model for any system consists of a set of "use cases". Intuitively, use cases represent the different ways in which a system can be used by the users. A simple way to find all the use cases of a system is to ask the question: "What can the users do by using the system?" Thus for the Library Information System (LIS), the use cases could be:

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

Use cases correspond to the *high-level functional requirements*. The use cases partition the system behaviour into transactions, so that each transaction performs some useful action from the user's point of view. Each transaction may involve either a single message or multiple message exchanges between the user and the system to complete.

2.3.1. Purpose of Use Cases

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 or the internal data representation, internal structure of the software, etc. A use case typically represents a sequence of interactions between the user and the system. These interactions consist of one mainline sequence. The mainline sequence represents the normal interaction between a user and the system. The mainline sequence is the most occurring sequence of interaction. For example, the mainline sequence of the withdraw cash use case supported by a bank ATM drawn, complete the transaction, and get the amount. Several variations to the main line sequence may also exist. Typically, a variation from the mainline sequence occurs when some specific conditions hold. For the bank ATM example, variations or alternate scenarios may occur, if the password is invalid or the amount to be withdrawn exceeds the amount balance. The variations are also called alternative paths. A use case can be viewed as a set of related scenarios tied together by a common goal. The mainline sequence and each of the variations are called scenarios or instances of the use case. Each scenario is a single path of user events and system activity through the use case.

2.3.2. Representation of Use Cases

Use cases can be represented by drawing a use case diagram and writing an accompanying text elaborating the drawing. 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 modelled (such as Library Information System) appears inside the rectangle.

The different users of the system are represented by using the stick person icon. Each stick person icon is normally referred to as an actor. An actor is a role played by a user with respect to the system use. It is possible that the same user may play the role of multiple actors. Each actor can participate in one or more use cases. The line connecting the actor and the use case is called

the communication relationship. It indicates that the actor makes use of the functionality provided by the use case. Both the human users and the 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>>.

Example

The use case model for the Tic-Tac-Toe problem is shown in fig. 37.10. This software has only one use case "play move". Note that the use case "get-user-move" is not used here. The name "get-user-move" would be inappropriate because the use cases should be named from the users' perspective.

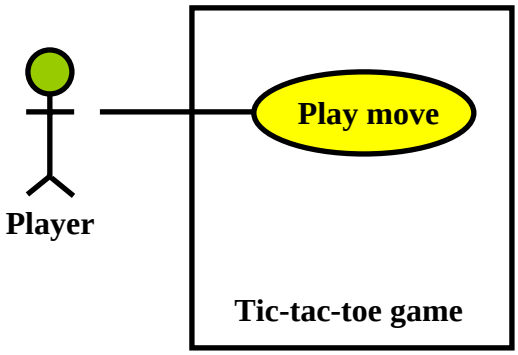

Fig. 37.10 Use case model for tic-tac-toe game

Text Description

Each ellipse on the 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 and other aspects of the use case. It should include all the behaviour associated with the use case in terms of the mainline sequence, different variations to the normal behaviour, 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 recommended. 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 alternative scenarios.

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

Actors: In addition to identifying the actors, some information about actors using this use case which may help the implementation of the use case may be recorded.

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 only the domain-related errors such as lack of user's access rights, invalid entry in the input fields, etc. Obviously, errors that are not domain related, such as software errors, need not be discussed here.

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.

2.3.3. Utility of Use Cases

Use cases (represented by ellipses) along with the accompanying text description serve as a type of requirements specification of the system and form the core model to which all other models must conform. But, what about the actors (stick person icons)? One possible use of identifying the different types of users (actors) is in identifying and implementing a security mechanism through a login system, so that each actor can involve only those functionalities to which he is entitled to. 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.

2.3.4. Factoring of Commonality Among Use Cases

It is often desirable to factor use cases into component use cases. Actually, factoring of use cases is required under two situations. First, 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 sized (A4) paper. Secondly, 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 whenever required. It is desirable to factor out common usage such as error handling from a set of use cases. This makes analysis of the class design much simpler and elegant. However, a word of caution here. Factoring of use cases should not be done except for achieving the above two objectives. From the design point of view, it is not advantageous to break up a use case into many smaller parts just for the sake of it.

UML offers three mechanisms for factoring of use cases, as follows:

Generalization

Use case generalization can be used when one use case is similar to another, but does something slightly differently or something more. Generalization works the same way with use cases as it does with classes. The child use case inherits the behaviour and meaning of the parent use case. The notation is the same too (as shown in fig. 37.11). It is important to remember that the base and the derived use cases are separate use cases and should have separate text descriptions.

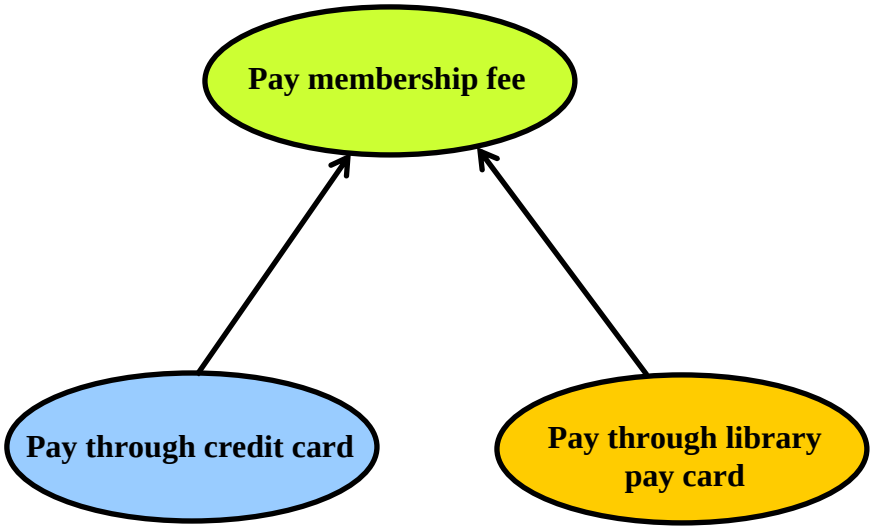

Fig. 37.11 Representation of use case generalization

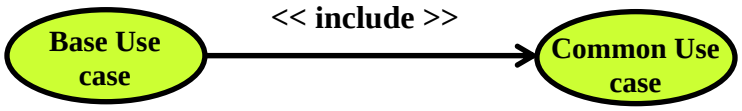

Fig. 37.12 Representation of use case inclusion

Includes

The *includes* relationship in the older versions of UML (prior to UML 1.1) was known as the uses relationship. The *includes* relationship involves one use case including the behaviour of another use case in its sequence of events and actions. The *includes* relationship occurs when a chunk of behaviour 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 cases into more manageable parts. As shown in fig. 37.12, 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 cases. As shown in example fig. 37.13, issue-book and renew-book both include check-reservation use case. The base use case may include several use cases. In such cases, it may interleave their associated common use cases together. The common use case becomes a separate use case and the independent text description should be provided for it.

