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

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

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

### 7.5.4 Optimize and Package

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

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

**Saving Derived Attributes.** A derived attribute is one whose value can be determined from the values of other attributes. As such an attribute is not independent, it may not have been specified in the initial design. However, if it is needed very often or if its computation is complex, its value can be computed and stored once and then

accessed later. This may require new objects to be created for the derived attributes. However, it should be kept in mind that by doing this the consistency between derived attributes and base attributes will have to be maintained and any changes to the base attributes may have to be reflected in the derived attributes.

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

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

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

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

### 7.5.5 Examples

Before we apply the methodology on some examples, it should be remembered again that no design methodology reduces the activity of producing a design to a series of steps that can be mechanically executed; each step requires some amount of engineering judgment. Furthermore, the design produced by following a methodology should not be considered the final design. The design can and should be modified using the design principles and the ultimate objectives of the project in mind. Methodologies are essentially guidelines to help the designer in the design activity; they are not hard-and-fast rules. The examples we give here are relatively small, and all aspects of the methodol-

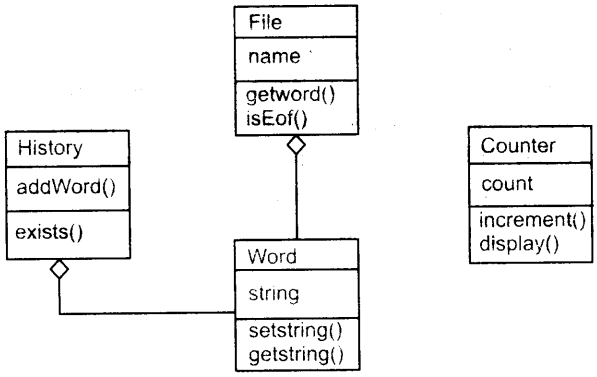


Figure 7.16: Class diagram for the word counting problem.

ogy do not get reflected in them. However, the design of the case studies, given at the end of the chapter, will provide a more substantial example for design.

### The Word-Counting Problem

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

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

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

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

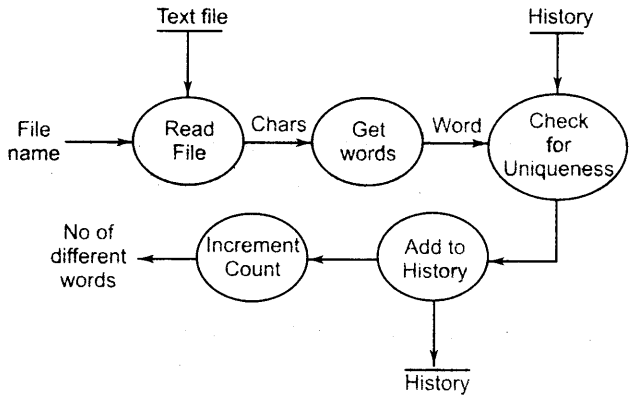


Figure 7.17: Functional Model for the word counting problem.

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

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

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

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

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

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

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

### Rate of Returns Problem

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

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

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

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

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

Now we have to perform the last two steps of the design methodology, where implementation and optimization concerns are used to enhance the classes. While considering the implementation of computation of total initial investment, computation of overall return rate, overall withdrawals and so on, we notice that for all of these, appropriate data from each investment is needed. Hence, to the class `Investments`, appropriate operations need to be added. Further, we note that all the computations for total initial

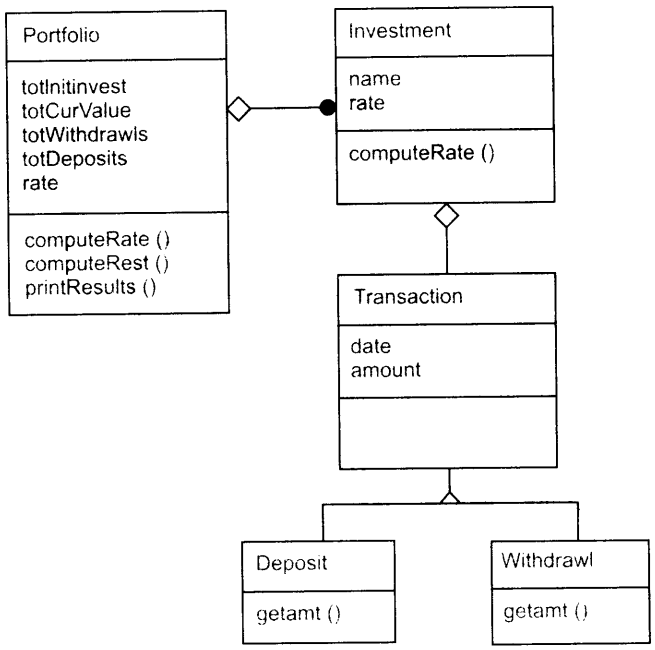


Figure 7.18: Class diagram for rate of return problem.

investment, total current value, and so on are all done together, and each of these is essentially adding values from various investments. Hence, we combine them in a single operation in `Portfolio` and a corresponding single operation in **`Investment`**. Studying the class hierarchy, we observe that the only difference in the two subclasses `Withdrawal` and `Deposit` is that in one case the amount is subtracted and in the other it is added. In such a situation, the two types can be easily considered a single type by keeping the amount as negative for a withdrawal and positive for a deposit. So we remove the subclasses, thereby simplifying the design and implementation. Instead of giving the class diagram for the final design, we provide the specification of the classes:

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

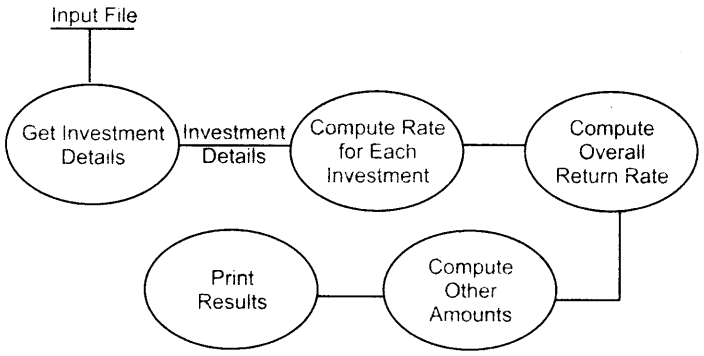


Figure 7.19: Functional model for the rate of return problem.

```
        Transaction(amount, month, year); // sets values
};


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

class Portfolio {
    private:
        Investment *investArray; // List of investments
        int noOfInvestments; // Total number of investments
        int totalInitInvest;
        int totalDeposits;
        int totalCurVal;
        int totalWithdrawl;
        float RateOfReturns; // Overall rate of returns
    public:
        getInvestDetails( char * fname ); // Parse the input file
        computeRate(); // Compute rates of return
```

```
        compute(); // Compute other totals
        printResults(); // Print return rates, total values, etc.
};
```

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

## 7.6 Metrics

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

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

### Weighted Methods per Class (WMC)

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

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

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

If the complexity of each method is considered 1, WMC gives the total number of methods in the class.

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

### Depth of Inheritance Tree (DIT)

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

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

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

### Number of Children (NOC)

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

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

### Coupling Between Classes (CBC)

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

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

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

### Response for a Class (RFC)

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

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

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

### Lack of Cohesion in Methods (LCOM)

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

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

$$LCOM = |P| - |Q|, \text{if } |P| > |Q| \ 0 \ \text{otherwise.}$$

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

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

## 7.7 Summary

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

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

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

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

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

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

To ensure that the design is modular, some general properties should be satisfied. The three properties we have discussed are cohesion, coupling, and open-closed principle. Coupling is an inter-class concept and captures how closely the different classes interact with each other and how much they depend on each other. Cohesion is an intra-class

concept and captures how strongly the elements of a class are related. Open-closed principle states that the classes should be designed in a manner that they are closed for modification but are open for extension. A good design should have low coupling, high cohesion, and should satisfy the open-closed principle—these make the design more modular and easier to change.

