Object-Oriented Design

Object-oriented (OO) approaches for software development have become extremely popular in recent years. Much of the new development is now being done using OO techniques and languages. There are many advantages that OO systems offer. An OO model closely represents the problem domain, which makes it easier to produce and understand designs. As requirements change, the objects in a system are less immune to these changes, thereby permitting changes more easily. Inheritance and close association of objects in design to problem domain entities encourage more reuse, that is, new applications can use existing modules more effectively, thereby reducing development cost and cycle time. Object-oriented approaches are believed to be more natural and provide richer structures for thinking and abstraction. Common design patterns have also been uncovered that allow reusability at a higher level. (Design patterns is an advanced topic which we will not discuss further; interested readers are referred to [69].)

The object-oriented design approach is fundamentally different from the function-oriented design approaches primarily due to the different abstraction that is used. It requires a different way of thinking and partitioning. It can be said that thinking in object-oriented terms is most important for producing truly object-oriented designs.

During design, as mentioned in the previous chapter, our focus is on what is called the module view in architecture. That is, the goal is to identify the modules that the system should have, and their interfaces and relationships. In OOD, we are therefore identifying the classes that should exist in the software and the relationship between these classes. During architecture design, the component and connector view is typically fixed. A goal of
design is to ensure that the architecture is preserved, and the relationship between the components and modules is clear.

In this chapter, we will discuss some important concepts that form the basis of object-orientation. Then we will discuss some concepts that influence a designer in creating an object-oriented design (OOD). We'll then describe the UML notation that can be used while doing an object-oriented design, followed by an OOD methodology. Then we'll discuss some metrics that are applicable on OOD and that can be used to evaluate the quality of design. We do not discuss verification methods, as the design verification methods discussed in the previous chapter are general methods that can be used regardless of the approach used for producing the design. Finally, as with other chapters, we'll end by doing the OOD design of the case studies. Before we proceed, let us understand the relationship between OO analysis and OOD design.

7.1 OO Analysis and OO Design

Pure object-oriented development requires that object-oriented techniques be used during the analysis, design, and implementation of the system. However, much of the focus of the object-oriented approach to software development has been on analysis and design. Various methods have been proposed for analysis and design, many of which propose a combined analysis and design technique. We will refer to a combined method as object-oriented analysis and design (OOAD). In OOAD the boundary between analysis and design is blurred. One reason for this blurring is the similarity of basic constructs (i.e., objects and classes) that are used in analysis and design. Though there is no agreement about what parts of the object-oriented development process belong to analysis and what parts to design, there is some general agreement about the domains of the two activities.

The fundamental difference between object-oriented analysis (OOA) and object-oriented design (OOD) is that the former models the problem domain, leading to an understanding and specification of the problem, while the latter models the solution to the problem. That is, analysis deals with the problem domain, while design deals with the solution domain. However, in OOAD it is believed that the problem domain representation created by OOA is generally subsumed in the solution domain representation. That is, the solution domain representation, created by OOD, generally contains much of the representation created by OOA, and more. This is shown in Figure
7.1. **OO ANALYSIS AND OO DESIGN**

As the objective of both OOA and OOD is to model some domain, frequently the OOA and OOD processes (i.e., the methodologies) and the representations look quite similar. This contributes to the blurring of the boundaries between analysis and design. It is often not clear where analysis ends and design begins. The separating line is a matter of perception. The lack of clear separation between analysis and design can also be considered one of the strong points of the object-oriented approach—the transition from analysis to design is "seamless." This is also the main reason OOAD methods—where analysis and design are both performed—have been proposed.

Despite the difference in perceptions on the boundary between OOA and OOD, one thing is clear. The main difference between OOA and OOD, due to the different domains of modeling, is in the type of objects that come out of the analysis and design processes. The objects during OOA focus on the problem domain and generally represent some things or concepts in the problem. These objects are sometimes called *semantic objects* as they have a meaning in the problem domain [118]. The solution domain, on the other hand, consists of semantic objects as well as other objects. During design, as the focus is on finding and defining a solution, the semantic objects identified during OOA may be refined and extended from the point of view of implementation, and other objects are added that are specific to the solution domain. The solution domain objects include *interface, application,* and *utility* objects [118]. The interface objects deal with the user interface, which

---

**Figure 7.1:** Relationship between OOA and OOD.
is not directly a part of the problem domain but represents some aspect of
the solution desired by the user. The application objects specify the control
mechanisms for the proposed solution. They are driver objects that are
specific to the application needs. Utility objects are those needed to support
the services of the semantic objects or to implement them efficiently (e.g.,
queues, trees, and tables). These objects are frequently general-purpose
objects and are not application-dependent.

The basic goal of the analysis and design activities is to identify the
classes in the system and their relationships, and frequently represented by
class diagrams. However, the system has to support some functionality and
behavior. Hence, in addition to concentrating on the static structure of the
problem or solution domains, the dynamic behavior of the system has to
be studied to make sure that the final design supports the desired dynamic
behaviors. Due to this, some dynamic modeling of the system is desired
before the design is complete. Whether this type of modeling is part of
analysis or design, i.e., where in the overall OOAD process the boundary
between analysis and design is, is not generally agreed on.

Another way to view the difference between modeling and design is that
in design, a model is built for the (eventual) implementation. As a con­
sequence, implementation issues drive the modeling process during design.
While in analysis, comprehension and representation issues drive the pro­
cess. This also results in OOA sometimes using primitives that are some­
what richer than the ones used in OOD, as the OOD primitives tend to be
closely associated with the features of the programming language to be used
for implementing the design. The models built during object-oriented anal­
ysis form the starting point of object-oriented design, and the model built
by OOD forms the basis for object-oriented implementation.

7.2 OO Concepts

Here we discuss the main concepts behind object-orientation. Though these
concepts were also used during object-oriented analysis, they are discussed
in more concrete terms here, as a design deals with the solution domain and
is therefore closer to the final implementation. As the discussion revolves
around the OO concepts as supported in programming languages, readers
who are very familiar with OO languages and their concepts can omit this
section. In the following section we discuss some design concepts.
7.2. OO CONCEPTS

7.2.1 Classes and Objects

Classes and objects are the basic building blocks of an OOD, just like functions (and procedures) are for a function-oriented design. During design, we are not dealing just with abstractions of real-world objects (as is the case with analysis), but we are also dealing with abstract software objects. During analysis, we viewed an object as an entity in the problem domain that had clearly defined boundaries and behavior. During design, this has to be extended to accommodate software objects.

**Encapsulation**

In general, we consider objects entities that provide some services to be used by a client, which could be another object, program, or a user. The basic property of an object is *encapsulation*: it encapsulates the data and information it contains, and supports a well-defined abstraction. For this, an object provides some well-defined services its clients can use, with the additional constraint that a client can access the object only through these services. This encapsulation of information along with the implementation of the operations performed on the information such that from outside a set of services is available is a key concept in object orientation. The set of services that can be requested from outside the object forms the *interface* of the object. An object may have operations defined only for internal use that cannot be used from outside. Such operations do not form part of the interface. The interface defines all ways in which an object can be used from outside.

For example, consider an object directory of telephone numbers that has add-name(), change-number(), and find-number() operations as part of the interface. These are the operations that can be invoked from outside on the object directory. It may also have internal operations like hash() and insert() that are used to support the operations in the interface but do not form part of the interface. These operations can only be invoked from within the object directory (i.e., by the operations defined on the object). Note that objects of other classes may also have the same interface (see the discussion on inheritance later).

A major advantage of encapsulation is that access to the encapsulated data is limited to the operations defined on the data. Hence, it becomes much easier to ensure that the integrity of data is preserved, something very hard to do if any program from outside can directly manipulate the data structures of an object. This is an extremely desirable property when building large
systems, without which things can be very chaotic. In function-oriented systems, this is usually supported through self-discipline by providing access functions to some data and requiring or suggesting that other programs access the information through the access functions. In OO languages, this is enforced by the language, and no program from outside can directly access the encapsulated data.

Encapsulation, leading to the separation of the interface and its implementation, has another major consequence. As long as the interface is preserved, implementation of an object can be changed without affecting any user of the object. For example, consider the directory object discussed earlier. Suppose the object uses an array of words to implement the operations defined on directory. Later, if the implementation is changed from the array to a B-tree or by using hashing, only the internals of the object need to be changed (i.e., the data definitions and the implementation of the operations). From the outside, the directory object can continue to be used in the same manner as before, because its interface is not changed.

State, Behavior, and Identity

An object has state, behavior, and identity [23, 124]. The encapsulated data for an object defines the state of the object. An important property of objects is that this state persists, in contrast to the data defined in a function or procedure, which is generally lost once the function stops being active (finishes its current execution). In an object, the state is preserved and it persists through the life of the object, i.e., unless the object is actively destroyed.

The various components of the information an object encapsulates can be viewed as “attributes” of the object. That is, an object can be viewed as having various attributes, whose values (together with the information about the relationship of the object to the other objects) form the state of the object. The relationship between attributes and encapsulated data is that the former is in terms of concepts that may have some meaning in the problem domain: they essentially represent the abstract information being modeled by the components of the data structures.

The state and services of an object together define its behavior. We can say that the behavior of an object is how an object reacts in terms of state changes when it is acted on, and how it acts upon other objects by requesting services and operations. Generally, for an object, the defined operations together specify the behavior of the object. However, it should
be pointed out that although the operations specify the behavior, the actual behavior also depends on the state of the object as an operation acts on the state and the sequence of actions it performs can depend on the state. A side effect of performing an operation may be that the state of the object is modified. As operations are the only means by which some activity can be performed by the object, it should also be clear that the current state of an object represents the sequence of operations that have been performed on it.

Finally, an object has *identity*. Identity is the property of an object that distinguishes it from all other objects. In most programming languages, variable names are used to distinguish objects from each other. So, for example, one can declare objects $s_1, s_2, \ldots$ of class type Stack. Each of these variables $s_1, s_2, \ldots$ will refer to a unique stack having a state of its own (which depends on the operations performed on the stack represented by the variable).

**Classes**

Objects represent the basic run-time entities in an OO system; they occupy space in memory that keeps its state and is operated on by the defined operations on the object. A *class*, on the other hand, defines a possible set of objects. We have seen that objects have some attributes, whose values constitute much of the state of an object. What attributes an object has are defined by the class of the object. Similarly, the operations allowed on an object or the services it provides, are defined by the class of the object. But a class is merely a definition that does not create any objects and cannot hold any values. When objects of a class are created, memory for the objects is allocated.

A class can be considered a template that specifies the properties for objects of the class. Classes have [136]:

1. An interface that defines which parts of an object of a class can be accessed from outside and how

2. A class body that implements the operations in the interface

3. Instance variables that contain the state of an object of that class

Each object, when it is created, gets a private copy of the instance variables, and when an operation defined on the class is performed on the object, it is performed on the state of the particular object.
The relationship between a class and objects of that class is similar to the relationship between a type and elements of that type. A class represents a set of objects that share a common structure and a common behavior, whereas an object is an instance of a class. The interface of the objects of a class—the behavior and the state space (i.e., the states an object can take)—are all specified by the class. The class specifies the operations that can be performed on the objects of that class and the interface of each of the operations.

Note that classes can be viewed as abstract data types. Abstract data types (ADTs) were promulgated in the 1970s, and a considerable amount of work has been done on specification and implementation of ADTs. The major differences between ADTs and class are inheritance and polymorphism (discussed later). Classes without inheritance are essentially ADTs, but with inheritance, which is considered a central property of object orientation, their semantics are richer than that of an ADT.

Not all operations defined on a class can be invoked on objects of that class from outside the object—some operations are defined that are entirely for internal use. The case for data declarations within the class is similar. Although generally it is fully encapsulated, in some languages it is possible to have some data visible from outside. However, this distinction of what is visible from outside has to be enforced by the language. Using the C++ classification, the data and operations of a class (sometimes collectively referred to as features) can be declared as one of three types:

- **Public.** These are (data or operation) declarations that are accessible from outside the class to anyone who can access an object of this class.

