Chapter 9 *Design*

This chapter will discuss the following concepts:

9.1 Software Design
9.2 Design Principles
9.3 Design Concepts
9.4 Object-Oriented Design

9.1 Software Design

The designer's goal is to produce a model or representation of an entity that will later be built. Each of the elements of the analysis model provides information that is necessary to create the four design models required for a complete specification of design. the design task produces the following four design models:

1- **The data design:** transforms the information domain model created during analysis into the data structures that will be required to implement the software.
2- **The architectural design:** defines the relationship between major structural elements of the software, the "design patterns" that can be used to achieve the requirements that have been defined for the system, and the constraints that affect the way in which architectural design patterns can be applied
3- **The interface design:** describes how the software communicates within itself, with systems that interoperate with it, and with humans who use it.
4- **The component-level design:** transforms structural elements of the software architecture into a procedural description of software components.

9.2 Design Principles

Basic design principles enable the software engineer to navigate the design process. A set of principles for software design have been suggested, adapted and extended in the following list:

1- The design process should not suffer from "tunnel vision."
2- The design should be traceable to the analysis model.

3- The design should "minimize the distance" between the software and the problem as it exists in the real world.
4- The design should exhibit uniformity and integration.
5- The design should be structured to accommodate change.
6- Design is not coding, coding is not design.
7- The design should be assessed for quality as it is being created, not after the fact.
8- The design should be reviewed to minimize semantic errors.

9.3 Design Concepts

A set of fundamental software design concepts has evolved over the past four decades. Each provides the software designer with a foundation from which more complicated design methods can be applied. Fundamental software design concepts provide the necessary framework for "getting it right."

9.3.1 Abstraction

When we consider a modular solution to any problem, many levels of abstraction can be posed. At the highest level of abstraction, a solution is stated in broad terms using the language of the problem environment. At lower levels of abstraction, a more procedural orientation is taken. Finally, at the lowest level of abstraction, the solution is stated in a manner that can be directly implemented.

As we move through the design process, the level of abstraction is reduced. Finally, the lowest level of abstraction is reached when source code is generated.

9.3.2 Refinement

A program is developed by successively refining levels of procedural detail. Refinement is actually a process of elaboration. We begin with a statement of function (or description of information) that is defined at a high level of abstraction. That is, the statement describes function or information conceptually but provides no information about the internal workings of the function or the internal structure of the information. Refinement causes the designer to elaborate on the original statement, providing more and more detail as each successive refinement (elaboration) occurs.

Abstraction and refinement are complementary concepts. Abstraction enables a designer to specify procedure and data and yet suppress low-level details. Refinement helps the designer to reveal low-level details as design progresses. Both concepts aid the designer in creating a complete design model as the design evolves.

9.3.3 Modularity

The concept of modularity in computer software has been developed for almost five decades. Software architecture embodies modularity, that is, software is divided into separately named and addressable components, often called modules, which are integrated to satisfy problem requirements.

9.3.4 Software Architecture

Software architecture refers to the hierarchical structure of program components (modules), the manner in which these components interact and the structure of data that are used by the components. In a broader sense, however, components can be generalized to represent major system elements and their interactions.

9.3.5 Control Hierarchy

Control hierarchy, also called program structure, represents the organization of program components (modules) and implies a hierarchy of control. It does not represent procedural aspects of software such as sequence of processes, occurrence or order of decisions, or repetition of operations; nor is it necessarily applicable to all architectural styles.

Different notations are used to represent control hierarchy for those architectural styles that are amenable to this representation. The most common is the treelike diagram (Figure 9.1) that represents hierarchical control for call and return architectures.

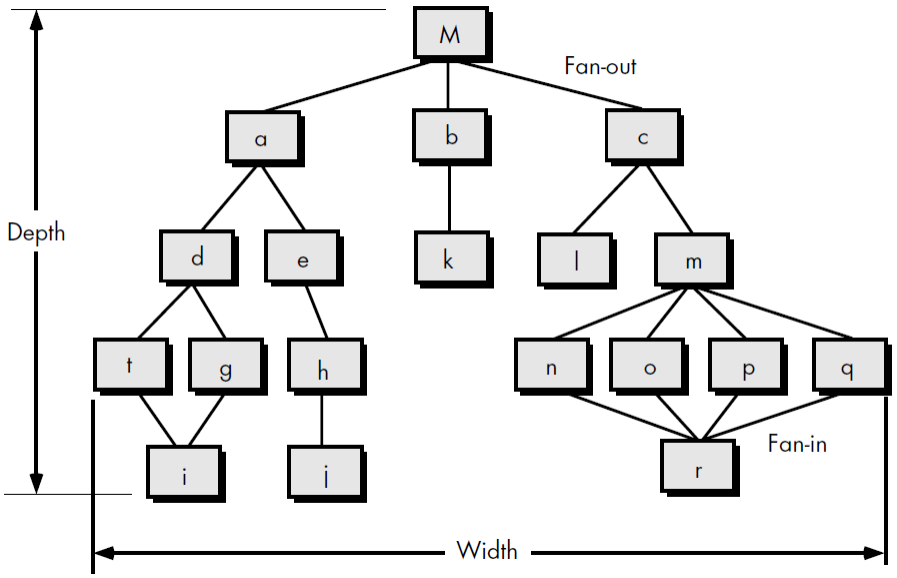

Figure 9.1 Structure terminology

In order to facilitate discussions of structure, we define a few simple measures and terms. Referring to Figure 9.1:

- **Depth and width:** provide an indication of the number of levels of control and overall span of control, respectively.
- **Fan-out:** is a measure of the number of modules that are directly controlled by another module.
- **Fan-in:** indicates how many modules directly control a given module.

