Module 7

Object Modeling using UML

Lesson 16

Class and Interaction Diagrams

Specific Instructional Objectives

At the end of this lesson the student will be able to:

- Explain the features represented by a class diagram.
- Explain the relationships among different types of classes by means of association.
- Explain the relationships among different types of classes by means of aggregation.
- Explain the relationships among different types of classes by means of composition.
- Draw interaction diagrams for any given problem.
- Explain the tradeoff between inheritance and aggregation/ composition
- Bring out a comparison of the three relationships: association, aggregation and composition

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.

Classes

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. 6.2 (Lesson 6.1).

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.

Attributes

An attribute is a named property of a class. It represents the kind of data that an object might contain. Attributes are listed with their names, and may optionally contain specification of their type, an initial value, and constraints. The type of the attribute is written by appending a colon and the type name after the attribute name. Typically, the first letter of a class name is a small letter. An example for an attribute is given.

```
bookName : String
```

Operation

Operation is the implementation of a service that can be requested from any object of the class to affect behaviour. An object's data or state can be changed by invoking an operation of the object. A class may have any number of operations or no operation at all. Typically, the first letter of an operation name is a small letter. Abstract operations are written in italics. The parameters of an operation (if any), may have a kind specified, which may be 'in', 'out' or 'inout'. An operation may have a return type consisting of a single return type expression. An example for an operation is given.

`issueBook(in bookName):Boolean`

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. 7.9 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. 7.9 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. 7.9: 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.

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. 7.10.

Fig. 7.10: Representation of aggregation

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.

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 closely ties 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. 7.11.

Fig. 7.11: Representation of composition

Association vs. Aggregation vs. Composition

- Association is the most general (m:n) relationship. Aggregation is a stronger relationship where one is a part of the other. Composition is even stronger than aggregation, ties the lifecycle of the part and the whole together.
- Association relationship can be reflexive (objects can have relation to itself), but aggregation cannot be reflexive. Moreover, aggregation is anti-symmetric (If B is a part of A, A can not be a part of B).
- Composition has the property of exclusive aggregation i.e. an object can be a part of only one composite at a time. For example, a **Frame** belongs to exactly one **Window** whereas in simple aggregation, a part may be shared by several objects. For example, a **Wall** may be a part of one or more **Room** objects.
- In addition, in composition, the whole has the responsibility for the disposition of all its parts, i.e. for their creation and destruction.

    - in general, the lifetime of parts and composite coincides
    - parts with non-fixed multiplicity may be created after composite itself
    - parts might be explicitly removed before the death of the composite

For example, when a **Frame** is created, it has to be attached to an enclosing **Window**. Similarly, when the **Window** is destroyed, it must in turn destroy its **Frame** parts.

Inheritance vs. Aggregation/Composition

- Inheritance describes *'is a' / 'is a kind of'* relationship between classes (base class - derived class) whereas aggregation describes *'has a'* relationship between classes. Inheritance means that the object of the derived class inherits the properties of the base class; aggregation means that the object of the whole has objects of the part. For example, the relation "cash payment *is a kind of* payment" is modeled using inheritance; "purchase order has a few items" is modeled using aggregation.

  Inheritance is used to model a "generic-specific" relationship between classes whereas aggregation/composition is used to model a "whole-part" relationship between classes.

