Function-Oriented Design

The design activity begins when the requirements document for the software to be developed is available and the architecture has been designed. During design we further refine the architecture. Generally, design focuses on the what we have called the module view in Chapter 4. That is, during design we determine what modules should the system have and which have to be developed. Sometimes, the module view may effectively be a module structure of each component in the architecture. That is, the design exercise determines the module structure of the components. However, this simple mapping of components and modules may not always hold. In that case we have to ensure that the module view created in design is consistent with the architecture.

The design of a system is essentially a blueprint or a plan for a solution for the system. Here we consider a system to be a set of modules with clearly defined behavior which interact with each other in a defined manner to produce some behavior or services for its environment. A module of a system can be considered a system, with its own modules.

The design process for software systems often has two levels. At the first level the focus is on deciding which modules are needed for the system, the specifications of these modules, and how the modules should be interconnected. This is what is called the system design or top-level design. In the second level, the internal design of the modules, or how the specifications of the module can be satisfied, is decided. This design level is often called detailed design or logic design. Detailed design essentially expands the system design to contain a more detailed description of the processing logic and data structures so that the design is sufficiently complete for coding.
A design methodology is a systematic approach to creating a design by applying of a set of techniques and guidelines. Most design methodologies focus on the system design, and do not reduce the design activity to a sequence of steps that can be blindly followed by the designer.

In this chapter we discuss the function-oriented methods for design and describe one particular methodology—the structured design methodology—in some detail. In a function-oriented design approach, a system is viewed as a transformation function, transforming the inputs to the desired outputs. The purpose of the design phase is to specify the components for this transformation function, so that each component is also a transformation function. That is, each module in design supports a functional abstraction. The basic output of the system design phase, when a function oriented design approach is being followed, is the definition of all the major data structures in the system, all the major modules of the system, and how the modules interact with each other.

In this chapter, we first discuss some general design principles. Then we discuss a notation for expressing function-oriented designs and describe the structured design methodology for developing a design. Then we discuss some verification methods for design and some metrics that are applicable to function-oriented designs. As in most chapters, we will end with the case studies.

### 6.1 Design Principles

The design of a system is correct if a system built precisely according to the design satisfies the requirements of that system. Clearly, the goal during the design phase is to produce correct designs. However, correctness is not the sole criterion during the design phase, as there can be many correct designs. The goal of the design process is not simply to produce a design for the system. Instead, the goal is to find the best possible design within the limitations imposed by the requirements and the physical and social environment in which the system will operate.

To evaluate a design, we have to specify some properties and criteria that can be used for evaluation. Ideally, these properties should be as quantitative as possible. In that situation we can precisely evaluate the “goodness” of a design and determine the best design. However, criteria for quality of software design is often subjective or non-quantifiable. In such a situation, criteria are essentially thumb rules that aid design evaluation.
A design should clearly be verifiable, complete (implements all the specifications), and traceable (all design elements can be traced to some requirements). However, the two most important properties that concern designers are efficiency and simplicity. Efficiency of any system is concerned with the proper use of scarce resources by the system. The need for efficiency arises due to cost considerations. If some resources are scarce and expensive, it is desirable that those resources be used efficiently. In computer systems, the resources that are most often considered for efficiency are processor time and memory. An efficient system is one that consumes less processor time and requires less memory. In earlier days, the efficient use of CPU and memory was important due to the high cost of hardware. Now that the hardware costs are low compared to the software costs, for many software systems traditional efficiency concerns now take a back seat compared to other considerations. One of the exceptions is real-time systems, for which there are strict execution time constraints.

Simplicity is perhaps the most important quality criteria for software systems. We have seen that maintenance of software is usually quite expensive. Maintainability of software is one of the goals we have established. The design of a system is one of the most important factors affecting the maintainability of a system. During maintenance, the first step a maintainer has to undertake is to understand the system to be maintained. Only after a maintainer has a thorough understanding of the different modules of the system, how they are interconnected, and how modifying one will affect the others should the modification be undertaken. A simple and understandable design will go a long way in making the job of the maintainer easier.

These criteria are not independent, and increasing one may have an unfavorable effect on another. For example, often the "tricks" used to increase efficiency of a system result in making the system more complex. Therefore, design decisions frequently involve trade-offs. It is the designers’ job to recognize the trade-offs and achieve the best balance. For our purposes, simplicity is the primary property of interest, and therefore the objective of the design process is to produce designs that are simple to understand.

Creating a simple (and efficient) design of a large system can be an extremely complex task that requires good engineering judgment. As designing is fundamentally a creative activity, it cannot be reduced to a series of steps that can be simply followed, though guidelines can be provided. In this section we will examine some basic guiding principles that can be used to produce the design of a system. Some of these design principles are concerned with providing means to effectively handle the complexity of the
design process. Effectively handling the complexity will not only reduce the effort needed for design (i.e., reduce the design cost), but can also reduce the scope of introducing errors during design. The principles discussed here form the basis for most of the design methodologies.

It should be noted that the principles that can be used in design are the same as those used in problem analysis. In fact, the methods are also similar because in both analysis and design we are essentially constructing models. However, there are some fundamental differences. First, in problem analysis, we are constructing a model of the problem domain, while in design we are constructing a model for the solution domain. Second, in problem analysis, the analyst has limited degrees of freedom in selecting the models as the problem is given, and modeling has to represent it. In design, the designer has a great deal of freedom in deciding the models, as the system the designer is modeling does not exist; in fact the designer is creating a model for the system that will be the basis of building the system. That is, in design, the system depends on the model, while in problem analysis the model depends on the system. Finally, as pointed out earlier, the basic aim of modeling in problem analysis is to understand, while the basic aim of modeling in design is to optimize (in our case, simplicity and performance). In other words, though the basic principles and techniques might look similar, the activities of analysis and design are very different.

6.1.1 Problem Partitioning and Hierarchy

When solving a small problem, the entire problem can be tackled at once. The complexity of large problems and the limitations of human minds do not allow large problems to be treated as huge monoliths. For solving larger problems, the basic principle is the time-tested principle of “divide and conquer.” Clearly, dividing in such a manner that all the divisions have to be conquered together is not the intent of this wisdom. This principle, if elaborated, would mean “divide into smaller pieces, so that each piece can be conquered separately.”

For software design, therefore, the goal is to divide the problem into manageably small pieces that can be solved separately. It is this restriction of being able to solve each part separately that makes dividing into pieces a complex task and that many methodologies for system design aim to address. The basic rationale behind this strategy is the belief that if the pieces of a problem are solvable separately, the cost of solving the entire problem is more than the sum of the cost of solving all the pieces.
6.1. DESIGN PRINCIPLES

However, the different pieces cannot be entirely independent of each other, as they together form the system. The different pieces have to cooperate and communicate to solve the larger problem. This communication adds complexity, which arises due to partitioning and may not have existed in the original problem. As the number of components increases, the cost of partitioning, together with the cost of this added complexity, may become more than the savings achieved by partitioning. It is at this point that no further partitioning needs to be done. The designer has to make the judgment about when to stop partitioning.

As discussed earlier, two of the most important quality criteria for software design are simplicity and understandability. It can be argued that maintenance is minimized if each part in the system can be easily related to the application and each piece can be modified separately. If a piece can be modified separately, we call it independent of other pieces. If module A is independent of module B, then we can modify A without introducing any unanticipated side effects in B. Total independence of modules of one system is not possible, but the design process should support as much independence as possible between modules. Dependence between modules in a software system is one of the reasons for high maintenance costs. Clearly, proper partitioning will make the system easier to maintain by making the design easier to understand. Problem partitioning also aids design verification.

Problem partitioning, which is essential for solving a complex problem, leads to hierarchies in the design. That is, the design produced by using problem partitioning can be represented as a hierarchy of components. The relationship between the elements in this hierarchy can vary depending on the method used. For example, the most common is the "whole-part of" relationship. In this, the system consists of some parts, each part consists of subparts, and so on. This relationship can be naturally represented as a hierarchical structure between various system parts. In general, hierarchical structure makes it much easier to comprehend a complex system. Due to this, all design methodologies aim to produce a design that employs hierarchical structures.

6.1.2 Abstraction

Abstraction is a very powerful concept that is used in all engineering disciplines. It is a tool that permits a designer to consider a component at an abstract level without worrying about the details of the implementation of the component. Any component or system provides some services to its en-
6. FUNCTION-ORIENTED DESIGN

An abstraction of a component describes the external behavior of that component without bothering with the internal details that produce the behavior. Presumably, the abstract definition of a component is much simpler than the component itself.

Abstraction is an indispensable part of the design process and is essential for problem partitioning. Partitioning essentially is the exercise in determining the components of a system. However, these components are not isolated from each other; they interact with each other, and the designer has to specify how a component interacts with other components. To decide how a component interacts with other components, the designer has to know, at the very least, the external behavior of other components. If the designer has to understand the details of the other components to determine their external behavior, we have defeated the purpose of partitioning—isolating a component from others. To allow the designer to concentrate on one component at a time, abstraction of other components is used.

