

Design Concepts and Principles

9.1 Software design and software engineering

Software design is applied regardless of the software process model that is used. Beginning once software requirements have been analyzed and specified, software design is the first of three technical activities—design, code generation, and test—that are required to build and verify the software. Each activity transforms information in a manner that ultimately results in validated computer software.

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 flow of information during software design is illustrated in Figure 1.

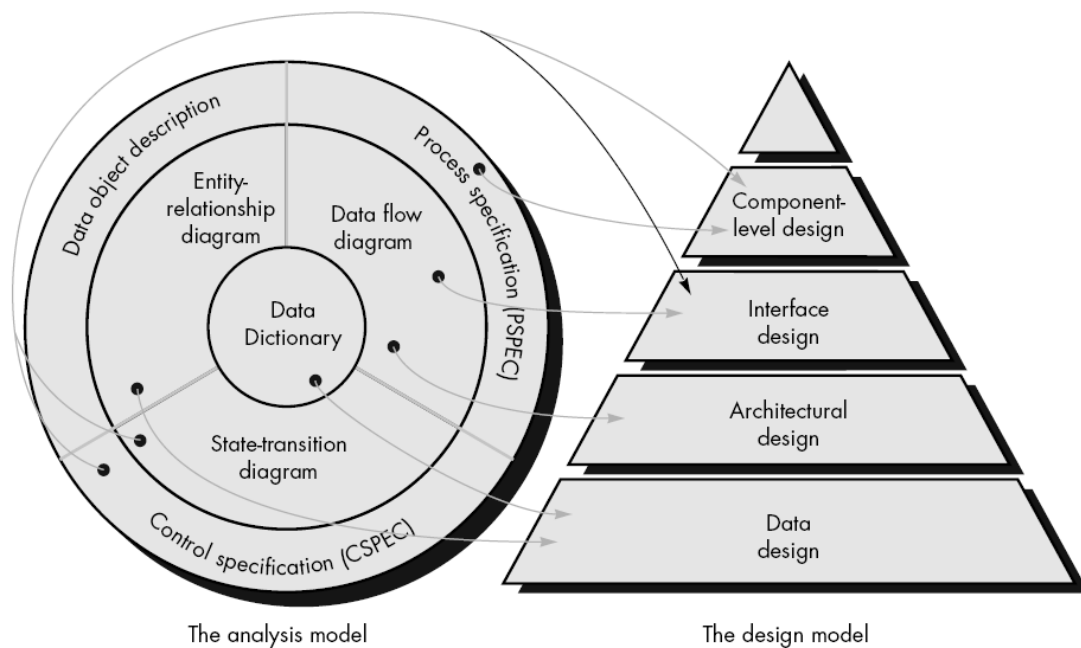


Figure 1: Translating the analysis model into a software design

Software requirements, manifested by the data, functional, and behavioral models, feed the design task. Using one of a number of design methods, the design task produces a data design, an architectural design, an interface design, and a component design.

The *data design* transforms the information domain model created during analysis into the data structures that will be required to implement the software. The data objects and relationships defined in the entity relationship diagram and the detailed data content depicted in the data dictionary provide the basis for the data design activity.

Part of data design may occur in conjunction with the design of software architecture. More detailed data design occurs as each software component is designed.

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. The architectural design representation—the framework of a computer-based system—can be derived from the system specification, the analysis model, and the interaction of subsystems defined within the analysis model.

The *interface design* describes how the software communicates within itself, with systems that interoperate with it, and with humans who use it. An interface implies a flow of information (e.g., data and/or control) and a specific type of behavior. Therefore, data and control flow diagrams provide much of the information required for interface design.

The *component-level design* transforms structural elements of the software architecture into a procedural description of software components. Information obtained from the PSPEC, CSPEC, and STD serve as the basis for component design. During design we make decisions that will ultimately affect the success of software construction and, as important, the ease with which software can be maintained.

But why is design so important?