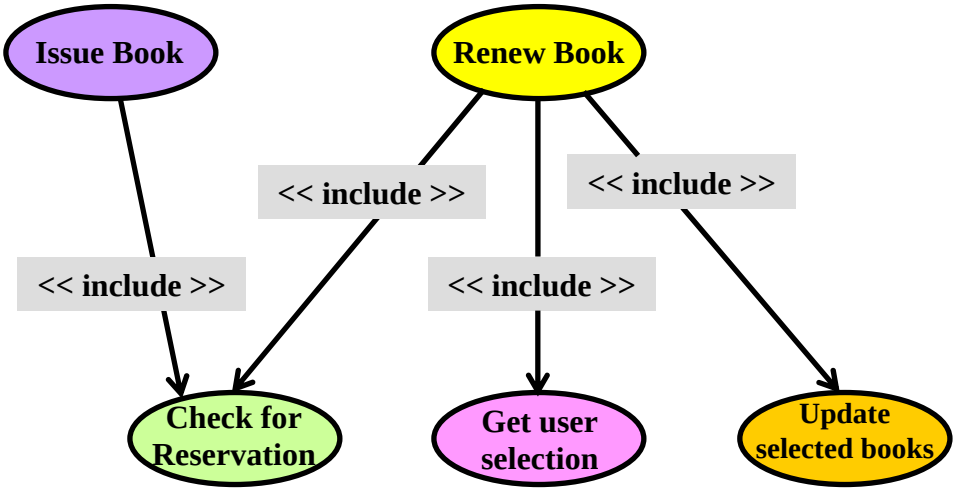

Fig. 37.13 Example use case inclusion

Extends

The main idea behind the *extends* relationship among the use cases is that it allows you to show optional system behaviour. An optional system behaviour is extended only under certain conditions. This relationship among use cases is also predefined as a stereotype as shown in fig.37.14. The *extends* relationship is similar to generalization. But unlike generalization, the extending use case can add additional behaviour only at an extension point only when certain conditions are satisfied. The extension points are points within the use case where variation to the mainline (normal) action sequence may occur. The *extends* relationship is normally used to capture alternate paths or scenarios.

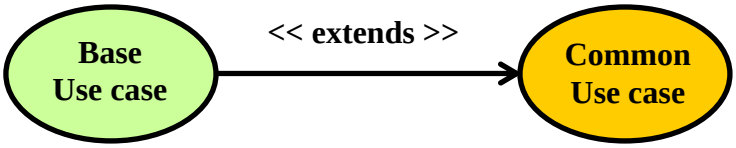

Fig. 37.14 Representation of use case extension

Organization of Use Cases

When the use cases are factored, they are organized hierarchically. The high-level use cases are refined into a set of smaller and more refined use cases as shown in fig. 37.15. Top-level use cases are super-ordinate to the refined use cases. The refined use cases are sub-ordinate to the top-level use cases. Note that only the complex use cases should be decomposed and organized in a hierarchy. It is not necessary to decompose simple use cases. The functionality of the super-ordinate use cases is traceable to their sub-ordinate use cases. Thus, the functionality provided by the super-ordinate use cases is composite of the functionality of the sub-ordinate use cases. In the highest level of the use case model, only the fundamental use cases are shown. The focus is on the application context. Therefore, this level is also referred to as the context diagram. In the context diagram, the system limits are emphasized. The top-level diagram contains only those use cases with which the external users of the system interact. The subsystem-level use cases specify the services offered by the subsystems to the other subsystems. Any number of levels involving the subsystems may be utilized. In the

lowest level of the use case hierarchy, the class-level use cases specify the functional fragments or operations offered by the classes.

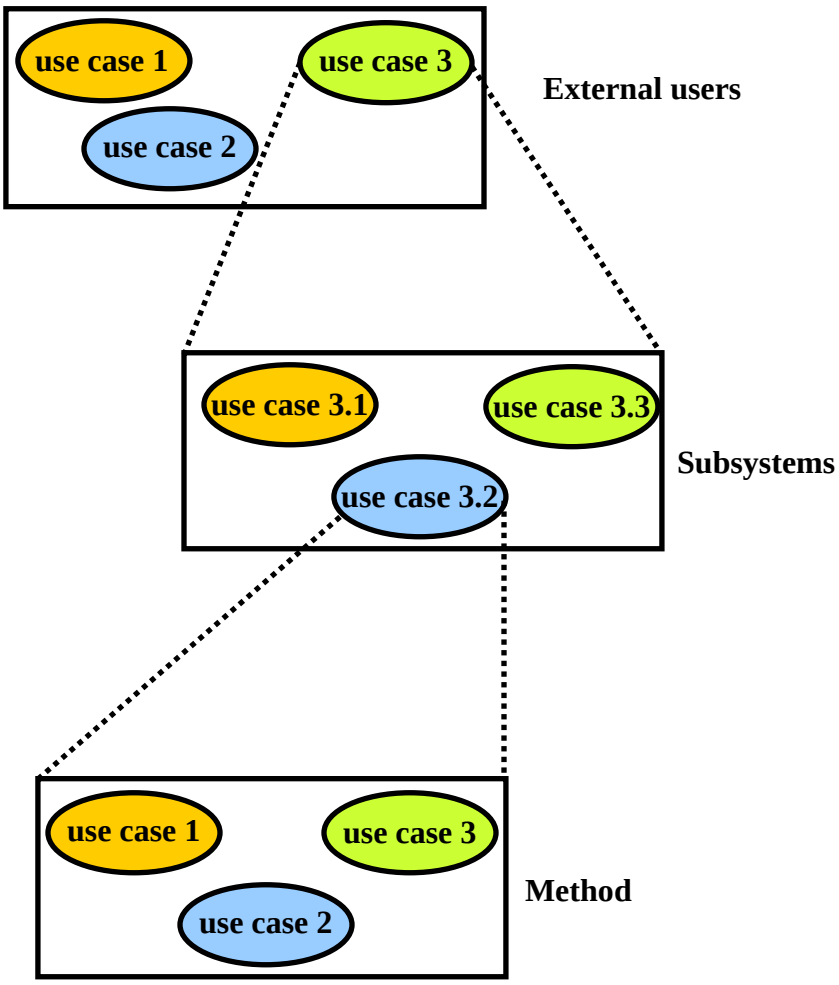

Fig. 37.15 Hierarchical organization of use cases

2.4. Class Diagrams

A class diagram describes the static structure of a system. It shows how a system is structured rather than how it behaves. The static structure of a system comprises of a number of class diagrams and their dependencies. The main constituents of a class diagram are classes and their relationships: generalization, aggregation, association, and various kinds of dependencies.

The classes represent entities with common features, i.e. attributes and operations. Classes are represented as solid outline rectangles 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. The class names are usually chosen to be singular nouns. An example of a class is shown in fig. 37.1.2. Classes have optional attributes and operations compartments. A class may appear on several diagrams. Its attributes and operations are suppressed on all but one diagram.

2.4.1. Association

Associations are needed to enable objects to communicate with each other. An association describes a connection between classes. The association relation between two objects is called object connection or link. Links are instances of associations. A link is a physical or conceptual connection between object instances. For example, suppose *Amit* has borrowed the book *Graph Theory*. Here, **borrowed** is the connection between the objects *Amit* and *Graph Theory* book. Mathematically, a link can be considered to be a tuple, i.e. an ordered list of object instances. An association describes a group of links with a common structure and common semantics. For example, consider the statement that Library Member borrows Books. Here, **borrows** is the association between the class *LibraryMember* and the class *Book*. Usually, an association is a binary relation (between two classes). However, three or more different classes can be involved in an association. A class can have an association relationship with itself (called recursive association). In this case, it is usually assumed that two different objects of the class are linked by the association relationship.

Association between two classes is represented by drawing a straight line between the concerned classes. Fig. 37.16 illustrates the graphical representation of the association relation. The name of the association is written along side the association line. An arrowhead may be placed on the association line to indicate the reading direction of the association. 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 noted as an individual number or as a value range. The multiplicity indicates how many instances of one class are associated with each other. Value ranges of multiplicity are noted by specifying the minimum and maximum value, separated by two dots, e.g. 1..5. An asterisk is a wild card and means many (zero or more). The association of fig. 37.16 should be read as "Many books may be borrowed by a Library Member". Observe that associations (and links) appear as verbs in the problem statement.