Abstraction is used for existing components as well as components that are being designed. Abstraction of existing components plays an important role in the maintenance phase. To modify a system, the first step is understanding what the system does and how. The process of comprehending an existing system involves identifying the abstractions of subsystems and components from the details of their implementations. Using these abstractions, the behavior of the entire system can be understood. This also helps determine how modifying a component affects the system.

During the design process, abstractions are used in the reverse manner than in the process of understanding a system. During design, the components do not exist, and in the design the designer specifies only the abstract specifications of the different components. The basic goal of system design is to specify the modules in a system and their abstractions. Once the different modules are specified, during the detailed design the designer can concentrate on one module at a time. The task in detailed design and implementation is essentially to implement the modules so that the abstract specifications of each module are satisfied.

There are two common abstraction mechanisms for software systems: functional abstraction and data abstraction. In functional abstraction, a module is specified by the function it performs. For example, a module to compute the log of a value can be abstractly represented by the function log. Similarly, a module to sort an input array can be represented by the specification of sorting. Functional abstraction is the basis of partitioning in function-oriented approaches. That is, when the problem is being parti-
tioned, the overall transformation function for the system is partitioned into smaller functions that comprise the system function. The decomposition of the system is in terms of functional modules.

The second unit for abstraction is *data abstraction*. Any entity in the real world provides some services to the environment to which it belongs. Often the entities provide some fixed predefined services. The case of data entities is similar. Certain operations are required from a data object, depending on the object and the environment in which it is used. Data abstraction supports this view. Data is not treated simply as objects, but is treated as objects with some predefined operations on them. The operations defined on a data object are the only operations that can be performed on those objects. From outside an object, the internals of the object are hidden; only the operations on the object are visible. Data abstraction forms the basis for *object-oriented design*, which is discussed in the next chapter. In using this abstraction, a system is viewed as a set of objects providing some services. Hence, the decomposition of the system is done with respect to the objects the system contains.

6.1.3 Modularity

As mentioned earlier, the real power of partitioning comes if a system is partitioned into modules so that the modules are solvable and modifiable separately. It will be even better if the modules are also separately compilable (then changes in a module will not require recompilation of the whole system). A system is considered *modular* if it consists of discreet components so that each component can be implemented separately, and a change to one component has minimal impact on other components.

Modularity is a clearly a desirable property in a system. Modularity helps in system debugging—isolating the system problem to a component is easier if the system is modular; in system repair—changing a part of the system is easy as it affects few other parts; and in system building—a modular system can be easily built by “putting its modules together.”

A software system cannot be made modular by simply chopping it into a set of modules. For modularity, each module needs to support a well-defined abstraction and have a clear interface through which it can interact with other modules. Modularity is where abstraction and partitioning come together. For easily understandable and maintainable systems, modularity is clearly the basic objective; partitioning and abstraction can be viewed as concepts that help achieve modularity.
6.1.4 Top-Down and Bottom-Up Strategies

A system consists of components, which have components of their own; indeed a system is a hierarchy of components. The highest-level component correspond to the total system. To design such a hierarchy there are two possible approaches: top-down and bottom-up. The top-down approach starts from the highest-level component of the hierarchy and proceeds through to lower levels. By contrast, a bottom-up approach starts with the lowest-level component of the hierarchy and proceeds through progressively higher levels to the top-level component.

A top-down design approach starts by identifying the major components of the system, decomposing them into their lower-level components and iterating until the desired level of detail is achieved. Top-down design methods often result in some form of stepwise refinement. Starting from an abstract design, in each step the design is refined to a more concrete level, until we reach a level where no more refinement is needed and the design can be implemented directly. The top-down approach has been promulgated by many researchers and has been found to be extremely useful for design. Most design methodologies are based on the top-down approach.

A bottom-up design approach starts with designing the most basic or primitive components and proceeds to higher-level components that use these lower-level components. Bottom-up methods work with layers of abstraction. Starting from the very bottom, operations that provide a layer of abstraction are implemented. The operations of this layer are then used to implement more powerful operations and a still higher layer of abstraction, until the stage is reached where the operations supported by the layer are those desired by the system.

A top-down approach is suitable only if the specifications of the system are clearly known and the system development is from scratch. However, if a system is to be built from an existing system, a bottom-up approach is more suitable, as it starts from some existing components. So, for example, if an iterative enhancement type of process is being followed, in later iterations, the bottom-up approach could be more suitable (in the first iteration a top-down approach can be used).

Pure top-down or pure bottom-up approaches are often not practical. For a bottom-up approach to be successful, we must have a good notion of the top to which the design should be heading. Without a good idea about the operations needed at the higher layers, it is difficult to determine what operations the current layer should support. Top-down approaches
require some idea about the feasibility of the components specified during design. The components specified during design should be implementable, which requires some idea about the feasibility of the lower-level parts of a component. A common approach to combine the two approaches is to provide a layer of abstraction for the application domain of interest through libraries of functions, which contains the functions of interest to the application domain. Then use a top-down approach to determine the modules in the system, assuming that the abstract machine available for implementing the system provides the operations supported by the abstraction layer. This approach is frequently used for developing systems. It can even be claimed that it is almost universally used these days, as most developments now make use of the layer of abstraction supported in a system consisting of the library functions provided by operating systems, programming languages, and special-purpose tools.

6.2 Module-Level Concepts

In the previous section we discussed some general design principles. Now we turn our attention to some concepts specific to function-oriented design. Before we discuss these, let us define what we mean by a module. A module is a logically separable part of a program. It is a program unit that is discreet and identifiable with respect to compiling and loading. In terms of common programming language constructs, a module can be a macro, a function, a procedure (or subroutine), a process, or a package. In systems using functional abstraction, a module is usually a procedure of function or a collection of these.

To produce modular designs, some criteria must be used to select modules so that the modules support well-defined abstractions and are solvable and modifiable separately. In a system using functional abstraction, coupling and cohesion are two modularization criteria, which are often used together.

6.2.1 Coupling

Two modules are considered independent if one can function completely without the presence of other. Obviously, if two modules are independent, they are solvable and modifiable separately. However, all the modules in a system cannot be independent of each other, as they must interact so that together they produce the desired external behavior of the system. The more connections between modules, the more dependent they are in the sense
that more knowledge about one module is required to understand or solve the other module. Hence, the fewer and simpler the connections between modules, the easier it is to understand one without understanding the other. The notion of coupling [138, 154] attempts to capture this concept of "how strongly" different modules are interconnected.

**Coupling** between modules is the strength of interconnections between modules or a measure of interdependence among modules. In general, the more we must know about module A in order to understand module B, the more closely connected A is to B. "Highly coupled" modules are joined by strong interconnections, while "loosely coupled" modules have weak interconnections. Independent modules have no interconnections. To solve and modify a module separately, we would like the module to be loosely coupled with other modules. The choice of modules decides the coupling between modules. Because the modules of the software system are created during system design, the coupling between modules is largely decided during system design and cannot be reduced during implementation.

Coupling increases with the complexity and obscurity of the interface between modules. To keep coupling low we would like to minimize the number of interfaces per module and the complexity of each interface. An interface of a module is used to pass information to and from other modules. Coupling is reduced if only the defined entry interface of a module is used by other modules (for example, passing information to and from a module exclusively through parameters). Coupling would increase if a module is used by other modules via an indirect and obscure interface, like directly using the internals of a module or using shared variables.

Complexity of the interface is another factor affecting coupling. The more complex each interface is, the higher will be the degree of coupling. For example, complexity of the entry interface of a procedure depends on the number of items being passed as parameters and on the complexity of the items. Some level of complexity of interfaces is required to support the communication needed between modules. However, often more than this minimum is used. For example, if a field of a record is needed by a procedure, often the entire record is passed, rather than just passing that field of the record. By passing the record we are increasing the coupling unnecessarily. Essentially, we should keep the interface of a module as simple and small as possible.

The type of information flow along the interfaces is the third major factor affecting coupling. There are two kinds of information that can flow along an interface: data or control. Passing or receiving control information means
6.2. MODULE-LEVEL CONCEPTS

<table>
<thead>
<tr>
<th>Interface Complexity</th>
<th>Type of Connection</th>
<th>Type of Communication</th>
</tr>
</thead>
<tbody>
<tr>
<td>Low</td>
<td>Simple obvious</td>
<td>To module by name</td>
</tr>
<tr>
<td>High</td>
<td>Complicated obscure</td>
<td>To internal elements</td>
</tr>
</tbody>
</table>

Table 6.1: Factors affecting coupling.

that the action of the module will depend on this control information, which makes it more difficult to understand the module and provide its abstraction. Transfer of data information means that a module passes as input some data to another module and gets in return some data as output. This allows a module to be treated as a simple input-output function that performs some transformation on the input data to produce the output data. In general, interfaces with only data communication result in the lowest degree of coupling, followed by interfaces that only transfer control data. Coupling is considered highest if the data is hybrid, that is, some data items and some control items are passed between modules. The effect of these three factors on coupling is summarized in Table 6.1 [138].

