from which it is flowing. The module size is considered an insignificant factor, and complexity D_c for a module is defined as [155]:

$$D_c = fan_in * fan_out + inflow * outflow$$

where fan_in represents the number of modules that call this module and fan_out is the number of modules this module calls.

The main question that arises is how good these metrics are. For "good," we will have to define their purpose, or how we want to use them. Just having a number signifying the complexity is, in itself, of little use, unless it can be used to make some judgment about cost or quality. One way to use the information about complexity could be to identify the complex modules, as these modules are likely to be more error prone and form "hot spots" later, if they are left as is. Once these modules are identified, the design can be evaluated to see if the complexity is inherent in the problem or if the design can be changed to reduce the complexity.

To identify modules that are "extra complex," we will have to define what complexity number is normal. Having a threshold complexity above which a module is considered complex assumes the existence of a globally accepted threshold value. This may not be possible, as designs in different problem domains produce different types of modules. Another alternative is to consider a module against other modules in the current design only, instead of comparing the modules against a prespecified standard. That is, evaluate the complexity of the modules in the design and highlight modules that are, relatively speaking, more complex. In this approach, the criteria for marking a module complex is also determined from the current design.

One such method for highlighting the modules was suggested in [155]. Let *avg_complexity* be the average complexity of the modules in the design being evaluated, and let *std_deviation* be the standard deviation in the design complexity of the modules of the system. The proposed method classifies the modules in three categories: error-prone, complex, and normal. If D_c is the complexity of a module, it can be classified as follows:

Error-prone	If $D_c > avg_complexity + std_deviation$
Complex	If $avg_complexity < D_c < avg_complexity + std_deviation$
Normal	Otherwise

Note that this definition of error-prone and complex is independent of the metric definition used to compute the complexity of modules. With this approach, a design can be evaluated by itself, not for overall design quality, but to draw attention to the error-prone and complex modules. This information can then be used to redesign the system to reduce the complexity of these modules (which also results in overall complexity reduction). This approach has been found to be very effective in identifying error-prone modules [155]. In evaluations of some completed projects, it has been shown

that error-prone and complex modules together highlight the modules in which most errors occurred [155]. This suggests that for a project, modules thus highlighted during design time point to modules that will be "hot spots" if the design is not improved by reducing their complexity. Another use of this is that even if the complexity of these modules is not reduced (perhaps because the complexity is intrinsic in the problem), identification of error-prone modules can help in quality assurance later; these modules can be required to undergo more rigorous quality assurance.

6.7 Summary

The design of a system is a plan for a solution such that if the plan is implemented, the implemented system will satisfy the requirements of the system and will preserve its architecture. The design activity is a two-level process. The first level produces the *system design* which defines the modules needed for the system, and how the components interact with each other. The *detailed design* refines the system design, by providing more description of the processing logic of components and data structures. A design methodology is a systematic approach to creating a design. Most design methodologies concentrate on system design. During system design a module view of the system is developed, which should be consistent with the component view created during architecture design.

The design process uses the time tested strategy of problem partitioning, through which the complexity of designing large systems is broken into smaller problems that can be solved separately. Effective partitioning depends on the use of abstraction, which permits a designer to concentrate on one module or component at a time by using the abstraction of other modules or components.

Modularity is a means of problem partitioning in software design. A system is considered modular if each component has a well-defined abstraction and if change in one component has minimal impact on other components. Two criteria used for deciding the modules during design are *coupling* and *cohesion.* Coupling is a measure of interdependence between modules, while cohesion is a measure of the strength with which the different elements of a module are related. There are different levels of cohesion, functional and type cohesion being the highest levels and incidental being the lowest. In general, other properties being equal, coupling should be minimized and cohesion maximized.

The structured design method is one of the best known methods for developing the design of a software system. This method creates a structure chart that can be used to implement the system. The goal is to produce a structure where the modules have minimum dependence on each other (low coupling) and a high level of cohesion. The basic methodology has four steps: (1) restate the problem as a data flow graph;

(2) identify the most abstract input and output data elements; (3) perform first-level factoring, which is done by specifying an input module for each of the most abstract inputs, an output module for each of the most abstract outputs, and a transform module for each of the central transforms; and (4) factor each of the input, output, and transform modules.

The methodology does not reduce the problem of design to a series of steps that can be followed blindly. The essential goal is to get a clear hierarchical structure. A number of design heuristics can be used to improve the structure resulting from the application of the basic methodology. The basic guiding principles are simplicity, high cohesion, and low coupling.

The most common method for verifying a design is design reviews or inspection, in which a team of people reviews the design for the purpose of finding defects. If the design is expressed in some formal notation, then some amount of consistency checking can be done automatically through the aid of tools.

There are a number of metrics that can be used to evaluate function-oriented designs. Network metrics evaluate the structure chart and consider deviation from the tree as the metric signifying the quality of design. The stability metric we discussed tries to quantify how resistant the design is to the ripple effects caused by changes by explicitly counting the number of assumptions modules make about each other. The information flow complexity metrics define design complexity based on the internal complexity of the module and the number of connections between modules.