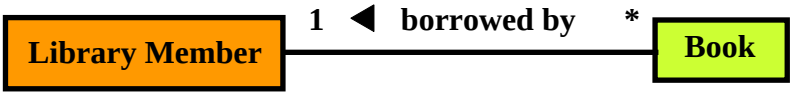

Fig. 37.16 Association between two classes

Associations are usually realized by assigning appropriate reference attributes to the classes involved. Thus, associations can be implemented using pointers from one object class to another. Links and associations can also be implemented by using a separate class that stores which objects of a class are linked to which objects of another class. Some CASE tools use the role names of the association relation for the corresponding automatically generated attribute.

2.4.2. Aggregation

Aggregation is a special type of association where the involved classes represent a whole-part relationship. The aggregate takes the responsibility of forwarding messages to the appropriate parts. Thus, the aggregate takes the responsibility of delegation and leadership. When an instance of one object contains instances of some other objects, then aggregation (or composition) relationship exists between the composite object and the component object. Aggregation is

represented by the diamond symbol at the composite end of a relationship. The number of instances of the component class aggregated can also be shown as in fig. 37.17 (a).

Fig. 37.17(a) Representation of aggregation

The aggregation relationship cannot be reflexive (i.e. recursive). That is, an object cannot contain objects of the same class as itself. Also, the aggregation relation is not symmetric. That is, two classes A and B cannot contain instances of each other. However, the aggregation relationship can be transitive. In this case, aggregation may consist of an arbitrary number of levels.

2.4.3. Composition

Composition is a stricter form of aggregation, in which the parts are existence-dependent on the whole. This means that the life of the parts are closely tied to the life of the whole. When the whole is created, the parts are created and when the whole is destroyed, the parts are destroyed. A typical example of composition is an invoice object with invoice items. As soon as the invoice object is created, all the invoice items in it are created and as soon as the invoice object is destroyed, all invoice items in it are also destroyed. The composition relationship is represented as a filled diamond drawn at the composite-end. An example of the composition relationship is shown in fig. 37.17 (b).

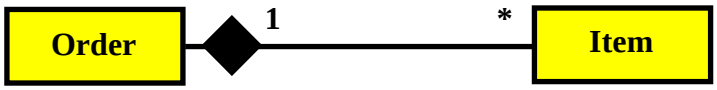

Fig. 37.17(b) Representation of composition

2.5. Interaction Diagrams

Interaction diagrams are models that describe how a group of objects collaborate to realize some behaviour. Typically, each interaction diagram realizes the behaviour of a single use case. An interaction diagram shows a number of example objects and the messages that are passed between the objects within the use case.

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. However, they are both useful. These two actually portray different perspectives of behaviour of the system and different types of inferences can be drawn from them. The interaction diagrams can be considered as a major tool in the design methodology.

2.5.1. Sequence Diagrams

A sequence diagram shows interaction among objects as a two dimensional chart. The chart is read from top to bottom. The objects participating in the interaction are shown 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 class are underlined. The objects appearing at the top signify that the object already existed when the use case execution was initiated. However, if some object is created during the execution of the 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 is called the object's lifeline. The lifeline indicates the existence of the object at any particular point of time. The rectangle drawn on the lifetime is called the activation symbol and indicates that the object is active as long as the rectangle exists. Each message is indicated as an arrow between the lifelines of two objects. The messages are shown 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. Each message is labeled with the message name. Some control information can also be included. Two types of control information are particularly valuable.

- A condition (e.g. [invalid]) indicates that a message is sent, only if the condition is true.
- An iteration marker shows the message is sent many times to multiple receiver objects as would happen when a collection or the elements of an array are being iterated. The basis of the iteration can also be indicated e.g. [for every book object].

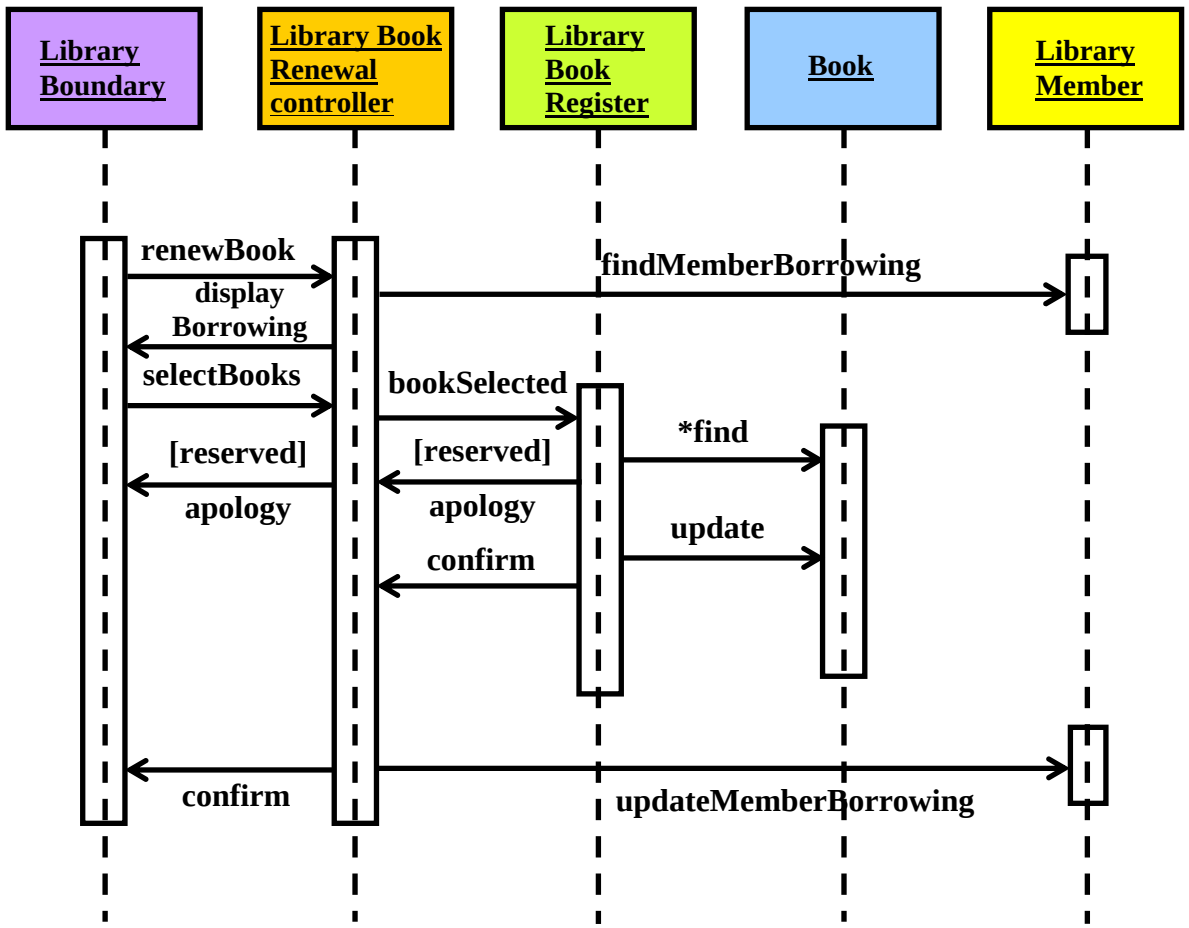

Fig. 37.18 Sequence diagram for the renew book use case

The sequence diagram for the book renewal use case for the Library Automation Software is shown in fig. 37.18. The development of the sequence diagram in the development methodology would help us in determining the responsibilities of the different classes; i.e. what methods should be supported by each class.

2.5.2. Collaboration Diagrams

A collaboration diagram shows both structural and behavioural aspects explicitly. This is unlike a sequence diagram which shows only the behavioural aspects. The structural aspect of a collaboration diagram consists of objects and the links existing between them. In this diagram, an object is also called a collaborator. The behavioural aspect is described by the set of messages exchanged among the different collaborators. The link between objects is shown as a solid line and can be used to send messages between two objects. The message is shown as a labeled arrow 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 fig. 37.18 is shown in fig. 37.19. The use of the collaboration diagrams in our development process would be to help us to determine which classes are associated with which other classes.