6.2.2 Cohesion

We have seen that coupling is reduced when the relationships among elements in different modules are minimized. That is, coupling is reduced when elements in different modules have little or no bonds between them. Another way of achieving this effect is to strengthen the bond between elements of the same module by maximizing the relationship between elements of the same module. Cohesion is the concept that tries to capture this intramodule [138, 154]. With cohesion, we are interested in determining how closely the elements of a module are related to each other.

Cohesion of a module represents how tightly bound the internal elements of the module are to one another. Cohesion of a module gives the designer an idea about whether the different elements of a module belong together in the same module. Cohesion and coupling are clearly related. Usually, the greater the cohesion of each module in the system, the lower the coupling between modules is. This correlation is not perfect, but it has been observed
in practice. There are several levels of cohesion:

- Coincidental
- Logical
- Temporal
- Procedural
- Communicational
- Sequential
- Functional

Coincidental is the lowest level, and functional is the highest. These levels do not form a linear scale. Functional binding is much stronger than the rest, while the first two are considered much weaker than others. Often, many levels can be applicable when considering cohesion between two elements of a module. In such situations, the highest level is considered. Cohesion of a module is considered the highest level of cohesion applicable to all elements in the module.

Coincidental cohesion occurs when there is no meaningful relationship among the elements of a module. Coincidental cohesion can occur if an existing program is "modularized" by chopping it into pieces and making different pieces modules. If a module is created to save duplicate code by combining some part of code that occurs at many different places, that module is likely to have coincidental cohesion. In this situation, the statements in the module have no relationship with each other, and if one of the modules using the code needs to be modified and this modification includes the common code, it is likely that other modules using the code do not want the code modified. Consequently, the modification of this "common module" may cause other modules to behave incorrectly. The modules using these modules are therefore not modifiable separately and have strong interconnection between them. We can say that, generally speaking, it is poor practice to create a module merely to avoid duplicate code (unless the common code happens to perform some identifiable function, in which case the statements will have some relationship between them) or to chop a module into smaller modules to reduce the module size.

A module has logical cohesion if there is some logical relationship between the elements of a module, and the elements perform functions that
fall in the same logical class. A typical example of this kind of cohesion is a module that performs all the inputs or all the outputs. In such a situation, if we want to input or output a particular record, we have to somehow convey this to the module. Often, this will be done by passing some kind of special status flag, which will be used to determine what statements to execute in the module. Besides resulting in hybrid information flow between modules, which is generally the worst form of coupling between modules, such a module will usually have tricky and clumsy code. In general, logically cohesive modules should be avoided, if possible.

Temporal cohesion is the same as logical cohesion, except that the elements are also related in time and are executed together. Modules that perform activities like “initialization,” “clean-up,” and “termination” are usually temporally bound. Even though the elements in a temporally bound module are logically related, temporal cohesion is higher than logical cohesion, because the elements are all executed together. This avoids the problem of passing the flag, and the code is usually simpler.

A procedurally cohesive module contains elements that belong to a common procedural unit. For example, a loop or a sequence of decision statements in a module may be combined to form a separate module. Procedurally cohesive modules often occur when modular structure is determined from some form of flowchart. Procedural cohesion often cuts across functional lines. A module with only procedural cohesion may contain only part of a complete function or parts of several functions.

A module with communicational cohesion has elements that are related by a reference to the same input or output data. That is, in a communicationally bound module, the elements are together because they operate on the same input or output data. An example of this could be a module to “print and punch record.” Communicationally cohesive modules may perform more than one function. However, communicational cohesion is sufficiently high as to be generally acceptable if alternative structures with higher cohesion cannot be easily identified.

When the elements are together in a module because the output of one forms the input to another, we get sequential cohesion. If we have a sequence of elements in which the output of one forms the input to another, sequential cohesion does not provide any guidelines on how to combine them into modules. Different possibilities exist: combine all in one module, put the first half in one and the second half in another, the first third in one and the rest in the other, and so forth. Consequently, a sequentially bound module may contain several functions or parts of different functions. Sequentially cohe-
sive modules bear a close resemblance to the problem structure. However, they are considered to be far from the ideal, which is functional cohesion.

Functional cohesion is the strongest cohesion. In a functionally bound module, all the elements of the module are related to performing a single function. By function, we do not mean simply mathematical functions; modules accomplishing a single goal are also included. Functions like "compute square root" and "sort the array" are clear examples of functionally cohesive modules.

How does one determine the cohesion level of a module? There is no mathematical formula that can be used. We have to use our judgment for this. A useful technique for determining if a module has functional cohesion is to write a sentence that describes, fully and accurately, the function or purpose of the module. The following tests can then be made [138]:

1. If the sentence must be a compound sentence, if it contains a comma, or if it has has more than one verb, the module is probably performing more than one function, and it probably has sequential or communicational cohesion.

2. If the sentence contains words relating to time, like "first," "next," "when," and "after" the module probably has sequential or temporal cohesion.

3. If the predicate of the sentence does not contain a single specific object following the verb (such as "edit all data") the module probably has logical cohesion.

4. Words like "initialize" and "cleanup" imply temporal cohesion.

Modules with functional cohesion can always be described by a simple sentence. However, if a description is a compound sentence, it does not mean that the module does not have functional cohesion. Functionally cohesive modules can also be described by compound sentences. If we cannot describe it using a simple sentence, the module is not likely to have functional cohesion.

6.3 Design Notation and Specification

During the design phase there are two things of interest: the design of the system, the producing of which is the basic objective of this phase, and the
6.3. DESIGN NOTATION AND SPECIFICATION

Process of designing itself. It is for the latter that principles and methods are needed. In addition, while designing, a designer needs to record his thoughts and decisions and to represent the design so that he can view it and play with it. For this, design notations are used.

Design notations are largely meant to be used during the process of design and are used to represent design or design decisions. They are meant largely for the designer so that he can quickly represent his decisions in a compact manner that he can evaluate and modify. These notations are frequently graphical.

Once the designer is satisfied with the design he has produced, the design is to be precisely specified in the form of a document. Whereas a design represented using the design notation is largely to be used by the designer, a design specification has to be so precise and complete that it can be used as a basis of further development by other programmers. Often, design specification uses textual structures, with design notation helping understanding.

6.3.1 Structure Charts

For a function-oriented design, the design can be represented graphically by structure charts. The structure of a program is made up of the modules of that program together with the interconnections between modules. Every computer program has a structure, and given a program its structure can be determined. The structure chart of a program is a graphic representation of its structure. In a structure chart a module is represented by a box with the module name written in the box. An arrow from module A to module B represents that module A invokes module B. B is called the subordinate of A, and A is called the superordinate of B. The arrow is labeled by the parameters received by B as input and the parameters returned by B as output, with the direction of flow of the input and output parameters represented by small arrows. The parameters can be shown to be data (unfilled circle at the tail of the label) or control (filled circle at the tail). As an example consider the structure of the following program, whose structure is shown in Figure 6.1.

```c
main()
{
    int sum, n, N, a[MAX];
    readnums(a, &N); sort(a, N); scanf(&n);
    sum = add_n(a, n); printf(sum);
}

readnums(a, N)
```
In general, procedural information is not represented in a structure chart, and the focus is on representing the hierarchy of modules. However, there are situations where the designer may wish to communicate certain procedural information explicitly, like major loops and decisions. Such information can

```c
int a[], *N;
{
    :
}

sort(a, N)
int a[], N;
{
    :
        if (a[i] > a[t]) switch(a[i], a[t]);
    :
}

/* Add the first n numbers of a */
add_n(a, n)
int a[], n;
{
    :
}
```
also be represented in a structure chart. For example, let us consider a situation where module A has subordinates B, C, and D, and A repeatedly calls the modules C and D. This can be represented by a looping arrow around the arrows joining the subordinates C and D to A, as shown in Figure 6.2. All the subordinate modules activated within a common loop are enclosed in the same looping arrow.

Major decisions can be represented similarly. For example, if the invocation of modules C and D in module A depends on the outcome of some decision, that is represented by a small diamond in the box for A, with the arrows joining C and D coming out of this diamond, as shown in Figure 6.2.