The importance of software design can be stated with a single word—*quality*. Design is the place where quality is fostered in software engineering. Design provides us with representations of software that can be assessed for quality. Design is the only way that we can accurately translate a customer's requirements into a finished software product or system. Software design serves as the foundation for all the software engineering and software support steps that follow. Without design, we risk building an unstable system—one that will fail when small changes are made; one that may be difficult to test; one whose quality cannot be assessed until late in the software process, when time is short and many dollars have already been spent.

9.2 Design and Software Quality

Three characteristics have been suggested that serve as a guide for the evaluation of a good design:

- 1-The design must implement all of the explicit requirements contained in the analysis model, and it must accommodate all of the implicit requirements desired by the customer.
- 2-The design must be a readable, understandable guide for those who generate code and for those who test and subsequently support the software.
- 3-The design should provide a complete picture of the software, addressing the data, functional, and behavioral domains from an implementation perspective.

9.3 Design Concepts

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. Problem-oriented terminology is coupled with implementation- oriented terminology in an effort to state a solution. Finally, at the lowest level of abstraction, the solution is stated in a manner that can be directly implemented.

Each step in the software process is a refinement in the level of abstraction of the software solution. During system engineering, software is allocated as an element of a computer-based system. During software requirements analysis, the software solution is stated in terms "that are familiar in the problem environment." 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.

As we move through different levels of abstraction, we work to create procedural and data abstractions. A *procedural abstraction* is a named sequence of instructions that has a specific and limited function. An example of a procedural abstraction would be the word *open* for a door. *Open* implies a long sequence of procedural steps (e.g., walk to the door, reach out and grasp knob, turn knob and pull door, step away from moving door, etc.).

A *data abstraction* is a named collection of data that describes a data object. In the context of the procedural abstraction *open*, we can define a data abstraction called **door**. Like any data object, the data abstraction for **door** would encompass a set of attributes that describe the door (e.g., door type, swing direction, opening mechanism, weight, dimensions). It follows that the procedural abstraction *open* would make use of information contained in the attributes of the data abstraction **door**.

9.3.2-Refinement

Stepwise refinement is a top-down design strategy. A program is developed by successively refining levels of procedural detail.

A hierarchy is developed by decomposing a macroscopic statement of function (a procedural abstraction) in a stepwise fashion until programming language statements are reached.

The process of program refinement is analogous to the process of refinement and partitioning that is used during requirements analysis. The difference is in the level of implementation detail that is considered, not the approach.

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

Software architecture embodies modularity; that is, software is divided into separately named and addressable components, often called *modules* that are integrated to satisfy problem requirements.

It has been stated that "modularity is the single attribute of software that allows a program to be intellectually manageable". Monolithic software (i.e., a large program

composed of a single module) cannot be easily grasped by a reader. The number of control paths, span of reference, number of variables, and overall complexity would make understanding close to impossible. To illustrate this point, consider the following argument based on observations of human problem solving.

Let $C(x)$ be a function that defines the perceived complexity of a problem x , and $E(x)$ be a function that defines the effort (in time) required to solve a problem x . For two problems, $p1$ and $p2$, if

$$C(p1) > C(p2) \tag{9.1a}$$

it follows that

$$E(p1) > E(p2) \tag{9.1b}$$

As a general case, this result is intuitively obvious. It does take more time to solve a difficult problem.

Another interesting characteristic has been uncovered through experimentation in human problem solving. That is,

$$C(p1 + p2) > C(p1) + C(p2) \tag{9.2}$$

Expression (9-2) implies that the perceived complexity of a problem that combines $p1$ and $p2$ is greater than the perceived complexity when each problem is considered separately. Considering Expression (19-2) and the condition implied by Expressions (9.1), it follows that

$$E(p1 + p2) > E(p1) + E(p2) \tag{9.3}$$

This leads to a "divide and conquer" conclusion—it's easier to solve a complex problem when you break it into manageable pieces. The result expressed in Expression (9.3) has important implications with regard to modularity and software. It is, in fact, an argument for modularity.