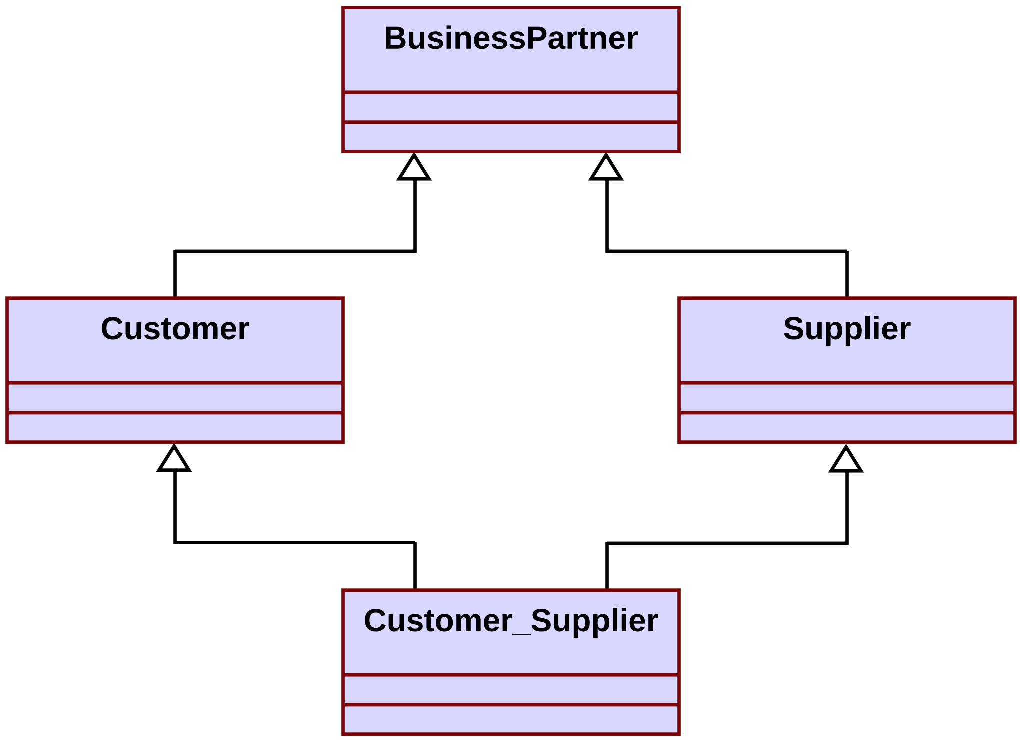
- Inheritance means that the objects of the subclass can be used anywhere the super class may appear, but not the reverse; i.e. wherever we could use instances of 'payment' in the system, we could substitute it with instances of 'cash payment', but the reverse can not be done.

- Inheritance is defined statically. It can not be changed at run-time. Aggregation is defined dynamically and can be changed at run-time. Aggregation is used when the type of the object can change over time.

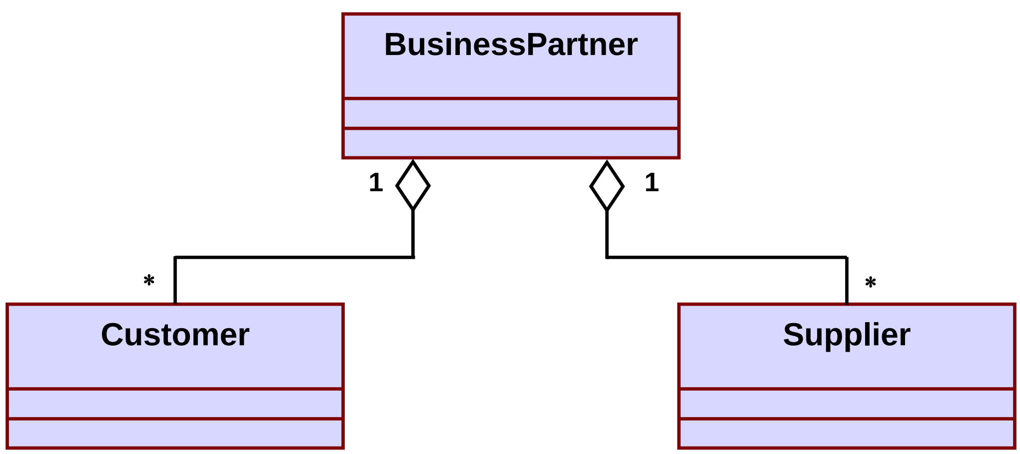
  For example, consider this situation in a business system. A **BusinessPartner** might be a **Customer** or a **Supplier** or both. Initially we might be tempted to model it as in Fig 7.12(a). But in fact, during its lifetime, a business partner might become a customer as well as a supplier, or it might change from one to the other. In such cases, we prefer aggregation instead (see Fig 7.12(b). Here, a business partner is a **Customer** if it has an aggregated **Customer** object, a **Supplier** if it has an aggregated **Supplier** object and a "C**ustomer_Supplier**" if it has both. Here, we have only two types. Hence, we are able to model it as inheritance. But what if there were several different types and combinations there of? The inheritance tree would be absolutely incomprehensible.