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

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

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

## Exercises

1. What is the relationship between abstract data types and classes?
2. Why are private parts of a superclass generally not made accessible to subclasses?
3. In C++, *friends* of a class $C$ can access the private parts of $C$. For declaring a class $F$ a friend of $C$, where should it be declared—in $C$ or in $F$? Why?

4. What are the different ways in which an object can access another object in a language like C++? (Do not consider the access allowed by being a friend.)

5. What are the potential problems that can arise in software maintenance due to different types of inheritance?

6. What is the relationship between OOA, SRS, and OOD?

7. In the word-counting example, a different functional model was used from the one proposed in Chapter 6. Use the model given in Chapter 6 and modify the OO design.

8. Suppose a simulator for a disk is to be written (for teaching an Operating Systems course). Use OMT to design the simulator.

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

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

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

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

## Case Studies

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

### Case Study 1—Course Scheduling

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

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

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

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

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

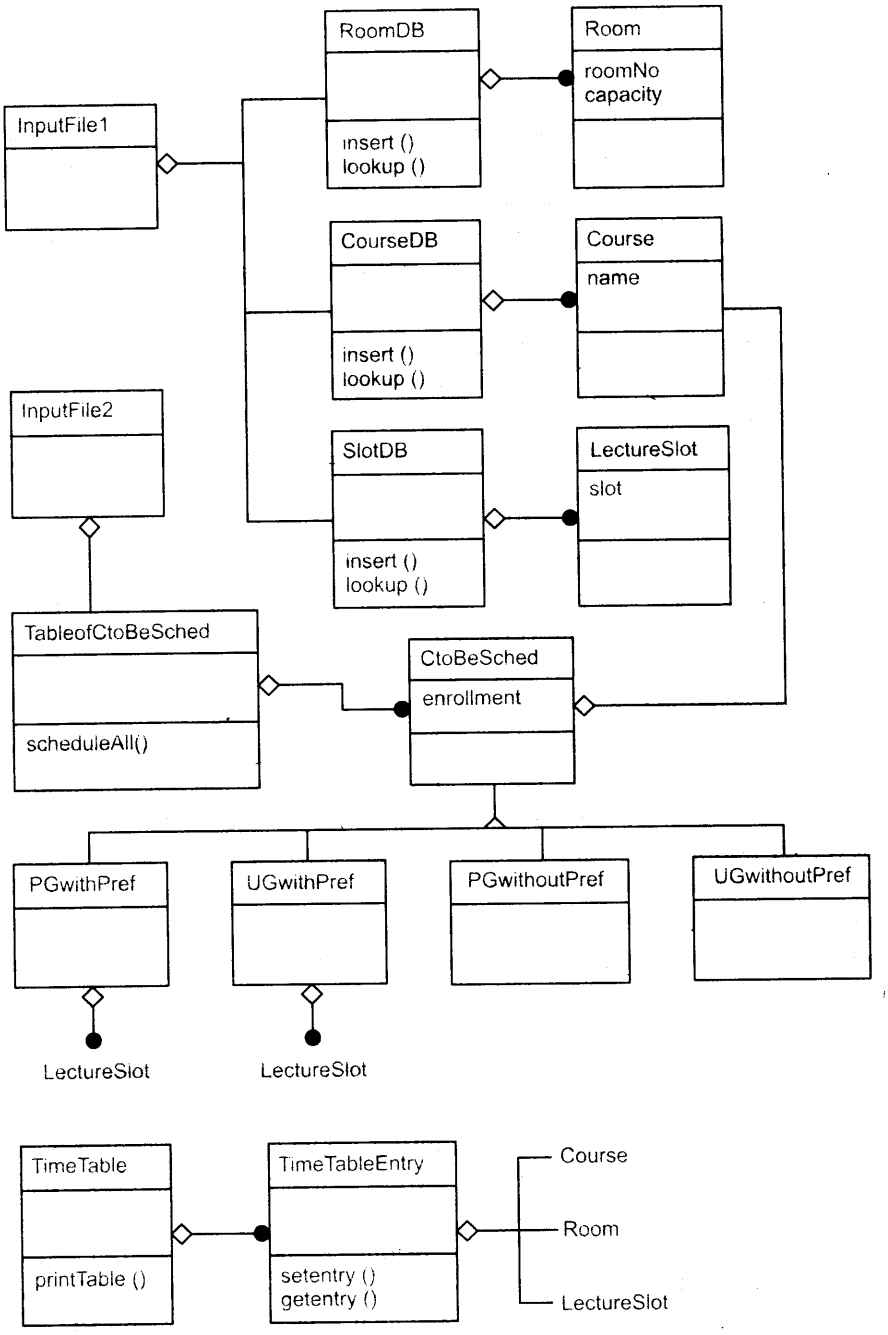


Figure 7.20: Initial class diagram for the case study.

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

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

Now we come to the last steps of considering implementation concerns. Many new issues come up here. First, we decided to have a generic template class, which can be used to implement the various DBs, as all DBs are performing similar functions. Hence, we defined a template class `List`. When considering the main issue of scheduling, we notice that scheduling UG courses with preferences, as discussed in the Chapter 6, is not straightforward, as the system has to ensure that it does not make any PG course without preference "unschedulable." To handle this, we take a simple approach of having a data structure that will reserve slots for PG courses and will then be used to check for

the safety of an assignment while scheduling PG courses with preferences. This adds an internal class `PGReserve`, with operations like *isAllotmentSafe()* (to check if making an allotment for UG course is "safe"), *Initialize()* (to initially "mark" all possible slots where `PGwithoutPref` courses can be scheduled). The structure is then used to schedule the PG courses without preferences after the UG courses with preferences are scheduled, leading to the operation *getSuitableSchedule()*.

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

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

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

### Case Study 2—PIMS

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

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

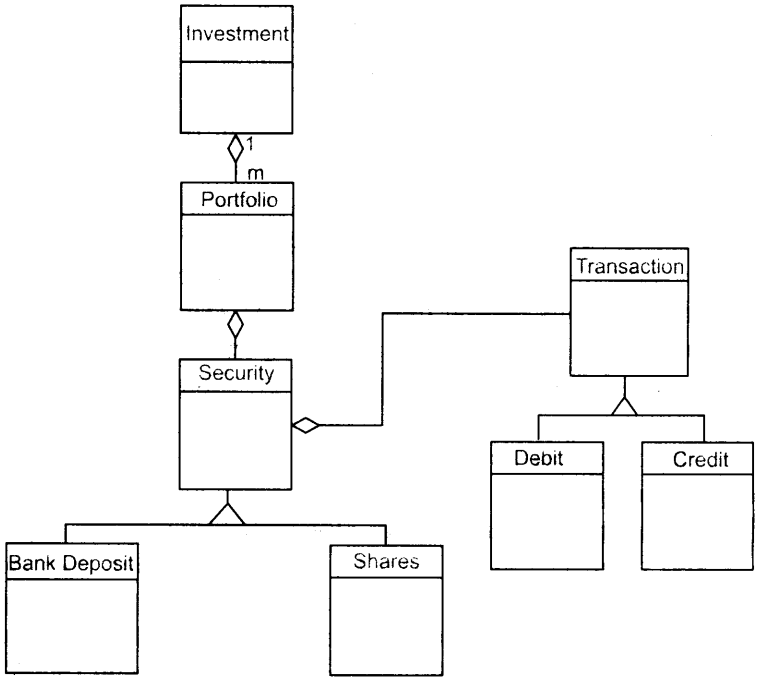


Figure 7.21: Class diagram for PIMS

The relationship between them is relatively straightforward. The class diagram containing some of the classes is shown in Figure 7.21. Though this initial class structure

was evolved during modeling, later the subtypes of transaction were eliminated as they provided little useful value. Subtypes of security type were also eliminated.

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

# Chapter 8

# Detailed Design

In previous chapters we discussed two different approaches for system design. In system design we concentrate on the modules in a system and how they interact with each other. Once a module is precisely specified, the internal logic that will implement the given specifications can be decided, and is the focus of this chapter. In this chapter we discuss methods for developing and specifying the detailed design of a module. We also discuss the metrics that can be extracted from a detailed design.