- **Protected.** These are declarations that are accessible from within the class itself and from within subclasses (actually also to those classes that are declared as friends).

- **Private.** These are declarations that are accessible only from within the class itself (and to those classes that are declared as friends).

Different programming languages provide different access restrictions, but public and private separation are generally needed. At least one operation is needed to create (and initialize) an object and one is needed to destroy an object. The operation creating and initializing objects is called *constructor*, and the operation destroying objects is called *destructor*. The remaining
7.2. **OO CONCEPTS**

```cpp
class List{
    private:
        // data definitions to implement bag
        int list[MAX];
        int size;

    public:
        List() {size = 0};
        add (number); // add a number
        int ispresent (number); // check if number is present
        int delete (number); // delete a number, if present
}
```

Figure 7.2: Class List of numbers.

operations can be broadly divided into two categories: modifiers and value-ops. Modifiers are operations that modify the state of the object, while value-ops are operations that access the object state but do not alter it. The operations defined on a class are also called *methods* of that class.

When a client requests some operations on an object, the request is actually bound to a method defined on the class of the object. Then that method is executed, using the state of the object on which the operation is to be executed. In other words, the object itself provides the state while the class provides the actual procedure for performing the operation on the object.

**An Example**

An example will illustrate these concepts. Suppose we need to have an object that represents a list of integers. The list consists of the numbers we put in it. We want it to be such that we can check if a number exists, and add or remove a number. In C++, the class definition List (to be used for obtaining the object list) could be something like Figure 7.2.

With this definition, a particular list, list, can be created by declaring `List list`. We can declare as many objects of the type `List` as we want. Whenever an object is declared of the type `List`, the constructor operator `List()` is executed, which sets the size of that list to 0. In C++, the operator with the same name as the name of the class is the constructor operator invoked to initialize the object whenever the object is created by declaration. We can add a number `n` to this bag by invoking `list.add(n)`. The history of
whatever numbers we add to list is preserved within the list (in its private data members). Much later, when we want to check if a number is present, it will return that the number is present if at any time in the past the number was added to list and it has not been deleted.

Note that the fact that the list is implemented as an array and a size pointer is not visible from outside. Other programs use lists by declaring objects of the type List and then performing operations on them. If at a later time, due to efficiency reasons we want to change the implementation of List to use a binary search tree, we will have to change the data structures and the code of the operations. However, no change needs to be made to the programs that declare and use various lists.

In C++, the interface of the object is whatever is defined as public. Generally, it will contain only the operations. The declarations in the private part can only be used from within the object; they cannot be accessed from outside. If some function is declared as private, then that function cannot be invoked from outside; it can only be used by the other operations defined on the class. The code for a function defined in a class can either be given with the definition of the function interface (as was done with the constructor List()) or defined elsewhere. If it is defined elsewhere, the definition has to be prefixed with the class name. For example, the function add(n) will be declared as List::add(int n).

### 7.2.2 Relationships Among Objects

An object, as a stand-alone entity, has very limited capabilities—it can only provide the services defined on it. Any complex system will be composed of many objects of different classes, and these objects will interact with each other so that the overall system objectives are met. In object-oriented systems, an object interacts with another by sending a message to the object to perform some service it provides. On receiving the message, the object invokes the requested service or the method and sends the result, if needed. Frequently, the object providing the service is called the server and the object requesting the service is called the client. This form of client-server interaction is a direct fall out of encapsulation and abstraction supported by objects.

If an object invokes some services in other objects, we can say that the two objects are related in some way to each other. All objects in a system are not related to all other objects. In fact, in most programming languages, an object cannot even access all objects, but can access only those objects
that have been explicitly programmed or located for this purpose. During
design, which objects are related has to be clearly defined so that the system
can be properly implemented.

If an object uses some services of another object, there is an *association*
between the two objects. This association is also called a *link*—a link exists
from one object to another if the object uses some services of the other object.
Links frequently show up as pointers when programming. A link captures
the fact that a message is flowing from one object to another. However,
when a link exists, though the message flows in the direction of the link,
information can flow in both directions (e.g., the server may return some
results).

With associations comes the issue of visibility, that is, which object is
visible to which. This is an issue that is very pertinent for implementation
and therefore comes up during design. However, this is not an important
issue during analysis and is therefore rarely dealt with during OOA. The
basic issue here is that if there is a link from object A to object B, for A to
be able to send a message to B, B must be visible to A in the final program.
There are different ways to provide this visibility. Some of the important
possibilities are [23]:

- The supplier object is global to the client.
- The supplier object is a parameter to some operation of the client that
  sends the message.
- The supplier object is a part of the client object.
- The supplier object is locally declared in some operation.

Each of these has some consequences. For example, if the supplier object is a
global object to the client, then the scoping of languages may make the client
visible to many other objects. This is, in general, not very desirable, and
should be done only when there is common information that many different
classes need. If the supplier object is a parameter of a method, then the
intention is to show that the object belongs elsewhere, and this object may
access it only through this method. If the supplier object is a part of the
client, it means that the supplier object is declared as a data member of this
class. This implies that when the life of the client object finishes, the supplier
object is also destroyed. This clearly can have implications on sharing of
objects and services. Overall, how an object is made visible to an object
7. OBJECT-ORIENTED DESIGN

that needs to access it is an important design issue to be kept in mind when designing associations.

If the supplier object is declared in the client object, there are different ways to implement associations. They can be implemented by a pointer in one of the objects (generally the client object) to the other object. The problem with this approach comes if the link is to be traversed in the reverse direction from the object to which it is pointed. For this, a search needs to be performed on all existing objects of the class with which this class has an association to find which object has the pointer to this object. Hence, this method of implementation should be used only if it is clear that the application is such that the reverse traversal of the link will never be needed.

Another way of implementing the association is by making the link bidirectional, which is what links generally mean in modeling. This can be done by keeping a pointer to the other object in each of the two objects. This is more expensive in terms of storage, but it solves the problem. However, care must be taken to see that the links are consistent; whenever one of the pointers is modified, the other pointer needs to be modified accordingly.

Yet another way of implementing association is to create a new object, whose only duty is to keep track of the links between objects. This approach separates the link maintenance job from the two objects. This is useful when there are many links. Each object will register its link with this special-purpose object.

Links between objects capture the client/server type of relationship. Another type of relationship between objects is aggregation, which reflects the whole/part-of relationship. Though not necessary, aggregation generally implies containment. That is, if an object A is an aggregation of objects B and C, then objects B and C will generally be within object A (though there are situations where the conceptual relationship of aggregation may not get reflected as actual containment of objects). The main implication of this is that a contained object cannot survive without its containing object. With links, that is not the case. An example of aggregation in C++ notation is shown next:

```cpp
class Disk {
    private:
        Track *tracks;
        disk information
};
```
In this example, a class of type Disk is declared, which specifies that any object of this type will have within it a pointer to another object of class Track, and this pointer is private information of the object that cannot be accessed from outside the object. The definition of the class Track states that each object of this type will have an array of elements of class Sector within it as private data members. The example captures the fact that a disk consists of many tracks, and each track contains many sectors. As shown by class definitions, aggregation can be implemented by declaring the parts as objects within the class, as is done while defining the class Track. Or it can be implemented as a pointer to the part, as is done while defining Disk. The latter method is also used for defining aggregation; hence representing aggregation is used only for efficiency reasons or if the object is to be accessed by many other objects outside the container object.

### 7.2.3 Inheritance and Polymorphism

Inheritance is a concept unique to object orientation. Some of the other concepts, such as information hiding, can be supported by non-object-oriented languages through self-discipline, but inheritance cannot generally be supported by such languages. It is also the concept central to many of the arguments claiming that software reuse can be better supported with object orientation.

Inheritance is a relation between classes that allows for definition and implementation of one class based on the definition of existing classes [107]. Let us try to understand this better. When a class B inherits from another class A, B is referred to as the subclass or the derived class and A is referred to as the superclass or the base class. In general, a subclass B will have two parts: a derived part and an incremental part [107]. The derived part
is the part inherited from A and the incremental part is the new code and definitions that have been specifically added for B. This is shown in Figure 7.3 [107]. Objects of type B have the derived part as well as the incremental part. Hence, by defining only the incremental part and inheriting the derived part from an existing class, we can define objects that contain both.

Inheritance is often called an "is-a" relation, implying that an object of type B is also an instance of type A. That is, an instance of a subclass, though more than an instance of the superclass, is also an instance of the superclass.

In general, an inherited feature of A may be redefined in various forms in B. This redefinition may change the visibility of the operation (e.g., a public operation of A may be made private in B), changed (e.g., by defining a different sequence of instructions for this operation), renamed, voided, and so on.

The inheritance relation between classes forms a hierarchy. As inheritance represents an "is-a" relation, it is important that the hierarchy represent a structure present in the application domain and is not created simply to reuse some parts of an existing class. That is, the hierarchy should be such that an object of a class is also an object of all its super classes in the problem domain.

The power of inheritance lies in the fact that all common features of the subclasses can be accumulated in the superclass. In other words, a feature is placed in the higher level of abstractions. Once this is done, such features can be inherited from the parent class and used in the subclass directly.
This implies that if there are many abstract class definitions available, when a new class is needed, it is possible that the new class is a specialization of one or more of the existing classes. In that case, the existing class can be tailored through inheritance to define the new class.

Inheritance promotes reuse by defining the common operations of the subclasses in a superclass. However, inheritance makes the subclasses dependent on the superclass, and a change in the superclass will directly affect the subclasses that inherit from it. As classes may change as design is refined, with each change in a class, its impact on the subclasses will also have to be analyzed. This also has an impact on the testing of classes. We will discuss the issue of testing later in the book.

Let us illustrate inheritance through the use of an example. Consider a graphics package that has the class `GraphicalObject` representing all graphical objects. A graphical object can have a zero area or a non-zero area, giving two subclasses `ZeroAreaObject` and `NonZeroAreaObject`. Line and Curve are two specific object classes of the first category, and Polygon and Circle are two specific object classes of the latter category. This hierarchy of classes is shown in Figure 7.4.

Figure 7.4: An inheritance example.
As we can see, the `GraphicalObject` has attributes of color and draw-style (which represents the style of drawing the figure)—both of which each graphical object has. It has many operations defined on it—move(), rotate(), scale(), etc.—the ones that are needed for every object by the graphics package. Note, however, that even though operations like rotate() and scale() are defined for an object, they are totally conceptual in that their exact specification depends on the nature of the object (e.g., rotate() on a circle has to do different things than rotate() on a line). Hence, these operations have to be defined for each object. In C++, such operations that are declared in a superclass and redefined in a subclass are declared as virtual in the superclass. If an operation specified in a class is always redefined in its subclass, then the operation can be defined as pure virtual (in C++, this is done by equating it to 0), implying that the operation has no body. The implication of existence of these operations is that no objects of this class can be created, as some of the operations declared in the class are not defined and hence cannot be performed. Such a class is sometimes called an abstract base class.

The C++ class skeletons for this hierarchy are shown next:

class GraphicalObject {
  protected:
    unsigned int colpr;
    unsigned int draw_style;
  public:
    virtual void move( Point &newLocation );
    virtual void rotate(double angle );
    virtual void scale( double XScale , double YScale);
    void setColor( unsigned int col );
    void setDrawStyle( unsigned int style );
};

class ZeroAreaObject: public GraphicalObject {};

class NonZeroAreaObject: public GraphicalObject {
  protected:
    unsigned int fillColor;
    unsigned int fillStyle;
  public:
    virtual fill();
};

class Line: public ZeroAreaObject {
Inheritance can be broadly classified as being of two types: strict inheritance and nonstrict inheritance [136]. In strict inheritance a subclass takes all the features from the parent class and adds additional features to specialize it. That is, all data members and operations available in the base class are also available in the derived class. This form supports the "is-a" relation and is the easiest form of inheritance. Nonstrict inheritance occurs when the sub-
class does not have all the features of the parent class or some features have been redefined. This form of inheritance has consequences in the dynamic behavior and complicates testing.