Exercises

1. What is the relationship between an architecture and system-level design?

2. Consider a program containing many modules. If a global variable x must be used to share data between two modules A and B, how would you design the modules to minimize coupling?

3. List a set of poor programming practices, based on the criteria of coupling and cohesion.

4. What is the cohesion of the following module? How would you change the module to increase cohesion?

```
procedure file (file_ptr, file_name, op_name);
begin
    case op_name of
        "open": perform activities for opening the file.
        "close": perform activities for opening the file.
        "print": print the file
    end case
end
```

5. If some existing modules are to be re-used in building a new system, will you use a top-down or bottom-up approach? Why?
6. If a module has logical cohesion, what kind of coupling is this module likely to have with others?
7. What is the difference between a flow chart and a structure chart?
8. Draw the structure chart for the following program:

```
main();
{     int x, y;
      x = 0; y = 0;
      a(); b(); }
a()
{     x = x+y; y = y+5; }
b()
{     x = x+5; y = y+x; a(); }
```

   How would you modify this program to improve the modularity?
9. If a '+' or a '*' is present between two output streams from a transform in a data flow graph, state some specific property about the module for that transform.
10. Use the structured design methodology to produce a design for the following:
    (a) A system to convert ASCII to EBSDIC.
    (b) A system to analyze your diet when given your daily intake (and some data files about different types of food and recommended intakes).
    (c) A system to do student registration in the manner it is done at your college.
    (d) A system to manage the inventory at a hardware store.
    (e) A system for a drug store that will manage inventory, keep track of expiration dates, and track allergy records of patients to avoid issuing medicines that might be harmful.
    (f) A system that acts as a calculator with only basic arithmetic functions.
11. Is this statement true: "If we follow the structured design methodology (without applying any heuristics), the resulting structure will always have one transform module for each bubble in the data flow graph"? Explain your answer.
12. Given a structure with high fan-out, how would you convert it to a structure with a low fan-out?
13. Discuss some approaches on how you can use metrics to guide you in design to produce a design that is easy to modify.
14. Design an experiment to study whether the information flow metrics and stability metrics are correlated.
15. If you have all the metrics data available for design, how will you use this data? Specify your objectives, the metrics you will use, how you will interpret the value, and what possible actions you will take based on the interpretation.

Case Studies

Here we discuss how we went about creating the design for Case Study 1 using the structured design methodology. Here we discuss only the process of creating the design; the design document giving the final design is available from the Web site.

The function-oriented design for the case study 2 was not done and hence is not discussed here.

Structured Design

We first discuss creating the design for Case Study 1 (course scheduling) using structured design methodology. We describe how the design was obtained; the details of the design are available from the Web site.

Data Flow Diagram: This is the first step in the structured design method. In our case study, there are two inputs: file1 and file2. Three outputs are required: the timetable, the conflict table, and the explanations for the schedule. A high-level data flow diagram of this problem is given in Figure 6.12.

The diagram is fairly clear. First we get from file1 the information about classrooms, lecture times, and courses, and we validate their format. The validated input from file1 is used for cross-validating information in file2. After validating the file2 input, we get an array of valid course records (with preferences, etc.) that must be scheduled. Because PG courses have to be scheduled before UG courses, these course records are separated into different groups: PG courses with preferences, UG courses with preferences, PG courses with no preference, and UG courses with no preference. This separated course list is the input to the schedule transform, the output of which is the three desired outputs.

The most abstract input and most abstract output are fairly obvious here. The "separated course schedule" is the most abstract input and the three outputs of the schedule transform are the most abstract outputs. There is only one central transform: schedule.

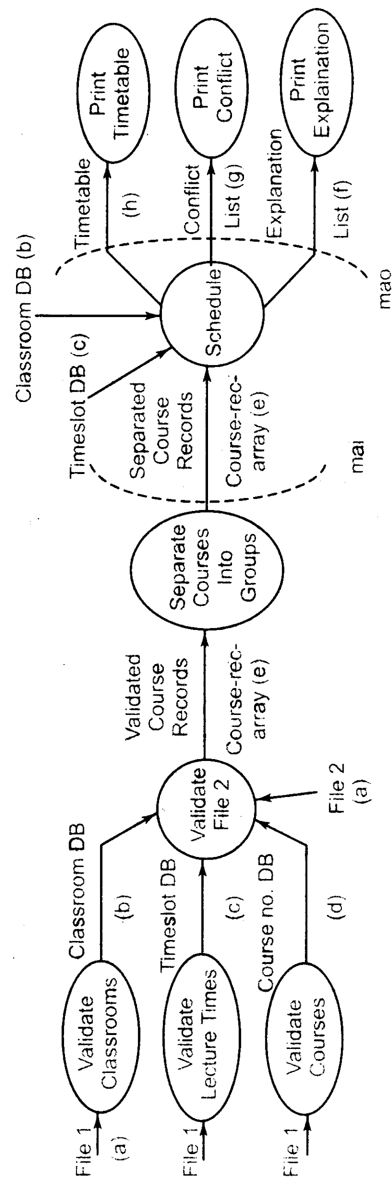


Figure 6.12: Data flow diagram for the case study.