Modules in a system can be categorized into few classes. There are some modules that obtain information from their subordinates and then pass it to their superordinate. This kind of module is an input module. Similarly, there are output modules that take information from their superordinate and pass it on to its subordinates. As the name suggests, the input and output modules are typically used for input and output of data from and to the environment. The input modules get the data from the sources and get it ready to be processed, and the output modules take the output produced and prepare it for proper presentation to the environment. Then there are modules that exist solely for the sake of transforming data into some other form. Such a module is called a transform module. Most of the computational modules typically fall in this category. Finally, there are modules whose primary concern is managing the flow of data to and from different subordinates. Such modules are called coordinate modules. The structure chart representation of the different types of modules is shown in Figure 6.3.
A module can perform functions of more than one type of module. For example, the composite module in Figure 6.3 is an input module from the point of view of its superordinate, as it feeds the data $Y$ to the superordinate. Internally, $A$ is a coordinate module and views its job as getting data $X$ from one subordinate and passing it to another subordinate, which converts it to $Y$. Modules in actual systems are often composite modules.

A structure chart is a nice representation mechanism for a design that uses functional abstraction. It shows the modules and their call hierarchy, the interfaces between the modules, and what information passes between modules. It is a convenient and compact notation that is very useful while creating the design. That is, a designer can make effective use of structure charts to represent the model he is creating while he is designing. However, it is not sufficient for representing the final design, as it does not give all the information needed about the design. For example, it does not specify the scope, structure of data, specifications of each module, etc. Hence, it is generally supplemented with textual specifications to convey design to the
6.3. DESIGN NOTATION AND SPECIFICATION

implementer.

We have seen how to determine the structure of an existing program. But once the program is written, its structure is fixed and little can be done about altering the structure. However, for a given set of requirements many different programs can be written to satisfy the requirements, and each program can have a different structure. That is, although the structure of a given program is fixed, for a given set of requirements, programs with different structures can be obtained. The objective of the design phase using function-oriented method is to control the eventual structure of the system by fixing the structure during design.

6.3.2 Specification

Using some design rules or methodology, a conceptual design of the system can be produced in terms of a structure chart. As seen earlier, in a structure chart each module is represented by a box with a name. The functionality of the module is essentially communicated by the name of the box, and the interface is communicated by the data items labeling the arrows. This is alright while the designer is designing but inadequate when the design is to be communicated. To avoid these problems, a design specification should define the major data structures, modules and their specifications, and design decisions.

During system design, the major data structures for the software are identified; without these, the system modules cannot be meaningfully defined during design. In the design specification, a formal definition of these data structures should be given.

Module specification is the major part of system design specification. All modules in the system should be identified when the system design is complete, and these modules should be specified in the document. During system design only the module specification is obtained, because the internal details of the modules are defined later. To specify a module, the design document must specify (a) the interface of the module (all data items, their types, and whether they are for input and/or output), (b) the abstract behavior of the module (what the module does) by specifying the module’s functionality or its input/output behavior, and (c) all other modules used by the module being specified—this information is quite useful in maintaining and understanding the design.

Hence, a design specification will necessarily contain specification of the major data structures and modules in the system. After a design is ap-
proved (using some verification mechanism), the modules will have to be implemented in the target language. This requires that the module “heads­ers” for the target language first be created from the design. This translation of the design for the target language can introduce errors if it’s done man­ually. To eliminate these translation errors, if the target language is known (as is generally the case after the requirements have been specified), it is better to have a design specification language whose module specifications can be used almost directly in programming. This not only minimizes the translation errors that may occur, but also reduces the effort required for translating the design to programs. It also adds incentive for designers to properly specify their design, as the design is no longer a “mere” document that will be thrown away after review—it will now be used directly in coding. In the case study, a design specification language close to C has been used. From the design, the module headers for C can easily be created with some simple editing.

To aid the comprehensibility of the design, all major design decisions made by the designers during the design process should be explained explicit­ly. The choices that were available and the reasons for making a particular choice should be explained. This makes a design more visible and will help in understanding the design.

6.4 Structured Design Methodology

Creating the software system design is the major concern of the design phase. Many design techniques have been proposed over the years to provide some discipline in handling the complexity of designing large systems. The aim of design methodologies is not to reduce the process of design to a sequence of mechanical steps but to provide guidelines to aid the designer during the design process. Here we describe the structured design methodology [138, 154] for developing system designs.

Structured design methodology (SDM) views every software system as having some inputs that are converted into the desired outputs by the software system. The software is viewed as a transformation function that transforms the given inputs into the desired outputs, and the central problem of designing software systems is considered to be properly designing this transformation function. Due to this view of software, the structured design methodology is primarily function-oriented and relies heavily on functional abstraction and functional decomposition.
The concept of the structure of a program lies at the heart of the structured design method. During design, structured design methodology aims to control and influence the structure of the final program. The aim is to design a system so that programs implementing the design would have a hierarchical structure, with functionally cohesive modules and as few interconnections between modules as possible.

In properly designed systems it is often the case that a module with subordinates does not actually perform much computation. The bulk of actual computation is performed by its subordinates, and the module itself largely coordinates the data flow between the subordinates to get the computation done. The subordinates in turn can get the bulk of their work done by their subordinates until the “atomic” modules, which have no subordinates, are reached. *Factoring* is the process of decomposing a module so that the bulk of its work is done by its subordinates. A system is said to be completely factored if all the actual processing is accomplished by bottom-level atomic modules and if non-atomic modules largely perform the jobs of control and coordination. SDM attempts to achieve a structure that is close to being completely factored.

The overall strategy is to identify the input and output streams and the primary transformations that have to be performed to produce the output. High-level modules are then created to perform these major activities, which are later refined. There are four major steps in this strategy:

1. Restate the problem as a data flow diagram
2. Identify the input and output data elements
3. First-level factoring
4. Factoring of input, output, and transform branches

We will now discuss each of these steps in more detail. The design of the case study using structured design will be given later. For illustrating each step of the methodology as we discuss them, we consider the following problem: there is a text file containing words separated by blanks or new lines. We have to design a software system to determine the number of unique words in the file.

6.4.1 Restate the Problem as a Data Flow Diagram

To use the SD methodology, the first step is to construct the data flow diagram for the problem. We studied data flow diagrams in Chapter 3.
However, there is a fundamental difference between the DFDs drawn during requirements analysis and those drawn during structured design. In the requirements analysis, a DFD is drawn to model the problem domain. The analyst has little control over the problem, and hence his task is to extract from the problem all the information and then represent it as a DFD.

During design activity, we are no longer modeling the problem domain, but rather are dealing with the solution domain and developing a model for the eventual system. That is, the DFD during design represents how the data will flow in the system when it is built. In this modeling, the major transforms or functions in the software are decided, and the DFD shows the major transforms that the software will have and how the data will flow through different transforms. So, drawing a DFD for design is a very creative activity in which the designer visualizes the eventual system and its processes and data flows. As the system does not yet exist, the designer has complete freedom in creating a DFD that will solve the problem stated in the SRS. The general rules of drawing a DFD remain the same; we show what transforms are needed in the software and are not concerned with the logic for implementing them. Consider the example of the simple automated teller machine that allows customers to withdraw money. A DFD for this ATM is shown in Figure 6.4.

There are two major streams of input data in this diagram. The first is the account number and the code, and the second is the amount to be debited. The DFD is self-explanatory. Notice the use of * at different places in the DFD. For example, the transform "validate," which verifies if the account number and code are valid, needs not only the account number and code, but also information from the system database to do the validation. And the transform debit account has two outputs, one used for recording the transaction and the other to update the account.

As another example, consider the problem of determining the number of different words in an input file. The data flow diagram for this problem is shown in Figure 6.5.

This problem has only one input data stream, the input file, while the desired output is the count of different words in the file. To transform the input to the desired output, the first thing we do is form a list of all the words in the file. It is best to then sort the list, as this will make identifying different words easier. This sorted list is then used to count the number of different words, and the output of this transform is the desired count, which is then printed. This sequence of data transformation is what we have in the data flow diagram.
6.4. STRUCTURED DESIGN METHODOLOGY

6.4.2 Identify the Most Abstract Input and Output Data Elements

Most systems have some basic transformations that perform the required operations. However, in most cases the transformation cannot be easily applied to the actual physical input and produce the desired physical output. Instead, the input is first converted into a form on which the transformation can be applied with ease. Similarly, the main transformation modules often
produce outputs that have to be converted into the desired physical output. The goal of this second step is to separate the transforms in the data flow diagram that convert the input or output to the desired format from the ones that perform the actual transformations.

For this separation, once the data flow diagram is ready, the next step is to identify the highest abstract level of input and output. *The most abstract input data elements* are those data elements in the data flow diagram that are farthest removed from the physical inputs but can still be considered inputs to the system. The most abstract input data elements often have little resemblance to the actual physical data. These are often the data elements obtained after operations like error checking, data validation, proper formatting, and conversion are complete.

Most abstract input (MAI) data elements are recognized by starting from the physical inputs and traveling toward the outputs in the data flow diagram, until the data elements are reached that can no longer be considered incoming. The aim is to go as far as possible from the physical inputs, without losing the incoming nature of the data element. This process is performed for each input stream. Identifying the most abstract data items represents a value judgment on the part of the designer, but often the choice is obvious.