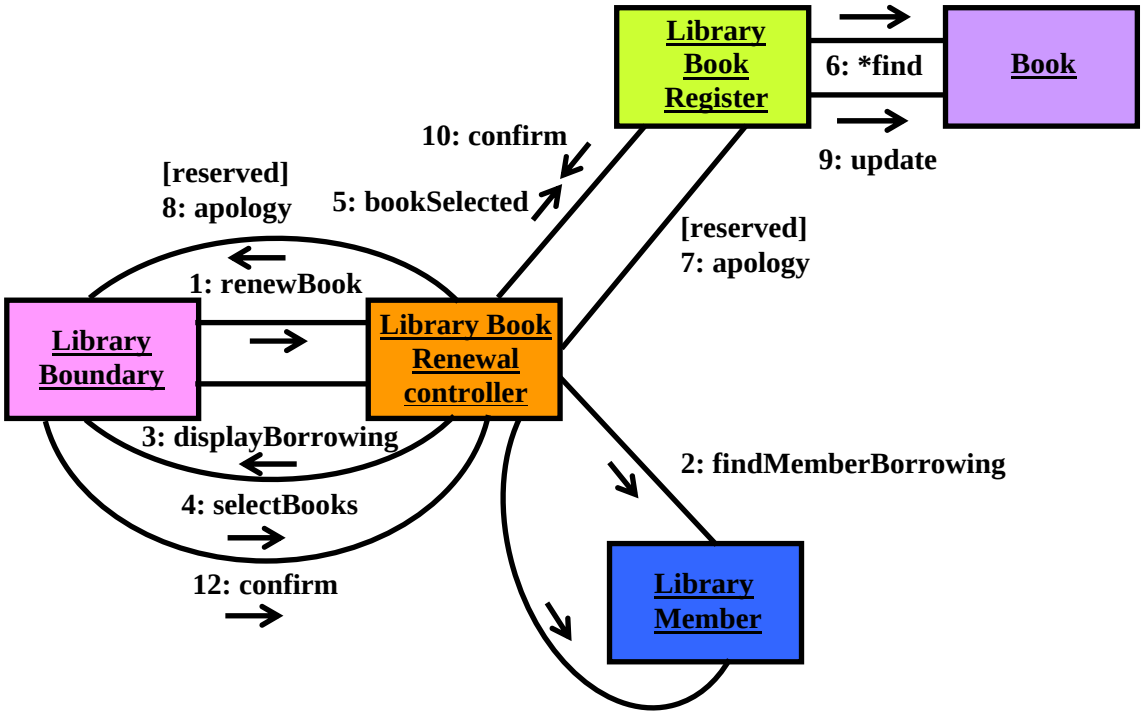

Fig. 37.19 Collaboration diagram for the renew book use case

2.6. Activity Diagrams

The activity diagram is possibly one modelling element which was not present in any of the predecessors of UML. No such diagrams were present either in the works of Booch, Jacobson, or Rumbaugh. It is possibly based on the event diagram of Odell [1992] though the notation is very different from that used by Odell. The activity diagram focuses on representing activities or chunks of processing which may or may not correspond to the methods of classes. An activity is

a state with an internal action and one or more outgoing transitions which automatically follow the termination of the internal activity. If an activity has more than one outgoing transition, then these must be identified through conditions. An interesting feature of the activity diagrams is the swim lanes. Swim lanes enable you to group activities based on who is performing them, e.g. academic department vs. hostel office. Thus swim lanes subdivide activities based on the responsibilities of some components. The activities in a swim lane can be assigned to some model elements, e.g. classes or some component, etc.

Activity diagrams are normally employed in business process modelling. This is carried out during the initial stages of requirements analysis and specification. Activity diagrams can be very useful to understand complex processing activities involving many components. Later these diagrams can be used to develop interaction diagrams which help to allocate activities (responsibilities) to classes.

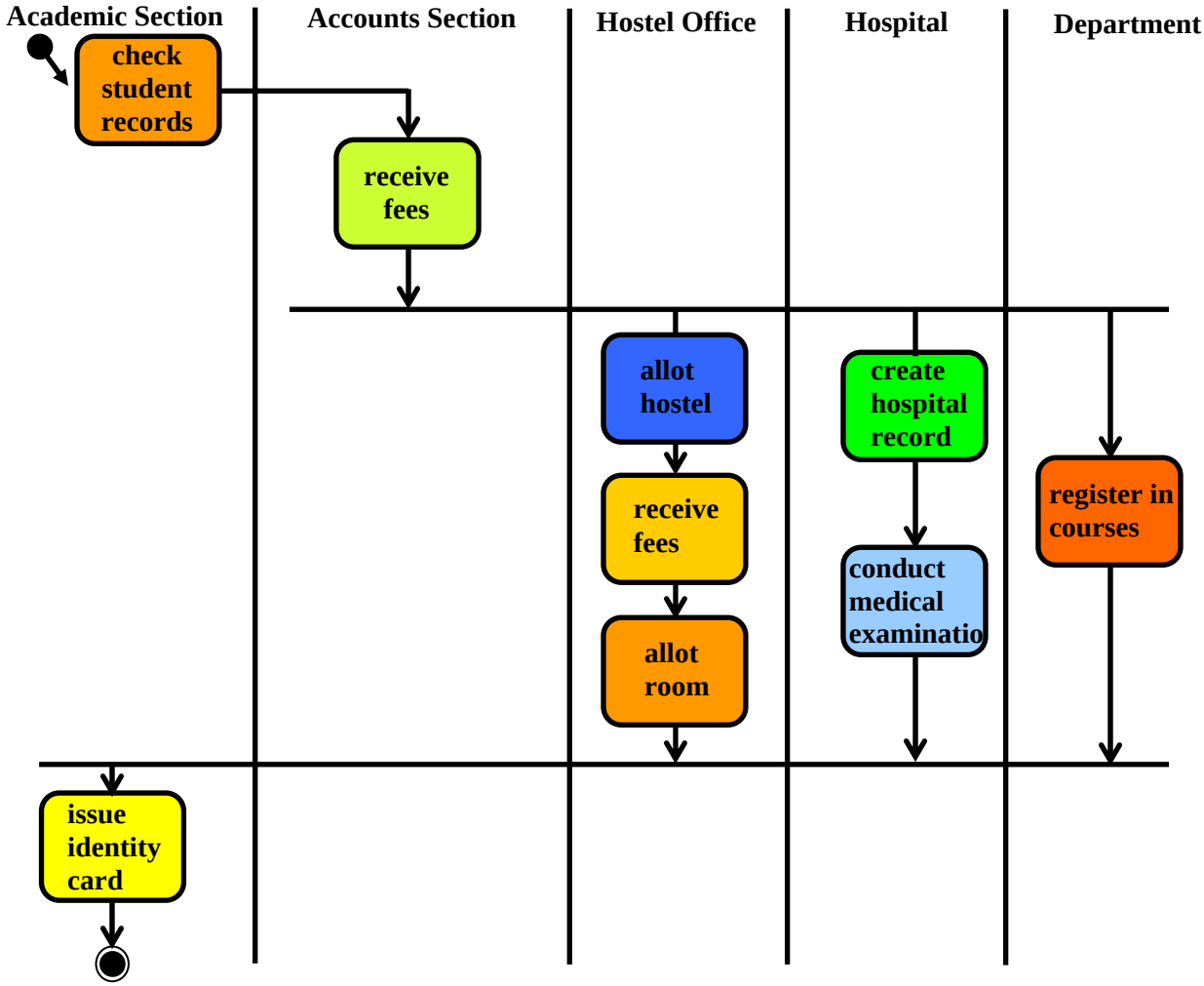

Fig. 37.20 Activity diagram for student admission procedure at IIT

The student admission process in IIT is shown as an activity diagram in fig. 37.20. This shows the part played by different components 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 are completed (this synchronization is represented as a horizontal line), the identity card can be issued to a student by the Academic section.