First-Level Factoring: The first-level structure chart can easily be obtained and is shown in Figure 6.13. In the structure chart, instead of having one output module for each of the three outputs, as is shown in the data flow diagram, we have only one output module, which then invokes three output modules for the different outputs.

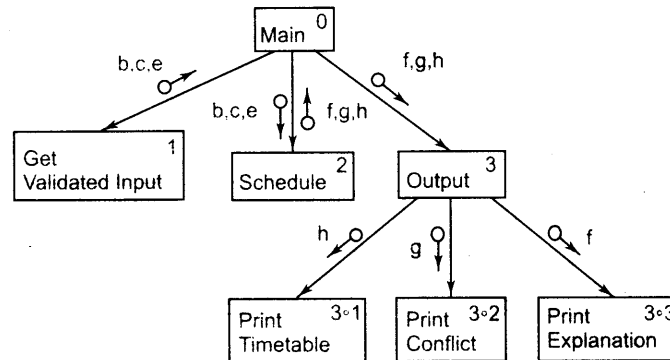


Figure 6.13: First level factoring.

Factoring the Input and Output Modules: The output module does not need any factoring. According to the design methodology, the input module get_validated_input will have one input module to get the array of validated course records and one transform module to separate into course groups. This input module can then be further factored into three input modules to get different validated inputs from file1, one input module to get data from file2, and one module for validating the file2 data. Because the data from file1 is also needed for the central transform, we modify the structure of the input branch. The structure chart for the input branch is shown in Figure 6.14.

Factoring the Central Transform: Now the central transform has to be factored. According to the requirements, PG courses have to be given preference over UG courses, and the highest priority of each course must be satisfied. This means that the courses with no priority should be scheduled after the courses with priority. Hence, we have four major subordinate modules to the central transform: schedule PG courses with preferences, schedule UG courses with preferences, schedule PG courses with no preferences, and schedule UG courses with no preferences. The structure of the central transform is shown in Figure 6.15.

These can then be combined into a structure chart for the system. The overall structure chart is shown in Figure 6.16. This structure chart gives an overall view of the strategy for structuring the programs. Further details about each module will evolve during detailed design and coding.

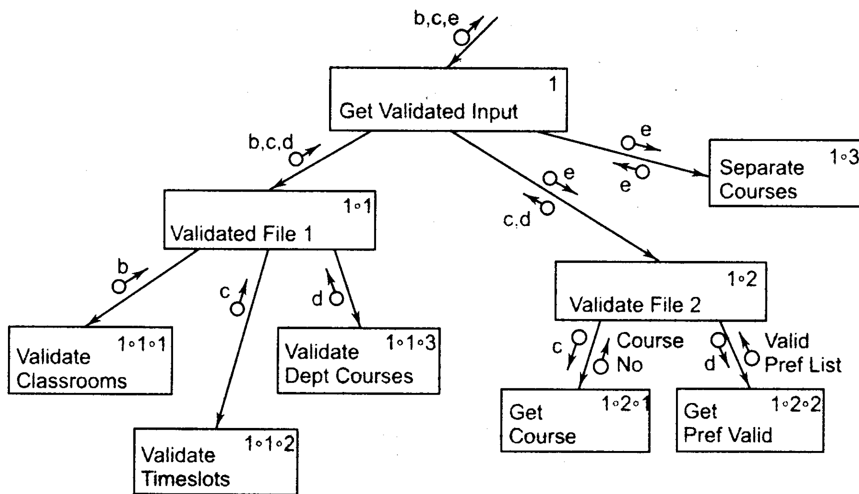


Figure 6.14: Factoring of the input branch.

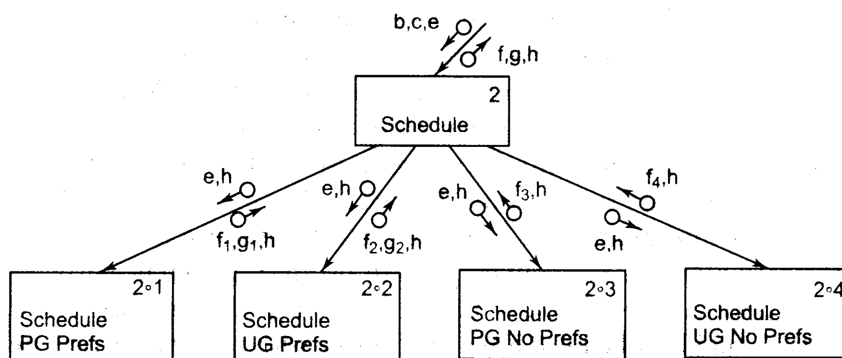


Figure 6.15: Factoring the central transform.