It is possible to conclude from Expression (9-3) that, if we subdivide software indefinitely, the effort required to develop it will become negligibly small!

Unfortunately, other forces come into play, causing this conclusion to be (sadly) invalid. Referring to Figure.2,

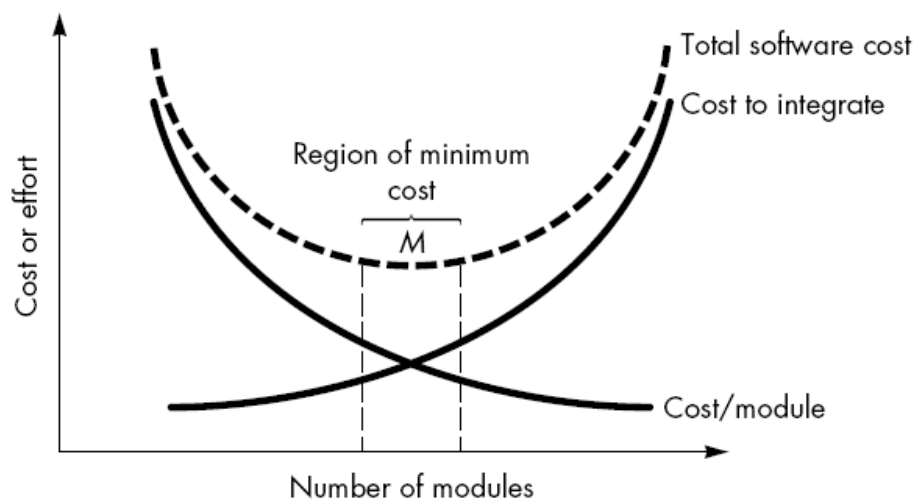


FIGURE 2: Modularity and software cost

The effort (cost) to develop an individual software module does decrease as the total number of modules increases. Given the same set of requirements, more modules means smaller individual size. However, as the number of modules grows, the effort (cost) associated with integrating the modules also grows. These characteristics lead to a total cost or effort curve shown in the figure. There is a number, M , of modules that would result in minimum development cost, but we do not have the necessary sophistication to predict M with assurance.

The curves shown in Figure 2 do provide useful guidance when modularity is considered. We should modularize, but care should be taken to stay in the vicinity of M . Undermodularity or overmodularity should be avoided.

9.3.4-Software Architecture

Software architecture is 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.

One goal of software design is to derive an architectural rendering of a system. This rendering serves as a framework from which more detailed design activities are conducted. A set of architectural patterns enable a software engineer to reuse design level concepts.

A set of properties should be specified as part of an architectural design:

Structural properties. This aspect of the architectural design representation defines the components of a system (e.g., modules, objects) and the manner in which those components are packaged and interact with one another. For example, objects are packaged to encapsulate both data and the processing that manipulates the data and interact via the invocation of methods.

Extra-functional properties. The architectural design description should address how the design architecture achieves requirements for performance, capacity, reliability, security, adaptability, and other system characteristics.

Families of related systems. The architectural design should draw upon repeatable patterns that are commonly encountered in the design of families of similar systems. In essence, the design should have the ability to reuse architectural building blocks.

Given the specification of these properties, the architectural design can be represented using one or more of a number of different models.

1-Structural models represent architecture as an organized collection of program components.

2-Framework models increase the level of design abstraction by attempting to identify repeatable architectural design frameworks (patterns) that are encountered in similar types of applications.

3-Dynamic models address the behavioral aspects of the program architecture, indicating how the structure or system configuration may change as a function of external events.

4-Process models focus on the design of the business or technical process that the system must accommodate.

5-Functional models can be used to represent the functional hierarchy of a system.

A number of different *architectural description languages* (ADLs) have been developed to represent these models. Although many different ADLs have been proposed, the majority provide mechanisms for describing system components and the manner in which they are connected to one another.