Similarly, we identify the *most abstract output data elements* (MAO) by starting from the outputs in the data flow diagram and traveling toward the inputs. These are the data elements that are most removed from the actual outputs but can still be considered outgoing. The MAO data elements may also be considered the logical output data items, and the transforms in the data flow diagram after these data items are basically to convert the logical output into a form in which the system is required to produce the output.

There will usually be some transforms left between the most abstract input and output data items. These *central transforms* perform the basic transformation for the system, taking the most abstract input and transforming it into the most abstract output. The purpose of having central transforms deal with the most abstract data items is that the modules implementing these transforms can concentrate on performing the transformation without being concerned with converting the data into proper format, validating the data, and so forth. It is worth noting that if a central transform has two outputs with a + between them, it often indicates the presence of a major decision in the transform (which can be shown in the structure chart).

Consider the data flow diagram shown in Figure 6.5. The arcs in the data flow diagram are the most abstract input and most abstract output.
The choice of the most abstract input is obvious. We start following the input. First, the input file is converted into a word list, which is essentially the input in a different form. The sorted word list is still basically the input, as it is still the same list, in a different order. This appears to be the most abstract input because the next data (i.e., count) is not just another form of the input data. The choice of the most abstract output is even more obvious; count is the natural choice (a data that is a form of input will not usually be a candidate for the most abstract output). Thus we have one central transform, count-number-of-different-words, which has one input and one output data item.

Consider now the data flow diagram of the automated teller shown in Figure 6.4. The two most abstract inputs are the dollar amount and the validated account number. The validated account number is the most abstract input, rather than the account number read in, as it is still the input—but with a guarantee that the account number is valid. The two abstract outputs are obvious. The abstract inputs and outputs are marked in the data flow diagram.

6.4.3 First-Level Factoring

Having identified the central transforms and the most abstract input and output data items, we are ready to identify some modules for the system. We first specify a main module, whose purpose is to invoke the subordinates. The main module is therefore a coordinate module. For each of the most abstract input data items, an immediate subordinate module to the main module is specified. Each of these modules is an input module, whose purpose is to deliver to the main module the most abstract data item for which it is created.

Similarly, for each most abstract output data item, a subordinate module that is an output module that accepts data from the main module is specified. Each of the arrows connecting these input and output subordinate modules are labeled with the respective abstract data item flowing in the proper direction.

Finally, for each central transform, a module subordinate to the main one is specified. These modules will be transform modules, whose purpose is to accept data from the main module, and then return the appropriate data back to the main module. The data items coming to a transform module from the main module are on the incoming arcs of the corresponding transform in the data flow diagram. The data items returned are on the outgoing arcs
of that transform. Note that here a module is created for a transform, while input/output modules are created for data items. The structure after the first-level factoring of the word-counting problem (its data flow diagram was given earlier) is shown in Figure 6.6.

In this example, there is one input module, which returns the sorted word list to the main module. The output module takes from the main module the value of the count. There is only one central transform in this example, and a module is drawn for that. Note that the data items traveling to and from this transformation module are the same as the data items going in and out of the central transform.

Let us examine the data flow diagram of the ATM. We have already seen that this has two most abstract inputs, two most abstract outputs, and two central transforms. Drawing a module for each of these, we get the structure chart shown in Figure 6.7.

As we can see, the first-level factoring is straightforward, after the most abstract input and output data items are identified in the data flow diagram. The main module is the overall control module, which will form the main program or procedure in the implementation of the design. It is a coordinate module that invokes the input modules to get the most abstract data items, passes these to the appropriate transform modules, and delivers the results of the transform modules to other transform modules until the most abstract data items are obtained. These are then passed to the output modules.
6.4.4 Factoring the Input, Output, and Transform Branches

The first-level factoring results in a very high-level structure, where each subordinate module has a lot of processing to do. To simplify these modules, they must be factored into subordinate modules that will distribute the work of a module. Each of the input, output, and transformation modules must be considered for factoring. Let us start with the input modules.

The purpose of an input module, as viewed by the main program, is to produce some data. To factor an input module, the transform in the data flow diagram that produced the data item is now treated as a central transform. The process performed for the first-level factoring is repeated here with this new central transform, with the input module being considered the main module. A subordinate input module is created for each input data stream coming into this new central transform, and a subordinate transform module is created for the new central transform. The new input modules now created can then be factored again, until the physical inputs are reached. Factoring of input modules will usually not yield any output subordinate modules.

The factoring of the input module get-sorted-list in the first-level structure is shown in Figure 6.8. The transform producing the input returned by this module (i.e., the sort transform) is treated as a central transform. Its input is the word list. Thus, in the first factoring we have an input module to get the list and a transform module to sort the list. The input module can be factored further, as the module needs to perform two functions, getting a word and then adding it to the list. Note that the looping arrow is used
Figure 6.8: Factoring the input module.

to show the iteration.

The factoring of the output modules is symmetrical to the factoring of the input modules. For an output module we look at the next transform to be applied to the output to bring it closer to the ultimate desired output. This now becomes the central transform, and an output module is created for each data stream going out of this transform. During the factoring of output modules, there will usually be no input modules. In our example, there is only one transform after the most abstract output, so this factoring need not be done.

If the data flow diagram of the problem is sufficiently detailed, factoring of the input and output modules is straightforward. However, there are no such rules for factoring the central transforms. The goal is to determine sub-transforms that will together compose the overall transform and then repeat the process for the newly found transforms, until we reach the atomic modules. Factoring the central transform is essentially an exercise in functional decomposition and will depend on the designers' experience and judgment.

One way to factor a transform module is to treat it as a problem in its own right and start with a data flow diagram for it. The inputs to the data flow diagram are the data coming into the module and the outputs are the data being returned by the module. Each transform in this data flow diagram represents a subtransform of this transform. The central transform can be factored by creating a subordinate transform module for each of the transforms in this data flow diagram. This process can be repeated for the
new transform modules that are created, until we reach atomic modules. The factoring of the central transform count-the-number-of-different-words is shown in Figure 6.9.

This was a relatively simple transform, and we did not need to draw the data flow diagram. To determine the number of words, we have to get a word repeatedly, determine if it is the same as the previous word (for a sorted list, this checking is sufficient to determine if the word is different from other words), and then count the word if it is different. For each of the three different functions, we have a subordinate module, and we get the structure shown in Figure 6.9.

It should be clear that the structure that is obtained depends a good deal on what are the most abstract inputs and most abstract outputs. And as mentioned earlier, determining the most abstract inputs and outputs requires making a judgment. However, if the judgment is different, though the structure changes, it is not affected dramatically. The net effect is that a bubble that appears as a transform module at one level may appear as a transform module at another level. For example, suppose in the word-counting problem we make a judgment that word-list is another form of the basic input but sorted-word-list is not. If we use word-list as the most abstract input, the net result is that the transform module corresponding to the sort bubble shows up as a transform module one level above. That is, now it is a central transform (i.e., subordinate to the main module) rather than a subordinate to the input module “get-sorted-word-list.” So, the SDM has the desired property that it is not very sensitive to some variations in the identification of the most abstract input and most abstract output.
6.4.5 Design Heuristics

The design steps mentioned earlier do not reduce the design process to a series of steps that can be followed blindly. The strategy requires the designer to exercise sound judgment and common sense. The basic objective is to make the program structure reflect the problem as closely as possible. With this in mind the structure obtained by the methodology described earlier should be treated as an initial structure, which may need to be modified. Here we mention some heuristics that can be used to modify the structure, if necessary. Keep in mind that these are merely pointers to help the designer decide how the structure can be modified. The designer is still the final judge of whether a particular heuristic is useful for a particular application or not.

Module size is often considered an indication of module complexity. In terms of the structure of the system, modules that are very large may not be implementing a single function and can therefore be broken into many modules, each implementing a different function. On the other hand, modules that are too small may not require any additional identity and can be combined with other modules.

However, the decision to split a module or combine different modules should not be based on size alone. Cohesion and coupling of modules should be the primary guiding factors. A module should be split into separate modules only if the cohesion of the original module was low, the resulting modules have a higher degree of cohesion, and the coupling between modules does not increase. Similarly, two or more modules should be combined only if the resulting module has a high degree of cohesion and the coupling of the resulting module is not greater than the coupling of the submodules. Furthermore, a module usually should not be split or combined with another module if it is subordinate to many different modules. As a rule of thumb, the designer should take a hard look at modules that will be larger than about 100 lines of source code or will be less than a couple of lines.

Another parameter that can be considered while "fine-tuning" the structure is the fan-in and fan-out of modules. Fan-in of a module is the number of arrows coming in the module, indicating the number of superordinates of a module. Fan-out of a module is the number of arrows going out of that module, indicating the number of subordinates of the module. A very high fan-out is not very desirable, as it means that the module has to control and coordinate too many modules and may therefore be too complex. Fan-out can be reduced by creating a subordinate and making many of the
current subordinates subordinate to the newly created module. In general the fan-out should not be increased above five or six.