A class hierarchy need not be a simple tree structure. It may be a graph, which implies that a class may inherit from multiple classes. This type of inheritance, when a subclass inherits from many superclasses, is called *multiple inheritance*. Consider part of the class hierarchy of logic gates for a system for simulating digital logic of circuits as shown in Figure 7.5. In this example, there are separate classes to represent And gates, Nor gates, and Or gates. The class for representing Nand gates inherits from both the class for And gates and the class for Not gates. That is, all the definitions (instances and operations) that have been declared as public (or protected) in the classes NotGate and AndGate are available for use to the class NandGate. Similarly, the class NorGate inherits from the OrGate and NotGate. Like in regular inheritance, a subclass can redefine any feature if it desires.

Multiple inheritance brings in some new issues. First, some features of two-parent classes may have the same name. So, for example, there may be an operation \( O() \) in class A and class B. If a class C inherits from class A and class B, then when \( O() \) is invoked from an object of class C, if \( O() \) is not defined locally within C, it is not clear from where the definition of \( O() \) should be taken—from class A or from class B. This ambiguity does not arise if there is no multiple inheritance; the operation of the closest ancestor in which \( O() \) is defined is executed. Different language mechanisms or rules can be used to resolve this ambiguity. In C++, when such an ambiguity arises, the programmer has to resolve it by explicitly specifying the superclass from which the definition of the feature is to be taken.

Multiple inheritance also brings in the possibility of *repeated inheritance*,
where a class inherits more than once from the same class [136]. For example, consider the situation shown in Figure 7.6 where classes B and C inherit from class A and class D inherits from both B and C. A situation like this means that effectively class D is inheriting twice from A—once through B and once through C. This form of inheritance is even more complex, as features of A may have been renamed in B and C, and can lead to run-time errors.

Due to the complexity that comes with multiple inheritance and its variations and the possibility of confusion that comes with them, it is generally advisable to avoid their usage.

Inheritance brings in polymorphism, a general concept widely used in type theory, that deals with the ability of an object to be of different types. In OOD, polymorphism comes in the form that a reference in an OO program can refer to objects of different types at different times. Here we are not talking about “type coercion,” which is allowed in languages like C; these are features that can be avoided if desired. In object-oriented systems, with inheritance, polymorphism cannot be avoided—it must be supported. The reason is the “is-a” relation supported by inheritance—an object x declared to be of class B is also an object of any class A that is the superclass of B. Hence, anywhere an instance of A is expected, x can be used.

With polymorphism, an entity has a static type and a dynamic type [107]. The static type of an object is the type of which the object is declared in the program text, and it remains unchanged. The dynamic type of an entity, on the other hand, can change from time to time and is known only at reference time. Once an entity is declared, at compile time the set of types

![Figure 7.6: Repeated inheritance.](image)
that this entity belongs to can be determined from the inheritance hierarchy that has been defined. The dynamic type of the object will be one of this set, but the actual dynamic type will be defined at the time of reference of the object. In the preceding example, the static type of \( x \) is \( B \). Initially, its dynamic type is also \( B \). Suppose an object \( y \) is declared of type \( A \), and in some sequence of instructions there is an instruction \( x := y \). Due to the "is-a" relation between \( A \) and \( B \), this is a valid statement. After this statement is executed, the dynamic type of \( x \) will change to \( A \) (though its static type remains \( B \)). This type of polymorphism is called \textit{object polymorphism} [136], in which wherever an object of a superclass can be used, objects of subclasses can be used.

This type of polymorphism requires \textit{dynamic binding} of operations, which brings in \textit{feature polymorphism}. Dynamic binding means that the code associated with a given procedure call is not known until the moment of the call [107]. Let us illustrate with an example. Suppose \( x \) is a polymorphic reference whose static type is \( B \) but whose dynamic type could be either \( A \) or \( B \). Suppose that an operation \( O \) is defined in the class \( A \), which is redefined in the class \( B \). Now when the operation \( O \) is invoked on \( x \), it is not known statically what code will be executed. That is, the code to be executed for the statement \( x.O \) is decided at run time, depending on the dynamic type of \( x \)—if the dynamic type is \( A \), the code for the operation \( O \) in class \( A \) will be executed; if the dynamic type is \( B \), the code for operation \( O \) in class \( B \) will be executed. This dynamic binding can be used quite effectively during application development to reduce the size of the code. For example, take the case of the graphical object hierarchy discussed earlier. In an application, suppose the elements of a figure are stored in an array \( A \) (of \texttt{GraphicalObject} type). Suppose element 1 of this array is a line, element 2 is a circle, and so on. Now if we want to rotate each object in the figure, we simply loop over the array performing \( A[i].\texttt{rotate()} \). For each \( A[i] \), the appropriate rotate function will be executed. That is, which function \( A[i].\texttt{rotate()} \) refers to is decided at run time, depending on the dynamic type of object \( A[i] \).

This feature polymorphism, which is essentially overloading of the feature (i.e., a feature can mean different things in different contexts and its exact meaning is determined only at run time) causes no problem in strict inheritance because all features of a superclass are available in the subclasses. But in nonstrict inheritance, it can cause problems, because a child may lose a feature. Because the binding of the feature is determined at run time, this can cause a run-time error as a situation may arise where the object is
bound to the superclass in which the feature is not present.

7.3 Design Concepts

In an OO system, the basic module is a class, and during design the key activity is to identify and specify the modules that should be there in the system being built. The goal of the design activity is to create a design that, besides being correct, has other attributes that make it a good design.

There are many desirable attributes for an OO system. However, here we will focus on three main concepts. If we can create a design that is satisfactory from these three perspectives (and is correct,) then we can be fairly sure that we have a good design. These key concepts govern the quality of a design, and should therefore drive the design process and the design choices. The three concepts are cohesion, coupling, and open-closed principle. Our goal is to create a design in which the modules are low in coupling, high in cohesion (we will soon understand what low and high means), and which satisfy the open-closed principle. Besides these, we also discuss a few design guidelines that suggest more concrete ways of putting these principles in practice.

7.3.1 Coupling

As mentioned in the previous chapter, coupling is an inter-module concept which captures the strength of interconnection between modules. The more tightly coupled the modules are, the more dependent they are on each other, and the more difficult it is to understand and modify them. Low coupling is desirable for making the system more understandable and modifiable.

The degree of coupling between a module and another module depends on how much information is needed about the other module for understanding and modifying this module, and how complex and explicit this information is. Low coupling occurs when this information is as little as possible, as simple as possible, and is easily visible or identifiable. In the previous chapter we discussed this concept for systems with functional modules. Although the concept remains the same, its manifestation in OO systems is somewhat different as objects are semantically richer than functions. In OO systems, three different types of coupling exists between modules [53]

- Interaction coupling
- Component coupling
• Inheritance coupling

Interaction coupling occurs due to methods of a class invoking methods of other classes. Note that as we are looking at coupling between classes we focus on interaction between classes, and not within a class. In many ways, this situation is similar to a function calling another function and hence this coupling is similar to coupling between functional modules. Like with functions, the worst form of coupling here is if methods directly access internal parts of other methods. (This type of interaction is disallowed in many languages but is allowed where concepts like friend classes, which allow a friend to delve into the internals of a class, exist.)

Interaction coupling reduces, though is still very high, if methods of a class interact with methods in another class by directly manipulating instance variables or attributes of objects of the other class. This form of interaction is also bad as one has to understand the code of other classes to understand what changes they are making to the class. It also violates the encapsulation principle of OO. This form of interaction is worse if variables are used to communicate temporary data, that is, the variables are used not to hold the state of the object but to pass state of the computation from one object to another. If this temp-value holder variable happens to be in the super class, then the coupling worsens since the variable is visible to all sub classes.

Coupling is least (like in coupling with functional modules) if methods communicate directly through parameters. Within this category, coupling is lower if only data is passed, but is higher if control information is passed since the invoked method impacts the execution sequence in the calling method. Also, coupling is higher if the amount of data being passed is more. So, if whole data structures are passed when only some parts are needed, coupling is being unnecessarily increased. Similarly, if an object is passed to a method when only some of its component objects (or objects the passed object refers to) are used within the method, coupling increases unnecessarily. The least coupling situation therefore is when communication is with parameters only, with only necessary variables being passed, and these parameters only pass data.

Component coupling refers to the interaction between two classes where a class has variables of the other class. Three clear situations exist when this can happen. A class C can be component coupled with another class C, if C has an instance variable of type C, or C has a method whose parameter is of type C, or if C has a method which has a local variable of type C (which can
then be passed as parameter to some method it invokes.) Note that when C is component coupled with C, it has the potential of being component coupled with all subclasses of C as at runtime an object of any subclass may actually be used. It should be clear that whenever there is component coupling, there is likely to be interaction coupling. Component coupling is considered to be weakest (i.e., most desired) if in a class C, the variables of class C are either in the signatures of the methods of C, or some attributes are of type C. If interaction is through local variables, then this interaction is not visible from outside, and therefore increases coupling.

Inheritance coupling is due to the inheritance relationship between classes. Two classes are considered inheritance coupled if one class is a direct or indirect subclass of the other. If inheritance adds coupling, one can ask the question why not do away with inheritance altogether. The reason is that inheritance may reduce the overall coupling in the system. Let us consider two situations. If a class A is coupled with another class B, and if B is a hierarchy with B and B as two subclasses, then if a method m is factored out of B and B and put in the super class B, the coupling reduces as A is now only coupled with B, whereas earlier it was coupled with both B and B. Similarly, if B is a class hierarchy which supports specialization-generalization relationship, then if new subclasses are added to B, no changes need to be made to a class A which calls methods in B. That is, for changing B’s hierarchy, A need not be disturbed. Without this hierarchy, changes in B would most likely result in changes in A.

Within inheritance coupling there are some situations that are worse than others. The worst form is when a subclass B modifies the signature of a method in B (or deletes the method). This situation can easily lead to a run-time error, besides violating the true spirit of the is-a relationship. If the signature is preserved but the implementation of a method is changed, that also violates the is-a relationship, though may not lead to a run-time error, and should be avoided. The least coupling scenario is when a subclass only adds instance variables and methods but does not modify any inherited ones.

7.3.2 Cohesion

Whereas coupling is an inter-module concept, cohesion is an intra-module concept. It focuses on why elements of a module are together in the same module. The objective here is to have elements that are tightly related to belong to the same module. This will make the modules easier to understand,
and as they capture clear concepts and abstractions, easier to modify. Generally, higher cohesion will lead to lower coupling as many elements that need to interact a lot will reside together in strongly coupled modules, lessening the need for interaction with other modules. On the other hand, modules that have low cohesion will often need to interact with other modules to perform their task. Clearly, for making a system more understandable and modifiable, we would like it to consist of modules that are highly cohesive. In other words, the goal is to have a high degree of cohesion in the modules in the system. Cohesion in OO systems also has three aspects [53]:

- Method cohesion
- Class cohesion
- Inheritance cohesion

*Method cohesion* is same as cohesion in functional modules, which we discussed at length in the previous chapter. It focuses on why the different code elements of a method are together within the method. The highest form of cohesion is if each method implements a clearly defined function, and all statements in the method contribute to implementing this function. In general, with functionally cohesive methods, what the method does can be stated easily with a simple statement. That is, in a short and simple statement of the type “this method does...,” we can express the functionality of the method.

*Class cohesion* focuses on why different attributes and methods are together in this class. The goal is to have a class that implements a single concept or abstraction with all elements contributing towards supporting this concept. In general, whenever there are multiple concepts encapsulated within a class, the cohesion of the class is not as high as it could be, and a designer should try to change the design to have each class encapsulate a single concept.

One symptom of the situation where a class has multiple abstractions is that the set of methods can be partitioned into two (or more) groups, each accessing a distinct subset of the attributes. That is, the set of methods and attributes can be partitioned into separate groups, each encapsulating a different concept. Clearly, in such a situation, by having separate classes encapsulating separate concepts, we can have modules with improved cohesion.
7.3. 