  Also, the aggregation model allows the possibility for a business partner to be neither - i.e. has neither a customer nor a supplier object aggregated with it.

- The advantage of aggregation is the integrity of encapsulation. The operations of an object are the interfaces of other objects which imply low implementation dependencies. The significant disadvantage of aggregation is the increase in the number of objects and their relationships. On the other hand, inheritance allows for an easy way to modify implementation for reusability. But the significant disadvantage is that it breaks encapsulation, which implies implementation dependence.



(a)



(b)

Fig. 7.12 Representation of **BusinessPartner, Customer, Supplier** relationship **(a)** using inheritance **(b)** using aggregation

Interaction Diagrams

Interaction diagrams are models that describe how group of objects collaborate to realize some behavior. Typically, each interaction diagram realizes the behavior 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 behavior 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.

Sequence Diagram

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 the 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 lifeline 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].

The sequence diagram for the book renewal use case for the Library Automation Software is shown in fig. 7.13. 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.

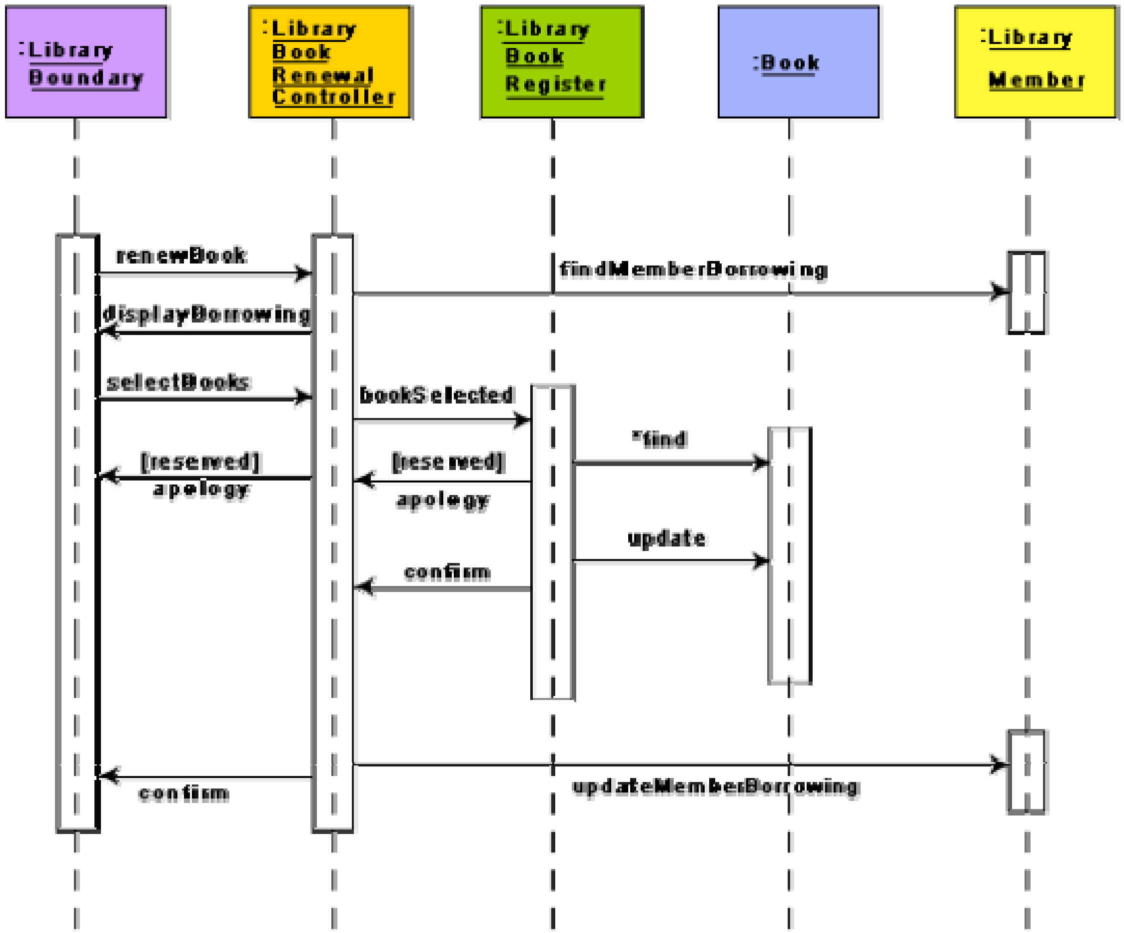


Fig. 7.13: Sequence diagram for the renew book use case

Collaboration Diagram

A collaboration diagram shows both structural and behavioral aspects explicitly. This is unlike a sequence diagram which shows only the behavioral 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 behavioral 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 only way to describe the relative sequencing of the messages in this diagram. The collaboration diagram for the example of fig. 7.13 is shown in fig. 7.14. 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.

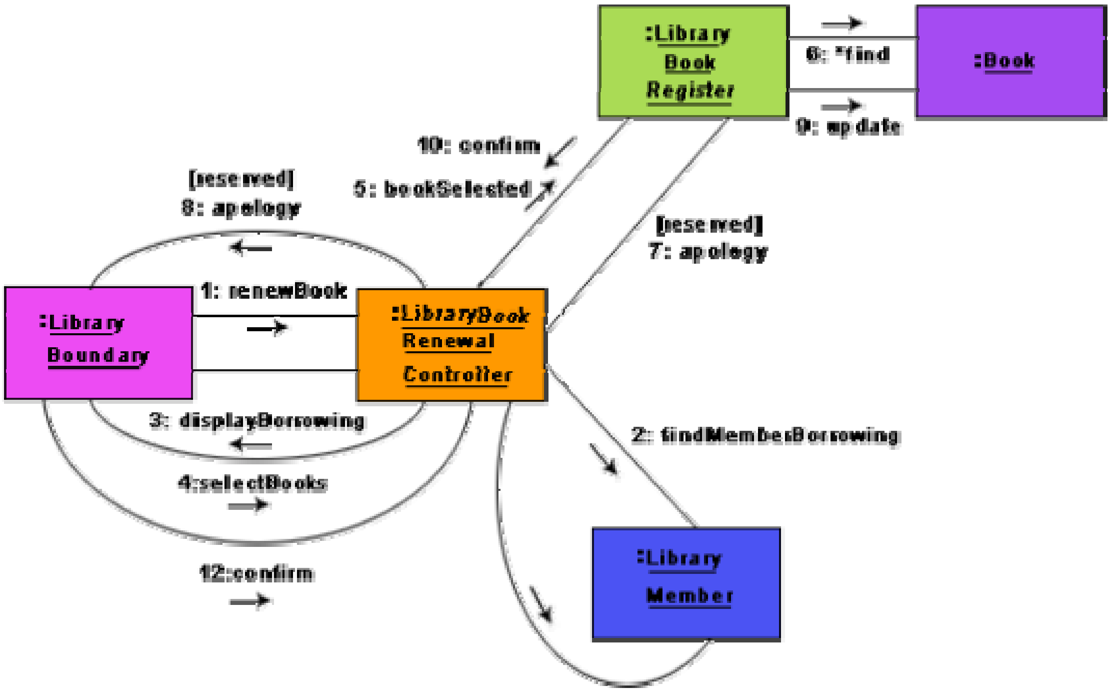


Fig. 7.14: Collaboration diagram for the renew book use case s