Whenever possible, the fan-in should be maximized. Of course, this should not be obtained at the cost of increasing the coupling or decreasing the cohesion of modules. For example, implementing different functions into a single module, simply to increase the fan-in, is not a good idea. Fan-in can often be increased by separating out common functions from different modules and creating a module to implement that function.

Another important factor that should be considered is the correlation of the scope of effect and scope of control. The scope of effect of a decision (in a module) is the collection of all the modules that contain any processing that is conditional on that decision or whose invocation is dependent on the outcome of the decision. The scope of control of a module is the module itself and all its subordinates (not just the immediate subordinates). The system is usually simpler when the scope of effect of a decision is a subset of the scope of control of the module in which the decision is located. Ideally, the scope of effect should be limited to the modules that are immediate subordinates of the module in which the decision is located. Violation of this rule of thumb often results in more coupling between modules.

There are some methods that a designer can use to ensure that the scope of effect of a decision is within the scope of control of the module. The decision can be removed from the module and “moved up” in the structure. Alternatively, modules that are in the scope of effect but are not in the scope of control can be moved down the hierarchy so that they fall within the scope of control.

6.4.6 Transaction Analysis

The structured design technique discussed earlier is called transform analysis, where most of the transforms in the data flow diagram have a few inputs and a few outputs. There are situations where a transform splits an input stream into many different substreams, with a different sequence of transforms specified for the different substreams. For example, this is the case with systems where there are many different sets of possible actions and the actions to be performed depend on the input command specified. In such situations the transform analysis can be supplemented by transaction analysis, and the detailed data flow diagram of the transform splitting the input may look like the DFD shown in Figure 6.10.

The module splitting the input is called the transaction center; it need
not be a central transform and may occur on either the input branch or
the output branch of the data flow diagram of the system. One of the
standard ways to convert a data flow diagram of the form shown in Figure
6.10 into a structure chart is to have an input module that gets the analyzed
transaction and a dispatch module that invokes the modules for the different
transactions. This structure is shown in Figure 6.11.

For smaller systems the analysis and the dispatching can be done in
the transaction center module itself, giving rise to a flatter structure. For
designing systems that require transaction analysis, start with a data flow diagram, as in transform analysis, and identify the transform centers. Factor the data flow diagram, as is done in transform analysis. For the modules corresponding to the transform centers, draw the detailed data flow diagram, which will be of the form shown in Figure 6.11. Choose one of the transaction-centered organizations, either one with a separate dispatch and input module or one with all combined in one module. Specify one subordinate module for each transaction. Temptations to combine many similar transactions into one module should be avoided, as it would result in a logically cohesive module. Then each transaction module should be factored, as is done in transform analysis. There are usually many distinct actions that need to be performed for a transaction; they are often specified in the requirements for each transaction. In such cases one subordinate module to the transaction module should be created for each action. Further factoring of action modules into many detailed action modules may be needed. In many transaction-oriented systems, there is a lot of commonality of actions among the different transactions. This commonality should be exploited by sharing the modules at either the action level or the detailed action level.

6.4.7 Discussion

No design methodology reduces design to a series of steps that can be mechanically executed. All design methodologies are, at best, a set of guidelines that, if applied, will most likely give a design that will satisfy the design objectives. The basic objective is to produce a design that is modular and simple. One way to achieve modularity is to have a design that has highly cohesive modules with low coupling between different modules. In other words, the basic objective of the design activity using a function-oriented approach is to create an architecture, that, if implemented, will satisfy the SRS, and that contains cohesive modules that have low coupling with others. Structured design methodology is an approach for creating a design that is likely to satisfy this objective. Now that we have studied the methodology, let us see how it actually achieves this goal.

The basic principle behind the SDM, as with most other methodologies, is problem partitioning, in which the problem is partitioned into subproblems that can be solved separately. In SDM, at the very basic level, this is done by partitioning the system into subsystems that deal with input, subsystems that deal with output, and subsystems that deal with data transformation.

The rationale behind this partitioning is that in many systems, partic-
ularly data processing systems, a good part of the system code deals with managing the inputs and outputs. The components dealing with inputs have to deal with issues of screens, reading data, formats, errors, exceptions, completeness of information, structure of the information, etc. Similarly, the modules dealing with output have to prepare the output in presentation formats, make charts, produce reports, etc. Hence, for many systems, it is indeed the case that a good part of the software has to deal with inputs and outputs. The actual transformation in the system is frequently not very complex—it is dealing with data and getting it in proper form for performing the transformation or producing the output in the desired form that requires considerable processing.

Structured design methodology clearly separates the system at the very top level into various subsystems, one for managing each major input, one for managing each major output, and one for each major transformation. The modules performing the transformation deal with data at an abstract level, that is, in the form that is most convenient for processing. Due to this, these modules can focus on the conceptual problem of how to perform the transformation without bothering with how to obtain “clean” inputs or how to “present” the output. And these subsystems are quite independent of each other, interacting only through the main module. Hence, this partitioning leads to independent subsystems that do not interact directly, and hence can be designed and developed separately.

This partitioning is at the heart of SDM. In the SDM itself, this partitioning is obtained by starting with a data flow diagram. However, the basic idea of the SDM can be effectively used even if one wants to go directly to the first structure (without going through a DFD).

Besides this central idea, another basic idea behind the SDM is that processing of an input subsystem should be done in a progressive manner, starting from the raw input and progressively applying transformations to eventually reach the most abstract input level (what this input subsystem has to produce). Similar is the case with the structure for the subsystems dealing with outputs. The basic idea here is to separate the different transformations performed on the input before it is in a form ready to be “consumed.” And if the SDM is followed carefully, this leads to a “thin and tall” tree as a structure for the input or output subsystem. For example, if an input goes through a series of bubbles in the DFD before it is considered most abstract, the structure for this will be a tree with each node having two subordinates—one obtaining the input data at its level of abstraction and the other a transform module that is used to transform the data to the next
abstract level (which is passed to the superordinate). Similar effect can also
be obtained by the main input module having one input module and then a
series of transform modules, each performing one transform. In other words,
the basic idea in SDM for processing an input is to partition the processing
of an input into a series of transforms. As long as this approach is followed,
it is not terribly important how the structure for the input subsystem is
obtained.

These ideas that the methodology uses to partition the problem into
smaller modules lead to a structure in which different modules can be solved
separately and the connections between modules are minimized (i.e., the
coupling is reduced)—most connections between modules go through some
coordinate modules. These ideas of structuring are sound and lead to a
modular structure. It is important that these fundamental ideas behind the
SDM be kept in mind when using this approach. It may not be so impor­
tant to follow SDM down to the smallest detail. This is how experienced
designers use most methodologies; the detailed steps of the methodology are
not necessarily followed, but the philosophy is. Many experienced design­
ers do not start with a detailed DFD when using the SDM; they prefer to
work directly with the structure or with a very high-level DFD. But they
do use these principles when creating the structure. Such an approach is
recommended only when one has some experience with the SDM.

6.5 Verification

The output of the system design phase, like the output of other phases in the
development process, should be verified before proceeding with the activities
of the next phase. If the design is expressed in some formal notation for
which analysis tools are available, then through tools it can be checked for
internal consistency (e.g., those modules used by another are defined, the
interface of a module is consistent with the way others use it, data usage is
consistent with declaration, etc.) If the design is not specified in a formal,
executable language, it cannot be processed through tools, and other means
for verification have to be used. The most common approach for verification
is design review or inspections. We discuss this approach here.

The purpose of design reviews is to ensure that the design satisfies the
requirements and is of “good quality.” If errors are made during the design
process, they will ultimately reflect themselves in the code and the final
system. As the cost of removing faults caused by errors that occur during
design increases with the delay in detecting the errors, it is best if design errors are detected early, before they manifest themselves in the system. Detecting errors in design is the purpose of design reviews.

The system design review process is similar to the inspection process, in that a group of people get together to discuss the design with the aim of revealing design errors or undesirable properties. The review group must include a member of both the system design team and the detailed design team, the author of the requirements document, the author responsible for maintaining the design document, and an independent software quality engineer. As with any review, it should be kept in mind that the aim of the meeting is to uncover design errors not to try to fix them; fixing is done later.

The number of ways in which errors can come in a design is limited only by the creativity of the designer. However, there are some forms of errors that are more often observed. Here we mention some of these [52]. Perhaps the most significant design error is omission or misinterpretation of specified requirements. Clearly, if the system designer has misinterpreted or not accounted for some requirement it will be reflected later as a fault in the system. Sometimes, this design error is caused by ambiguities in the requirements.

There are some other quality factors that are not strictly design errors but that have implications on the reliability and maintainability of the system. An example of this is weak modularity (that is, weak cohesion and/or strong coupling). During reviews, elements of design that are not conducive to modification and expansion or elements that fail to conform to design standards should also be considered "errors."