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 3) that represents hierarchical control for call and return architectures.

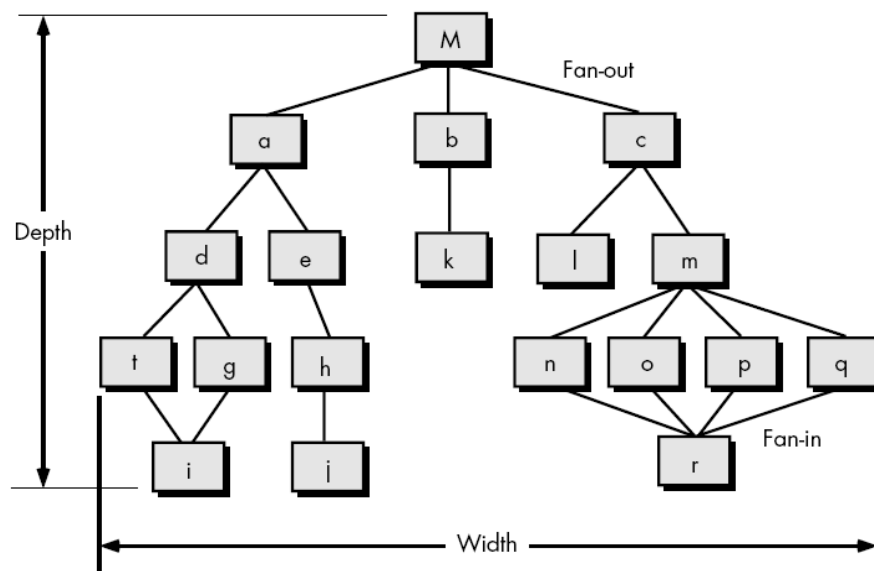


FIGURE 3: Structure terminology for a call and return architectural style

Note:

A call and return architecture is a classic program structure that decomposes function into a control hierarchy where a “main” program invokes a number of program components, which in turn may invoke still other components.

In order to facilitate later discussions of structure, we define a few simple measures and terms. Referring to Figure 3, **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 3, module *M* is superordinate to modules *a*, *b*, and *c*. Module *h* is subordinate to module *e* and is ultimately subordinate to module *M*. Width-oriented relationships (e.g., between modules *d* and *e*) although possible to express in practice, need not be defined with explicit terminology.

The control hierarchy also represents two subtly different characteristics of the software architecture: visibility and connectivity.

Visibility indicates the set of program components that may be invoked or used as data by a given component, even when this is accomplished indirectly. For example, a module in an object-oriented system may have access to a wide array of data objects that it has inherited, but makes use of only a small number of these data objects. All of the objects are visible to the module.

Connectivity indicates the set of components that are directly invoked or used as data by a given component. For example, a module that directly causes another module to begin execution is connected to it.

9.3.6 Data Structure

Data structure is a representation of the logical relationship among individual elements of data. Because the structure of information will invariably affect the final procedural design, data structure is as important as program structure to the representation of software architecture.

Data structure dictates the organization, methods of access, degree of associativity, and processing alternatives for information. The organization and complexity of data structure are limited only by the ingenuity of the designer. There are, however, a limited number of classic data structures that form the building blocks for more sophisticated structures.

A **scalar item** is the simplest of all data structures. As its name implies, a scalar item represents a single element of information that may be addressed by an identifier; that is, access may be achieved by specifying a single address in memory. The size and format of a scalar item may vary within bounds that are dictated by a programming language. For example, a scalar item may be a logical entity one bit long, an integer

or floating point number that is 8 to 64 bits long, or a character string that is hundreds or thousands of bytes long.

When scalar items are organized as a list or contiguous group, a *sequential vector* is formed. When the sequential vector is extended to two, three, and ultimately, an arbitrary number of dimensions, an *n-dimensional space* is created. The most common *n-dimensional space* is the two-dimensional matrix. In many programming languages, an *n-dimensional space* is called an *array*.