**DESIGN CONCEPTS**

In many situations, even though two (or more) concepts may be encapsulated within a class, there are some methods that access attributes of both the encapsulated concepts. This happens, when the class represents different entities which have a relationship between them. For cohesion, it is best to represent them as two separate classes with relationship among them. That is, we should have multiple classes, with some methods in these classes accessing objects of the other class. In a way, this improvement in cohesion results in an increased coupling. However, for modifiability and understandability, it is better if each class encapsulates a single concept.

*Inheritance cohesion* focuses on why classes are together in an hierarchy. The two main reasons for inheritance are to model generalization-specialization relationship, and for code reuse. Cohesion is considered high if the hierarchy supports generalization-specialization of some concept (which is likely to naturally lead to reuse of some code). It is considered lower if the hierarchy is primarily for sharing code with weak conceptual relationship between superclass and subclasses. In other words, it is desired that in an OO system the class hierarchies should be such that they support clearly identified generalization-specialization relationship.

### 7.3.3 The Open-Closed Principle

This is a design concept which came into existence in the OO context. Like with cohesion and coupling, the basic goal here is again to promote building of systems that are easily modifiable, as modification and change happen frequently and a design that cannot easily accommodate change will result in systems that will die fast and will not be able easily adapt to the changing world.

The basic principle, as stated by Bertrand Myers is “Software entities should be open for extension, but closed for modification”[15]. A module being “open for extension” means that its behavior can be extended to accommodate new demands placed on this module due to changes in requirements and system functionality. The modules being “closed for modification” means that the existing source code of the module is not changed when making enhancements.

Then how does one make enhancements to a module without changing the existing source code? This principle restricts the changes to modules to extension only, i.e., it allows addition of code, but disallows changing of existing code. If this can be done, clearly, the value is tremendous. Code changes involve heavy risk and to ensure that a change has not “broken”
things that were working often requires a lot of regression testing. This risk can be minimized if no changes are made to existing code. But if changes are not made, how will enhancements be made? This principle says that enhancements should be made by adding new code, rather than altering old code.

There is another side benefit of this. Programmers typically prefer writing new code rather than modifying old code. But the reality is that systems that are being built today are being built on top of existing software. If this principle is satisfied, then we can expand existing systems by mostly adding new code to old systems, and minimizing the need for changing code.

This principle can be satisfied in OO designs by properly using inheritance and polymorphism. Inheritance allows creating new classes that will extend the behavior of existing classes without changing the original class. And it is this property that can be used to support this principle. As an example consider an application in which a client object (of type Client) interacts with a printer object (of class Printer1) and invokes the necessary methods for completing its printing needs. The class diagram for this will be as shown in Figure 7.7.

In this design, the client directly calls the methods on the printer object for printing something. Now suppose the system has to be enhanced to allow another printer to be used by the client. Under this design, to implement this change, a new class Printer2 will have to be created and the code of the client class will have to be changed to allow using object of Printer2 type as well. This design does not support the open-closed principle as the Client class is not closed against change.

The design for this system, however, can be done in another manner that supports the open-closed principle. In this design, instead of directly implementing the Printer1 class, we create an abstract class Printer that defines the interface of a printer and specifies all the methods a printer object should support. Printer1 is implemented as a specialization of this class. In this design, when Printer2 is to be added, it is added as another subclass of type Printer. The client does not need to be aware of this subtype as it
interacts with objects of type Printer. That is, the client only deals with a
generic Printer, and its interaction is same whether the object is actually of
type Printer1 or Printer2. The class diagram for this is shown in Figure 7.8.

It is this inheritance property of OO that is leveraged to support the
open-closed principle. The basic idea is to have a class encapsulate the ab­
straction of some concept. If this abstraction is to be extended, the extension
is done by creating new subclasses of the abstraction, thereby keeping all the
existing code unchanged.

If inheritance hierarchies are built in this manner, they are said to satisfy
the Liskov Substitution Principle [112]. According to this principle, if a
program is using object o1 of a (base) class C, that program should remain
unchanged if o1 is replaced by an object o2 of a class C, where C is a subclass
of C. If this principle is satisfied for class hierarchies, and hierarchies are used
properly, then the open-closed principle can be supported. It should also be
noted that recommendations for both inheritance coupling and inheritance
cohesion support that this principle be followed in class hierarchies.

7.3.4 Some Design Guidelines

In an OO design, class definitions make up the bulk of the system definition.
Therefore, the design of classes has a major impact on the overall quality
of the design. Here we present a set of guidelines for class design that can
be used to produce “good quality” classes [107], or reusable classes [103].
Most of these rules, and their intent, are self-explanatory and based on the
preceding discussion of design concepts.

1. The public interface of a class should only contain the operations defined on the class. That is, the data definitions should not be a part of the public interface.

2. Only the operations that form the interface for a class, that is, the ones needed by the users of the class, should be the public members of the class.

3. An instance of a class should not send messages directly to components of another class. That is, if there is a class C defined inside a class B, then objects of a class A should not directly perform operations on objects of class C (though many languages will permit it).

4. Each operation defined on a class should be such that it either modifies or accesses some data defined in the class.

5. A class should be dependent on as few classes as possible.

6. The interaction between two classes should be explicit. That is, global objects should be avoided, and any objects needed by an object should be explicitly passed as a parameter or accessed through other explicitly defined means.

7. Each subclass should be developed as a specialization of the superclass with the public interface of the superclass becoming part of the public interface of the subclass.

8. The inheritance hierarchy should model some hierarchy that naturally exists, and the class definition at each level should represent some concept. The top of the hierarchy should be an abstract class.

9. Inside a class, case analysis on object type should be avoided. If this is needed, it should be done by sending messages.

10. The number of arguments and the size of methods should be kept small.
7.4 Unified Modeling Language (UML)

Most design approaches have two aspects to them—a language or a notation to express the design, particularly while it is being developed, and a methodology for developing the design. As design is a creative and iterative activity, a good notation should aid the designer during the design activity. This means that the notation should allow the designer to succinctly capture the key aspects of the design (and refine it later), and allow easy communication to encourage brainstorming. With good notation, often the methodology for design becomes a set of general rules, and the notation becomes the primary tool for creating the design.

Unified Modeling Language (UML) is a graphical notation for expressing object oriented designs [24, 124, 64]. It is called a modeling language and not a design notation as it allows representing various aspects of the system, not just the design that has to be implemented. For a design, a specification of the classes that exist in the system might suffice. However, while modeling, during the design process, the designer also tries to understand how the different classes are related and how they interact to provide the desired functionality. This aspect of modeling helps build designs that are more likely to satisfy all the requirements of the system. Due to the ability of UML to create different models, it has become an aid for understanding the system, designing the system, as well as a notation for representing design.

Though UML has now evolved into a fairly comprehensive and large modeling notation, we will focus on a few central concepts and notations relating to classes and their relationships and interactions. Though we have already seen some of the notation when discussing OO analysis, we discuss it here independently for sake of completeness. For a more detailed discussion on UML, the reader is referred to [24, 124, 64].

7.4.1 Class Diagram

The class diagram of UML is the central piece in a design or model. As the name suggests, these diagrams describe the classes that are there in the design. As the final code of an OO implementation is mostly classes, these diagrams have a very close relationship with the final code. There are many tools that translate the class diagrams to code skeletons, thereby avoiding errors that might get introduced if the class diagrams are manually translated to class definitions by programmers. A class diagram defines
1. **Classes that exist in the system**—besides the class name, the diagrams are capable of describing the key fields as well as the important methods of the classes.

2. **Associations between classes**—what types of associations exist between different classes.

3. **Subtype, supertype relationship**—classes may also form subtypes giving type hierarchies using polymorphism. The class diagrams can represent these hierarchies also.

A class itself is represented as a rectangular box which is divided into three areas. The top part gives the class name. By convention the class name is a word with the first letter in uppercase. (In general, if the class name is a combination of many words, then the first letter of each word is in uppercase.) The middle part lists the key attributes or fields of the class. These attributes are the state holders for the objects of the class. By convention, the name of the attributes starts with a lowercase, and if multiple words are joined, then each new word starts with an uppercase. The bottom part lists the methods or operations of the class. These represent the behavior that the class can provide. Naming conventions are same as for attributes but to show that it is a function, the names end with “( )”. (The parameters of the methods can also be specified, if desired.)

Sometimes, designers may like to specify the responsibility of a class. The responsibility is what the entire class is meant to do using its attributes and methods. Some designers feel that cohesive classes have clearly defined responsibility. If responsibility needs to be specified, it is typically done by having a 4th part at the bottom of the class box and specifying the responsibility in it as plain text.

If a class is an interface (having specifications but no body,) this can be specified by marking the class with the stereotype “<< interface >>”, which is generally written above the class name. Similarly, if a class/method/attribute has some properties that we want to specify, it can be done by tagging the entity by specifying the property next to the entity name within “{” and “}” or by putting some special symbol. Example of a class, an interface, and a class with some tagged values is shown in Figure 7.9.

The divided-box notation is to describe the key features of a class as a stand alone entity. However, classes have relationships between them, and objects of different classes interact. Therefore, to model a system or an application, we must represent relationship between classes. One common
relationship is the generalization-specialization relationship between classes, which finally gets reflected as the inheritance hierarchy. In this hierarchy, properties of general significance are assigned to a more general class—the superclass—while properties which can specialize an object further are put in the subclass. All properties of the superclass are inherited by the subclass, so a subclass contains its own properties as well as those of the superclass.

The generalization-specialization relationship is specified by having arrows coming from the subclass to the superclass, with the empty triangle shaped arrow-head touching to the superclass. Often, when there are multiple subclasses of a class, this may be specified by having one arrow head on the superclass, and then drawing lines from this to the different subclasses. In this hierarchy, often specialization is done on the basis of some discriminator—a distinguishing property that is used to specialize superclass into different subclasses. In other words, by using the discriminator, objects of the superclass type are partitioned into sets of objects of different subclass types. The discriminator used for the generalization-specialization relationship can be specified by labeling the arrow. An example of how this relationship is modeled in UML is shown in 7.10.

In this example, the IITKPerson class represents all people belonging to the IITK. These are broadly divided into two subclasses—Student and Employee, as both these types have many different properties (some common ones also) and different behavior. Similarly, students have two different subclasses, UnderGraduate and PostGraduate, both requiring some different attributes and having different constraints. The Employee class has subtypes representing the faculty, staff, and research staff. (This hierarchy is from an actual working system developed for the author's Institute.)
Besides the generalization-specialization relationship, another common relationship is association, which allows objects to communicate with each other. An association between two classes means that an object of one class needs some services from objects of the other class to perform its own service. The relationship is that of peers in that objects of both the classes can use services of the other. The association is shown by a line between the two classes. An association may have a name which can be specified by labeling the association line. (The association can also be assigned some attributes of its own.) And if the roles of the two ends of the association need to be named, that can also be done. In an association, an end may also have multiplicity allowing relationships like 1 to 1, or 1 to many be modeled. Where there is a fixed multiplicity, it is represented by putting a number at that end; a zero or many multiplicity is represented by a *.
Another type of relationship is the part-whole relationship which represents the situation when an object is composed of many parts, each part itself is an object. This situation represents containment or aggregation—i.e. object of a class are contained inside the object of another class. (Containment and aggregation can be treated separately and shown differently, but we will consider them as the same.) For representing this aggregation relationship, the class which represents the “whole” is shown at the top and a line emanating from a little diamond connecting it to classes which represent the parts. Often in an implementation this relationship is implemented in the same manner as an association, hence, this relationship is also sometimes modeled as an association.

The association and aggregation are shown in Figure 7.11, expanding the example given above. An object of IITKPerson type contains two objects of type Address, representing the permanent address and the current address. It also contains an object of type BiometricInfo, which keeps information like the person’s picture and signature. As these objects are common to all people, they belong in the parent class rather than a subclass. An IITKPerson is allowed to take some advances from the Institute to meet expenses for travel, medical, etc. Hence, Advances is a different class (which, incidentally, has a hierarchy of its own) to which IITKPerson class as a 1 to \( m \) association. (These relations are also from the system.)