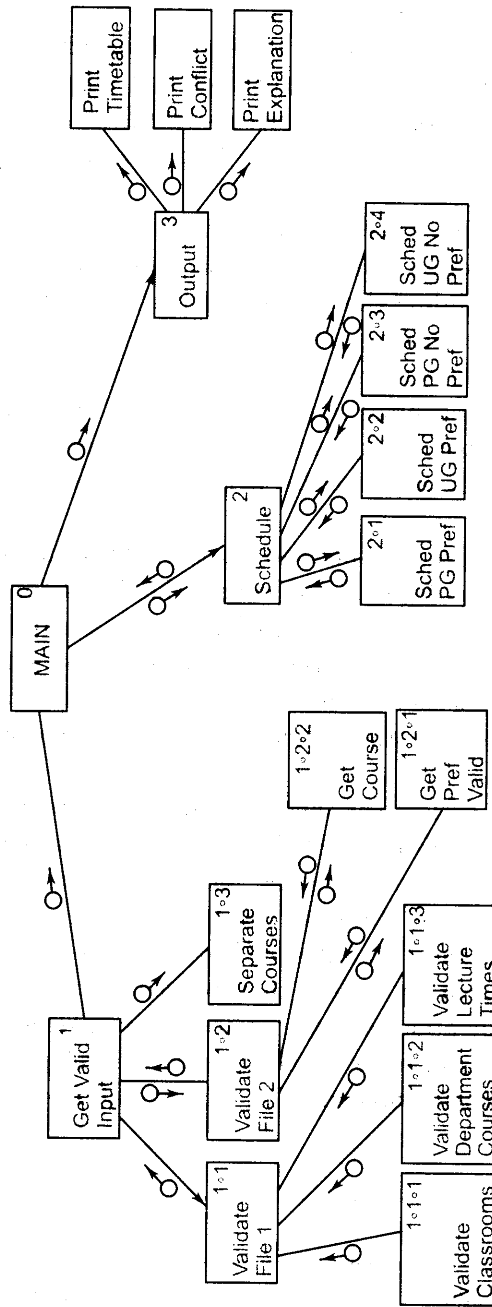


Figure 6.16: Structure chart for the system.

Analysis Using Information Flow Metrics

Based on the structure chart, the design of the system was first specified completely: this required formally specifying the data structures and all the modules. For each module, we specified the purpose of the module, its interface, the modules it invokes, and the estimated size of the module (in LOC). This formed the first version of the design document.

The first thing that could be noted was that when specifying a complete design from the structure chart, the design usually expands. For example, we found that for supporting the module for scheduling the UG courses with preferences (SchedUgPrefs) a lot more needs to be done. The reason is as follows. The UG courses with preferences are scheduled before PG courses with no preferences. However, PG courses are to be given preferences and no two PG courses can be scheduled in the same time slot. Hence, a UG course should not be allotted a slot that makes a PG course "unschedulable." This requires that "safety" of a room and time for a UG course should be checked before allocation.

For this, another data structure was specified. Essentially, a three-dimensional linked list was defined, which contained for each PG course the list of time slots for which it could be allotted, and for each time slot a list of all rooms where it could be allotted was maintained. This structure can be used for checking the safety—an allocation should not make a PG course unschedulable. In addition to this, a lot of utility routines needed to be defined to support the other functions, e.g., sort_rooms(), get_index(), and chk_fmt_course_no().

The complete first version design was then analyzed using information flow metrics described earlier in the chapter. We followed the approach of comparing modules of the design among themselves and then highlight the "error-prone" and "complex modules" (as described earlier). In the case study we used the metric where complexity of a module is defined as $D_c = fan_in * fan_out + inflow * outflow$. The definition of error-prone and complex is as given earlier, except that we also use size for classification; the size of the module must also be above average or above (average + standard deviation) for it to be classified as complex or error prone. A locally developed tool called dmetric was used to extract the information flow metrics. The overall metrics and results of the analysis are given here.

```
-----------------------------------------------------------------
OVERALL METRICS
-----------------------------------------------------------------
#modules: 35  Total size: 1330      Avg. size: 38   Std.Deviation: 27
Total complexity: 595         Avg. complexity: 17   Std.Deviation: 33
```

```
Deviation of the structure chart from a tree  = 0
(without considering leaves)

-----------------------------------------------------------------
 ERROR-PRONE MODULES
-----------------------------------------------------------------
8) sched_ug_pref
    call_in: 1   call_out: 4   inflow: 5   outflow:13   size:100
    design complexity:   69

-----------------------------------------------------------------
 COMPLEX  MODULES
-----------------------------------------------------------------
5) validate_file2
    call_in: 1   call_out: 4   inflow: 4   outflow: 8   size:100
    design complexity:   36

7) sched_pg_pref
    call_in: 1   call_out: 1   inflow: 1   outflow: 6   size:75
    design complexity:    7

13) is_safe_allotment
    call_in: 1   call_out: 0   inflow: 3   outflow: 1   size:80
    design complexity:    3

15) validate_classrooms
    call_in: 1   call_out: 5   inflow: 3   outflow: 7   size:80
    design complexity:   26

16) validate_dept_courses
    call_in: 1   call_out: 3   inflow: 2   outflow: 5   size:75
    design complexity:   13

17) validate_lec_times
    call_in: 1   call_out: 3   inflow: 2   outflow: 5   size:70
    design complexity:   13
-----------------------------------------------------------------
```

This data flow analysis clearly points out that the module to schedule UG courses with preferences is the most complex, with a complexity considerably higher than the average. It also shows that the overall structure is a tree (with a 0 deviation). Hence, we considered the structure to be alright. Based on this analysis, parts of the design dealing with scheduling of UG courses was re-examined in an effort to reduce complexity.

