**Figure 7.21:** Use case packaging.

## 7.5 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. We now discuss the UML syntax for representation of the classes and their relationships.

### 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 centred 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 are shown in Figure 7.22.

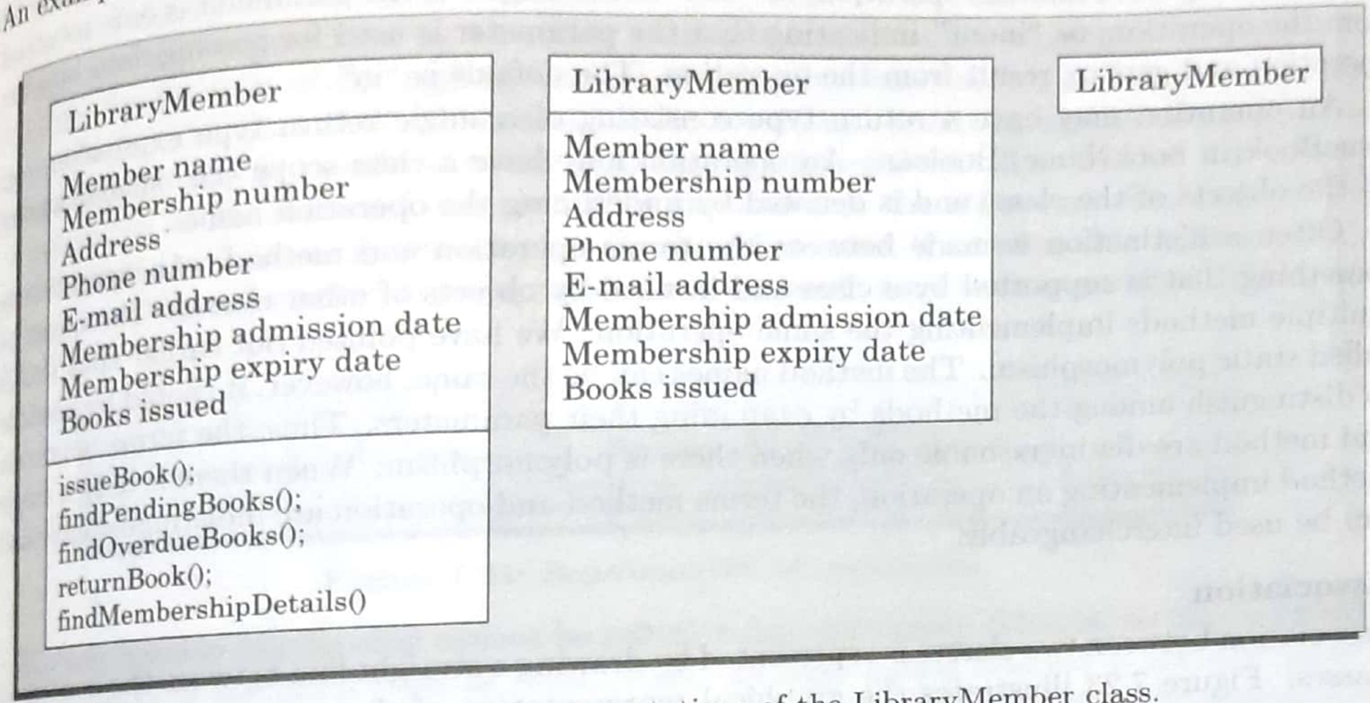

**Figure 7.22:** Different representations of the LibraryMember class.

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 (that is, their class, e.g. Int, Book, Employee, etc.), an initial value, and constraints. Attribute names are written left-justified using plain type letters, and the names should begin with a lower case letter.

Attribute names may be followed by square brackets containing a multiplicity expression, e.g. sensorStatus[10]. The multiplicity expression indicates the number of attributes 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[1]: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 the newly created objects, e.g. sensorStatus[1]:Int=0.

**Operation**

The operation names are typically left justified, in plain type, and always begin with a lower case letter. Abstract operations are written in italics[5]. (Remember that abstract operations

[5]Many UML symbols are only suitable for drawing using a CASE tool and 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 in place of difficult to draw symbols.

are those for which the implementation is not provided during the class definition.) The parameters of a function may have a 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".