Class diagrams focus on classes, and should not be confused with object diagram. Objects are specific instances of classes. Sometimes, it is desirable to model specific objects and relationship between them, and for that object diagrams are used. An object is represented like a class, except that its name also specifies the name of the class to which it belongs. Generally, the object name starts with lowercase, and the class name is specified after a colon. To further clarify, the entire name is underlined. An example is, myList: List. The attributes of an object may have specific values. These values can be specified by giving them along with the attribute name (E.g. name = “John”).

7.4.2 Sequence and Collaboration Diagrams

Class diagrams represent the static structure of the system, or they capture what is the structure of the code that may implement it, and how the different classes in the code are related. Class diagrams, however, do not represent the dynamic behavior of the system. That is, how the system behaves when it performs some of its functions cannot be represented by class diagrams.
This is done through sequence diagrams or collaboration diagrams, together called interaction diagrams. An interaction diagram typically captures the behavior of a use case and models how the different objects in the system collaborate to implement the use case. Let us first discuss sequence diagrams, which is perhaps more common of the two interaction diagrams.

A sequence diagram shows the series of messages exchanged between some objects, and their temporal ordering, when objects collaborate to provide some desired system functionality (or implement a use case). The sequence diagram is generally drawn to model the interaction between objects for a particular use case. Note that in a sequence diagram (and also in collaboration diagrams), it is objects that participate and not classes. When capturing dynamic behavior, the role of classes are limited as during execution it is objects that exist.

In a sequence diagram, all the objects that participate in the interaction are shown at the top as boxes with object names. For each object, a vertical bar representing its lifeline is drawn downwards. A message from one object to another is represented as an arrow from the lifeline of one to the lifeline of the other. Each message is labeled with the message name, which typically should be the name of a method in the class of the target object. An object can also make a self call, which is shown as an message starting and ending in the same objects lifeline. To clarify the sequence of messages and relative timing of each, time is represented as increasing as one moves farther away downwards from the object name in the object life. That is, time is represented by the y-axis, increasing downwards.

Using the lifeline of objects and arrows, one can model objects lives and how messages flow from one object to another. However, frequently a message is sent from one object to another only under some condition. This condition can be represented in the sequence diagram by specifying it within brackets before the message name. If a message is sent to multiple receiver objects, then this multiplicity is shown by having a "*" before the message name.

Each message has a return, which is when the operation finishes and returns the value (if any) to the invoking object. Though often this message can be implied, sometimes it may be desirable to show the return message explicitly. This is done by showing a dashed arrow. A sequence diagram for an example is shown in Figure 7.12. This example is for printing the graduation report for students. The object for GradReport (which has the responsibility for printing the report) sends a message to the Student objects for the relevant information, which request the CourseTaken objects for the
courses the student has taken. These objects get information about the courses from the Course objects. (This example is discussed in greater length later in Chapter 9, where this implementation is improved through refactoring. The class diagram is also given in that Chapter in Figure 9.5.)

A collaboration diagram also shows how objects communicate. Instead of using a timeline-based representation that is used by sequence diagrams, a collaboration diagram looks more like a state diagram. Each object is represented in the diagram, and the messages sent from one object to another are shown as numbered arrows from one object to the other. In other words, the chronological ordering of messages is captured by message numbering, in contrast to a sequence diagram where ordering of messages is shown pictorially. As should be clear, the two types of interaction diagrams are semantically equivalent and have the same representation power. The collaboration diagram for the above example is shown in Figure 7.13. Over the years, however, sequence diagrams have become more popular, as people find the visual representation of sequencing quicker to grasp.

As we can see, an interaction diagram models the internal dynamic behavior of the system, when the system performs some function. The internal dynamics of the system is represented in terms of how the objects interact with each other. Through an interaction diagram, one can clearly see how a system internally implements an operation, and what messages are sent
between different objects. If a convincing interaction diagram cannot be constructed for a system operation with the classes that have been identified in the class diagram, then it is safe to say that the system structure is not capable of supporting this operation and that it must be enhanced. So, it can be used to validate if the system structure being designed through class diagrams is capable of providing the desired services.

As a system has many functions, each involving different objects in different ways, there will be a dynamic model for each of these functions or use cases. In other words, whereas one class diagram can capture the structure of the system’s code, for the dynamic behavior many diagrams are needed. Many systems may be performing many functions and it may not be feasible or practical to draw the interaction diagram for each of these. Typically, during design, interaction diagram of some key use cases or functions will be drawn to make sure that the classes that exist can indeed support the desired use cases, and to understand their dynamics. So, while creating the design, it should be kept in mind that while one class diagram is needed to represent the structure of the system, an interaction diagram represents interactions of objects for one of the many scenarios.
7.4.3 Other Diagrams and Capabilities

UML is an extensible and quite elaborate modeling notation. Above we have discussed notation related to two of the most common models developed while modeling a system—class diagrams and interaction diagrams. These two together help model the static structure of the system as well as the dynamic behavior. There are, however, many other aspects that might need to be modeled for which extra notation is required. UML provides notation for many different types of models.

In modeling and building systems, often instead of classes, components are used. Components often encapsulate "larger" elements, and are semantically simpler than classes. Components often encapsulate subsystems and provide clearly defined interfaces through which these components can be used by other components in the system. While designing an architecture, as we have seen, components are very useful. UML provides a notation for specifying a component. UML also provides a separate notation for a subsystem. In a large system, many classes may be combined together to form packages, where a package is a collection of many elements, possibly of different types. UML also provides a notation to specify packages. These are shown in Figure 7.14.

In the chapter on Architecture we discussed the deployment view of the system, which may be quite different from the component or module view. In deployment view, the focus is what software element uses which hardware, that is, how is the system deployed. UML has notation for representing a
deployment view. The main element is a *node*, represented as a named cube, which represents a computing resource like the CPU which physically exists. The name of the cube identifies the resource as well as its type. Within the cube for the node the software elements it deploys (which can be components, packages, classes, etc.) are shown using their respective notation. If different nodes communicate with each other, this is shown by connecting the nodes by lines.

The notation for packages and deployment view provide structural views of the system from different perspectives. UML also provides notation to express different types of behavior. A *state diagram* is a model in which the entity being modeled is viewed as a set of states, with transitions between the states taking place when some event occurs. A state is represented as a rectangle with rounded edges or as ellipses or circles; transitions are represented by arrows connecting two states. Details can also be attached to transitions. State diagrams are often used to model the behavior of objects of a class—the state represents the different states of the object and transition captures the performing of the different operations on that object. So, whereas interaction diagrams capture how objects collaborate, a state diagram models how an object itself evolves as operations are performed on it. This can help clearly elucidate and specify the behavior of a class. We will discuss it further in the next chapter, as we view state diagrams as helping in doing the detailed design of a class.

*Activity Diagrams* provide another method for modeling dynamic behavior. These diagrams model a system by modeling the activities that take place in it when the system executes for performing some function. Each activity is represented like an oval, with the name of the activity within it. From the activity, the system proceeds to other activities. Often, which activity to perform next depends on some decision. This decision is shown as a diamond leading to multiple activities (which are the options for this decision). Repeated execution of some activities can also be shown. These diagrams are like flow charts, but also have notation to specify parallel execution of activities in a system by specifying an activity splitting into multiple activities or many activities joining (synchronizing) after their completion.

UML is an extensible notation allowing a modeler the flexibility to represent newer concepts as well. There are many situations in which a modeler needs some notation which is similar to an existing one but is not exactly the same. For example, in some cases, one may want to specify if a class is an abstract class or an interface. Instead of having special notation for these concepts, UML has the concept of a stereotype, through which existing
notation can be used to model different concepts. An existing notation, for example of a class, can be used to represent some other similar concept by specifying it as a stereotype by giving the name of the new concept within << and >>. We have already seen an example earlier. A metaclass can be specified in a similar manner; and so can a utility class (one which has some utility functions which are directly used and whose objects are not created).

Tagged values can be used to specify additional properties of the elements to which they are attached. They can be attached to any name, and are specified within “{ }”. Though tagged values can be anything a modeler wants, it is best to limit its use to a few clearly defined (and pre-agreed) properties like private, abstract, query, and readonly. Notes can also be attached to the different elements in a model. We have earlier seen the use of some tagged values in Figure 7.9.

We discussed use cases and use case diagrams in an earlier chapter. Use case diagrams are part of the UML. However, as discussed earlier, use case diagrams add little additional information that use cases do not provide. They are mostly used for providing a high-level summary of use cases.

7.5 A Design Methodology

Many design and analysis methodologies have been proposed. Some of the earlier ones are [23, 37, 95, 133]. As we stated earlier, a methodology basically uses the concepts (of OO in this case) to provide guidelines and notation for the design activity. Though methodologies are useful, they do not reduce the activity of design to a sequence of steps that can be followed mechanically. Due to this, the overall approach and the principles behind it are often more useful than the details of the methodologies. In fact, most experienced designers tailor the methodology to suit their way of thinking and working. We will discuss only one particular methodology here, as at an abstract level most methodologies start to seem very similar and vary mostly in details. Even though it is one of the earlier methodologies, its basic concepts are still applicable.

We assume that during architecture design the system has been broken into high-level subsystems or components. The problem we address is how to produce an object-oriented design for a subsystem, which can itself be viewed as a system.

As we discussed earlier, the OO design of consists of specification of all the classes and objects that will exist in the system implementation. A
complete OO design should be such that in the implementation phase only further details about methods or attributes need to be added. A few low-level objects may be added later, but most of the classes and objects and their relationships are identified during design.

In OO design, the OO analysis forms the starting step. Using the model produced during analysis, a detailed model of the final system is built. As we discussed earlier, in an object-oriented approach, the separation between analysis and design is not very clear and depends on the perception. We will follow what we defined in Chapter 4 regarding what constitutes the output of an OOA—a class diagram of the problem. The OMT methodology that we discuss for design considers dynamic modeling and functional modeling parts of the analysis [133]. As these two models have little impact on the object model produced in OOA or on the SRS, we view these modeling as part of the design activity. Hence, performing the object modeling can be viewed as the first step of design. With this point of view, the design methodology for producing an OO design consists of the following sequence of steps:

- Produce the class diagram
- Produce the dynamic model and use it to define operations on classes
- Produce the functional model and use it to define operations on classes
- Identify internal classes and operations
- Optimize and package

We discussed object-oriented modeling in Chapter 4, along with a methodology for performing the modeling. Any methodology can be followed, as long as the output of the modeling activity is the class diagram representing the problem structure. Hence, the first step of the design is generally performed during the requirements phase when the problem is being modeled for producing the SRS. Briefly, during analysis, the basic goal is to produce a class diagram of the problem domain. This requires identification of object types in the problem domain, the structures between classes (both inheritance and aggregation), attributes of the different classes, associations between the different classes, and the services each class needs to provide to support the system. For further details, the reader should refer to Chapter 4.
7.5.1 Dynamic Modeling

The class diagram models the static structure of the system. However, just modeling the static structure is not sufficient for designing the system, as the desired effect of the events on the system state will also impact the final structure of the system. So, a better understanding of the dynamic behavior of the system will help in further refining the design.

The dynamic model of a system aims to specify how the state of various objects changes when events occur. An event is something that happens at some time instance. For an object, an event is essentially a request for an operation. An event typically is an occurrence of something and has no time duration associated with it. Each event has an initiator and a responder. Events can be internal to the system, in which case the event initiator and the event responder are both within the system. An event can be an external event, in which case the event initiator is outside the system (e.g., the user or a sensor).

A scenario is a sequence of events that occur in a particular execution of the system, as we have seen while discussing use cases in Chapter 3. From the scenarios, the different events being performed on different objects can be identified, which are then used to identify services on objects. The different scenarios together can completely characterize the behavior of the system. If the design is such that it can support all the scenarios, we can be sure that the desired dynamic behavior of the system can be supported by the design. This is the basic reason for performing dynamic modeling. With use cases, dynamic modeling involves preparing interaction diagrams for the important scenarios, identifying events on classes, ensuring that events can be supported, and perhaps build state models for the classes.