During analysis we observed that much of the complexity was due to the 3-D linked data structure being used for determining safety. Through discussions, we then developed a different approach for determining safety. The idea was that instead of using a separate data structure, before allocating a UG course, we would "simulate" the scheduling of the PgNoPref courses, using the regular function for scheduling these courses. If the number of courses the function sched_pg_no_prefs() returns is the same before and after the planned UG course scheduling, then the current allocation is safe. For this approach, we just have to make sure that is_safe_allotment() invokes sched_pg_no_prefs() with temporary data structures such that the actual timetable is not affected during this "simulation." The design was then modified to incorporate this approach. On analyzing the complexity again, we found that this approach reduced the complexity of the sched_ug_pref() module significantly and the complexity of this module was now similar to complexity of other modules. Overall, we considered the modified design satisfactory.

This demonstrates how highlighting of "hot spots" can be used to focus the attention of the designer or analysts and to improve the quality of the design. Note that this is done before the coding has started, which makes it very efficient from the point of view of cost. For example, if the same decision of changing the method of determining safety was taken after the code was developed, it would require that some parts of the old code be discarded, new code developed, and the design document changed to reflect the new design. All this will require considerably more effort than what was spent to change the design. Metrics-based analysis can also be used for monitoring by the project management; a quick look at the results of complexity and structure analysis will reveal if the structure and complexity are "acceptable" or if the design needs improvement.

The specification of the final design is available from the book's Web site.

Chapter 7

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 a 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 OO design of the case studies. Before we proceed, let us understand the relationship between OO analysis and OO 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 [118].

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

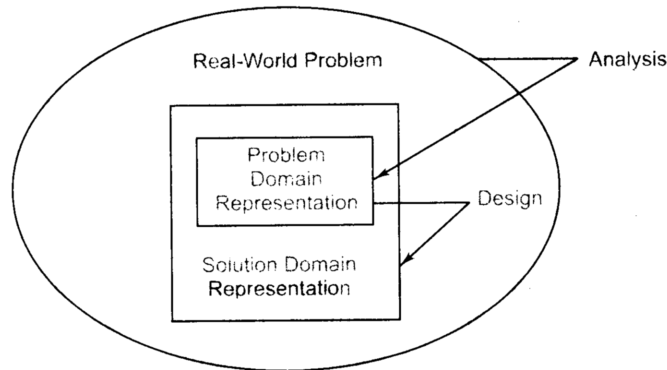


Figure 7.1: Relationship between OOA and OOD.

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 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 consequence, implementation issues drive the modeling process during design. While in analysis, comprehension and representation issues drive the process. This also results in OOA sometimes using

primitives that are somewhat 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 analysis 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.

7.2.1 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 op-

erations 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 s1, s2, ... of class type `Stack`. Each of these variables s1, s2, ... 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 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

```
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.

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 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 bi-directional, 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:

```
class Disk {
     private:
          Track *tracks;
```

```
                disk information
                     :
        };

        class Track {
            private:
                Sector sectors[MAX];
                     :
        };

