CHAPTER 5

SOFTWARE DESIGN

During the software design phase, the design document is produced, based on the customer requirements as documented in the SRS document. We can state the main objectives of the design phase, in other words, as follows.

> The activities carried out during the design phase (called the design process) transform the SRS document into the design document.

This view of a design process has been shown schematically in Figure 5.1. The design document produced at the end of the design phase should be implementable using a programming language in the subsequent (coding) phase.

5.1 OUTCOME OF A DESIGN PROCESS

The following items are designed and documented during the design phase.

Different modules required

The different modules in the solution should be clearly identified. Each module is a collection of functions and the data shared by the functions of the module. Each module should accomplish some well-defined task out of the overall responsibility of the software. Each module should be named according to the task it performs. For example, in an academic automation software, the module consisting of the functions and data necessary to accomplish the task of registration of the students should be named `handle student registration`.

Control relationships among modules

A control relationship between two modules essentially arises due to *function calls* between the two modules. The control relationships existing among various modules should be identified in the design document.

Interfaces among different modules

The interfaces between two modules identifies the exact data items exchanged between the two modules when a one module invokes a function of the other module.

Data structures of the individual modules

Each module normally stores some data that the functions of the module need to share to accomplish the overall responsibility of the module. Suitable data structures for the data to be stored in a module needs to be properly designed and documented.

Algorithms required to implement the individual modules

Each function in a module usually performs some processing activity. The algorithms required to accomplish the processing activities of various modules need to be carefully designed and documented with due considerations given to the accuracy of the results, space and time complexities.

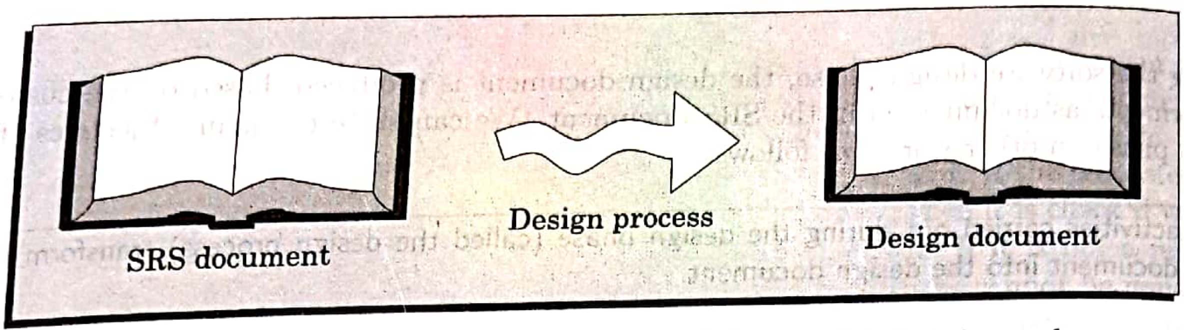


Figure 5.1: The design process transforms the SRS document into a design document.

Starting with the SRS document (as shown in Figure 5.1), the design documents are produced through iterations over a series of steps that we are going to discuss in this chapter and the subsequent three chapters. The design documents are reviewed by the members of the development team to ensure that the design solution conforms to the requirements specification.

5.1.1 Classification of Design Activities

A good software design is seldom arrived by using a single step procedure but rather by iterating over a series of steps called the **design activities**. Let us first classify the design activities before discussing them in detail. Depending on the order in which various design activities are performed, we can broadly classify them into two important stages: preliminary (or high-level) design and detailed design.

The meaning and scope of these two stages can vary considerably from one design methodology to another. However, for the traditional function-oriented design approach, it is possible to define the objective of the high-level design as follows:

> Through high-level design, a problem is decomposed into a set of modules, the control relationships among various modules identified, and also the interfaces among various modules are identified.