It is best to start by modeling scenarios being triggered by external events. The scenarios should not necessarily cover all possibilities, but the major ones should be considered. First the main success scenarios should be modeled, then scenarios for "exceptional" cases should be modeled. For example, in the system for a restaurant that we discussed in Chapter 4, the main success scenario for placing an order could be:
Customer reads the menu.
Customer places the order.
Order is sent to the kitchen for preparation.
Ordered items are served.
Customer requests for a bill for the order.
Bill is prepared for this order;
Customer is given the bill;
Customer pays the bill.

An "exception" scenario could be if the ordered item was not available or if the customer cancels his order. From each scenario, events have to be identified. Events are interactions with the outside world and object-to-object interactions. All the events that have the same effect on the flow of control in the system are grouped as a single event type. Each event type is then allocated to the object classes that initiate it and that service the event. With this done, a scenario can be represented as a sequence (or collaboration) diagram showing the events that will take place on the different objects if the execution corresponding to the scenario takes place. A possible sequence diagram of the preceding scenario is given in Figure 7.15.

Once the main scenarios are modeled, various events on objects that are needed to support executions corresponding to the various scenarios are known. This information is then used to expand our view of the classes in the design. The main reason for performing dynamic modeling is that scenarios and sequence diagrams extend the initial design. Generally speaking, for each event in the sequence diagrams, there will be an operation on the object on which the event is invoked. So, by using the scenarios and sequence diagrams we can further refine our view of the objects and add operations that are needed to support some scenarios but may not have been identified during initial modeling. For example, from the event trace diagram in Figure 7.15, we can see that "placeOrder" and "getBill" will be two operations required on the object of type Order if this interaction is to be supported.

The effect of these different events on a class itself can be modeled using the state diagrams. We believe that the state transition diagram is of limited use during system design but may be more useful during detailed design. Hence, we will discuss state modeling of classes in the next chapter.
7.5.2 Functional Modeling

The functional model describes the computations that take place within a system. It is the third dimension in modeling—object modeling looks at the static structure of the system, dynamic modeling looks at the events in the system, and functional modeling looks at the functionality of the system. In other words, the functional model of a system specifies what happens in the system, the dynamic model specifies when it happens, and the class model specifies what it happens to [133].

A functional model of a system specifies how the output values are computed in the system from the input values, without considering the control aspects of the computation. This represents the functional view of the system—the mapping from inputs to outputs and the various steps involved in the mapping. Generally, when the transformation from the inputs to outputs is complex, consisting of many steps, the functional modeling is likely to be useful. In systems where the transformation of inputs to outputs is not complex, the functional model is likely to be straightforward.

As we have seen, the functional model of a system (either the problem domain or the solution domain) can be represented by a data flow diagram.
(DFD). We have used DFDs in problem modeling, and the structured design methodology, discussed in Chapter 6. Just as with dynamic modeling, the basic purpose of doing functional modeling, when the goal is to obtain an object-oriented design for the system, is to use the model to make sure that the object model can perform the transformations required from the system. As processes represent operations and in an object-oriented system, most of the processing is done by operations on classes, all processes should show up as operations on classes. Some operations might appear as single operations on an object; others might appear as multiple operations on different classes, depending on the level of abstraction of the DFD. If the DFD is sufficiently detailed, most processes will occur as operations on classes. The DFD also specifies the abstract signature of the operations by identifying the inputs and outputs.

7.5.3 Defining Internal Classes and Operations

The classes identified so far are the ones that come from the problem domain. The methods identified on the objects are the ones needed to satisfy all the interactions with the environment and the user and to support the desired functionality. However, the final design is a blueprint for implementation. Hence, implementation issues have to be considered. While considering implementation issues, algorithm and optimization issues arise. These issues are handled in this step.

First, each class is critically evaluated to see if it is needed in its present form in the final implementation. Some of the classes might be discarded if the designer feels they are not needed during implementation.

Then the implementation of operations on the classes is considered. For this, rough algorithms for implementation might be considered. While doing this, a complex operation may get defined in terms of lower-level operations on simpler classes. In other words, effective implementation of operations may require heavy interaction with some data structures and the data structure to be considered an object in its own right. These classes that are identified while considering implementation concerns are largely support classes that may be needed to store intermediate results or to model some aspects of the object whose operation is to be implemented. The classes for these objects are called container classes.

Once the implementation of each class and each operation on the class has been considered and it has been satisfied that they can be implemented, the system design is complete. The detailed design might also uncover some
very low-level objects, but most such objects should be identified during system design.

7.5.4 Optimize and Package

In the design methodology used, the basic structure of the design was created during analysis. As analysis is concerned with capturing and representing various aspects of the problem, some inefficiencies may have crept in. In this final step, the issue of efficiency is considered, keeping in mind that the final structures should not deviate too much from the logical structure produced by analysis, as the more the deviation, the harder it will be to understand a design. Some of the design optimization issues are discussed next [133].

Adding Redundant Associations. The association in the initial design may make it very inefficient to perform some operations. In some cases, these operations can be made more efficient by adding more associations. Consider the example where a Company has a relationship to a person (a company employs many persons) [133]. A person may have an attribute languages-spoken, which lists the languages the person can speak. If the company sometimes needs to determine all its employees who know a specific language, it has to access each employee object to perform this operation. This operation can be made more efficient by adding an index in the Company object for different languages, thereby adding a new relationship between the two types of objects. This association is largely for efficiency. For such situations, the designer must consider each operation and determine how many objects in an association are accessed and how many are actually selected. If the hit ratio is low, indexes can be considered.

Saving Derived Attributes. A derived attribute is one whose value can be determined from the values of other attributes. As such an attribute is not independent, it may not have been specified in the initial design. However, if it is needed very often or if its computation is complex, its value can be computed and stored once and then accessed later. This may require new objects to be created for the derived attributes. However, it should be kept in mind that by doing this the consistency between derived attributes and base attributes will have to be maintained and any changes to the base attributes may have to be reflected in the derived attributes.

Use of Generic Types. A language like C++ allows "generic" classes to be declared where the base type or the type of some attribute is kept "generic" and the actual type is specified only when the object is actually defined. (The approach of C++ does not support true generic types, and
this type of definition is actually handled by the compiler.) By using generic
types, the code size can be reduced. For example, if a list is to be used in
different contexts, a generic list can be defined and then instantiated for an
integer, real, and char types.

**Adjustment of Inheritance.** Sometimes the same or similar opera­tions are defined in various classes in a class hierarchy. By making the
operation slightly more general (by extending interface or its functionality),
it can be made a common operation that can be “pushed” up the hierarchy.
The designer should consider such possibilities. Note that even if the same
operation has to be used in only some of the derived classes, but in other
derived classes the logic is different for the operation, inheritance can still
be used effectively. The operation can be pushed to the base class and then
redefined in those classes where its logic is different.

Another way to increase the use of inheritance, which promotes reuse,
is to see if abstract classes can be defined for a set of existing classes and
then the existing classes considered as a derived class of that. This will
require identifying common behavior and properties among various classes
and abstracting out a meaningful common superclass. Note that this is
useful only if the abstract superclass is meaningful and the class hierarchy is
“natural.” A superclass should not be created simply to pack the common
features on some classes together in a class.

Besides these, the general design principles discussed earlier should be
applied to improve the design—to make it more compact, efficient, and mod­
ular. Often these goals will conflict. In that case, the designer has to use
his judgment about which way to go. In general, as we stated earlier in the
chapter, understandability and modularity should be given preference over
efficiency and compactness.

### 7.5.5 Examples

Before we apply the methodology on some examples, it should be remem­
bered again that no design methodology reduces the activity of producing
a design to a series of steps that can be mechanically executed; each step
requires some amount of engineering judgment. Furthermore, the design
produced by following a methodology should not be considered the final de­
sign. The design can and should be modified using the design principles and
the ultimate objectives of the project in mind. Methodologies are essentially
guidelines to help the designer in the design activity; they are not hard-and­
fast rules. The examples we give here are relatively small, and all aspects of
the methodology do not get reflected in them. However, the design of the case studies, given at the end of the chapter, will provide a more substantial example for design.

**The Word-Counting Problem**

Let us first consider the word counting problem discussed in Chapter 6 (for which the structured design was done). The initial analysis clearly shows that there is a `File` object, which is an aggregation of many `Word` objects. Further, one can consider that there is a `Counter` object, which keeps track of the number of different words. It is a matter of preference and opinion whether `Counter` should be an object, or counting should be implemented as an operation. If counting is treated as an operation, the question will be to which object it belongs. As it does not belong "naturally" to either the class `Word` nor the class `File`, it will have to be "forced" into one of the classes. For this reason, we have kept `Counter` as a separate object. The basic problem statement finds only these three objects. However, further analysis for services reveals that some history mechanism is needed to check if the word is unique. The class diagram obtained after doing the initial modeling is shown in Figure 7.16.

Now let us consider the dynamic modeling for this problem. This is essentially a batch processing problem, where a file is given as input and some output is given by the system. Hence, the use case and scenario for
this problem are straightforward. For example, the scenario for the "normal" case can be:

- System prompts for the file name; user enters the file name.
- System checks for existence of the file.
- System reads the words from the file.
- System prints the count.

From this simple scenario, no new operations are uncovered, and our object diagram stays unchanged. Now we consider the functional model. One possible functional model is shown in Figure 7.17. The model reinforces the need for some object where the history of what words have been seen is recorded. This object is used to check the uniqueness of the words. It also shows that various operations like increment(), isunique(), and addToHistory() are needed. These operations should appear as operations in classes or should be supported by a combination of operations. In this example, most of these processes are reflected as operations on classes and are already incorporated in the design.

Now we are at the last two steps of design methodology, where implementation and optimization concerns are used to enhance the object model. The first decision we take is that the history mechanism will be implemented by
a binary search tree. Hence, instead of the class History, we have a different class Btree. Then, for the class Word, various operations are needed to compare different words. Operations are also needed to set the string value for a word and retrieve it. The final class diagram is similar in structure to the one shown in Figure 7.16, except for these changes.

The final step of the design activity is to specify this design. This is not a part of the design methodology, but it is an essential step, as the design specification is what forms the major part of the design document. The design specification, as mentioned earlier, should specify all the classes that are in the design, all methods of the classes along with their interfaces. We use C++ class structures for our specification. The final specification of this problem is given next. This specification can be reviewed for design verification and can be used as a basis of implementing the design.

class Word {
    private:
        char *string; // string representing the word
    public:
        bool operator == ( Word ); // Checks for equality
        bool operator < ( Word );
        bool operator > ( Word );
        Word operator = ( Word ); // The assignment operator
        void setWord ( char * ); // Sets the string for the word
        char *getWord ( ); // gets the string for the word
};

class File {
    private:
        FILE inFile;
        char *fileName;
    public:
        Word getWord ( ); // get a word; Invokes operations of Word
        bool isEof ( ); // Checks for end of file
        void fileOpen ( char * );
};

class Counter {
    private:
        int counter;
    public:
        void increment ( );
}
void display ( );
};

class Btree: GENERIC in <ELEMENT_TYPE> {
private:
    ELEMENT_TYPE element;
    Btree < ELEMENT_TYPE > *left;
    Btree < ELEMENT_TYPE > *right;
public:
    void insert( ELEMENT_TYPE ); // to insert an element
    bool lookup( ELEMENT_TYPE ); // to check if an element exists
};

As we can see, all the class definitions complete with data members and
operations and all the major declarations are given in the design specifica-
tion. Only the implementation of the methods are not provided. This design
was later implemented in C++. The conversion to code required only minor
additions and modifications to the design. The final code was about 240
lines of C++ code (counting noncomment and nonblank lines only).

Rate of Returns Problem

Let us consider a slightly larger problem: that of determining the rate of
returns on investments. An investor has made investments in some compa-
nies. For each investment, in a file, the name of the company, all the money
he has invested (in the initial purchase as well as in subsequent purchases),
and all the money he has withdrawn (through sale of shares or dividends)
are given, along with the dates of each transaction. The current value of the
investment is given at the end, along with the date. The goal is to find the
rate of return the investor is getting for each investment, as well as the rate
of return for the entire portfolio. In addition, the amounts he has invested
initially, amounts he has invested subsequently, amounts he has withdrawn,
and the current value of the portfolio also is to be output.