2.7. State Chart Diagrams

A state chart diagram is normally used to model how the state of an object changes in its lifetime. State chat 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, state chart diagram is not appropriate. State chart diagrams are based on the finite state machine (FSM) formalism. An FSM consists of a finite number of states corresponding to those of the object being modelled. The object undergoes state changes when specific events occur. The FSM formalism existed long before the object-oriented technology and has been used for a wide variety of applications. Apart from modelling, it has even been used in theoretical computer science as a generator for regular languages.

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. 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 quickly and are not interruptible. Activities are associated with states and can take a longer time. An activity can be interrupted by an event.

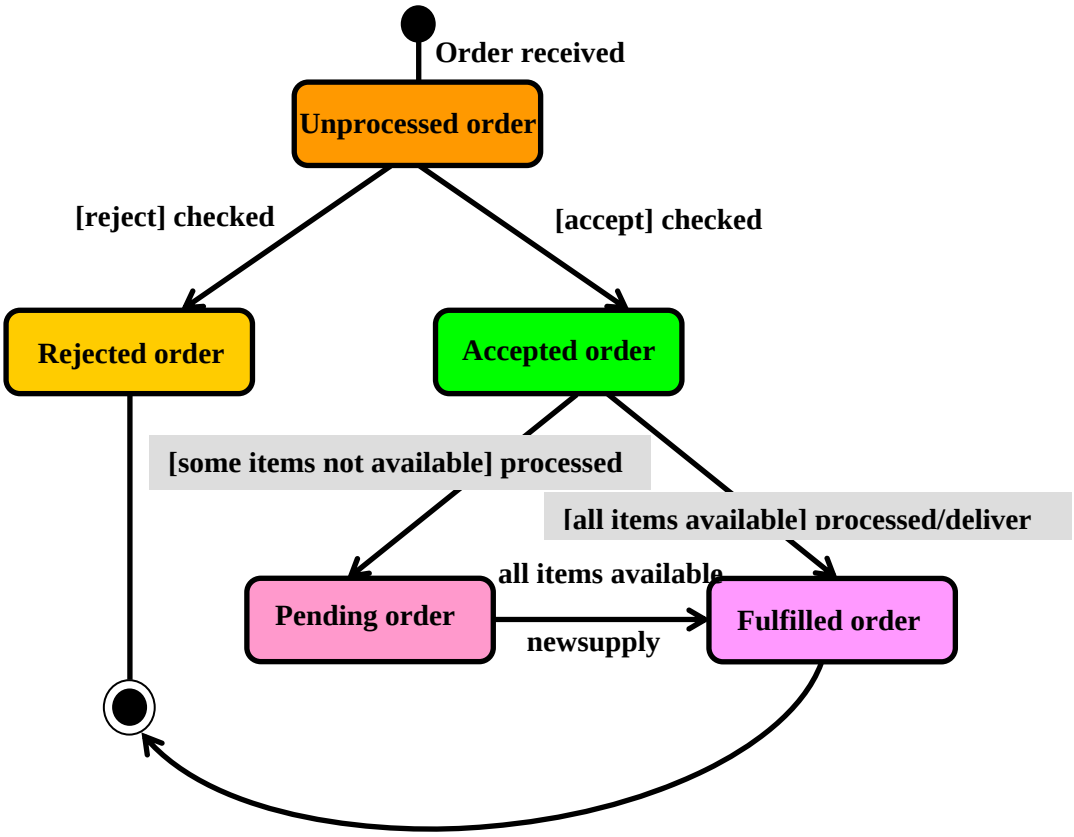

Fig. 37.21 State chart diagram for an order object

The basic elements of the state chart diagram are as follows:

- **Initial state.** This is represented as 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 along side the arrow. A guard to the transition can also be assigned. A guard is a Boolean logic condition. The transition can take place only if the grade evaluates to true. The syntax for the label of the transition is shown in 3 parts: event[guard]/action.

An example state chart for the order object of the Trade House Automation software is shown in fig. 37.21.

3. Object-Oriented Software Development

The object-modelling concepts introduced in the earlier sections can be put together to develop an object-oriented analysis and design methodology. Object-oriented design (OOD) advocates a radically different design approach compared to the traditional function-oriented design approach. OOD paradigm suggests that the natural objects (i.e. the entities) occurring in a problem should be identified first and then implemented. Object-oriented design techniques not only identify objects, but also identify the internal details of these identified objects. Also, the relationships existing among different objects are identified and represented in such a way that the objects can be easily implemented using a programming language.

The term object-oriented analysis (OOA) refers to a method of developing an initial model of the software from the requirements specification. The analysis model is refined into a design model. The design model can be implemented using a programming language. The term object-oriented programming refers to the implementation of programs using object-oriented concepts.

3.1. Design Patterns

Design patterns are reusable solutions to problems that recur in many applications. A pattern serves as a guide for creating a “good” design. Patterns are based on sound common sense and the application of fundamental design principles. These are created by people who spot repeating themes across designs. The pattern solutions are typically described in terms of class and interaction diagrams. Examples of design patterns are expert pattern, creator pattern, controller pattern, etc.

In addition to providing the model of a good solution, design patterns include a clear specification of the problem, and also explain the circumstances in which the solution would and would not work. Thus, a design pattern has four important parts:

- The problem
- The context in which the problem occurs
- The solution
- The context within which the solution works

3.1.1. Design Pattern Solutions

The design pattern solutions are typically described in terms of class and interaction diagrams.

Expert Pattern

Problem: Which class should be responsible for doing certain things?

Solution: Assign responsibility to the information expert – the class that has the information necessary to fulfill the required responsibility. The expert pattern expresses the common intuition that objects do things related to the information they have. The class diagram and collaboration diagrams for this solution to the problem of which class should compute the total sales is shown in the fig. 37.1.1.

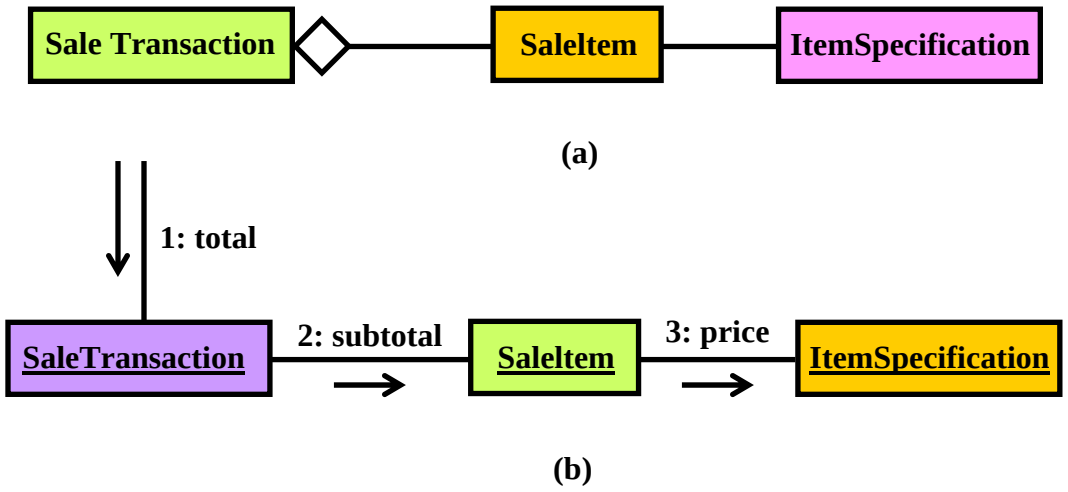

Fig. 37.22 Expert pattern: (a) Class diagram (b) Collaboration diagram

Creator Pattern

Problem: Which class should be responsible for creating a new instance of some class?
Solution: Assign a class C1 the responsibility to create an instance of class C2, if one or more of the following are true:

- C1 is an aggregation of objects of type C2
- C1 contains objects of type C2
- C1 closely uses objects of type C2
- C1 has the data that would be required to initialize the objects of type C2, when they are created

Controller Pattern

Problem: Who should be responsible for handling the actor requests?
Solution: For every use case, there should be a separate controller object which would be responsible for handling requests from the actor. Also, the same controller should be used for all the actor requests pertaining to one use case so that it becomes possible to maintain the necessary information about the state of the use case. The state information maintained by a controller can be used to identify the out-of-sequence actor requests, e.g. whether voucher request is received before arrange payment request.
Model View Separation Pattern
Problem: How should the non-GUI classes communicate with the GUI classes?
Context in which the problem occurs: This is a very commonly occurring pattern which is found in almost every problem. Here, model is a synonym for the domain layer objects, view is a synonym for the presentation layer objects such as the GUI objects.

Solution: The model view separation pattern states that model objects should not have direct knowledge (or be directly coupled) of the view objects. This means that there should not be any direct calls from other objects to the GUI objects. This results in a good solution, because the GUI classes are related to a particular application whereas the other classes may be reused.

There are actually two solutions to this problem which work in different circumstances. These are as follows:

Solution 1: Polling or Pull from above

It is the responsibility of a GUI object to ask for the relevant information from the other objects, i.e. the GUI objects pull the necessary information from the other objects whenever required.

This model is frequently used. However, it is inefficient for certain applications. For example, simulation applications which require visualization, the GUI objects would not know when the necessary information becomes available. Other examples are, monitoring applications such as network monitoring, stock market quotes, and so on. In these situations, a "push-from-below" model of display update is required. Since "push-from-below" is not an acceptable solution, an indirect mode of communication from the other objects to the GUI objects is required.

Solution 2: Publish- subscribe pattern

An event notification system is implemented through which the publisher can indirectly notify the subscribers as soon as the necessary information becomes available. An event manager class can be defined as one which keeps track of the subscribers and the types of events they are interested in. An event is published by the publisher by sending a message to the event manager object. The event manager notifies all registered subscribers usually via a parameterized message (called a callback). Some languages specifically support event manager classes. For example, Java provides the EventListener interface for such purposes.

3.2. Domain Modelling

Domain modelling is known as conceptual modelling. A domain model is a representation of the concepts or objects appearing in the problem domain. It also captures the obvious relationships among these objects. Examples of such conceptual objects are the *Book, BookRegister, MemeberRegister, LibraryMember,* etc. The recommended strategy is to quickly create a rough conceptual model where the emphasis is in finding the obvious concepts expressed in the requirements while deferring a detailed investigation. Later during the development process, the conceptual model is incrementally refined and extended.

The objects identified during domain analysis can be classified into three types:

- Boundary objects
- Controller objects
- Entity objects

The boundary and controller objects can be systematically identified from the use case diagram whereas identification of entity objects requires practice. So, the crux of the domain modeling activity is to identify the entity models.

3.2.1. Boundary objects

The boundary objects are those with which the actors interact. These include screens, menus, forms, dialogs, etc. The boundary objects are mainly responsible for user interaction. Therefore,

they normally do not include any processing logic. However, they may be responsible for validating inputs, formatting, outputs, etc. The boundary objects were earlier being called the interface objects. However, the term interface class is being used for Java, COM/DCOM, and UML with different meaning. A recommendation for the initial identification of the boundary classes is to define one boundary class per actor/use case pair.

3.2.2. Entity objects

These normally hold information such as data tables and files that need to outlive use case execution, e.g. Book, BookRegister, LibraryMember, etc. Many of the entity objects are “dumb servers”. They are normally responsible for storing data, fetching data, and doing some fundamental kinds of operation that do not change often.

3.2.3. Controller objects

The controller objects coordinate the activities of a set of entity objects and interface with the boundary objects to provide the overall behavior of the system. The responsibilities assigned to a controller object are closely related to the realization of a specific use case. The controller objects effectively decouple the boundary and entity objects from one another making the system tolerant to changes of the user interface and processing logic. The controller objects embody most of the logic involved with the use case realization (this logic may change from time to time). A typical interaction of a controller object with boundary and entity objects is shown in fig. 37.22. Normally, each use case is realized using one controller object. However, some use cases can be realized without using any controller object, i.e. through boundary and entity objects only. This is often true for use cases that achieve only some simple manipulation of the stored information.

3.2.4. Example

Let’s consider the “query book availability” use case of the Library Information System (LIS). Realization of the use case involves only matching the given book name against the books available in the catalog. More complex use cases may require more than one controller object to realize the use case. A complex use case can have several controller objects such as transaction manager, resource coordinator, and error handler. There is another situation where a use case can have more than one controller object. Sometimes the use cases require the controller object to transit through a number of states. In such cases, one controller object might have to be created for each execution of the use case.

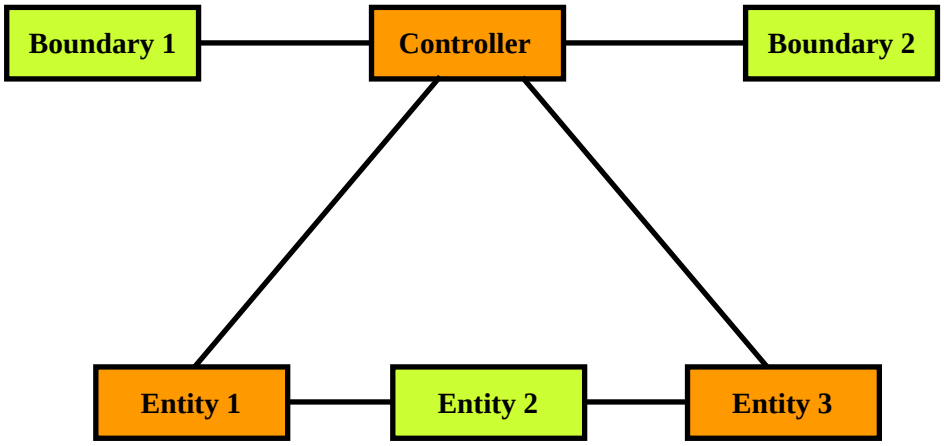

Fig. 37.23 A typical realization of a use case through the collaboration of boundary, controller, and entity objects

3.2.5. Identification of Entity Objects

One of the most important steps in any object-oriented design methodology is the identification of objects. In fact, the quality of the final design depends to a great extent on the appropriateness of the objects identified. However, to date no formal methodology exists for identification of objects. Several semi-formal and informal approaches have been proposed for object identification. These can be classified into the following broad classes:

- Grammatical analysis of the problem description
- Derivation from data flow
- Derivation from the entity relationship (E-R) diagram

A widely accepted object identification approach is the grammatical analysis approach. Grady Booch originated the grammatical analysis approach [1991]. In Booch's approach, the nouns occurring in the extended problem description statement (processing narrative) are mapped to objects and the verbs are mapped to methods.

3.3. Booch's Object Identification Method

Booch's object identification approach requires a processing narrative of the given problem to be first developed. The processing narrative describes the problem and discusses how it can be solved. The objects are identified by noting down the nouns in the processing narrative. Synonym of a noun must be eliminated. If an object is required to implement a solution, then it is said to be part of the solution space. Otherwise, if an object is necessary only to describe the problem, then it is said to be a part of the problem space. However, several of the nouns may not be objects. An imperative procedure name, i.e., noun form of a verb actually represents an action and should not be considered as an object. A potential object found after lexical analysis is usually considered legitimate, only if it satisfies the following criteria:

Retained information: Some information about the object should be remembered for the system to function. If an object does not contain any private data, it can not be expected to play any important role in the system.