The outcome of high-level design is called the **program structure** or the **software architecture**. High-level design is a crucial step in the overall design of a software. After the high-level design is complete, we have managed to decompose the problem into many small functionally independent modules that are cohesive, have low coupling among themselves, and are arranged in a hierarchy. Many different types of notations have been used to represent a high-level design. A notation that is widely being used for procedural development is a tree-like diagram called the **structure chart**. Another popular design representation techniques called **UML** that is being used to document object-oriented design, involves developing several types of diagrams to document the object-oriented design of a systems. Though other notations such as Jackson diagram [1975] or Warnier-Orr [1977, 1981] diagram are available to document a software design, we confine our attention in this text to structure charts and UML diagrams only.

Once the high-level design is complete, detailed design is undertaken.

During detailed design, each module is examined carefully to design its data structures and the algorithms.

The outcome of the detailed design stage is usually documented documented in the form of a *module specification* (MSPEC) document. Since after the high-level design is complete, the problem is decomposed into small modules that can be under easy grasp of good programmers, in this text we shall not delve into how to write the MSPECs.

5.1.3 Analysis versus Design

Analysis and design activities differ in goal and scope.

> The goal of any analysis technique is to elaborate the customer requirements through careful thinking and at the same time consciously avoiding making any decisions regarding the exact way the system is to be implemented.

The analysis model is usually documented using some graphical formalism. In case of the function-oriented approach that we are going to discuss, the analysis model would be documented using data flow diagrams (DFDs), whereas the design would be documented using structure chart. On the other hand, for object-oriented approach, both the design model and the analysis model will be documented using UML (Unified Modelling Language). The analysis model would normally be very difficult to implement using a programming language.

The design model is obtained from the analysis model through transformations over a series of steps. In contrast to the analysis model, the design model reflects several decisions taken regarding the exact way system is to be implemented. The design model should be detailed enough to be easily implementable using a programming language.

5.2 HOW CAN WE CHARACTERIZE A GOOD SOFTWARE DESIGN?

Coming up with an accurate characterization of a good software design that would hold across diverse problem domains is certainly not easy. In fact, the definition of a good software design can vary depending on the exact application being designed. For example, memory size used up by a program may be an important issue to characterize a good solution for embedded software development—since embedded applications are often required to work under severely limited memory sizes due to cost, space, or power consumption considerations. For embedded applications, factors such as design comprehensibility may take a back seat while judging the goodness of design. Thus, for embedded applications, one may sacrifice design comprehensibility to achieve code compactness. Similarly, it is not usually true that a criterion that is crucial for some application, needs to be almost completely ignored for another application. It is therefore clear that the criteria used to judge a design solution can vary widely across different types of applications. Not only do the criteria used to judge a design solution depend on the exact application being designed, but to make the matter worse, there is no general agreement among software engineers and researchers on the exact criteria to use for judging a design even for a specific category of application. However, most researchers and software engineers agree on a few desirable characteristics that every good software design for general applications must possess. These characteristics are listed below:

Correctness

A good design should first of all be correct. That is, it should correctly implement all the functionalities of the system.

Understandability

A good design should be easily understandable. Unless a design solution is easily understandable, it would be difficult to implement and maintain it.

Efficiency

A good design solution should adequately address resource, time, and cost optimization issues.

Maintainability

A good design should be easy to change. This is an important requirement, since change requests usually keep coming from the customer even after product release.

Understandability of a design—a major concern

While performing the design of a certain problem, assume that we have arrived at a large number of design solutions and need to choose the best one. Obviously all incorrect designs have to be discarded first. Out of the correct design solutions, how can we identify the best one?

> Given that we are choosing from only correct design solutions, understandability of a design solution is possibly the most important issue to be considered while judging the goodness of a design.