**A Sample Checklist:** The use of checklists can be extremely useful for any review. The checklist 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 problem-specific errors. Here we list a few general items that can be used to construct a checklist for a design review [52]:

- Is each of the functional requirements taken into account?
- Are there analyses to demonstrate that performance requirements can be met?
- Are all assumptions explicitly stated, and are they acceptable?
6.6. METRICS

- Are there any limitations or constraints on the design beyond those in the requirements?
- Are external specifications of each module completely specified?
- Have exceptional conditions been handled?
- Are all the data formats consistent with the requirements?
- Are the operator and user interfaces properly addressed?
- Is the design modular, and does it conform to local standards?
- Are the sizes of data structures estimated? Are provisions made to guard against overflow?

6.6 Metrics

We have already seen that the basic purpose of metrics is to provide quantitative data to help monitor the project. Here we discuss some of the metrics that can be extracted from a design and that could be useful for evaluating the design. We do not discuss the standard metrics of effort or defect that are collected (as per the project plan) for project monitoring.

Size is always a product metric of interest, as size is the single most influential factor deciding the cost of the project. As the actual size of the project is known only when the project ends, at early stages the project size is only an estimate. As we saw in Figure 5.1, our ability to estimate size becomes more accurate as development proceeds. Hence, after design, size (and cost) re-estimation are typically done by project management. After design, as all the modules in the system and major data structures are known, the size of the final system can be estimated quite accurately.

For estimating the size, the total number of modules is an important metric. This can be easily obtained from the design. By using an average size of a module, from this metric the final size in LOC can be estimated. Alternatively, the size of each module can be estimated, and then the total size of the system will be estimated as the sum of all the estimates. As a module is a small, clearly specified programming unit, estimating the size of a module is relatively easy.

Another metric of interest is complexity, as one of our goals is to strive for simplicity and ease of understanding. A possible use of complexity metrics at design time is to improve the design by reducing the complexity of the
modules that have been found to be most complex. This will directly improve the testability and maintainability. If the complexity cannot be reduced because it is inherent in the problem, complexity metrics can be used to highlight the more complex modules. As complex modules are often more error-prone, this feedback can be used by project management to ensure that strict quality assurance is performed on these modules as they evolve. Overall, complexity metrics are of great interest at design time and they can be used to evaluate the quality of design, improve the design, and improve quality assurance of the project. We will describe some of the metrics that have been proposed to quantify the complexity of design.

6.6.1 Network Metrics

Network metrics for design focus on the structure chart (mostly the call graph component of the structure chart) and define some metrics of how “good” the structure or network is in an effort to quantify the complexity of the call graph. As coupling of a module increases if it is called by more modules, a good structure is considered one that has exactly one caller. That is, the call graph structure is simplest if it is a pure tree. The more the structure chart deviates from a tree, the more complex the system. Deviation of the tree is then defined as the graph impurity of the design [153]. Graph impurity can be defined as

\[ \text{Graph impurity} = n - e - 1 \]

where \( n \) is the number of nodes in the structure chart and \( e \) is the number of edges. As in a pure tree the total number of nodes is one more than the number of edges, the graph impurity for a tree is 0. Each time a module has a fan-in of more than one, the graph impurity increases. The major drawback of this approach is that it ignores the common use of some routines like library or support routines. An approach to handle this is not to consider the lowest-level nodes for graph impurity because most often the lowest-level modules are the ones that are used by many different modules, particularly if the structure chart was factored. Library routines are also at the lowest level of the structure chart (even if they have a structure of their own, it does not show in the structure chart of the application using the routine).

Other network metrics have also been defined. For most of these metrics, significant correlations with properties of interest have not been established. Hence, their use is limited to getting some idea about the structure of the design.
6.6. METRICS

6.6.2 Stability Metrics

We know that maintainability of software is a highly desired quality attribute. Maintenance activity is hard and error-prone as changes in one module require changes in other modules to maintain consistency, which require further changes, and so on. It is clearly desirable to minimize this ripple effect of performing a change, which is largely determined by the structure of the software. Stability of a design is a metric that tries to quantify the resistance of a design to the potential ripple effects that are caused by changes in modules [151]. The higher the stability of a program design, the better the maintainability of the program. Here we define the stability metric as defined in [151].

At the lowest level, stability is defined for a module. From this, the stability of the whole system design can be obtained. The aim is to define a measure so that the higher the measure the less the ripple effect on other modules that in some way are related to this module. The modules that can be affected by change in a module are the modules that invoke the module or share global data (or files) with the module. Any other module will clearly not be affected by change in a module. The potential ripple effect is defined as the total number of assumptions made by other modules regarding the module being changed. Hence, counting the number of assumptions made by other modules is central to determining the stability of a module.

As at design time only the interfaces of modules are known and not their internals, for calculating design stability only the assumptions made about the interfaces need be considered. The interface of a module consists of all elements through which this module can be affected by other modules, i.e., through which this module can be coupled with other modules. Hence, it consists of the parameters of the modules and the global data the module uses. Once the interface is identified, the structure of each element of the interface is examined to determine all the minimal entities in this element for which assumptions can be made. The minimal entities generally are the constituents of the interface element. For example, a record is broken into its respective fields as a calling module can make assumptions about a particular field.

For each minimal entity at least two categories of assumptions can be made—about the type of the entity and about the value of the entity. (The assumption about the type is typically checked by a compiler if the programming language supports strong typing.) Each minimal entity in the interface is considered as contributing one assumption in each category. A structured
6. FUNCTION-ORIENTED DESIGN

type is considered as contributing one more assumption about its structure in addition to the assumptions its minimal elements contribute. The procedure for determining the stability of a module \( x \) and the stability of the program can be broken into a series of steps [151]:

**Step 1:** From the design, analyze the module \( x \) and all the modules that call \( x \) or share some file or data structure with \( x \), and obtain the following sets.

\[
J_x = \{\text{modules that invoke } x\}
\]
\[
J'_x = \{\text{modules invoked by } x\}
\]
\[
R_{xy} = \{\text{passed parameters returned from } x \text{ to } y, y \in J_x\}
\]
\[
R'_{xy} = \{\text{parameters passed from } x \text{ to } y, y \in J'_x\}
\]
\[
GR_x = \{\text{Global data referenced in } x\}
\]
\[
GD_x = \{\text{Global data defined in } x\}
\]

Note that determining \( GR_x \) and \( GD_x \) is not always possible when pointers and indirect referencing are used. In that case, a conservative estimate is to be used. From these, for each global data item \( i \), define the set \( G_i \) as

\[
G_i = \{x | i \in GR_x \cup GD_x\}.
\]

The set \( G_i \) represents the set of modules where the global data \( i \) is either referenced or defined. Where it is not possible to compute \( G \) accurately, the worst case should be taken.

**Step 2:** For each module \( x \), determine the number of assumptions made by a caller module \( y \) about elements in \( R_{xy} \) (parameters returned from module \( x \) to \( y \)) through these steps:

1. Initialize assumption count to 0.

2. If \( i \) is a structured data element, decompose it into base types, and increment the assumption count by 1; else consider \( i \) minimal.

3. Decompose base types, and if they are structured, increment the count by 1.

4. For each minimal entity \( i \), if module \( y \) makes some assumption about the value of \( i \), increment the count by 2; else increment by 1.
Let $TP_{xy}$ represent the total number of assumptions made by a module $y$ about parameters in $R_{xy}$.

*Step 3:* Determine $TP'_{xy}$, the total number of assumptions made by a module $y$ called by the module $x$ about elements in $R'_{xy}$ (parameters passed from module $x$ to $y$). The method for computation is the same as in the previous step.

*Step 4:* For each data element $i \in GD_x$ (i.e., the global data elements modified by the module $x$), determine the total number of assumptions made by other modules about $i$. These will be the modules other than $x$ that use or modify $i$, i.e., the set of modules to be considered is $\{G_i - \{x\}\}$. The counting method of step 2 is used. Let $TG_x$ be the total number of assumptions made by other modules about the elements in $GD_x$.

*Step 5:* For a module $x$, the design logical ripple effect (DLRE) is defined as:

$$DLRE_x = TG_x + \sum_{y \in J_x} TP_{xy} + \sum_{y \in J'_x} TP'_{xy}.$$  

$DLRE_x$ is the total number of assumptions made by other modules that interact with $x$ through either parameters or global data. The design stability (DS) of a module $x$ is then defined as

$$DS_x = 1/(1 + DLRE_x).$$

*Step 6:* The program design stability (PDS) is computed as

$$PDS = 1/(1 + \sum_x DLRE_x).$$

By following this sequence of steps, the design stability of each module and the overall program can be computed. The stability metric, in a sense, is trying to capture the notion of coupling of a module with other modules. The stability metrics can be used to compare alternative designs—the larger the stability, the more maintainable the program. It can also be used to identify modules that are not very stable and that are highly coupled with other modules with a potential of high ripple effect. Changes to these modules will not be easy, hence a redesign can be considered to enhance the stability. Only a limited validation has been done for this metric. Some validation has been given in [151], showing that if programming practices are followed which are generally recognized as enhancing maintainability, then higher program stability results.
Another stability metric was described in [121]. In this formulation, the effect of a change in a module $i$ on another module $j$ is represented as a probability. For the entire system, the effect of change is captured by the probability of change metrics $C$. An element $C[i, j]$ of the matrix represents the probability that a change in module $i$ will result in a change in module $j$. With this matrix the ripple effect of a change in a module can also be easily computed. This can then be used to model the stability of the system. The main problem with this metric is to estimate the elements of the matrix.