This is a practical problem that is frequently needed by investors (and
forms the basis of our second Case Study). The computation of rate of re-
turn is not straightforward and cannot be easily done through spreadsheets.
Hence, such a software can be of practical use. Besides the basic functional-
ity given earlier, the software needs to be robust and catch errors that can
be caught in the input data.

We start with the analysis of the problem. Initial analysis clearly shows
that there are a few object classes of interest—Portfolio, Investment,
and Transaction. A portfolio consists of many investments, and an investment consists of many transactions. Hence, the class Portfolio is an aggregation of many Investments, and an Investment is an aggregation of many Transactions. A transaction can be of Withdrawal type or Deposit type, resulting in a class hierarchy, with Investment being the superclass and Withdrawal and Deposit subclasses.

For an object of class Investment, the major operation we need to perform is to find the rate of return. For the class Portfolio we need to have operations to compute rate of return, total initial investment, total withdrawal, and total current value of the portfolio. Hence, we need operations for these. The class diagram obtained from analysis of the problem is shown in Figure 7.18.

In this problem, as the interaction with the environment is not much, the dynamic model is not significant. Hence, we omit the dynamic modeling for this problem. A possible functional model is given in Figure 7.19. The classes are then enhanced to make sure that each of the processes of the functional model is reflected as operations on various objects. As we can see, most of the processes already exist as operations.

Now we have to perform the last two steps of the design methodology,
where implementation and optimization concerns are used to enhance the classes. While considering the implementation of computation of total initial investment, computation of overall return rate, overall withdrawals and so on, we notice that for all of these, appropriate data from each investment is needed. Hence, to the class Investments, appropriate operations need to be added. Further, we note that all the computations for total initial investment, total current value, and so on are all done together, and each of these is essentially adding values from various investments. Hence, we combine them in a single operation in Portfolio and a corresponding single operation in Investment. Studying the class hierarchy, we observe that the only difference in the two subclasses Withdrawal and Deposit is that in one case the amount is subtracted and in the other it is added. In such a situation, the two types can be easily considered a single type by keeping the amount as negative for a withdrawal and positive for a deposit. So we remove the subclasses, thereby simplifying the design and implementation. Instead of giving the class diagram for the final design, we provide the specification of the classes:

```cpp
class Transaction {
    private:
        int amount; // money amount for the transaction
        int month;   // month of the transaction
        int year;    // year of the transaction
```
public:
    getAmount();
    getMonth();
    getYear();
    Transaction(amount, month, year); // sets values
};

class Investment {
private:
    char *investmentName; // Name of the company
    Transaction *transactArray; // List of transactions
    int noOfTransacts; // Total number of transactions
    float rateOfReturn; // rate of return
public:
    getTransactDetails(); // Set details of transactions
    computeRate();
    float getRate(); // Return the rate of the returns
    compute(initVal, totWithdrawls, totCurVal, totDeposits);
        // Returns these values for this investment
};

class Portfolio {
private:
    Investment *investArray; // List of investments
    int noOfInvestments; // Total number of investments
    int totalInitInvest;
    int totalDeposits;
    int totalCurVal;
    int totalWithdrawl;
    float RateOfReturns; // Overall rate of returns
public:
    getInvestDetails( char * fname ); // Parse the input file
    computeRate(); // Compute rates of return
    compute(); // Compute other totals
    printResults(); // Print return rates, total values, etc.
};

The design is self-explanatory. This design was later implemented in C++ code, and we found that only minor implementation details were added during the implementation, showing the correctness and completeness of the design. The final size of the program was about 470 lines of C++ code (counting noncomment and nonblank lines only).
7.6 Metrics

We have already seen that the basic paradigm behind OOD is fundamentally different from the paradigm of function-oriented design. This has brought in a different building block and concepts related to this building block. The definition of modularity has also changed for this new building block, and new methodologies have been proposed for creating designs using this paradigm. It is, therefore, natural to expect that a new set of metrics will be required to evaluate an OO design. A few attempts have been made to propose metrics for object-oriented software [1, 32, 111].

Here we present some metrics that have been proposed for evaluating the complexity of an OOD. As design of classes is the central issue in OOD and the major output of any OOD methodology is the class definition, these metrics focus on evaluating classes. Note that for measuring the size of a system, conventional approaches, which measure the size in LOC or function points, can be used, even if OO is used for design. It is the metrics for evaluating the quality or complexity of the design that need to be redefined for OOD. The metrics discussed were proposed in [32], and the discussion is based on this work. The results of an experiment described in [6] for validating these metrics and the metrics data presented in [32] are used to discuss the role of these metrics.

**Weighted Methods per Class (WMC)**

The effort in developing a class will in some sense be determined by the number of methods the class has and the complexity of the methods. Hence, a complexity metric that combines the number of methods and the complexity of methods can be useful in estimating the overall complexity of the class. The weighted methods per class (WMC) metric does precisely this.

Suppose a class $C$ has methods $M_1, M_2, \ldots, M_n$ defined on it. Let the complexity of the method $M_i$ be $c_i$. As a method is like a regular function or procedure, any complexity metric that is applicable for functions can be used to define $c_i$ (e.g., estimated size, interface complexity, and data flow complexity). The WMC is defined as:

$$WMC = \sum_{i=1}^{i=n} c_i.$$  

If the complexity of each method is considered 1, WMC gives the total number of methods in the class.
The data given in [6, 32], which is based on evaluation of some existing programs, shows that in most cases, the classes tend to have only a small number of methods, implying that most classes are simple and provide some specific abstraction and operations. Only a few classes have many methods defined on them. The analysis in [6] showed that the WMC metric has a reasonable correlation with fault-proneness of a class. As can be expected, the larger the WMC of a class the better the chances that the class is fault-prone.

**Depth of Inheritance Tree (DIT)**

Inheritance is, as we have mentioned, one of the unique features of the object-oriented paradigm. As we have said before, inheritance is one of the main mechanisms for reuse in OOD—the deeper a particular class is in a class hierarchy, the more methods it has available for reuse, thereby providing a larger reuse potential. At the same time, as we have mentioned, inheritance increases coupling, which makes changing a class harder. In other words, a class deep in the hierarchy has a lot of methods it can inherit, which makes it difficult to predict its behavior. For both these reasons, it is useful to have some metric to quantify inheritance. The depth of inheritance tree (DIT) is one such metric.

The DIT of a class $C$ in an inheritance hierarchy is the depth from the root class in the inheritance tree. In other words, it is the length of the shortest path from the root of the tree to the node representing $C$ or the number of ancestors $C$ has. In case of multiple inheritance, the DIT metric is the maximum length from a root to $C$.

The data in [6, 32] suggests that most classes in applications tend to be close to the root, with the maximum DIT metric value (in the applications studied) being around 10. Most the classes have a DIT of 0 (that is, they are the root). This seems to suggest that the designers tend to keep the number of abstraction levels (reflected by the levels in the inheritance tree) small, presumably to aid understanding. In other words, designers (of the systems evaluated) might be giving up on reusability in favor of comprehensibility. The experiments in [6] show that DIT is very significant in predicting defect-proneness of a class: the higher the DIT the higher the probability that the class is defect-prone.
Number of Children (NOC)

The number of children (NOC) metric value of a class $C$ is the number of immediate subclasses of $C$. This metric can be used to evaluate the degree of reuse, as a higher NOC number reflects reuse of the definitions in the superclass by a larger number of subclasses. It also gives an idea of the direct influence of a class on other elements of a design—the larger the influence of a class, the more important that the class is correctly designed.

In the empirical observations, it was found that classes generally had a small NOC metric value, with a vast majority of classes having no children (i.e., NOC is 0). This suggests that in the systems analyzed, inheritance was not used very heavily. However, the data in [6] seems to suggest that the larger the NOC, the lower the probability of detecting defects in a class. That is, the higher NOC classes are less defect-prone. The reasons for this are not definitive.

Coupling Between Classes (CBC)

As discussed earlier, coupling between modules of a system, in general, reduces modularity and makes module modification harder. In OOD, as the basic module is a class, it is desirable to reduce the coupling between classes. The less coupling of a class with other classes, the more independent the class, and hence it will be more easily modifiable. Coupling between classes (CBC) is a metric that tries to quantify coupling that exists between classes.

The CBC value for a class $C$ is the total number of other classes to which the class is coupled. Two classes are considered coupled if methods of one class use methods or instance variables defined in the other class. In general, whether two classes are coupled can easily be determined by looking at the code and the definitions of all the methods of the two classes. However, note that there are indirect forms of coupling (through pointers, etc.) that are hard to identify by evaluating the code.

The experimental data indicates that most of the classes are self-contained and have a CBC value of 0, that is, they are not coupled with any other class, including superclasses [32]. Some types of classes, for example the ones that deal with managing interfaces (called interface objects earlier), generally tend to have higher CBC values. The data in [6] found that CBC is significant in predicting the fault-proneness of classes, particularly those that deal with user interfaces.
Response for a Class (RFC)

Although the CBC for a class captures the number of other classes to which this class is coupled, it does not quantify the "strength" of interconnection. In other words, it does not explain the degree of connection of methods of a class with other classes. Response for a class (RFC) tries to quantify this by capturing the total number of methods that can be invoked from an object of this class.

The RFC value for a class $C$ is the cardinality of the response set for a class. The response set of a class $C$ is the set of all methods that can be invoked if a message is sent to an object of this class. This includes all the methods of $C$ and of other classes to which any method of $C$ sends a message. It is clear that even if the CBC value of a class is 1 (that is, it is coupled with only one class), the RFC value may be quite high, indicating that the "volume" of interaction between the two classes is very high. It should be clear that it is likely to be harder to test classes that have higher RFC values.

The experimental data found that most classes tend to invoke a small number of methods of other classes. Again, classes for interface objects tend to have higher RFC values. The data in [6] found that RFC is very significant in predicting the fault-proneness of a class—the higher the RFC value the larger the probability that the class is defect-prone.

Lack of Cohesion in Methods (LCOM)

This last metric in the suite of metrics proposed in [32] tries to quantify cohesion of classes. As we have seen, along with low coupling between modules, high cohesion is a highly desirable property for modularity. For classes, cohesion captures how closely bound are the different methods of the class. One way to quantify this is given by the LCOM metric.

Two methods of a class $C$ can be considered "cohesive" if the set of instance variables of $C$ that they access have some elements in common. That is, if $I_1$ and $I_2$ are the set of instance variables accessed by the methods $M_1$ and $M_2$, respectively, then $M_1$ and $M_2$ are similar if $I_1 \cap I_2 \neq \phi$. Let $Q$ be the set of all cohesive pairs of methods, that is, all $(M_i, M_j)$ such that $I_i$ and $I_j$ have a non-null intersection. Let $P$ be the set of all noncohesive pairs of methods, that is, pairs such that the intersection of sets of instance variables they access is null. Then LCOM is defined as

$$LCOM = |P| - |Q|, \text{ if } |P| > |Q| \quad 0 \text{ otherwise.}$$
If there are \( n \) methods in a class \( C \), then there are \( n(n - 1) \) pairs, and LCOM is the number of pairs that are non cohesive minus the number of pairs that are cohesive. The larger the number of cohesive methods, the more cohesive the class will be, and the LCOM metric will be lower. A high LCOM value may indicate that the methods are trying to do different things and operate on different data entities, which may suggest that the class supports multiple abstractions, rather than one abstraction. If this is validated, the class can be partitioned into different classes. The data in [6] found little significance of this metric in predicting the fault-proneness of a class.

In [6], the first five metrics, which were found to be significant in predicting the fault-proneness of classes, were combined to predict the fault-proneness of classes. The experiments showed that the first five metrics, when combined (in this case the coefficients for combination were determined by multivariate analysis of the fault and metric data) are very effective in predicting fault-prone classes. In their experiment, out of a total of 58 faulty classes, 48 classes were correctly predicted as fault-prone. The prediction missed 10 classes and predicted 32 extra classes as fault-prone, although they were not so.