2.5.4 Problem Partitioning

- When solving a small problem, the entire problem can be tackled at once. For solving larger/bigger problems, the basic principle is the time-tested principle of "divide and conquer." This principle suggests dividing into smaller pieces, so that each piece can be conquered separately.
- When the system design follows a hierarchical nature, the structure of the program can be partitioned either horizontally or vertically.
- For software design, therefore, the goal is to divide the problem into manageably small pieces that can be solved separately.
- 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.
- 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.
- Problem partitioning can be divided into two categories namely, Horizontal partitioning and Vertical partitioning.
- In **horizontal partitioning**, the control modules are used to communicate between functions and execute the functions. In **vertical partitioning**, the functionality is distributed among the modules in a top-down manner.
- Horizontal partitioning defines separate branches of modular hierarchy for each major program function.
- Fig. 2.43 shows horizontal partitioning. The simplest approach to horizontal partitioning defines three partitions are input, data transformation (often called processing), and output.
- Partitioning their architecture horizontally provides a number of distinct benefits:
  1. Software that is easier to extend.
  2. Software that is easier to test.
  3. Propagation of fewer side effects.
  4. Software that is easier to maintain.

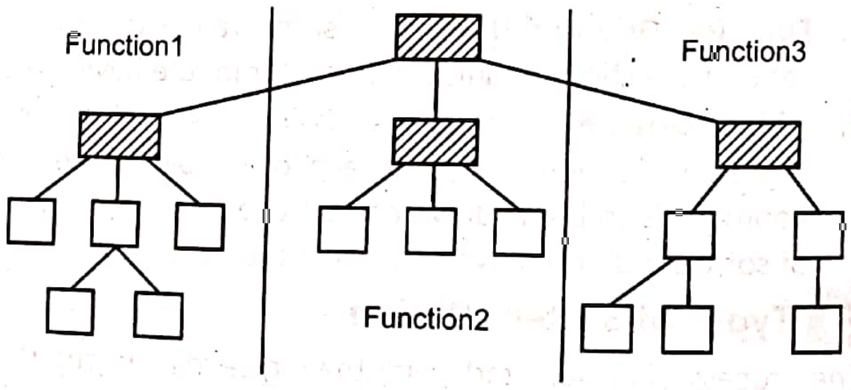


Fig. 2.43: Horizontal Partitioning

- Conversely, horizontal partitioning often causes more data to be passed across module interfaces and can complicate the overall control of program flow.
- Fig. 2.44 shows vertical partitioning. Vertical partitioning, often called factoring, suggests that control and work should be distributed from top-down in the program structure.

- Top-level modules should perform control functions and do actual processing work.
- Modules that reside low in the structure should be the workers, performing all input, compilation, and output tasks.