## 8.1 Detailed Design and PDL

Most design techniques, like structured design, identify the major modules and the major data flow among them. The methods used to specify the system design typically focus on the external interfaces of the modules and cannot be extended to specify the internals. Process design language (PDL) is one way in which the design can be communicated precisely and completely to whatever degree of detail desired by the designer. That is, it can be used to specify the system design and to extend it to include the logic design. PDL is particularly useful when using top-down refinement techniques to design a system or module.

### 8.1.1 PDL

One way to communicate a design is to specify it in a natural language, like English. This approach often leads to misunderstanding, and such imprecise communication is not particularly useful when converting the design into code. The other extreme is to communicate it precisely in a formal language, like a programming language. Such representations often have great detail, which is necessary for implementation but not important for communicating the design. These details are often a hindrance to easy communication of the basic design. Ideally we would like to express the design in

```
minmax(infile)

ARRAY a

        DO UNTIL end of input
              READ an item into a
        ENDDO
        max, min := first item of a
        DO FOR each item in a
              IF max < item THEN set max to item
              IF min > item THEN set min to item
        ENDDO
END
```

Figure 8.1: PDL description of the minmax program.

a language that is as precise and unambiguous as possible without having too much detail and that can be easily converted into an implementation. This is what PDL attempts to do.

PDL has an overall outer syntax of a structured programming language and has a vocabulary of a natural language (English in our case). It can be thought of as "structured English." Because the structure of a design expressed in PDL is formal, using the formal language constructs, some amount of automated processing can be done on such designs. As an example, consider the problem of finding the minimum and maximum of a set of numbers in a file and outputting these numbers in PDL as shown in Figure 8.1.

Notice that in the PDL program we have the entire logic of the procedure, but little about the details of implementation in a particular language. To implement this in a language, each of the PDL statements will have to be converted into programming language statements. Let us consider another example. Text is given in a file with one blank between two words. It is to be formatted into lines of 80 characters, except the last line. A word is not to be divided into two lines, and the numbers of blanks needed to fill the line are added at the end, with no more than two blanks between words. The PDL program is shown in Figure 8.2. Notice the use of procedure to express the design.

With PDL, a design can be expressed in whatever level of detail that is suitable for the problem. One way to use PDL is to first generate a rough outline of the entire solution at a given level of detail. When the design is agreed on at this level, more detail can be added. This allows a successive refinement approach, and can save considerable cost by detecting the design errors early during the design phase. It also aids design

```
Initialize buf to empty
DO FOREVER
       DO UNTIL (#chars in buf ≥ 80 & word boundary is reached)
                  OR (end-of-text reached)
             read chars in buf
       ENDDO
       IF #chars > 80 THEN
             remove last word from buf
             PRINT-WITH-FILL (buf)
             set buf to last word ELSEIF #chars = 80 THEN
             print (Buf)
             set buf to empty
       ELSE EXIT the loop
ENDDO

PROCEDURE PRINT-WITH-FILL (buf)

Determine #words and #character in buf
#of blanks needed = 80 - #character
DO FOR each word in the buf
       print (word)
       if #printed words ≥ (#word - #of blanks needed) THEN
             print (two blanks)
       ELSE print (single blank)
 ENDDO
```

Figure 8.2: PDL description of text-formatter.

verification by phases, which helps in developing error-free designs. The structured outer syntax of PDL also encourages the use of structured language constructs while implementing the design.

The basic constructs of PDL are similar to those of a structured language. The first is the IF construct. It is similar to the if-then-else construct of Pascal. However, the conditions and the statements to be executed need not be stated in a formal language. For a general selection, there is a CASE statement. Some examples of CASE statements are:

CASE OF transaction type
CASE OF operator type

The DO construct is used to indicate repetition. The construct is indicated by:

DO iteration criteria

```
        one or more statements
    ENDDO
```

The iteration criteria can be chosen to suit the problem, and unlike a formal programming language, they need not be formally stated. Examples of valid uses are:

```
DO WHILE there are characters in input file
DO UNTIL the end of file is reached
DO FOR each item in the list EXCEPT when item is zero
```

A variety of data structures can be defined and used in PDL such as lists, tables, scalar, and integers. Variations of PDL, along with some automated support, are used extensively for communicating designs.

### 8.1.2 Logic/Algorithm Design

The basic goal in detailed design is to specify the logic for the different modules that have been specified during system design. Specifying the logic will require developing an algorithm that will implement the given specifications. Here we consider some principles for designing algorithms or logic that will implement the given specifications.

The term *algorithm* is quite general and is applicable to a wide variety of areas. Essentially, an algorithm is a sequence of steps that need to be performed to solve a given problem. The problem need not be a programming problem. We can, for example, design algorithms for such activities as cooking dishes (the recipes are nothing but algorithms) and building a table. In the software development life cycle we are only interested in algorithms related to software. For this, we define an algorithm to be an unambiguous procedure for solving a problem [74]. A *procedure* is a finite sequence of well-defined steps or operations, each of which requires a finite amount of memory and time to complete. In this definition we assume that termination is an essential property of procedures. From now on we will use procedures, algorithms, and logic interchangeably.

There are a number of steps that one has to perform while developing an algorithm [74]. The starting step in the design of algorithms is *statement of the problem.* The problem for which an algorithm is being devised has to be precisely and clearly stated and properly understood by the person responsible for designing the algorithm. For detailed design, the problem statement comes from the system design. That is, the problem statement is already available when the detailed design of a module commences. The next step is development of a mathematical *model* for the problem. In modeling, one has to select the mathematical structures that are best suited for the problem. It

can help to look at other similar problems that have been solved. In most cases, models are constructed by taking models of similar problems and modifying the model to suit the current problem. The next step is the *design of the algorithm.* During this step the data structure and program structure are decided. Once the algorithm is designed, its correctness should be verified.

No clear procedure can be given for designing algorithms. Having such a procedure amounts to automating the problem of algorithm development, which is not possible with the current methods. However, some heuristics or methods can be provided to help the designer design algorithms for modules. The most common method for designing algorithms or the logic for a module is to use the *stepwise refinement technique* [148].

The stepwise refinement technique breaks the logic design problem into a series of steps, so that the development can be done gradually. The process starts by converting the specifications of the module into an abstract description of an algorithm containing a few abstract statements. In each step, one or several statements in the algorithm developed so far are decomposed into more detailed instructions. The successive refinement terminates when all instructions are sufficiently precise that they can easily be converted into programming language statements. During refinement, both data and instructions have to be refined. A guideline for refinement is that in each step the amount of decomposition should be such that it can be easily handled and that represents one or two design decisions.

The stepwise refinement technique is a top-down method for developing detailed design. We have already seen top-down methods for developing system designs. To perform stepwise refinement, a language is needed to express the logic of a module at different levels of detail, starting from the specifications of the module. We need a language that has enough flexibility to accommodate different levels of precision. Programming languages typically are not suitable as they do not have this flexibility. For this purpose, PDL is very suitable. Its formal outer syntax ensures that the design being developed is a "computer algorithm" whose statements can later be converted into statements of a programming language. Its flexible natural language-based inner syntax allows statements to be expressed with varying degrees of precision and aids the refinement process.

**An Example:** Let us again consider the problem of counting different words in a text file. Suppose that in the high-level structure chart of a large text processing system, a COUNT module is specified whose job is to determine the count of different words. During detailed design we have to determine the logic of this module so that the specifications are met. We will use the stepwise refinement method for this. For specification we will use PDL, adapted to C-style syntax. A simple strategy for the first step is shown in Figure 8.3.