Items, vectors, and spaces may be organized in a variety of formats. A *linked list* is a data structure that organizes noncontiguous scalar items, vectors, or spaces in a manner (called *nodes*) that enables them to be processed as a list. Each node contains the appropriate data organization (e.g., a vector) and one or more pointers that indicate the address in storage of the next node in the list. Nodes may be added at any point in the list by redefining pointers to accommodate the new list entry.

It is important to note that data structures, like program structure, can be represented at different levels of abstraction. For example, a stack is a conceptual model of a data structure that can be implemented as a vector or a linked list. Depending on the level of design detail, the internal workings of a **stack** may or may not be specified.

13.3.7 Software Procedure

Program structure defines control hierarchy without regard to the sequence of processing and decisions. Software procedure focuses on the processing details of each module individually. Procedure must provide a precise specification of processing, including sequence of events, exact decision points, repetitive operations, and even data organization and structure.

There is, of course, a relationship between structure and procedure. The processing indicated for each module must include a reference to all modules subordinate to the module being described. That is, a procedural representation of software is layered as illustrated in Figure 5

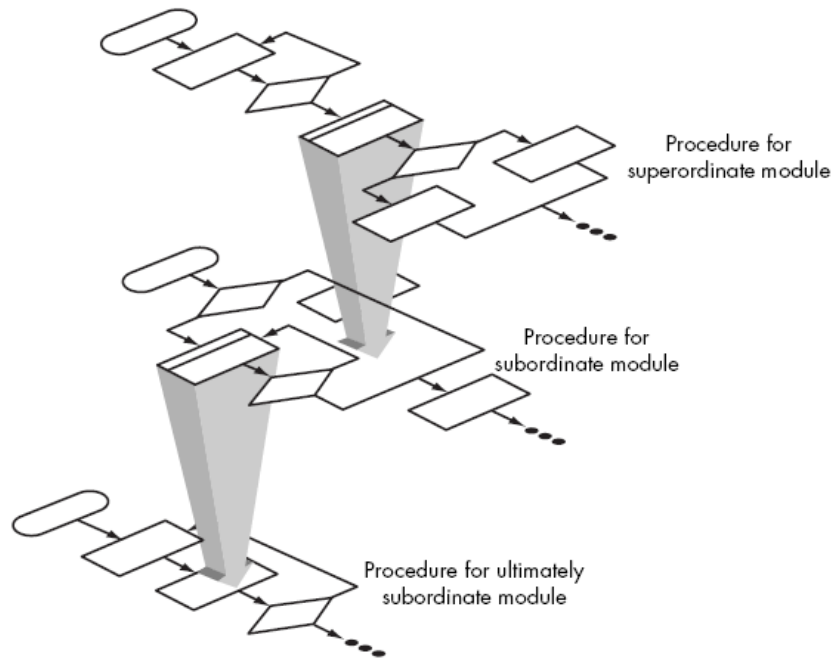


Figure 5: Procedure is layered

9.3.8 Information Hiding

The concept of modularity leads every software designer to a fundamental question: "How do we decompose a software solution to obtain the best set of modules?"

The principle of *information hiding* suggests that modules be "characterized by design decisions that (each) hides from all others." In other words, modules should be specified and designed so that information (procedure and data) contained within a module is inaccessible to other modules that have no need for such information.

Hiding implies that effective modularity can be achieved by defining a set of independent modules that communicate with one another only that information necessary to achieve software function. Abstraction helps to define the procedural (or informational) entities that make up the software. Hiding defines and enforces access constraints to both procedural detail within a module and any local data structure used by the module.

The use of information hiding as a design criterion for modular systems provides the greatest benefits when modifications are required during testing and later, during software maintenance. Because most data and procedure are hidden from other parts of the software, inadvertent errors introduced during modification are less likely to propagate to other locations within the software.