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 7.23 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 the 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 used as a wild card and means many (zero or more). The association of Figure 7.23 should be read as "Many books may be borrowed by a LibraryMember". Usually, associations (and links) appear as verbs in the problem statement.

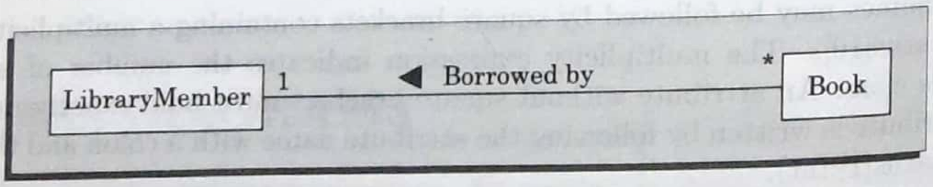

**Figure 7.23:** Association between two classes.

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 relation where the involved classes are not only associated to each other, but a whole-part relationship exists between them. That is, the aggregate object not only knows the addresses of its parts and therefore invoke the methods of its parts, but also takes the responsibility of creating and destroying its parts. An example of aggregation, a book register is an aggregation of book objects. Books 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 7.24. The figure represents the fact that a document can be considered as an aggregation of paragraphs. Each paragraph can in turn be considered as aggregation of lines. Observe that the number 1 is annotated at the diamond end, and a* 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 (*).

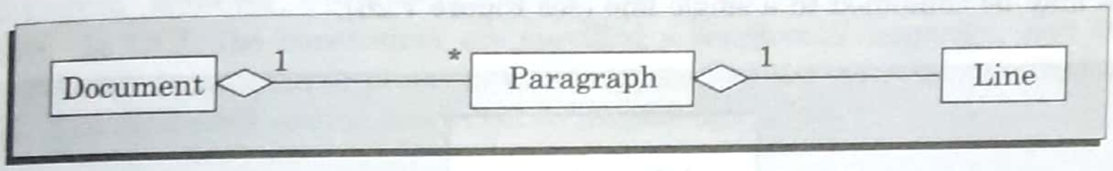

**Figure 7.24:** Representation of aggregation.

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. As an example of a transitive aggregation relationship, please see Figure 7.24.

### 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 cannot exist outside the whole. In other words, 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.

A typical example of composition is an order object where after placing the order, no item in the order cannot 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 the same as the aggregate (order). The composition relationship is represented as a filled diamond drawn at the composite-end. An example of the composition relationship is shown in Figure 7.25.

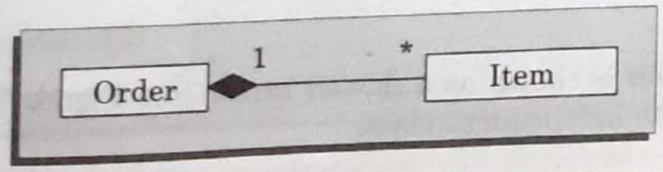

**Figure 7.25:** Representation of composition.

### Aggregation versus composition

Both aggregation and composition represent part/whole relationships. When the components can dynamically be added and removed from the aggregate, then the relationship is aggregation. If the components cannot be dynamically added/delete, then the components are have the same lifetime as the composite. In this case, the relationship is represented by composition.

As an example, consider the example of an order consisting 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 it.

