Software Design

Design phase of software development deals with transforming the customer requirements as described in the SRS document into a form implementable using a programming language. In order to be easily implementable in a conventional programming language, the following items must be designed during the design phase:

- Different modules required to implement the design solution.
- Control relationship among the identified modules, i.e. the call relationship (also known as the invocation relationship) among modules.
- Interface among different modules, i.e. details of the data items exchanged among different modules.
- Data structures of the individual modules.
- Algorithms required to implement the individual modules.

Thus the goal of the design phase is to take the SRS document as the input and to produce the above mentioned items at the completion stage of the design phase. A good software design is seldom arrived through a single step procedure but goes through a series of steps. However, we can broadly classify various design activities into two important parts:

- *Preliminary* (or *high-level*) design.
- *Detailed* design.

But the meaning and scope of these two stages may vary considerably from one methodology to another. We will assume in this text that during high-level design, different modules and the control relationships among them are identified and the interfaces among these modules are defined. The outcome of high-level design is called the *program structure* or *software architecture*. Many different notations are used to represent a high-level design. Usually a tree-like diagram called the *structure chart* is used to represent the control hierarchy in a high-level design. However, other notations such as Jackson diagrams [1975] or Warnier–Orr [1977, 1981] diagrams can

also be used. During detailed design, the data structure and the algorithms used by different modules are designed. The outcome of the detailed design is usually known as the *module specification* document.

A large number of software design techniques is available. We will provide a broad overview of these design approaches and discuss their important characteristics. Before discussing the different design approaches, let us discuss a fundamental question—that of how to distinguish between a good design and a bad design. In order to do this, we must be able to define a set of criteria which characterize a good software design. Because, unless we know what a good software design is, we cannot possibly design one. Also, as we will see, there is no unique way to design a system. Using the same design methodology, different engineers can arrive at totally different design solutions. So it is essential that we know some ways of distinguishing a good software design from a bad design.

4.1 WHAT IS A GOOD SOFTWARE DESIGN?

Judging the goodness of a design involves many subjective factors which depend on the particular application. For example, an embedded software may have to be implemented in a limited size of memory, and therefore one may have to sacrifice design comprehensibility in order to achieve compactness of code. Therefore, for embedded applications, factors like design comprehensibility may take a back seat while judging the goodness of design. In other words, a design solution which can be judged good for one application may not be considered good for another application. Not only the goodness of a design but also the notion of goodness of a design varies widely across software engineers and academicians. However, most researchers and software engineers agree that software design for general applications must have a few desirable characteristics. These are listed below:

- A good design should capture all the functionalities of the system correctly.
- It should be easily understandable.
- It should be efficient.
- It should be easily amenable to change, i.e. easily maintainable.

Understandability of a design is a major factor which is used to evaluate the goodness of a design, since a design that is easily understandable is also easy to maintain and change. Unless a design is easily understandable, it would require a tremendous effort to maintain it. It will be of interest to know that about 60% of the total

effort in the life cycle of a typical product is spent on maintenance. If the software is not easily understandable, the maintenance effort would increase manifold. In order to enhance the understandability of a design, it should have the following features:

- Use of consistent and meaningful names for various design components.
- Use of a cleanly decomposed set of modules.
- Neat arrangement of modules in a hierarchy, i.e. tree-like diagram.

Modular design is one of the fundamental principles of a good design. Decomposition of a problem into modules facilitates taking advantage of the *divide and conquer* principle. If different modules are almost independent of each other then each module can be understood separately, eventually reducing the complexity greatly. To understand why this is so, we know that it is very difficult to break a bunch of sticks tied together, but very easy to break the sticks individually. Let us see how we can further elaborate the idea of a clean decomposition of a problem into modules and their arrangement in a neat hierarchy.

Clean decomposition of a design problem into modules means that the modules in a software design should display *high cohesion* and *low coupling*. We will shortly introduce the concepts of cohesion and coupling and discuss what these terms mean.

Neat arrangement of modules in a hierarchy essentially means

- low fan-out, and
- abstraction

We will further elaborate these concepts in this chapter.

4.2 COHESION AND COUPLING

Most researchers and engineers agree that a good software design implies *clean* decomposition of a problem into modules, and the arrangement of these modules in a neat hierarchy. What do we really mean by *clean* decomposition of a problem into modules? The primary characteristics of clean decomposition are *high cohesion* and *low coupling*. Cohesion is a measure of the functional strength of a module whereas coupling of a module with another module is a measure of the degree of functional interdependence or interaction between the two modules.

A module having high cohesion and low coupling is said to be *functionally independent* of other modules. By the term *functional independence*, we mean that a cohesive module performs a single task or function. A functionally independent module has minimal

interaction with other modules. Functional independence is a key to good design primarily due to the following reasons:

- Functional independence reduces error propagation. Therefore, an error existing in one module does not directly affect other modules and also any error existing in other modules does not directly affect this module.
- Reuse of a module is possible, because each module performs some well-defined and precise function and the interface of the module with other modules is simple and minimal. Therefore, any such module can be easily taken out and reused in a different program.
- Complexity of the design is reduced, because different modules can be understood in isolation as modules are more or less independent of each other.