This strategy is simple and easy to understand. This is the strategy that we proposed in the data flow graph earlier. The "primitive" operations used in this strategy are very high-level and need to be further refined. Specifically, there are three operations that

```
int count (file)
FILE file;
word_list wl;
{
     read file into wl
     sort (wl);
     count = different_words (wl);
     printf (count);
}
```

Figure 8.3: Strategy for the first step in stepwise refinement.

need refinement. These are (1) read file into the word list, whose purpose is to read all the words from the file and create a word list, (2) sort(wl), which sorts the word list in ascending order, and (3) count different words from a sorted word list. So far, only one data structure is defined: the word list. As refinement proceeds, more data structures might be needed.

In the next refinement step, we should select one of the three operations to be refined and further elaborate it. In this step we will refine the reading procedure. One strategy for implementing the read module is to read words and add them to the word list. This is shown in Figure 8.4.

```
read_from_file (file, wl)
FILE file;
word_list wl;
{
     initialize wl to empty;
     while not end-of-file {
     get_a_word from file
     add word to wl
}
```

Figure 8.4: Refinement of the reading operation.

This is a straightforward strategy, simple enough to be easily handled in one refinement step. Another strategy could be to read large amounts of data from the file in a buffer and form the word list from this buffer. This might lead to a more efficient im-

plementation. For the next refinement step we select the counting function. A strategy for implementing this function is shown in Figure 8.5.

```
int different_words (wl)
word_list wl;
{
      word last, cur;
      int cnt;

      last = first word in wl
      cnt = 1;
      while not end of list {
            cur = next word from wl
            if (cur <> last) {
                  cnt = cnt + 1;
                  last = cur;
            }
      }
      return (cnt)
}
```

Figure 8.5: Refinement of the function different_words.

Similarly, we can refine the sort function. Once these refinements are done, we have a design that is sufficiently detailed and needs no further refinement. For more complex problems many successive refinements might be needed for a single operation. Design for such problems can proceed in two ways—depth first or breadth first. In the depth first approach, when an operation is being refined, its refinement is completely finished (which might require many levels of refinement) before refinement of other operations begins. In the breadth first approach, all operations needing refinement are refined once. Then all the operations specified in this refinement are refined once. This is done until no refinement is needed. A combination of the two approaches could also be followed.

It is worth comparing the structure of the PDL programs produced by this method as compared to the structure produced using the structured design methodology. The two structures are not the same. The basic difference is that in stepwise refinement, the function sort is subordinate to the main module, while in the design produced by using structured design methodology, it is a subordinate module to the input module. This is not just a minor point; it points to a difference in approaches. In stepwise refinement, in each refinement step we specify the operations that are needed (as we do while drawing

the data flow diagram). In structured design, the focus is on partitioning the problem into input, output, and transform modules, which usually results in a different structure.

### 8.1.3 State Modeling of Classes

For object-oriented design, the approach just discussed for obtaining the detailed design may not be sufficient, as it focuses on specifying the logic or the algorithm for the modules identified in the (function-oriented) high-level design. But a class is not a functional abstraction and cannot be viewed as an algorithm. A method of a class can be viewed as a functional module, and the methods can be used to specify the logic for the methods.

The technique for getting a more detailed understanding of the class as a whole, without talking about the logic of different methods, has to be different from the refinement-based approach. An object of a class has some state and many operations on it. To better understand a class, the relationship between the state and various operations and the effect of interaction of various operations have to be understood. This can be viewed as one of the objectives of the detailed design activity for object-oriented development. Once the overall class is better understood, the algorithms for its various methods can be developed. Note that the axiomatic specification approach for a class, discussed earlier in this chapter, also takes this view. Instead of specifying the functionality of each operation, it specifies, through axioms, the interaction between different operations.

A method to understand the behavior of a class is to view it as a finite state automata (FSA). An FSA consists of states and transitions between states, which take place when some events occur. When modeling an object, the state is the value of its attributes, and an event is the performing of an operation on the object. A *state diagram* relates events and states by showing how the state changes when an event is performed. A state diagram for an object will generally have an initial state, from which all states in the FSA are reachable (i.e., there is a path from the initial state to all other states).

A state diagram for an object does not represent all the actual states of the object, as there are many possible states. A state diagram attempts to represent only the logical states of the object. A *logical state* of an object is a combination of all those states from which the behavior of the object is similar for all possible events. Two logical states will have different behavior for at least one event. For example, for an object that represents a stack, all states that represent a stack of size more than 0 and less than some defined maximum are similar as the behavior of all operations defined on the stack will be similar in all such states (e.g., push will add an element, pop will remove one, etc.). However, the state representing an empty stack is different as the behavior of top and pop operations are different now (an error message may be returned). Similarly, the state representing a full stack is different. The state model for this bounded size stack is shown in Figure 8.6.

The finite state modeling of objects is an aid to understand the effect of various operations defined on the class on the state of the object. A good understanding of

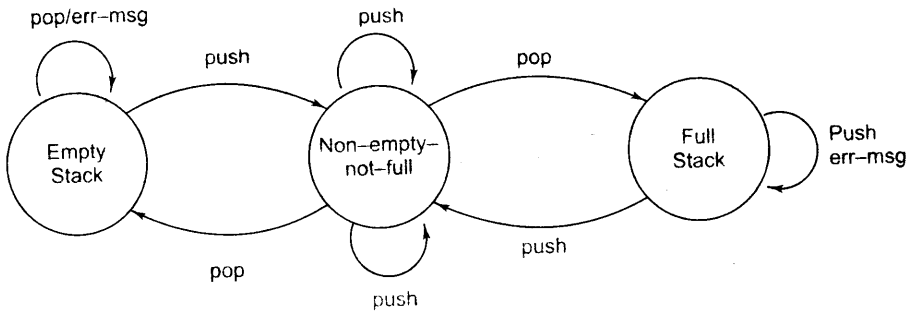


Figure 8.6: FSA model of a stack.

this can aid in developing the logic for each of the operations. To develop the logic of operations, regular approaches for algorithm development can be used. The model can also be used to validate if the logic for an operation is correct. As we have seen, for a class, typically the input-output specification of the operations is not provided. Hence, the FSA model can be used as a reference for validating the logic of the different methods. As we will see in Chapter 10, a state model can be used for generating test cases for validation.

State modeling of classes has also been proposed as a technique for analysis [133]. However, we believe that it has limited use during analysis, and its role is more appropriate during detailed design when the detailed working of a class needs to be understood. Even here, the scope of this modeling is limited. It is likely to be more of use if the interaction between the methods through the state is heavy and there are many states in which the methods need to behave differently.

## 8.2 Verification

There are a few techniques available to verify that the detailed design is consistent with the system design. The focus of verification in the detailed design phase is on showing that the detailed design meets the specifications laid down in the system design. Validating that the system as designed is consistent with the requirements of the system is not stressed during detailed design. The three verification methods we consider are design walkthroughs, critical design review, and consistency checkers.

### 8.2.1 Design Walkthroughs

A *design walkthrough* is a manual method of verification. The definition and use of walkthroughs change from organization to organization. Here we describe one walk-

through model. A design walkthrough is done in an informal meeting called by the designer or the leader of the designer's group. The walkthrough group is usually small and contains, along with the designer, the group leader and/or another designer of the group. The designer might just get together with a colleague for the walkthrough or the group leader might require the designer to have the walkthrough with him.

In a walkthrough the designer explains the logic step by step, and the members of the group ask questions, point out possible errors or seek clarification. A beneficial side effect of walkthroughs is that in the process of articulating and explaining the design in detail, the designer himself can uncover some of the errors.

Walkthroughs are essentially a form of peer review. Due to its informal nature, they are usually not as effective as the design review.

### 8.2.2 Critical Design Review

The purpose of *critical design review* is to ensure that the detailed design satisfies the specifications laid down during system design. The critical design review process is same as the inspections process in which a group of people get together to discuss the design with the aim of revealing design errors or undesirable properties. The review group includes, besides the author of the detailed design, a member of the system design team, the programmer responsible for ultimately coding the module(s) under review, and an independent software quality engineer. While doing design review it should be kept in mind that the aim is to uncover design errors, not try to fix them. Fixing is done later.