        Class Sector {
            private:
                     :
        }
```

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.

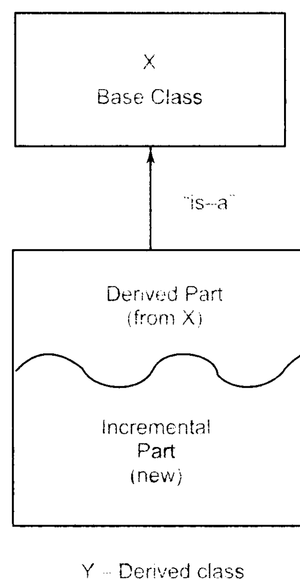


Figure 7.3: Inheritance.

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.

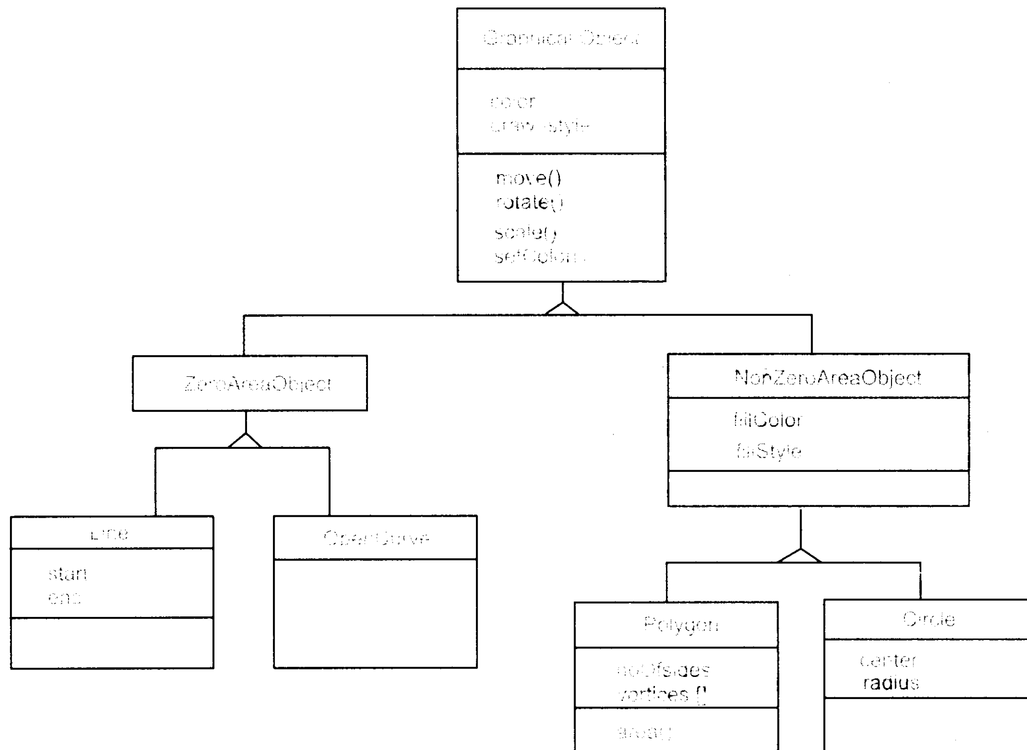


Figure 7.4: An inheritance example.

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:

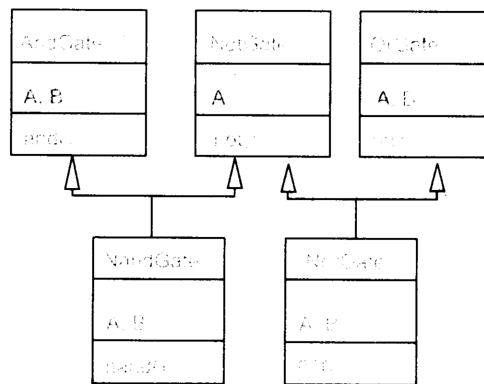
```
class GraphicalObject {
    protected:
        unsigned int color;
        unsigned int draw_style;
    public:
        virtual void move( Point &newLocation );
        virtual void rotate(double angle );
        virtual void scale( double XScale , double YScale);
        void setColor( unsigned 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 {
    private:
        Point start, end;
    public:
        int length();
        Point &midPoint();
        // Inherited virtual features are given definition here
        void move(Point &newLocation );
        void rotate( double angle );
        void scale( double XScale, double YScale);
```

```
};

class OpenCurve: public ZeroAreaObject {
    private:
        Point *controlPoints;
    public:
        // Inherited virtual features are given definition here
};

class Polygon: public NonZeroAreaObject {
    private:
        Point *vertices;
        unsigned int noOfSides;
    public:
        double area();
        // Inherited virtual features are defined here
};