### 7.7 Summary

In the previous chapter we studied how a software system can be designed using functional abstraction as the basic unit. In this chapter, we looked at how a system can be designed using objects and classes as the basic unit. The fundamental difference in this approach from functional approaches is that an object encapsulates state and provides some predefined operations on that state. That is, state (or data) and operations (i.e., functions) are considered together, whereas in the function-oriented approach the two are kept separate.

When using an object-oriented approach, an object is the basic design unit. For objects, during design, the class for the objects is identified. A class represents the type for the object and defines the possible state space for the objects of that class and the operations that can be performed on the objects. An object is an instance of a class and has state, behavior, and identity. Objects in a system do not exist in isolation but are related to each other. One of the goals of design is to identify the relationship between the objects of different classes.

Universal Modeling Language (UML) has become the de-facto standard
for building models of object-oriented systems. UML has various types of diagrams to model different types of properties, and allows both static structure as well as dynamic behavior to be modeled. It is an extensible notation that allows new types to be added.

For representing the static structure, the main diagram is the class diagram, which represents the classes in the system and relationships between the classes. The relationship between the classes may be generalization-specialization, which leads to class hierarchies. The relationship may be that of an aggregation which models the “whole-part of” relationship. Or it may be an association, which models the client-serve type of relationship between classes.

For modeling the dynamic behavior, sequence or collaboration diagrams (together called interaction diagrams) may be used. These diagrams represent how a scenario is implemented by involving different objects. The focus is on capturing the messages that are exchanged between objects to implement a scenario.

There are many other diagrams that UML has proposed that can be used to model other aspects. For example, the state diagram can be used to model behavior of a class. Activity diagrams can model the activities that take place in a system during some execution. For static structure, it provides notation for specifying subsystems, packages, and components.

To ensure that the design is modular, some general properties should be satisfied. The three properties we have discussed are cohesion, coupling, and open-closed principle. Coupling is an inter-class concept and captures how closely the different classes interact with each other and how much they depend on each other. Cohesion is an intra-class concept and captures how strongly the elements of a class are related. Open-closed principle states that the classes should be designed in a manner that they are closed for modification but are open for extension. A good design should have low coupling, high cohesion, and should satisfy the open-closed principle—these make the design more modular and easier to change.

A good modeling notation and principles to evaluate a design are the key necessities for creating good design. Design methodologies help by providing some guidelines of how to create a design. We discussed the object modeling technique for design, which first creates a class model for the system, and then refines it through dynamic modeling, and functional modeling. Identifying the internal classes and optimization are the final steps in this methodology for creating a design.

Finally, we discussed some metrics that can be used to study the com-
plexity of an object-oriented design. We presented one suite of metrics that were proposed, along with some data regarding their validation. The metric weighted methods per class is defined as the sum of complexities of all the methods and gives some idea about how much effort might be needed to develop the class. The depth of inheritance tree of a class is defined as the maximum depth in the class hierarchy of this class, and can represent the potential of reuse that exists for a class, and the degree of coupling between the class and its parent classes. The number of children metric is the number of immediate subclasses of a class, and it can be used to capture the degree of reuse of a class. Coupling of a class is the number of classes whose methods it uses or who use its methods. The response for a class metric is the number of methods that can be invoked by sending a message to this class. It tries to capture the strength of interconnection between classes. Finally, the lack of cohesion metric represents the number of method pairs whose set of access variables have nothing in common minus the number of method pairs that have some common instance variable.

Unlike in previous chapters, we have not discussed verification methods here. The reason is that verification methods discussed in the previous chapters are general techniques that are not specific to function-oriented approaches. Hence, the same general techniques can be used for object-oriented design.

Exercises

1. What is the relationship between abstract data types and classes?
2. Why are private parts of a superclass generally not made accessible to subclasses?
3. In C++, friends of a class $C$ can access the private parts of $C$. For declaring a class $F$ a friend of $C$, where should it be declared—in $C$ or in $F$? Why?
4. What are the different ways in which an object can access another object in a language like C++? (Do not consider the access allowed by being a friend.)
5. What are the potential problems that can arise in software maintenance due to different types of inheritance?
6. What is the relationship between OOA, SRS, and OOD?
7. In the word-counting example, a different functional model was used from the one proposed in Chapter 6. Use the model given in Chapter 6 and modify the OO design.
8. Suppose a simulator for a disk is to be written (for teaching an Operating Systems course). Use OMT to design the simulator.

9. If an association between classes has some attributes of its own, how will you implement it?

10. If we were to use the method described in Chapter 5 to identify error-prone and complex modules, which of the metrics will you use and why (you may also combine the metrics).

11. Design an experiment to validate your proposal for predicting error-prone modules. Specify data collection and analysis.

12. Compare the OO designs and the structured design of the case study to obtain some observations for comparing the two design strategies (this can be considered a research problem).
Case Studies

As with previous chapters, we end this chapter by performing the object-oriented design of the case studies. Here we discuss the application of the design process on the case study, i.e., how the design for the case studies is created. The final design specifications are given on the Web site. While discussing the creation of design, we provide only the main steps to give an idea of the design activity.

Case Study 1—Course Scheduling

We start the design activity by identifying classes of objects in the problem domain and relationship between the classes. From the problem specification, given in Chapter 3, we can clearly identify the following objects: TimeTable, Course, Room, LectureSlot, CToBeSched (course to be scheduled), InputFile_1, and InputFile_2. From the problem, it is clear that TimeTable, an important object in the problem domain, is an aggregation of many TimeTableEntry, each of which is a collection of a Course, a Room where the course is scheduled, and a LectureSlot in which the course is scheduled.

On looking at the description of file 1, we find that it contains a list of rooms, courses, and time slots that is later used to check the validity of entries in file 2. This results in the objects RoomDB, CourseDB, and SlotDB, each of which is an aggregation of many members of Room, Course, and Slot, respectively. Similarly, on looking at the description of file 2, we find that it contains a TableOfCToBeSched, which is an aggregation of many CToBeSched.

On studying the problem further and considering the scheduling constraints imposed by the problem, it is clear that for scheduling, the courses have to be divided into four different types—depending on whether the course is a UG course or a PG course, and whether or not preferences are given. In other words, we can specialize CToBeSched to produce four subclasses: PGwithPref, UGwithPref, PGwithoutPref, and UGwithoutPref. The classes that represent courses with preferences will contain a list of preferences, which is a list of LectureSlots. This is the only hierarchy that is evident from examining the problem.

Considering the attributes of the object classes, the problem clearly specifies that a Room has the attributes roomNo and capacity; a LectureSlot has one major attribute, the slot it represents; and a Course has courseName
as an attribute. A CToBeSched contains a Course and has enrollment as an attribute.

Considering the services for the classes, we identify from the problem specification services like scheduleAll() on TableOfCToBeSched, which schedules all the courses, printTable() for the TimeTable, setentry() and getentry() for a TimeTableEntry, and insert() and lookup() operations for the various lists. The initial class diagram is shown in Figure 7.20.

The system here is not an interactive system; hence dynamic modeling is rather straightforward. The normal scenario is that the inputs are given and the outputs are produced. There are at least two different normal scenarios possible, depending on whether there are any conflicts (requiring conflicts and their reasons to be printed) or not (in which case only the timetable is printed). The latter normal scenario does not reveal any new operations. However, a natural way to model the first scenario is to have an object ConflictTable into which different conflicts for the different time slots of different courses are stored, and from where they are later printed. Hence, we add this object and model it as an aggregation of ConflictTableEntry, with an operation insertEntry() to add a conflict entry in the table and an operation printTable() to print the conflicts. Then there are a number of exception scenarios—one for each possible error in the input. In each case, the scenario shows that a proper error message is to be output. This requires that operations needed on objects like Room, Course and Slot check their formats for correctness. Hence, validation operations are added to these objects.

The functional model for the problem was given in Chapter 6. It shows that from file 1, roomDB, courseDB, and slotDB need to be formed and the entries for each of these have to be obtained from the file and validated. As validation functions are already added, this adds the function for producing the three lists, called build_CRS_DBS(). Similarly, the DFD clearly shows that on InputFile_2 a function to build the table of courses to be scheduled is needed, leading to the adding of the operation buildCtoBeSched(). While building this table, this operation also divides them into the four groups of courses, as done in the DFD. The DFD shows that an operation to schedule the courses is needed. This operation (scheduleAll()) is already there. Although the high-level DFD does not show, but a further refinement of the bubble for “schedule” shows that bubbles are needed for scheduling PG courses with preferences, UG courses with preferences, PG courses without preferences, and UG courses without preferences (they are reflected in the structure chart as modules). These bubbles get reflected as schedule() op-
Figure 7.20: Initial class diagram for the case study.
erations on all four subclasses—PGwithPref, UGwithPref, PGwithoutPref, and UGwithoutPref. The DFD also has bubbles for printing the timetable and conflicts. These get translated into print operations on TimeTable, TimeTableEntry, ConflictTable, and ConflictTableEntry.

Now we come to the last steps of considering implementation concerns. Many new issues come up here. First, we decided to have a generic template class, which can be used to implement the various DBs, as all DBs are performing similar functions. Hence, we defined a template class List. When considering the main issue of scheduling, we notice that scheduling UG courses with preferences, as discussed in the Chapter 6, is not straightforward, as the system has to ensure that it does not make any PG course without preference “unschedulable.” To handle this, we take a simple approach of having a data structure that will reserve slots for PG courses and will then be used to check for the safety of an assignment while scheduling PG courses with preferences. This adds an internal class PGReserve, with operations like isAllotmentSafe() (to check if making an allotment for UG course is “safe”), Initialize() (to initially “mark” all possible slots where PGwithoutPref courses can be scheduled). The structure is then used to schedule the PG courses without preferences after the UG courses with preferences are scheduled, leading to the operation getSuitableSchedule().

To implement the scheduling operation, we decided to use the dynamic binding capability. For each subclass, the schedule() operation that has been defined is made to have the same signature, and a corresponding virtual function is added in the superclass CtoBeScheduled. With this, when the courses are to be scheduled, we can just go over all the courses that need to be scheduled and call the schedule operation. Dynamic binding will ensure that the appropriate schedule operation is called, depending on the type of course (i.e., to which of the four subclasses it belongs). All schedule operations will interact with the TimeTable for checking the conditions specified in the requirements. Various functions are added on TimeTable for this.

Having considered the scheduling operation, we considered the major operation on the files. It becomes clear that to implement these operations, various parsing functions are needed on the two files. These functions are then added. As these operations are only needed to implement the externally visible operations on the class, they are defined as private operations. Considering the public operations on these files reinforce the need for insert() and lookup() operations in the different DBs, these operations require operations to set the attributes of the independent object of which they are an aggregation. Hence, these operations are added. In a similar manner,
while considering implementation issues various other operations on the different object classes were revealed. Various other operations are revealed when considering implementation of other operations. The final class diagram after the design is given in the design document available from the Web site.

As we can see, the class diagram, even for this relatively small system, is quite complex and not easily manageable. Furthermore, it is not practical to properly capture the parameters of the various operations in object diagrams. The types of the various attributes is also frequently not shown to keep the diagram compact. Similarly, all associations do not get reflected. Hence, for specifying the design precisely, this class diagram is translated to a precise specification of the classes. The final design specifications are also given in the design document available from the Web site.

**Case Study 2—PIMS**

The requirements for this case study have been given before. After reviewing the use cases, the following classes clearly emerge.

- Investment
- Portfolio
- Security
- Transaction
- GUI
- NetLoader
- Current Value System
- Alerts
- SecurityManager
- DataRepository

The relationship between them is relatively straightforward. The class diagram containing some of the classes is shown in Figure 7.21. Though this initial class structure was evolved during modeling, later the subtypes
of transaction were eliminated as they provided little useful value. Subtypes of security type were also eliminated.

There are many use cases specified in the SRS for this system. After the initial modeling of these classes and their methods, sequence diagrams for some of the scenarios of some of the use cases are drawn. From this exercise, the specifications of the classes is refined. Some of the sequence diagrams and the specifications of the classes are given in the design document which is available from the Web site.