3

System Design

Contact hours: 6

3.1 Introduction

Software design is more creative process than analysis because it deals with the development of the actual mechanics for a new workable system. As such, it cannot be reduced to a series of steps that can be simply followed, though guidelines can be provided. The methods used in software design and analysis are similar in that both cases are essentially construction of 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 modelling has to represent it. In design, the designer has a great deal of freedom in deciding the models, as the system the designer is modelling 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. Though the basic principles and techniques might look similar, the activities of analysis and design are very different.

What is design? Design is the highly significant phase in the software development where the designer plans 'how' a software system should be produced in order to make it functional, reliable and reasonably easy to understand, modify and maintain.
A software requirements specifications (SRS) document tells us "what" a system does and becomes input to the design process, which tells us "how" a software system works. Designing software systems means determining how requirements are realized and the result is a software design document (SDD). Thus the purpose of design phase is to produce a solution to a problem given in SRS document. Figure.3.1 shows the design

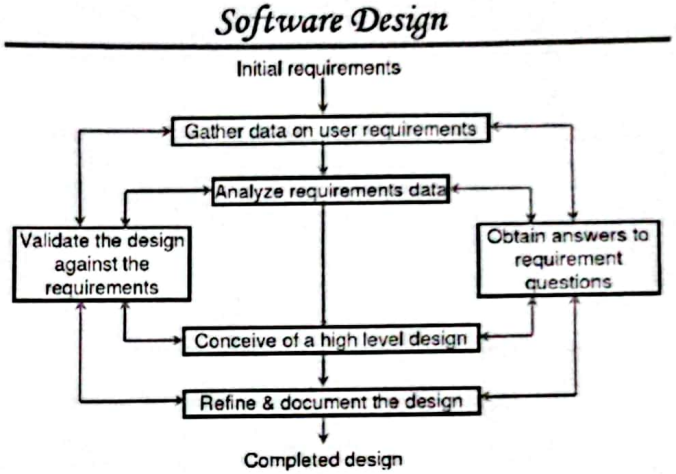


Figure 3.1: Design framework.

framework. It begins with initial requirements and ends up with the final design.

3.2 Conceptual and Technical Designs

The process of software design involves the transformation of ideas into detailed implementation descriptions, with the goal of satisfying the software requirements. To transform requirements into a working system, designers must satisfy both customers and the system builders (coding persons). The customers understand what the system is to do and at the same time, the system builders must understand how the system is to work. For this reason, design is really a two part, iterative process. First, we produce conceptual design that tells the customer exactly what the system will do. Once the customer approves the conceptual design, we translate the conceptual design into a much more detailed document, the technical design, that allows system builders to understand the actual hardware and software needed to solve the customer's problem. This two part design is shown in Figure 3.2. The two design documents describe the same system, but in different ways because of the different audiences for the documents. The conceptual design answers the following questions.

Software Design

Conceptual Design and Technical Design

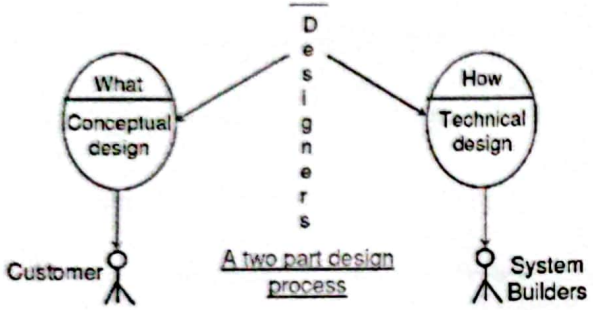


Figure 3.2: A two-part design process.

(*i*) Where will the data come from?

(*ii*) What will happen to the data in the system?

(*iii*) How will the system look to users?

(*iv*) What choices will be offered to users?

(*v*) What is the timing of events?

(*vi*) How will the reports and screens look like?

The conceptual design describes the system in language understandable to the customers. It does not contain any technical jargons and is independent of implementation. Whereas, the technical design describes the hardware configuration, the software needs, the communications interfaces, the input and output of the system, the network architecture, and anything else that translates the requirements into a solution to the customer's problem. Sometimes customers are very sophisticated and they can understand the 'what' and 'how' together. This can happen when customers are themselves software developers and may not require conceptual design. In such cases comprehensive design document may be produced.

Software Design

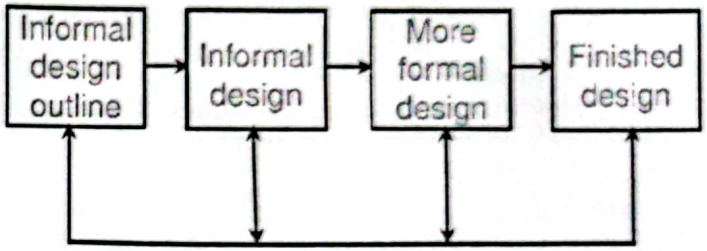


Figure 3.3: The transformation of an informal design to a detail design.