Multiple attributes: Usually objects have multiple attributes and support multiple methods. It is very rare to find useful objects which store only a single data element or support only a

single method, because an object having only a single data element or method is usually implemented as a part of another object.

Common operations: A set of operations can be defined for potential objects. If these operations apply to all occurrences of the object, then a class can be defined. An attribute or operation defined for a class must apply to each instance of the class. If some of the attributes or operations apply only to some specific instances of the class, then one or more subclasses can be needed for these special objects.

Normally, the actors themselves and the interactions among themselves should be excluded from the entity identification exercise. However, some times there is a need to maintain information about an actor within the system. This is not the same as modeling the actor. These classes are sometimes called surrogates. For example, in the Library Information System (LIS) we would need to store information about each library member. This is independent of the fact that the library member also plays the role of an actor of the system.

Although the grammatical approach is simple and intuitively appealing, yet through a naive use of the approach, it is very difficult to achieve high quality results. In particular, it is very difficult to come up with useful abstractions simply by doing grammatical analysis of the problem description. Useful abstractions usually result from clever factoring of the problem description into independent and intuitively correct elements.

3.3.1. An Example: Tic-Tac-Toe

Tic-tac-toe is a computer game in which a human player and the computer make alternative moves on a 3 x 3 square. A move consists of marking a previously unmarked square. A player who first places three consecutive marks along a straight line (i.e., along a row, column, or diagonal) on the square, wins the game. As soon as either the human player or the computer wins, a message congratulating the winner should be displayed. If neither player manages to get three consecutive marks along a straight line, but all the squares on the board are filled up, then the game is drawn. The computer always tries to win a game.

By performing a grammatical analysis of this problem statement, it can be seen that nouns have been underlined in the problem description and the actions or verbs have been italicized. However, on closer examination synonyms can be eliminated from the identified nouns. The list of nouns after eliminating the synonyms is the following: *Tic-tac-toe, computer game, human player, move, square, mark, straight line, board, row, column, and diagonal.*

From this list of possible objects, nouns can be eliminated e.g. *human player*, as it does not belong to the problem domain. Also, the nouns *square, game, computer, Tic-tac-toe, straight line, row, column,* and *diagonal* can be eliminated, as any data and methods can not be associated with them. The noun *move* can also be eliminated from the list of potential objects since it is an imperative verb and actually represents an action. Thus, there is only one object left – ***board***.

After being experienced in object identification, it is not normally necessary to really identify all nouns in the problem description by underlining them or actually listing them down, and systematically eliminate the non-objects to arrive at the final set of objects.

The step-by-step workout of the analysis and design procedure is given as follows:

- The use case model is shown in fig 37.10.
- The initial domain model is shown in fig 37.23(a).
- The domain model after adding the boundary and control classes is shown in fig 37.23(b).
- Sequence diagram for the play move use case is shown in fig. 37.25.

- Class diagram is shown in fig. 37.24. The messages of the sequence diagram have been populated as methods of the corresponding classes.