The use of checklists, as with other reviews, is considered important for the success of the review. The checklist is a means of focusing the discussion or the "search" of errors. Checklists can be used by each member during private study of the design and during the review meeting. For best results, the checklist should be tailored to the project at hand, to uncover project-specific errors. Here we list a few general items that can be used to construct a checklist for a design review [52].

**A Sample Checklist**

- Does each of the modules in the system design exist in detailed design?
- Are there analyses to demonstrate that the performance requirements can be met?
- Are all the assumptions explicitly stated, and are they acceptable?
- Are all relevant aspects of system design reflected in detailed design?
- Have the exceptional conditions been handled?
- Are all the data formats consistent with the system design?
- Is the design structured, and does it conform to local standards?

- Are the sizes of data structures estimated? Are provisions made to guard against overflow?
- Is each statement specified in natural language easily codable?
- Are the loop termination conditions properly specified?
- Are the conditions in the loops OK?
- Are the conditions in the if statements correct?
- Is the nesting proper?
- Is the module logic too complex?
- Are the modules highly cohesive?

### 8.2.3 Consistency Checkers

Design reviews and walkthroughs are manual processes; the people involved in the review and walkthrough determine the errors in the design. If the design is specified in PDL or some other formally defined design language, it is possible to detect some design defects by using consistency checkers.

*Consistency checkers* are essentially compilers that take as input the design specified in a design language (PDL in our case). Clearly, they cannot produce executable code because the inner syntax of PDL allows natural language and many activities are specified in the natural language. However, the module interface specifications (which belong to outer syntax) are specified formally. A consistency checker can ensure that any modules invoked or used by a given module actually exist in the design and that the interface used by the caller is consistent with the interface definition of the called module. It can also check if the used global data items are indeed defined globally in the design.

Depending on the precision and syntax of the design language, consistency checkers can produce other information as well. In addition, these tools can be used to compute the complexity of modules and other metrics, because these metrics are based on alternate and loop constructs, which have a formal syntax in PDL. The trade-off here is that the more formal the design language, the more checking can be done during design, but the cost is that the design language becomes less flexible and tends towards a programming language.

## 8.3 Metrics

After the detailed design the logic of the system and the data structures are largely specified. Only the implementation-oriented details, which are often specific to the

programming language used, need to be further defined. Hence, many of the metrics that are traditionally associated with code can be used effectively after detailed design. During detailed design all the metrics covered during the system design are applicable and useful. With the logic of modules available after detailed design, it is meaningful to talk about the complexity of a module. Traditionally, complexity metrics are applied to code, but they can easily be applied to detailed design as well. Here we describe some metrics applicable to detailed design.

### 8.3.1 Cyclomatic Complexity

Based on the capability of the human mind and the experience of people, it is generally recognized that conditions and control statements add complexity to a program. Given two programs with the same size, the program with the larger number of decision statements is likely to be more complex. The simplest measure of complexity, then, is the number of constructs that represent branches in the control flow of the program, like `if then else`, `while do`, `repeat until`, and `goto` statements.

A more refined measure is the *cyclomatic complexity measure* proposed by McCabe, which is a graph-theoretic–based concept. For a graph G with $n$ nodes, $e$ edges, and $p$ connected components, the cyclomatic number V(G) is defined as

$$V(G) = e - n + p.$$

To use this to define the cyclomatic complexity of a module, the control flow graph $G$ of the module is first drawn. To construct a control flow graph of a program module, break the module into blocks delimited by statements that affect the control flow, like `if`, `while`, `repeat`, and `goto`. These blocks form the nodes of the graph. If the control from a block $i$ can branch to a block $j$, then draw an arc from node $i$ to node $j$ in the graph. The control flow of a program can be constructed mechanically. As an example, consider the C-like function for bubble sorting, given next. The control flow graph for this is given in Figure 8.7.

```
0. {
1.      i = 1;
2.      while (i <= n) {
3.           j = i;
4.           while (j <= i) {
5.                if (A[i] < A[j])
6.                     swap(A[i], A[j]);
7.                j = j + 1; }
8.      i = i + 1; }
9. }
```

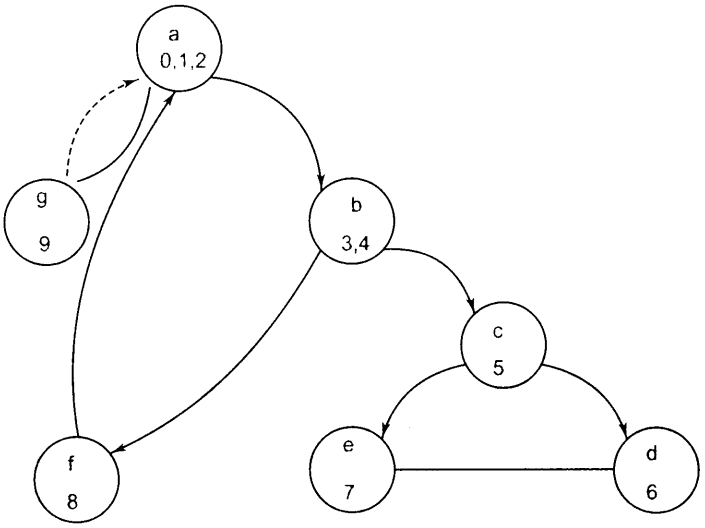


Figure 8.7: Flow graph of the example.

The graph of a module has an entry node and an exit node, corresponding to the first and last blocks of statements (or we can create artificial nodes for simplicity, as in the example). In such graphs there will be a path from the entry node to any node and a path from any node to the exit node (assuming the program has no anomalies like unreachable code). For such a graph, the cyclomatic number can be 0 if the code is a linear sequence of statements without any control statement. If we draw an arc from the exit node to the entry node, the graph will be strongly connected because there is a path between any two nodes. The cyclomatic number of a graph for any program will then be nonzero, and it is desirable to have a nonzero complexity for a simple program without any conditions (after all, there is some complexity in such a program). Hence, for computing the cyclomatic complexity of a program, an arc is added from the exit node to the start node, which makes it a strongly connected graph. For a module, the *cyclomatic complexity* is defined to be the cyclomatic number of such a graph for the module.

As it turns out the cyclomatic complexity of a module (or cyclomatic number of its graph) is equal to the maximum number of linearly independent circuits in the graph. A set of circuits is linearly independent if no circuit is totally contained in another circuit or is a combination of other circuits. So, for calculating the cyclomatic number of a module, we can draw the graph, make it connected by drawing an arc from the exit node to the entry node, and then either count the number of circuits or compute it by counting the number of edges and nodes. In the graph shown in Figure 8.7, the

cyclomatic complexity is

$$V(G) = 10 - 7 + 1 = 4.$$

The independent circuits are:

ckt 1: b c e b
ckt 2: b c d e b
ckt 3: a b f a
ckt 4: a g a

It can also be shown that the cyclomatic complexity of a module is the number of decisions in the module plus one, where a decision is effectively any conditional statement in the module [41]. Hence, we can also compute the cyclomatic complexity simply by counting the number of decisions in the module. For this example, as we can see, we get the same cyclomatic complexity for the module if we add 1 to the number of decisions in the module. (The module has three decisions: two in the two `while` statements and one in the `if` statement.)

The cyclomatic number is one quantitative measure of module complexity. It can be extended to compute the complexity of the whole program, though it is more suitable at the module level. McCabe proposed that the cyclomatic complexity of modules should, in general, be kept below 10. The cyclomatic number can also be used as a number of paths that should be tested during testing. Cyclomatic complexity is one of the most widely used complexity measures. Experiments indicate that the cyclomatic complexity is highly correlated to the size of the module in LOC (after all, the more lines of code the greater the number of decisions). It has also been found to be correlated to the number of faults found in modules.

### 8.3.2 Data Bindings

We have seen that coupling and cohesion are important concepts for evaluating a design. However, to be truly effective, metrics are needed to "measure" the coupling between modules or the cohesion of a module. During system design, we tried to quantify coupling based on information flow between modules. Now that the logic of modules is also available, we can come up with metrics that also consider the logic. One metric that attempts to capture the module-level concept of coupling is data binding. Data bindings are measures that capture the data interaction across portions of a software system [90]. In other words, data bindings try to specify how strongly coupled different modules in a software system are. Different types of data bindings are possible [90].