6.6.3 Information Flow Metrics

The network metrics of graph impurity had the basis that as the graph impurity increases, the coupling increases. However, it is not a very good approximation for coupling, as coupling of a module increases with the complexity of the interface and the total number of modules a module is coupled with, whether it is the caller or the callee. So, if we want a metric that is better at quantifying coupling between modules, it should handle these. The information flow metrics attempt to define the complexity in terms of the total information flowing through a module.

In one of the earliest work on information flow metrics [84, 85], the complexity of a module is considered as depending on the intramodule complexity and the intermodule complexity. The intramodule complexity is approximated by the size of the module in lines of code (which is actually the estimated size at design time). The intermodule complexity of a module depends on the total information flowing in the module ($inflow$) and the total information flowing out of the module ($outflow$). The inflow of a module is the total number of abstract data elements flowing in the module (i.e., whose values are used by the module), and the outflow is the total number of abstract data elements that are flowing out of the module (i.e., whose values are defined by this module and used by other modules). The module design complexity, $D_c$, is defined as

$$D_c = \text{size} \ast (\text{inflow} \ast \text{outflow})^2.$$

The term $(\text{inflow} \ast \text{outflow})$ refers to the total number of combinations of input source and output destination. This term is squared, as the interconnection between the modules is considered a more important factor (compared to the internal complexity) determining the complexity of a module. This is based on the common experience that the modules with more in-
6.6. METRICS

terconnections are harder to test or modify compared to other similar-size modules with fewer interconnections.

The metric defined earlier defines the complexity of a module purely in terms of the total amount of data flowing in and out of the module and the module size. A variant of this was proposed based on the hypothesis that the module complexity depends not only on the information flowing in and out, but also on the number of modules to or from which it is flowing. The module size is considered an insignificant factor, and complexity $D_c$ for a module is defined as [155]:

$$D_c = \text{fan\_in} \times \text{fan\_out} + \text{in\_flow} \times \text{out\_flow}$$

where fan\_in represents the number of modules that call this module and fan\_out is the number of modules this module calls.

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

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

One such method for highlighting the modules was suggested in [155]. Let \text{avg\_complexity} be the average complexity of the modules in the design being evaluated, and let \text{std\_deviation} be the standard deviation in the design complexity of the modules of the system. The proposed method classifies the modules in three categories: error-prone, complex, and normal. If $D_c$ is the complexity of a module, it can be classified as follows:
6. FUNCTION-ORIENTED DESIGN

Error-prone

If $D_c > \text{avg. complexity} + \text{std. deviation}$

Complex

If $\text{avg. complexity} < D_c < \text{avg. complexity} + \text{std. deviation}$

Normal

Otherwise

Note that this definition of error-prone and complex is independent of the metric definition used to compute the complexity of modules. With this approach, a design can be evaluated by itself, not for overall design quality, but to draw attention to the error-prone and complex modules. This information can then be used to redesign the system to reduce the complexity of these modules (which also results in overall complexity reduction). This approach has been found to be very effective in identifying error-prone modules [155]. In evaluations of some completed projects, it has been shown that error-prone and complex modules together highlight the modules in which most errors occurred [155]. This suggests that for a project, modules thus highlighted during design time point to modules that will be "hot spots" if the design is not improved by reducing their complexity. Another use of this is that even if the complexity of these modules is not reduced (perhaps because the complexity is intrinsic in the problem), identification of error-prone modules can help in quality assurance later; these modules can be required to undergo more rigorous quality assurance.

6.7 Summary

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

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

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

The structured design method is one of the best known methods for developing the design of a software system. This method creates a structure chart that can be used to implement the system. The goal is to produce a structure where the modules have minimum dependence on each other (low coupling) and a high level of cohesion. The basic methodology has four steps: (1) restate the problem as a data flow graph; (2) identify the most abstract input and output data elements; (3) perform first-level factoring, which is done by specifying an input module for each of the most abstract inputs, an output module for each of the most abstract outputs, and a transform module for each of the central transforms; and (4) factor each of the input, output, and transform modules.

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

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

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

Exercises

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

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

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

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

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

5. If some existing modules are to be re-used in building a new system, will you use a top-down or bottom-up approach? Why?

6. If a module has logical cohesion, what kind of coupling is this module likely to have with others?

7. What is the difference between a flow chart and a structure chart?

8. Draw the structure chart for the following program:

   ```
   main();
   { int x, y;
     x = 0; y = 0;
     a(); b(); }
   a()
   { x = x+y; y = y+5; }
   b()
   { x = x+5; y = y+x; a(); }
   ```
How would you modify this program to improve the modularity?

9. If a ‘+’ or a ‘*’ is present between two output streams from a transform in a data flow graph, state some specific property about the module for that transform.

10. Use the structured design methodology to produce a design for the following:

   (a) A system to convert ASCII to EBSDIC.
   (b) A system to analyze your diet when given your daily intake (and some data files about different types of food and recommended intakes).
   (c) A system to do student registration in the manner it is done at your college.
   (d) A system to manage the inventory at a hardware store.
   (e) A system for a drug store that will manage inventory, keep track of expiration dates, and track allergy records of patients to avoid issuing medicines that might be harmful.
   (f) A system that acts as a calculator with only basic arithmetic functions.

11. Is this statement true: “If we follow the structured design methodology (without applying any heuristics), the resulting structure will always have one transform module for each bubble in the data flow graph”? Explain your answer.

12. Given a structure with high fan-out, how would you convert it to a structure with a low fan-out?

13. Discuss some approaches on how you can use metrics to guide you in design to produce a design that is easy to modify.

14. Design an experiment to study whether the information flow metrics and stability metrics are correlated.

15. If you have all the metrics data available for design, how will you use this data? Specify your objectives, the metrics you will use, how you will interpret the value, and what possible actions you will take based on the interpretation.
Case Studies

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

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

Structured Design

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

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

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

The most abstract input and most abstract output are fairly obvious here. The “separated course schedule” is the most abstract input and the three outputs of the schedule transform are the most abstract outputs. There is only one central transform: schedule.
Figure 6.12: Data flow diagram for the case study.
First-Level Factoring: The first-level structure chart can easily be obtained and is shown in Figure 6.13. In the structure chart, instead of having one output module for each of the three outputs, as is shown in the data flow diagram, we have only one output module, which then invokes three output modules for the different outputs.

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

Factoring the Central Transform: Now the central transform has to be factored. According to the requirements, PG courses have to be given preference over UG courses, and the highest priority of each course must be satisfied. This means that the courses with no priority should be scheduled after the courses with priority. Hence, we have four major subordinate modules to the central transform: schedule PG courses with preferences, schedule UG courses with preferences, schedule PG courses with no preferences, and schedule UG courses with no preferences. The structure of the central transform is shown in Figure 6.15.
These can then be combined into a structure chart for the system. The overall structure chart is shown in Figure 6.16. This structure chart gives an overall view of the strategy for structuring the programs. Further details about each module will evolve during detailed design and coding.
Figure 6.16: Structure chart for the system.
Analysis Using Information Flow Metrics

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

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

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

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

OVERALL METRICS
6. FUNCTION-ORIENTED DESIGN

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

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

-----------------------------------------------
ERROR-PRONE MODULES
-----------------------------------------------
8) sched_ug_pref
   call in: 1  call out: 4  inflow: 5  outflow: 13  size: 100
   design complexity: 69

-----------------------------------------------
COMPLEX MODULES
-----------------------------------------------
5) validate_file2
   call in: 1  call out: 4  inflow: 4  outflow: 8  size: 100
   design complexity: 36

7) sched_pg_pref
   call in: 1  call out: 1  inflow: 1  outflow: 6  size: 75
   design complexity: 7

13) is_safe_allotment
   call in: 1  call out: 0  inflow: 3  outflow: 1  size: 80
   design complexity: 3

15) validate_classrooms
   call in: 1  call out: 5  inflow: 3  outflow: 7  size: 80
   design complexity: 26

16) validate_dept_courses
   call in: 1  call out: 3  inflow: 2  outflow: 5  size: 75
   design complexity: 13

17) validate_lec_times
   call in: 1  call out: 3  inflow: 2  outflow: 5  size: 70
   design complexity: 13

-----------------------------------------------

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

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

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

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