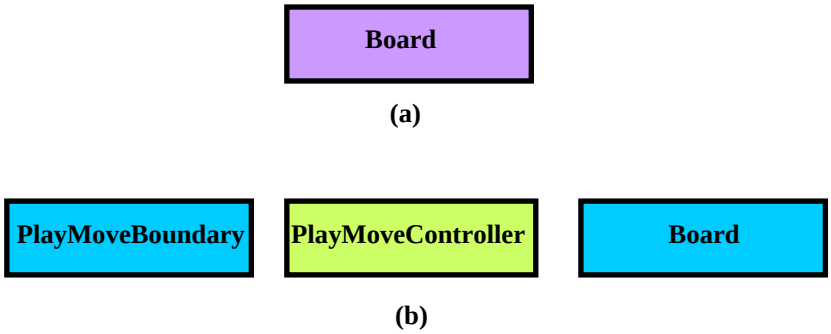

Fig. 37.24 (a) Initial domain model (b) Refined domain model

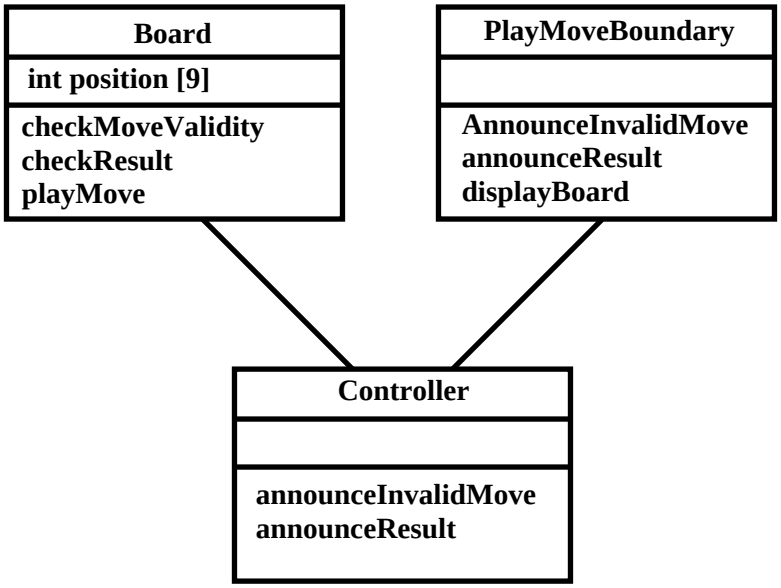

Fig. 37.25 Class diagram

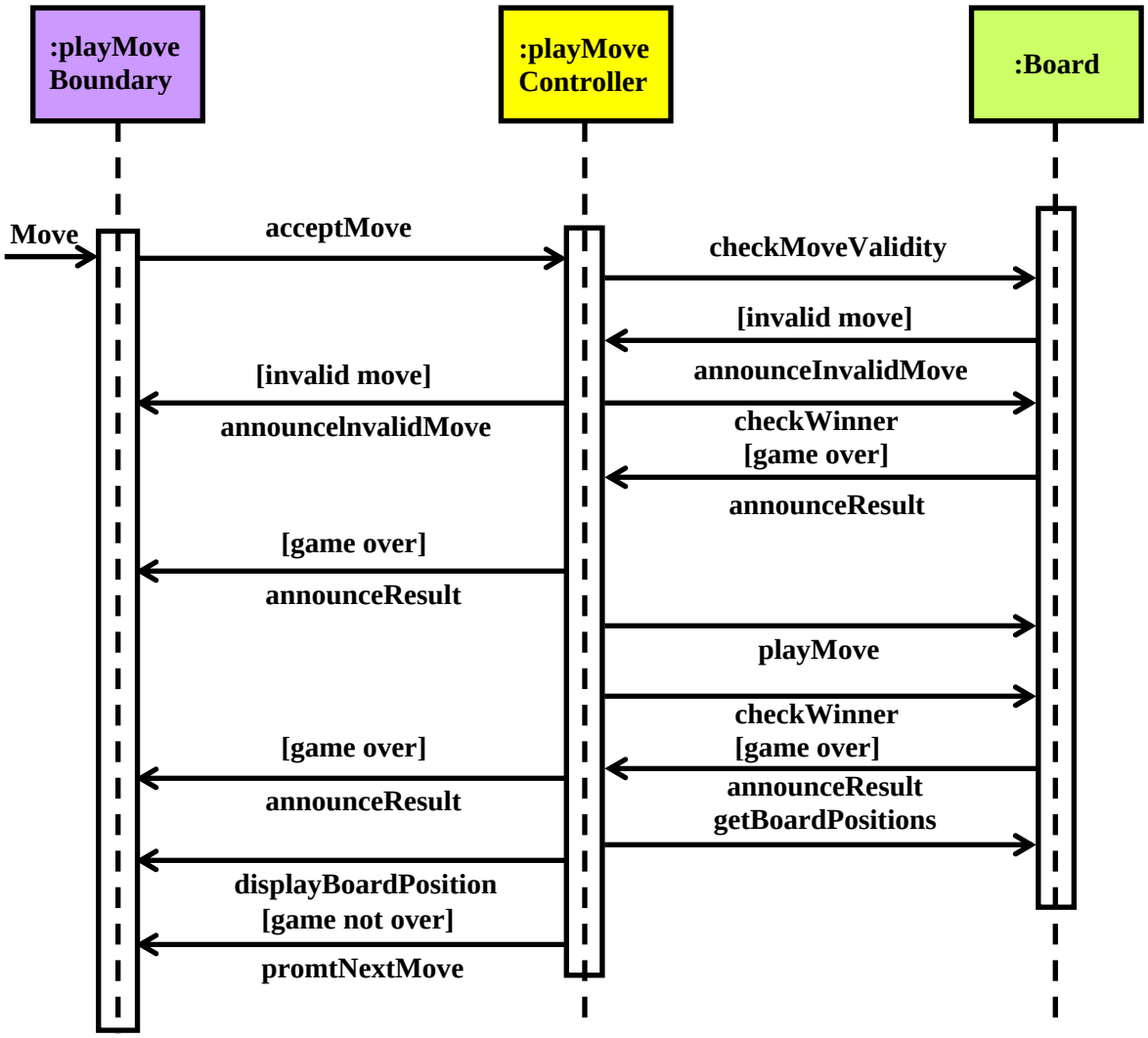

Fig. 37.26 Sequence diagram for the play move use case

4. Exercises

1. Mark the following as True or False. Justify your answer.
   a. All software engineering principles are backed by either scientific basis or theoretical proof.
   b. Data abstraction helps in easy code maintenance and code reuse.
   c. Classes can be considered equivalent to Abstract Data Types (ADTs).
   d. The inheritance relationship describes '*has a*' relationship among classes.
   e. Inheritance feature of the object oriented paradigm helps in code reuse.
   f. An important advantage of polymorphism is facilitation of reuse.
   g. Using dynamic binding a programmer can send a generic message to a set of objects which may be of different types i.e. belonging to different classes.
   h. In dynamic binding, address of an invoked method is known only at the compile time
   i. For any given problem, one should construct all the views using all the diagrams provided by UML.
   j. Use cases are explicitly dependent among themselves.

k. Each actor can participate in one and only one use case.
l. Class diagrams developed using UML can serve as the functional specification of a system.
m. The terms method and operation are equivalent concepts and can be used interchangeably.
n. The aggregation relationship can be recursively defined, i.e. an object can contain instances of itself.
o. In a UML class diagram, the aggregation relationship defines an equivalence relationship among objects.
p. The aggregation relationship can be considered to be a special type of association relationship.
q. Normally, you use an interaction diagram to represent how the behaviour of an object changes over its life time.
r. The interaction diagrams can be effectively used to describe how the behaviour of an object changes across several use cases.
s. A state chart diagram is good at describing behaviour that involves multiple objects cooperating with each other to achieve some behaviour.
t. Facade pattern tells how non-GUI classes should communicate with the GUI classes.
u. The use cases should be tightly tied to the GUI.
v. The responsibilities assigned to a controller object are closely related to the realization of a specific use case.
w. There is a one-to-one correspondence between the classes of the domain model and the final class diagram.
x. A large number of message exchanges between objects indicates good delegation and is a sure sign of a design well-done.
y. Deep class hierarchies are the hallmark of any good OOD.
z. Cohesiveness of the data and methods within a class is a sign of good OOD.

2. For the following, mark all options which are true.
   a. In the object-oriented approach, each object essentially consists of
      - some data that are private to the object
      - a set of functions (or operations) that operate on those data
      - the set of methods it provides to the other objects for accessing and manipulating the data
      - none of the above
   b. Redefinition of methods in a derived class which existed in the base class is called
      - function overloading
      - operator overloading
      - method overriding
      - none of the above
   c. The mechanism by which a subclass inherits attributes and methods from more than one base class is called
      - single inheritance
      - multiple inheritance
      - multi-level inheritance
      - hierarchical inheritance
   d. In the object-oriented approach, the same message can result in different actions when received by different objects. This feature is referred to as
      - static binding

- dynamic binding
- genericity
- overloading

e. UML is
- a language to model syntax
- an object-oriented development methodology
- an automatic code generation tool
- none of the above

f. In the context of use case diagram, the stick person icon is used to represent
- human users
- external systems
- internal systems
- none of the above

g. The design pattern solutions are typically described in terms of
- class diagrams
- object diagrams
- interaction diagrams
- both class and interaction diagrams

h. The class that should be responsible for doing certain things for which it has the necessary information – is the solution proposed by
- creator pattern
- controller pattern
- expert pattern
- facade pattern

i. The class that should be responsible for creating a new instance of some class – is the solution proposed by
- creator pattern
- controller pattern
- expert pattern
- facade pattern

j. The objects identified during domain analysis can be classified into
- boundary objects
- controller objects
- entity objects
- all of the above

k. The most critical part of the domain modelling activity is to identify
- controller objects
- boundary objects
- entity objects
- none of the above

l. The objects which effectively decouple the boundary and entity objects from one another making the system tolerant to changes of the user interface and processing logic are
- controller objects
- boundary objects
- entity objects

- none of the above

3. What is the basic difference between a class and its object? Also, identify the basic difference between methods and messages.
4. Explain what you understand by data abstraction. Identify its advantages.
5. Explain the different types of inheritance with examples. Identify the advantages of inheritance.
6. Explain encapsulation in the context of OO programming. State the advantages of encapsulation.
7. Identify the differences between static binding and dynamic binding. What are the advantages of dynamic binding?
8. Explain the advantages of object-oriented design.
9. Explain the need of a model in the context of software development.
10. Describe the different types of views of a system captured by UML diagrams.
11. What is the purpose of a use case? What is the necessity for developing use case diagram?
12. Which diagrams in UML capture the behavioural view of the system? Which UML diagrams capture the structural aspects of a system?
13. Which UML diagrams capture the important components of the system and their dependencies?
14. Represent the following relations among classes using UML diagram.
    a. Students credit 5 courses each semester. Each course is taught by one or more teachers.
    b. Bill contains a number of items. Each item describes some commodity, the price of unit, and total price.
    c. An order consists of one or more order items. Each order item contains the name of the item, its quantity and the date by which it is required. Each order item is described by an item type specification object having details such as its vendor addresses, its unit price, and the manufacturer.
15. How should you identify use cases of a system?
16. What is the difference between an operation and a method in the context of OOD technique?
17. What does the association relationship among classes represent? Give examples of the association relationship.
18. What does aggregation relationship between classes represent? Give examples of aggregation relationship between classes.
19. Why are objects always passed by reference in all popular programming languages?
20. What are design patterns? What are the advantages of using design patterns? Write down some popular design patterns and their necessities.
21. Give an outline of object-oriented development process.
22. What is meant by domain modelling? Differentiate the different types of objects that are identified during domain analysis.

References (Lessons 29 - 33)

1. Sommerville, Software Engineering, Addison Wesley, Reading, MA, USA, 2000.
2. Steve Heath, Embedded System Design: Real World Design", Butter-worth Heinemann, Newton, Mass., USA, May 2002.
3. Hatley D. and Pirbhai I., Strategies for Real-Time System Specification, Dorset House, New York, 1987.
4. Ward P.T. and Mellor S.J., Structured Development of Real-Time Systems, Yourdon Press, New York, 1985.