class Circle: public NonZeroAreaObject {
    private:
        Point centre;
        unsigned int radius;
    public:
        double area();
        // Inherited virtual features are defined here
};
```

Inheritance can be broadly classified as being of two types: strict inheritance and non-strict 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 subclass 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

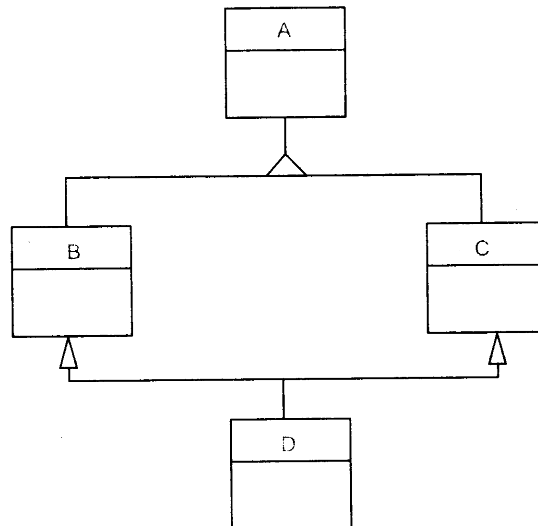


Figure 7.6: Repeated inheritance.

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 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 *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 *dynamic binding* of operations, which brings in *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 `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].rotate()`. For each A[i], the appropriate rotate function will be executed. That is, which function A[i].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.

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 cohe-

sion 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.

Figure 7.7: Example without using subtyping.

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 abstraction 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.

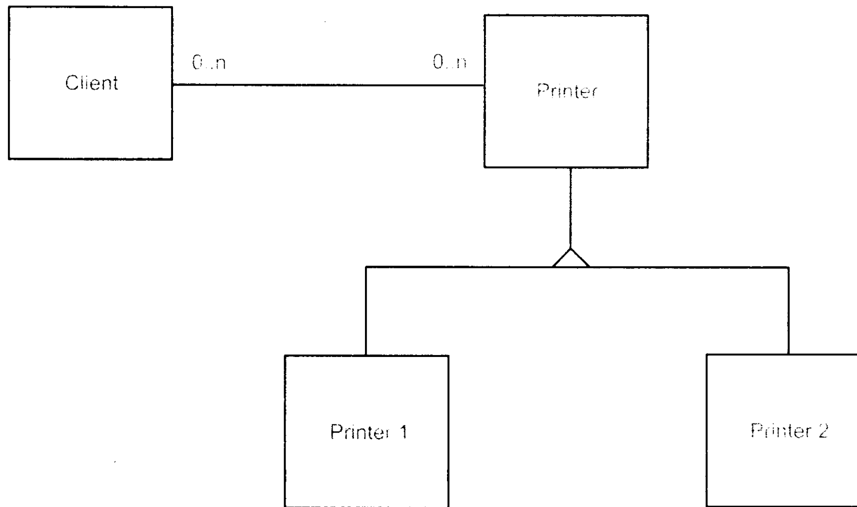