A *potential data binding* is defined as a triplet $(p, x, q)$, where $p$ and $q$ are modules and $x$ is a variable within the static scope of both $p$ and $q$. This reflects the possibility

that the modules $p$ and $q$ may communicate with each other through the shared variable $x$. This binding does not consider the internals of $p$ and $q$ to determine if the variable $x$ is actually accessed in any of the modules. This binding is based on data declaration.

A *used data binding* is a potential binding where both $p$ and $q$ use the variable $x$ for reference or assignment. This is harder to compute than potential data binding and requires more information about the internal logic of a module.

An *actual data binding* is a used data binding with the additional restriction that the module $p$ assigns a value to $x$ and $q$ references $x$. It is the hardest to compute, and it signifies the situation where information may flow from the module $p$ to module $q$ through the shared variable $x$. Computation of actual data binding requires detailed logic descriptions of modules $p$ and $q$.

All of these data bindings attempt to represent the strength of interconnections among modules. The greater the number of bindings between two modules, the higher the interconnection between these modules. For a particular type of binding, a matrix can be computed that contains the number of bindings between different modules. This matrix can be used for further statistical analysis to determine the interconnection strength of the system or a subsystem.

### 8.3.3 Cohesion Metric

Here we discuss one attempt at quantifying the cohesion of a module [54]. To compute the value of the cohesion metric for a module M, a flow graph G is constructed for M. Each vertex in G is an executable statement in M. For each node, we also record the variable referenced in the statement. An arc exists from a node $s_i$ to another node $s_j$ if the statement $s_j$ can immediately follow the statement $s_i$ in some execution of the module. In addition to these, we add an initial node I from where the execution of the module starts, and a final node T, at which the execution of the module terminates. For termination statements (e.g., return, exit) we draw an arc from the statement to T.

From G a *reduced* flow graph is constructed by deleting those nodes that do not refer to any variable (such as unconstrained gotos). All the arcs coming in the deleted node are redirected to the node that is the successor of the deleted node (such nodes will have only one successor).

Assume that the variables are sequentially numbered as 1, 2, ..., $n$. For a variable $i$, $R_i$ is the reference set, which is the set of all the executable statements that refer to the variable $i$. The union of all the $R_i$s is the set of all the nodes in the graph (minus the node for T, which is a nonexecuting node). Let $\mid G \mid$ refer to the (number of nodes - 1) for the reduced graph.

The cohesion of a set of statements S is defined as

$$C(S) = \frac{\mid S \mid dim(S)}{\mid G \mid dim(G)}$$

where $dim()$ is the dimension of a set of statements, which is the maximum number of linearly independent paths from I to T that pass through any element of S. Thus, the dimension of a set of statements S is the count of all the independent paths from the start statement to the end statement of a module that includes at least one statement from the set. If S is the set of all the statements in the module (if S is the same as G), then dim S is the same as the cyclomatic complexity of the module.

The cohesion of a module is defined as the average cohesion of the reference sets of the different statements or nodes in (reduced) G. Hence the cohesion of the module C(M) is

$$C(M) = \frac{\sum_{i=1}^{n} C(R_i)}{n}.$$

Essentially, this metric is trying to measure cohesion of a module by seeing how many independent paths of the module go through the different statements. The idea is that if a module has high cohesion, most of the variables will be used by statements in most paths. Hence for a high-cohesion module, the cohesion of the reference set of each variable will be high. The highest cohesion number achievable by this is when the dimension of all the reference sets is all the independent paths, thus the same as the cyclomatic complexity. In other words, the highest cohesion is when all the independent paths use all the variables of the module.

## 8.4 Summary

Detailed design starts after the module specifications are available, as part of the output of the system design. The goal of this activity is to develop the internal logic of the modules.

To express the internal logic of a module, we need a design language. The design language should be such that it is flexible enough to be easily usable, yet precise enough to be easily convertible into code. We have described a language, process design language (PDL), that satisfies the requirements. PDL can be used to express the detailed design of systems. It has a formal outer syntax and a flexible inner syntax and vocabulary, giving it a balance between formalism and ease of expression. Stepwise refinement and other algorithm development techniques can be used along with PDL to design as well as specify the logic.

For objects, state modeling can be used to understand the behavior of an object, as functional means are not sufficient. With state modeling, the state of an object is captured with methods causing transitions between states.

Like any phase, we need some metrics to evaluate the effectiveness of the phase and to evaluate the output of that phase. We considered a metric called cyclomatic complexity for evaluating the complexity of modules from their detailed design. This metric can be also used to assess the overall complexity of the system, or it can be used to identify the most complex modules, which are more likely to be "error-prone." In a module the cyclomatic complexity equals the number of decisions in the module plus one. We also discussed the data binding metric and a cohesion metric.

A few techniques exist for verifying the detailed design. The most common are design walkthroughs and critical design review. Automated tools can be used for some consistency checking if a well-defined design language, like PDL, is used. Even with automated consistency checkers, reviews and walkthroughs remain the most important methods for verifying the detailed design. We have described the review process and given a sample checklist that can be used in the review.

The detailed design activity is is frequently not performed formally because a detailed design description of the modules does not always adds much value, and experienced programmers feel that they can go directly to coding. Furthermore, the detailed design document has little archival value as it is almost impossible to keep the detailed design document consistent with the code. Hence the primary use of the detailed design phase is to help the programmer who can specify the logic and get it verified before writing the code. Due to this, developing the detailed design is of value mostly for the more complex and important modules. Even for these, the detailed design is often done informally by the programmer as part of the personal process of developing code. Due to these reasons, we will not give the detailed design of the case study.

## Exercises

1. The detailed design of a system can involve many persons, each developing the detailed design of a set of modules. Draw a process diagram for this method of detailed design development.
2. Extend the PDL with constructs to support classes. Then write the detailed design for classes String, Btree, SymbolTable.
3. Do a state modeling of these classes: String, Btree, and SymbolTable.
4. What features would you like to add to PDL if the target source language supports data abstraction?
5. If cyclomatic complexity of a module is much higher than the suggested limit of 10, what will you do? Give reasons and guidelines for whatever you propose.

6. Design an experiment to study the relationship between the cyclomatic complexity and size in LOC of modules. Collect a set of programs and then perform the experiment and determine the nature of the relationship between them for these programs.

7. Design an experiment to study the relation between cyclomatic complexity and "error-proneness" of modules. If you can collect error data, execute the experiment on the data you can collect.

# Chapter 9

# Coding

The goal of the coding or programming activity is to implement the design in the best possible manner. The coding activity affects both testing and maintenance profoundly. As we saw earlier, the time spent in coding is a small percentage of the total software cost, while testing and maintenance consume the major percentage. Thus, it should be clear that the goal during coding should *not* be to reduce the implementation cost, but the goal should be to reduce the cost of later phases, even if it means that the cost of this phase has to increase. In other words, the goal during this phase is *not* to simplify the job of the programmer. Rather, the goal should be to simplify the job of the tester and the maintainer.

This distinction is important, as programmers are often concerned about how to finish their job quickly, without keeping the later phases in mind. During coding, it should be kept in mind that the programs should not be constructed so that they are easy to write, but so that they are easy to read and understand. A program is read a lot more often and by a lot more people during the later phases.

There are many different criteria for judging a program, including readability, size of the program, execution time, and required memory. Having readability and understandability as a clear objective of the coding activity can itself help in producing software that is more maintainable. A famous experiment by Weinberg showed that if programmers are specified a clear objective for the program, they usually satisfy it [143]. In the experiment, five different teams were given the same problem for which they had to develop programs. However, each of the teams was specified a different objective, which it had to satisfy. The different objectives given were: minimize the effort required to complete the program, minimize the number of statements, minimize the memory required, maximize the program clarity, and maximize the output clarity. It was found that in most cases each team did the best for the objective that was specified to it. The rank of the different teams for the different objectives is shown in Figure 9.1.