**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 a single line (see Figure 7.26).

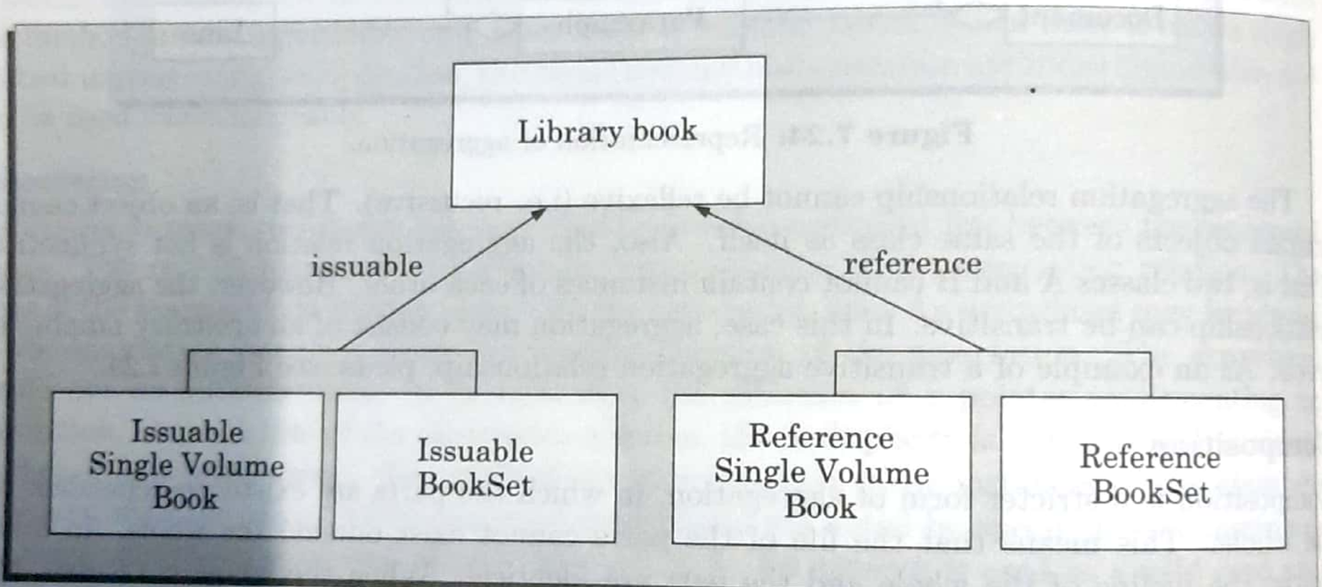

**Figure 7.26:** Representation of the inheritance relationship.

The direct arrows allow flexibility in laying out the diagram and can easily be drawn by hand. The combined arrows emphasize the collectivity of the subclasses, when specialization has been done on the basis of some discriminator. In the example of Figure 7.26, issuable and reference are the discriminators. The various subclasses of a superclass can then be differentiated by means of the discriminator. The set of subclasses of a class having the same discriminator is called a partition. It is often helpful to mention the discriminator during modelling, as these become documented design decisions.

**Dependency**

A dependency relationship is shown as a dotted arrow (see Figure 7.27) that is drawn from the dependent class to the independent class.

**Constraints**

A constraint describes a condition or an integrity rule. Constraints are typically used to describe the permissible set of values of an attribute, to specify the pre- and post-conditions for operations, to define certain ordering of items, etc. For example, to denote that the books in a library are sorted on ISBN number we can annotate the book class with the constraint

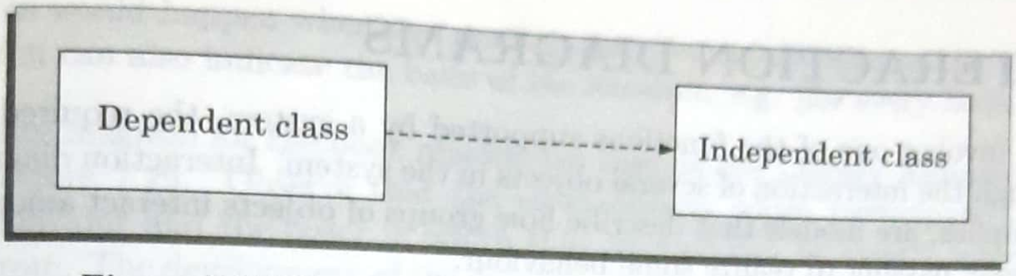

**Figure 7.27:** Representation of dependence between classes.

{sorted}. UML allows you to use any free form expression to describe constraints. The only rule is that they are to be enclosed within braces. Constraints can be expressed using informal English. However, UML also provides Object Constraint Language (OCL) to specify constraints. In OCL the constraints are specified a semiformal 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].