Figure 7.8: Example using subtyping.

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.

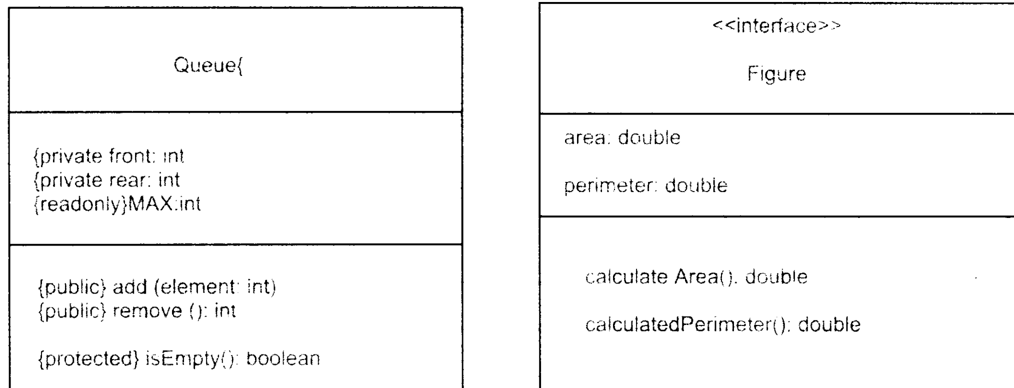


Figure 7.9: Class, stereotypes, and tagged values.

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`,

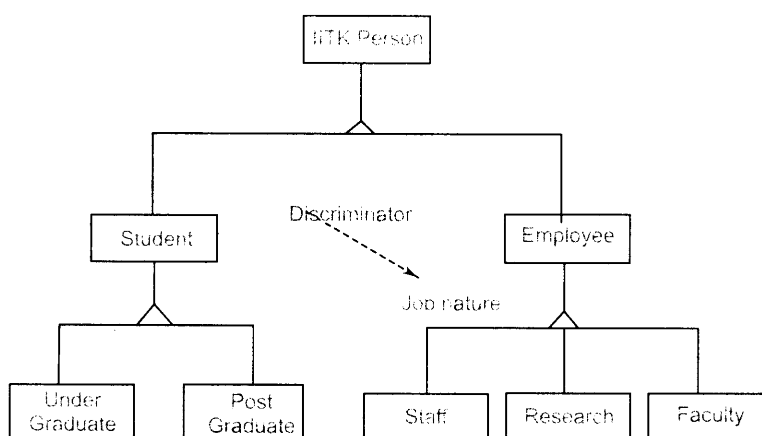


Figure 7.10: A class hierarchy.

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.

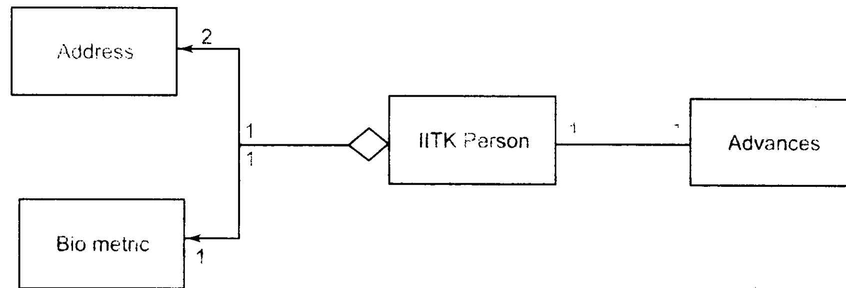


Figure 7.11. Aggregation and association among classes.

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, incidently, 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.

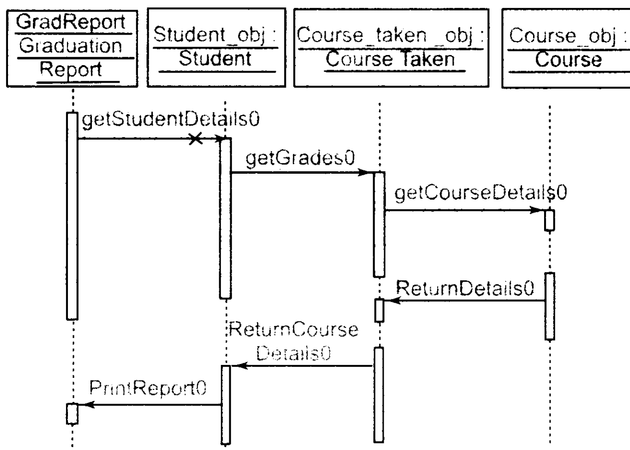


Figure 7.12: Sequence diagram for printing a graduation report.

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

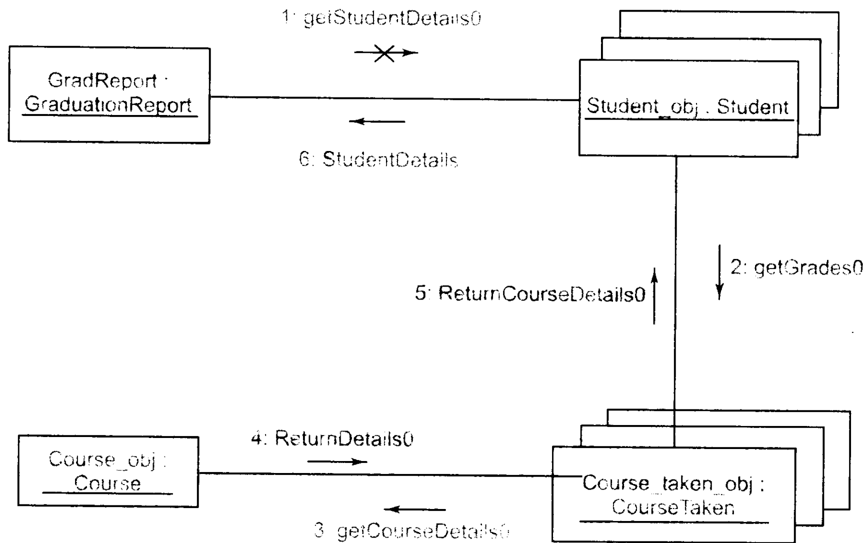


Figure 7.13: Collaboration diagram for printing a graduation report.

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.

7.1.3 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.

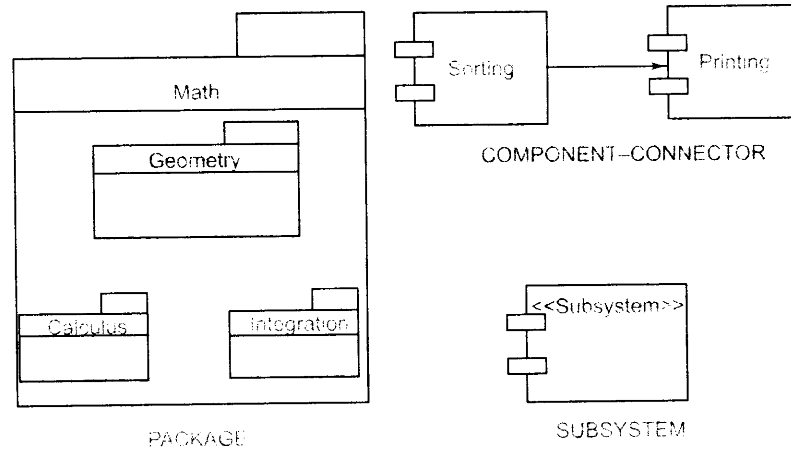


Figure 7.14. Subsystems, Components, and packages.

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.

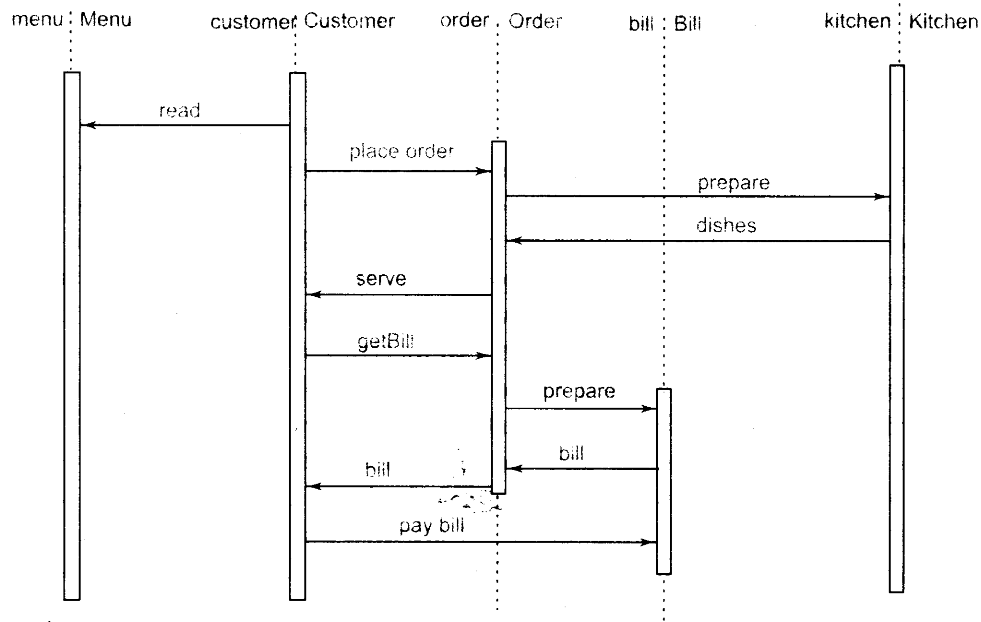


Figure 7.15: A sequence diagram for the restaurant.

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.