The experiment clearly shows that if objectives are clear, programmers tend to achieve that objective. Hence, if readability is an objective of the coding activity,

| | Resulting Rank (1 = Best) | | | | |
|---|---|---|---|---|---|
| | O1 | O2 | O3 | O4 | O5 |
| Minimize effort to complete (O1) | 1 | 4 | 4 | 5 | 3 |
| Minimize number of statements (O2) | 2–3 | 1 | 2 | 3 | 5 |
| Minimize memory required (O3) | 5 | 2 | 1 | 4 | 4 |
| Maximize program clarity (O4) | 4 | 3 | 3 | 2 | 2 |
| Maximize output clarity (O5) | 2–3 | 5 | 5 | 1 | 1 |

Figure 9.1: The Weinberg experiment.

then it is likely that programmers will develop easily understandable programs. For our purposes, ease of understanding and modification are the basic goals of the programming activity.

In this chapter we will first discuss some programming practices and guidelines, in which we will also discuss some common coding errors to make students aware of them. Then we discuss some processes that are followed while coding. Refactoring is discussed next, which is done during coding but is a distinct activity. We then discuss some verification methods, followed by discussion of some metrics. We end the chapter with a discussion of the implementation of the case studies.

## 9.1 Programming Principles and Guidelines

The main task before a programmer is to write quality code with few bugs in it. The additional constraint is to write code quickly. Writing solid code is a skill that can only be acquired by practice. However, based on experience, some general rules and guidelines can be given for the programmer. Good programming (producing correct and simple programs) is a practice independent of the target programming language, although well-structured programming languages make the programmer's job simpler. In this section, we will discuss some concepts and practices that can help a programmer write higher quality code. As a key task of a programmer is to avoid errors in the programs, we first discuss some common coding errors.

### 9.1.1 Common Coding Errors

Software errors (we will use the terms errors, defects and bugs interchangeably in our discussion here; precise definitions are given in the next chapter) are a reality that all programmers have to deal with. Much of effort in developing software goes in identifying and removing bugs. There are various practices that can reduce the occurrence of bugs, but regardless of the tools or methods we use, bugs are going to occur in programs. Though errors can occur in a wide variety of ways, some types of errors are found more

commonly. Here we give a list of some of the commonly occurring bugs. The main purpose of discussing them is to educate programmers about these mistakes so that they can avoid them. The compilation is based on various published articles on the topic (e.g., [28, 87, 57, 55, 150]), and a more detailed compilation is available in the TR [141].

### Memory Leaks

A memory leak is a situation where the memory is allocated to the program which is not freed subsequently. This error is a common source of software failures which occurs frequently in the languages which do not have automatic garbage collection (like C, C++). They have little impact in short programs but can have catastrophic effect on long running systems. A software system with memory leaks keeps consuming memory, till at some point of time the program may come to an exceptional halt because of the lack of free memory. An example of this error is:

```
char* foo(int s)
{
        char *output;
        if (s>0)
          output=(char*) malloc (size);
        if (s==1)
          return NULL; /* if s==1 then mem leaked */
        return(output);
}
```

### Freeing an Already Freed Resource

In general, in programs, resources are first allocated and then freed. For example, memory is first allocated and then deallocated. This error occurs when the programmer tries to free the already freed resource. The impact of this common error can be catastrophic. An example of this error is:

```
main()
{
  char *str;
  str=(char *)malloc(10);
  if (global==0)
        free(str);
  free(str); /* str is already freed
}
```

The impact of this error can be more severe if we have some malloc statement between the two free statements—there is a chance that the first freed location is now allocated to the new variable and the subsequent free will deallocate it!

### NULL Dereferencing

This error occurs when we try to access the contents of a location that points to NULL. This is a commonly occurring error which can bring a software system down. It is also difficult to detect as it the NULL dereferencing may occur only in some paths and only under certain situations. Often improper initialization in the different paths leads to the NULL reference statement. It can also be caused because of aliases—for example, two variables refer to the same object, and one is freed and an attempt is made to dereference the second. This code segment shows two instances of NULL dereference.

```
char *ch=NULL;
if (x>0)
{
  ch='c';
}
printf("\%c", *ch); /* ch may be NULL
*ch=malloc(size);
ch = 'c'; /* ch will be NULL if malloc returns NULL
```

Similar to NULL dereference is the error of accessing uninitialized memory. This often occurs if data is initialized in most cases, but some cases do not get covered. they were not expected. An example of this error is:

```
switch( i )
{
      case 0: s=OBJECT_1; break;
      case 1: s=OBJECT_2;break;
}
return (s); /* s not initialized for values
               other than 0 or 1 */
```

### Lack of Unique Addresses

Aliasing creates many problems, and among them is violation of unique addresses when we expect different addresses. For example in the string concatenation function, we expect source and destination addresses to be different. If this is not the case, as is the situation in the code segment below, it can lead to runtime errors.

```
strcat(src,destn);
/* In above function, if src is aliased to destn,
 * then we may get a runtime error */
```

#### Synchronization Errors

In a parallel program, where there are multiple threads possibly accessing some common resources, then synchronization errors are possible [43, 55]. These errors are very difficult to find as they don't manifest easily. But when they do manifest, they can cause serious damage to the system. There are different categories of synchronization errors, some of which are:

1. Deadlocks
2. Race conditions
3. Inconsistent synchronization

Deadlock is a situation in which one or more threads mutually lock each other. The most frequent reason for the cause of deadlocks is inconsistent locking sequence—the threads in deadlock wait for resources which are in turn locked by some other thread. Race conditions occur when two threads try to access the same resource and the result of the execution depends on the order of the execution of the threads. Inconsistent synchronization is also a common error representing the situation where there is a mix of locked and unlocked accesses to some shared variables, particularly if the access involves updates. Some examples of these errors are given in the [141].

#### Array Index Out of Bounds

Array index often goes out of bounds, leading to exceptions. Care needs to be taken to see that the array index values are not negative and do not exceed their bounds.

#### Arithmetic exceptions

These include errors like divide by zero and floating point exceptions. The result of these may vary from getting unexpected results to termination of the program.

#### Off by One

This is one of the most common errors which can be caused in many ways. For example, starting at 1 when we should start at 0 or vice versa, writing <= N instead of < N or vice versa, and so on.

#### Enumerated data types

Overflow and underflow errors can easily occur when working with enumerated types, and care should be taken when assuming the values of enumerated data types. An example of such an error is:

```
typedef enum {A, B,C, D} grade;
void foo(grade x)
{
      int l,m;
      l=GLOBAL_ARRAY[x-1]; /* Underflow possible */
      m=GLOBAL_ARRAY[x+1]; /* Overflow possible */
}
```

### Illegal use of & instead of &&

This bug arises if non short circuit logic (like & or |) is used instead of short circuit logic (&& or ||). Non short circuit logic will evaluate both sides of the expression. But short circuit operator evaluates one side and based on the result, it decides if it has to evaluate the other side or not. An example is:

```
if(object!= null & object.getTitle() != null)
/* Here second operation can cause a null dereference */
```

### String handling errors

There are a number of ways in which string handling functions like strcpy, sprintf, gets etc can fail. Examples are one of the operands is NULL, the string is not NULL terminated, or the source operand may have greater size than the destination. String handling errors are quite common.

### Buffer overflow

Though buffer overflow is also a frequent cause of software failures, in todays world its main impact is that it is a security flaw that can be exploited by a malicious user for executing arbitrary code.