Even though no quantitative measures are available to determine the degree of cohesion and coupling, an understanding of different kinds of cohesion and coupling will give us some idea regarding the degree of cohesiveness of a module. Therefore, by examining the type of cohesiveness exhibited by a module, we can roughly tell whether it displays high or low cohesion.

4.2.1 Classification of Cohesiveness

The different classes of cohesiveness that can exist in a design are depicted in Fig. 4.1 and elaborated below:

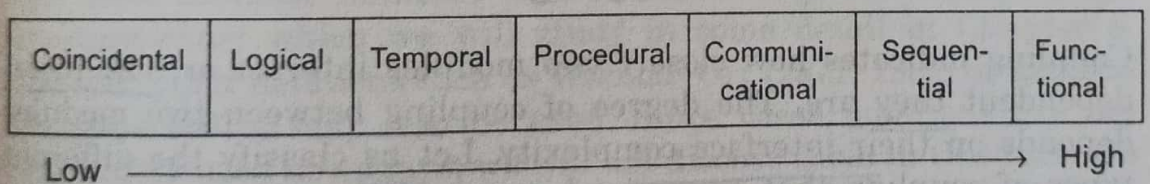

Fig. 4.1 Classification of cohesion.

Coincidental cohesion. A module is said to have coincidental cohesion if it performs a set of tasks that are related to each other very loosely. In this case, the module contains a random collection of functions. In this case it is likely that the functions have been put in the module, out of pure coincidence, without any thought or design.

Logical cohesion. A module is said to be logically cohesive, if all the elements of the module perform similar operations, e.g. error handling, data input, data output, etc. An example of logical cohesion is the case where a set of print functions generating an output report have been arranged into a single module.

Temporal cohesion. When a module contains tasks that are related by the fact that all the tasks must be executed in the same time-

span, the module is said to exhibit temporal cohesion. The set of functions responsible for initialization, start-up, shut-down of the same process, etc. exhibit temporal cohesion.

Procedural cohesion. A module is said to possess procedural cohesion, if the set of functions of the module are all part of a procedure (algorithm) in which a certain sequence of steps has to be carried out in a certain order for achieving an objective, e.g. the algorithm for decoding a message.

Communicational cohesion. A module is said to have communicational cohesion if all the functions of the module refer to or update the same data structure, e.g. the set of functions defined on an array or a stack.

Sequential cohesion. A module is said to possess sequential cohesion, if the elements of the module form different parts of a sequence, where output from one element of the sequence is input to the next element of the sequence.

Functional cohesion. Functional cohesion is said to exist if different elements of a module cooperate to achieve a single function, e.g. managing an employee's payroll. When a module displays functional cohesion, and if we are asked to describe what the module does we can describe it using a single sentence.

4.2.2 Classification of Coupling

Coupling indicates how closely two modules interact or how interdependent they are. The degree of coupling between two modules depends on their interface complexity. Let us classify the different types of coupling that can exist between different modules. Thus, even if there are no techniques to precisely and quantitatively estimate the coupling between two modules, classification of the different types of coupling will at least help us to estimate the degree of coupling between two modules. Five types of coupling can occur between any two modules.

Data coupling. Two modules are data coupled, if they communicate via an elementary data item which is passed as a parameter between the two, e.g. an integer, a float, a character, etc. This data item should be problem related and not used for the control purpose.

Stamp coupling. Two modules are stamp coupled, if they communicate via a composite data item such as a record in PASCAL or a structure in C.

Control coupling. Control coupling exists between two modules, if data from one module is used to direct the order of execution of instructions in another. An example of control coupling is a flag set in one module and tested in another module.

Common coupling. Two modules are common coupled, if they share some global data areas.

Content coupling. Content coupling exists between two modules, if their code is shared, e.g. a branch from one module into another module.

The degree of coupling is in ascending order from data coupling to content coupling. The different types of coupling are shown in Fig. 4.2.

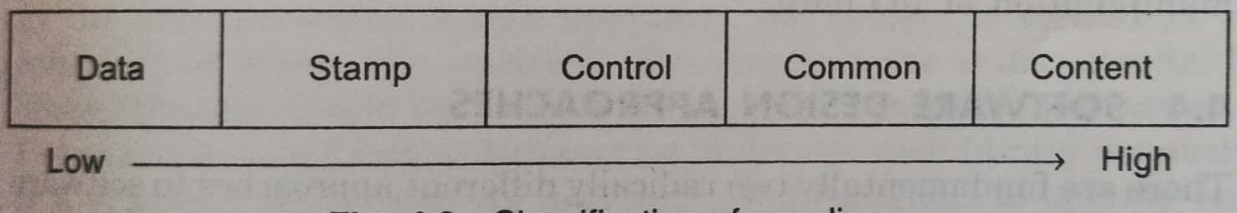

Fig. 4.2 Classification of coupling.