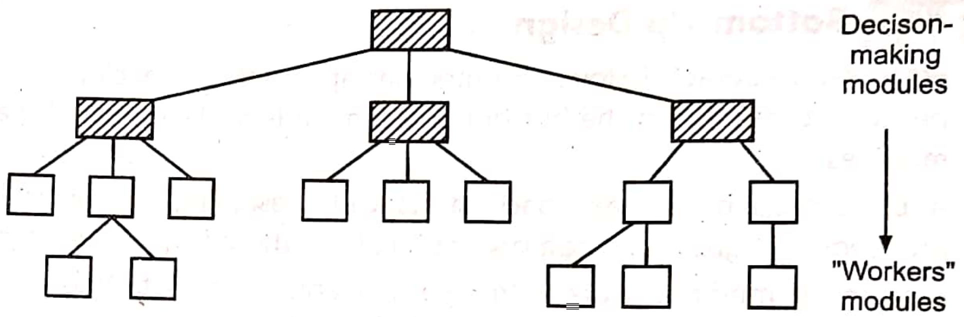


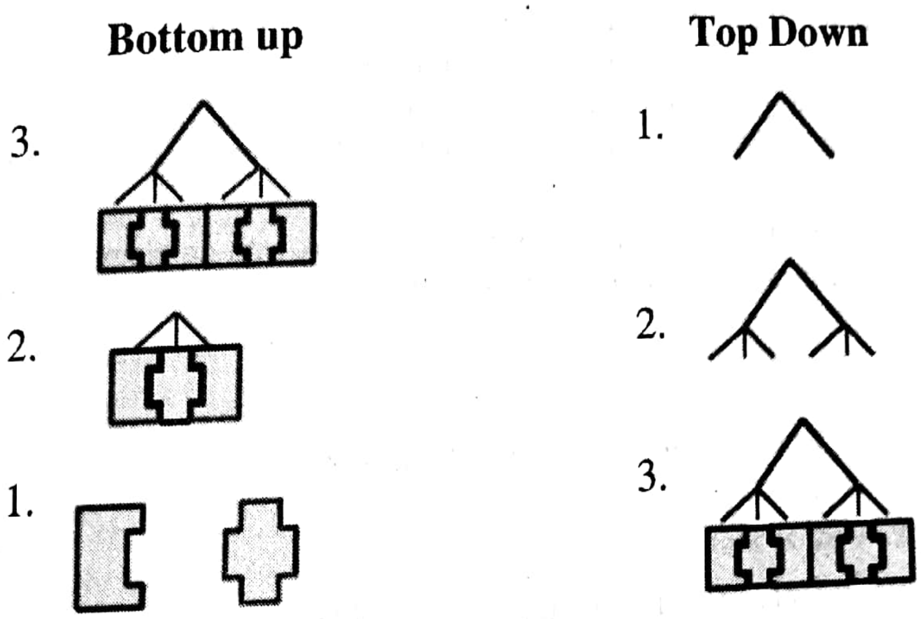
Fig. 2.44: Vertical Partitioning

- Certification standards ensure that only quality-tested product information is released to downstream users. A reliable system for releasing data ensures that assembly diagrams, for instance, are complete and correct.
- Management standards provide guidance for strategic use of design quality data. The organization's other systems, such as MRP and enterprise resource planning, are undermined when they receive poor quality design data.

3.11. SOFTWARE DESIGN STRATEGY

There are mainly two strategies used for software design:

- Top – Down Design
- Bottom –up Design



3.11.1. Top – Down Design

It is informal design approach, where breaking the problem into smaller problems. The design activity must begin with the analysis of the requirements definition and should not consider implementation details at first.

Top – Down Design has following objectives:

- To Systematize the design process
- To produce a modular program design
- To provide a framework in which problem solving can more effectively proceed

In this approach, software project is decomposed into subprojects or problem or task are decomposed into sub – task, sub – task into sub – sub – task, process is continue, until task become so simple that an algorithm can be formulated as a solution.

Advantages of top-down Programming

- Separating the low level work from the higher level objects leads to a modular design.
- Modular design means development can be self contained.
- Having "skeleton" code illustrates clearly how low level modules integrate.
- Less operations errors (To reduce errors , because each module have to be processed separately, So programmers get large amount of time for processing).

- Much less time consuming (Each programmer is only involved in a part of the big project).
- Very optimized way of processing. (Each programmer has to apply their own knowledge and experience to their parts (modules), so the project will become an optimized one)
- Easy to maintain (If an error occurs in the output, it is easy to identify the errors generated from which module of the entire program)

Disadvantages of top-down Programming

- Functionality either needs to be inserted into low level objects by making them return "canned answers" — manually constructed objects, similar to what you'd specify if you were mocking them in a test, or otherwise functionality will be lacking until development of low level objects is complete.

3.11.2. Bottom-up Approach

In a bottom-up approach the individual base elements of the system are first specified in great detail. These elements are then linked together to form larger subsystems, which then in turn are linked, sometimes in many levels, until a complete top-level system is formed. This strategy often resembles a "seed" model, whereby the beginnings are small, but eventually grow in complexity and completeness.

In bottom – up design, the approach is to start "at bottom", with problems that you already know how to solve. Here you can work upwards towards a solution to the overall problems.

Advantage:

- Economically is usefule, because here solution can be reused.
- it can be used to hide the low-level details of implementation and be merged with a top-down technique.

Disadvantage:

- It is not so closely related to the structure of the problem
- Its focus is not on specific requirements and thus its results may not fit a given problem.