When a program takes an input which is being copied in a buffer, by giving a large (and malicious) input, a malicious user can overflow the buffer on the stack. By doing this, the return address can get rewritten to whatever the malicious user has planned. So, when the function call ends, the control goes to where the malicious user has planned, which is typically some malicious code to take control of the computer or do some harmful actions. Basically, by exploiting the buffer overflow situation, a malicious user can execute arbitrary code. The following code fragment illustrates buffer overflow:

```
void mygets(char *str){
      int ch;
      while(ch=getchar() !='\n' && ch!='\0')
            *(str++)=ch;
      *str='\0';
}
```

```
main(){
      char s2[4];
      mygets(s2);
}
```

Here there is a possible buffer overflow attack. If the input given is large, it can overflow the buffer s2, and by carefully crafting the bytes that go on the stack the return address of mygets() can be replaced by an address of a malicious program. For further discussion on buffer overflow and on writing code that is secure, the reader is referred to [88].

### 9.1.2 Structured Programming

As stated earlier the basic objective of the coding activity is to produce programs that are easy to understand. It has been argued by many that structured programming practice helps develop programs that are easier to understand. The structured programming movement started in the 1970s, and much has been said and written about it. Now the concept pervades so much that it is generally accepted—even implied—that programming should be structured. Though a lot of emphasis has been placed on structured programming, the concept and motivation behind structured programming are often not well understood. Structured programming is often regarded as "goto-less" programming. Although extensive use of gotos is certainly not desirable, structured programs *can* be written with the use of gotos. Here we provide a brief discussion on what structured programming is.

A program has a static structure as well as a dynamic structure. The static structure is the structure of the text of the program, which is usually just a linear organization of statements of the program. The dynamic structure of the program is the sequence of statements executed during the execution of the program. In other words, both the static structure and the dynamic behavior are sequences of statements; where the sequence representing the static structure of a program is fixed, the sequence of statements it executes can change from execution to execution.

The general notion of correctness of the program means that when the program executes, it produces the desired behavior. To show that a program is correct, we need to show that when the program executes, its behavior is what is expected. Consequently, when we argue about a program, either formally to prove that it is correct or informally to debug it or convince ourselves that it works, we study the static structure of the program (i.e., its code) but try to argue about its dynamic behavior. In other words, much of the activity of program understanding is to understand the dynamic behavior of the program from the text of the program.

It will clearly be easier to understand the dynamic behavior if the structure in the dynamic behavior resembles the static structure. The closer the correspondence

between execution and text structure, the easier the program is to understand, and the more different the structure during execution, the harder it will be to argue about the behavior from the program text. The goal of structured programming is to ensure that the static structure and the dynamic structures are the same. That is, the objective of structured programming is to write programs so that the sequence of statements executed during the execution of a program is the same as the sequence of statements in the text of that program. As the statements in a program text are linearly organized, the objective of structured programming becomes developing programs whose control flow during execution is linearized and follows the linear organization of the program text.

Clearly, no meaningful program can be written as a sequence of simple statements without any branching or repetition (which also involves branching). So, how is the objective of linearizing the control flow to be achieved? By making use of structured constructs. In structured programming, a statement is not a simple assignment statement, it is a structured statement. The key property of a structured statement is that it has a *single-entry* and a *single-exit*. That is, during execution, the execution of the (structured) statement starts from one defined point and the execution terminates at one defined point. With single-entry and single-exit statements, we can view a program as a sequence of (structured) statements. And if all statements are structured statements, then during execution, the sequence of execution of these statements will be the same as the sequence in the program text. Hence, by using single-entry and single-exit statements, the correspondence between the static and dynamic structures can be obtained. The most commonly used single-entry and single-exit statements are:

*Selection:* if B then S1 else S2
if B then S1
*Repetition:* While B do S
repeat S until B
*Sequencing:* S1; S2; S3;...

It can be shown that these three basic constructs are sufficient to program any conceivable algorithm. Modern languages have other such constructs that help linearize the control flow of a program, which, generally speaking, makes it easier to understand a program. Hence, programs should be written so that, as far as possible, single-entry, single-exit control constructs are used. The basic goal, as we have tried to emphasize, is to make the logic of the program simple to understand. No hard-and-fast rule can be formulated that will be applicable under all circumstances. Structured programming practice forms a good basis and guideline for writing programs clearly.

It should be pointed out that the main reason structured programming was promulgated is formal verification of programs. As we will see later in this chapter, during verification, a program is considered a sequence of executable statements, and verifica-

tion proceeds step by step, considering one statement in the statement list (the program) at a time. Implied in these verification methods is the assumption that during execution, the statements will be executed in the sequence in which they are organized in the program text. If this assumption is satisfied, the task of verification becomes easier. Hence, even from the point of view of verification, it is important that the sequence of execution of statements is the same as the sequence of statements in the text.

A final note about the structured constructs. Any piece of code with a single-entry and single-exit cannot be considered a structured construct. If that is the case, one could always define appropriate units in any program to make it appear as a sequence of these units (in the worst case, the whole program could be defined to be a unit). The basic objective of using structured constructs is to linearize the control flow so that the execution behavior is easier to understand and argue about. In linearized control flow, if we understand the behavior of each of the basic constructs properly, the behavior of the program can be considered a composition of the behaviors of the different statements. For this basic approach to work, it is implied that we can clearly understand the behavior of each construct. This requires that we be able to succinctly capture or describe the behavior of each construct. Unless we can do this, it will not be possible to compose them. Clearly, for an arbitrary structure, we cannot do this merely because it has a single-entry and single-exit. It is from this viewpoint that the structures mentioned earlier are chosen as structured statements. There are well-defined rules that specify how these statements behave during execution, which allows us to argue about larger programs.

Overall, it can be said that structured programming, in general, leads to programs that are easier to understand than unstructured programs, and that such programs are easier (relatively speaking) to formally prove. However, it should be kept in mind that structured programming is not an end in itself. Our basic objective is that the program be easy to understand. And structured programming is a safe approach for achieving this objective. Still, there are some common programming practices that are now well understood that make use of unstructured constructs (e.g., break statement, continue statement). Although efforts should be made to avoid using statements that effectively violate the single-entry single-exit property, if the use of such statements is the simplest way to organize the program, then from the point of view of readability, the constructs should be used. The main point is that any unstructured construct should be used only if the structured alternative is harder to understand. This view can be taken only because we are focusing on readability. If the objective was formal verifiability, structured programming will probably be necessary.

### 9.1.3 Information Hiding

A software solution to a problem always contains data structures that are meant to represent information in the problem domain. That is, when software is developed to

solve a problem, the software uses some data structures to capture the information in the problem domain.

In general, only certain operations are performed on some information. That is, a piece of information in the problem domain is used only in a limited number of ways in the problem domain. For example, a ledger in an accountant's office has some very defined uses: debit, credit, check the current balance, etc. An operation where all debits are multiplied together and then divided by the sum of all credits is typically not performed. So, any information in the problem domain typically has a small number of defined operations performed on it.

When the information is represented as data structures, the same principle should be applied, and only some defined operations should be performed on the data structures. This, essentially, is the principle of information hiding. The information captured in the data structures should be hidden from the rest of the system, and only the access functions on the data structures that represent the operations performed on the information should be visible. In other words, when the information is captured in data structures and then on the data structures that represent some information, for each operation on the information an access function should be provided. And as the rest of the system in the problem domain only performs these defined operations on the information, the rest of the modules in the software should only use these access functions to access and manipulate the data structures.

Information hiding can reduce the coupling between modules and make the system more maintainable. Information hiding is also an effective tool for managing the complexity of developing software—by using information hiding we have separated the concern of managing the data from the concern of using the data to produce some desired results.

Many of the older languages, like Pascal, C, and FORTRAN, do not provide mechanisms to support data abstraction. With such languages, information hiding can be supported only by a disciplined use of the language. That is, the access restrictions will have to be imposed by the programmers; the language does not provide them. Most modern OO languages provide linguistic mechanisms to implement information hiding.

### 9.1.4 Some Programming Practices

The concepts discussed above can help in writing simple and clear code with few bugs. There are many programming practices that can also help towards that objective. We discuss here a few rules that have been found to make code easier to read as well as avoid some of the errors. Some of these practices are from [141].

**Control Constructs:** As discussed earlier, it is desirable that as much as possible single-entry, single-exit constructs be used. It is also desirable to use a few standard control constructs rather than using a wide variety of constructs, just because they are available in the language.