The control relationship among modules is expressed in the following way: A module that controls another module is said to be superordinate to it, and conversely, a module controlled by another is said to be subordinate to the controller. For example, referring to Figure 9.1, module M is superordinate to modules a, b, and c. Module h is subordinate to module e and is ultimately subordinate to module M.

9.3.6 Structural Partitioning

If the architectural style of a system is hierarchical, the program structure can be partitioned both horizontally and vertically.

Horizontal partitioning, (Figure 9.2a), defines separate branches of the modular hierarchy for each major program function. Control modules, represented in a darker shade are used to coordinate communication between and execution of the functions. The simplest approach to horizontal partitioning defines three partitions—input, data transformation (often called processing) and output. Partitioning the architecture horizontally provides a number of distinct benefits:

- software that is easier to test
- software that is easier to maintain
- propagation of fewer side effects
- software that is easier to extend

Vertical partitioning (Figure 9.2b), often called factoring, suggests that control (decision making) and work should be distributed top-down in the program structure. Top level modules should perform control functions and do little actual processing work. Modules that reside low in the structure should be the workers, performing all input, computation, and output tasks.

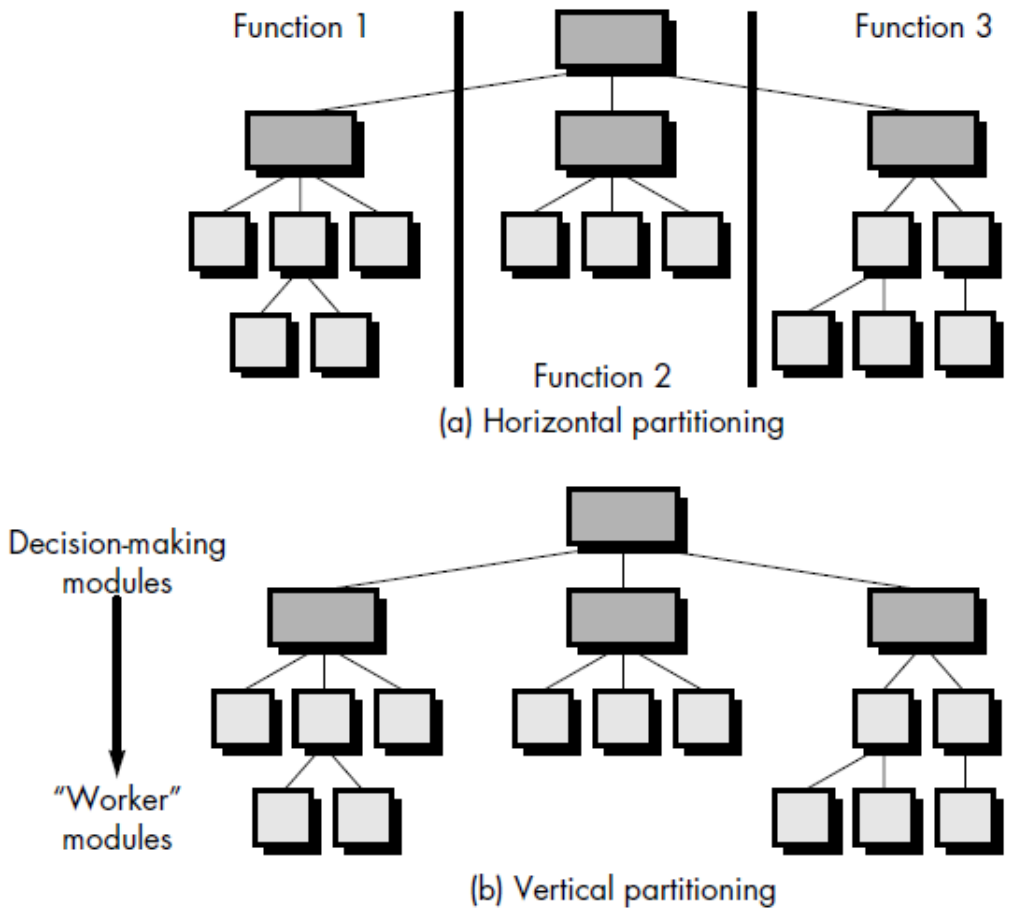

Figure 9.2 Structural partitioning