3.3 Objectives of design

Specification is seldom a document from which coding can directly be done. So design fills the gap between specifications and coding. If the specification calls for a large or complex program (or both), then the design is quite likely to work down through a number of levels. At each level, breaking the implementation problem into a combination of smaller and smaller problems. Filling a large gap will involve a number of stepping stones. The wider the gap, the larger the number of stepping-stones. The design needs to be: correct and complete, understandable, at the right level, maintainable, and facilitate maintenance of the produced code.

Software designers do not arrive at a finished design document immediately but develop the design iteratively through a number of different phases. The design process involves adding details as the design is developed with constant backtracking to correct earlier, less formal, designs. The starting point is an informal design which is refined by adding information to make it consistent and complete as shown in Figure 3.3.

3.4 Why is design important?

A good design is the key to successful product. A well-designed system is easy to implement, understandable and reliable and allows for smooth evolution. Without design, we risk building an unstable system. One that will fail when small changes are made. One

that will be difficult to maintain. One whose quality cannot be assessed until late in the software process.

Therefore, software design should contain a sufficiently complete, accurate and precise solution to a problem in order to ensure its quality implementation. There are three characteristics that serve as a guide for the evolution of a good design.

(*i*) 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.

(*ii*) The design must be readable, understandable guide for those who generate code and for those who test and subsequently support the software.

(*iii*) The design should provide a complete picture of the software, addressing the data, functional and behavioural domain from an implementation perspective.

3.5 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 unfavourable 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 judgement. 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 modelling has to represent it. In design, the designer has a great deal of freedom in deciding the models, as the system the designer is modelling

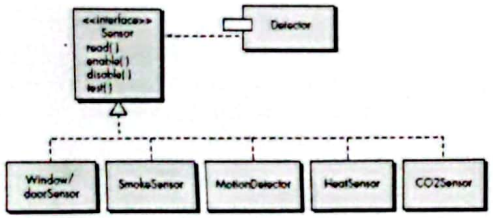


Figure 3.4: Following the OCP

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 modelling in problem analysis is to understand, while the basic aim of modelling 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.

3.5.1 The Open-Closed Principle (OCP)

"A module [component] should be open for extension but closed for modification." This statement seems to be a contradiction, but it represents one of the most important characteristics of a good component-level design. Stated simply, you should specify the component in a way that allows it to be extended (within the functional domain that it addresses) without the need to make internal (code or logic-level) modifications to the component itself. To accomplish this, you create abstractions that serve as a buffer between the functionality that is likely to be extended and the design class itself. For example, assume that the SafeHome security function makes use of a Detector class that must check the status of each type of security sensor. It is likely that as time passes, the number and types of security sensors will grow. If internal processing logic is implemented as a sequence of if-then-else constructs, each addressing a different sensor type, the addition of a new sensor type will require additional internal processing logic (still another if-then-else). This is a violation of OCP. One way to accomplish OCP for the Detector class is illustrated in Figure 10.4. The sensor interface presents a consistent view of sensors to the detector component. If a new type of sensor is added no change is required for the Detector class (component). The OCP is preserved.

3.6 Problem Partitioning

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. 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 sub-parts, 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.

3.7 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 environment. An abstraction of a component describes the external behaviour of that component without bothering with the internal details that produce the behaviour. 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 partitioningisolating 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. FYmctional abstraction is the basis of partitioning in function-oriented approaches. That is, when the problem is being partitioned, 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.

When you 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 detailed description of the solution is provided. 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. As different levels of abstraction are developed, you work to create both procedural and data abstractions. A procedural abstraction refers to a sequence of instructions that have a specific and limited function. The name of a procedural abstraction implies these functions, but specific details are suppressed. 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.).5 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.

3.8 Separation of concerns

Separation of concerns is a design concept that suggests that any complex problem can be more easily handled if it is subdivided into pieces that can each be solved and/or optimized independently. A concern is a feature or behaviour that is specified as part of the requirements model for the software. By separating concerns into smaller, and therefore more manageable pieces, a problem takes less effort and time to solve.

3.9 Modularity

Modularity is the most common manifestation of separation of concerns. Software is divided into separately named and addressable components, sometimes 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 software engineer. The number of control paths, span of reference, number of variables, and overall complexity would make understanding close to impossible. In almost all instances, you should break the design into many modules, hoping to make understanding easier and, as a consequence, reduce the cost required to build the software. It is possible to conclude that if you 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 Figure3.5, 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 Figure3.5 do provide useful qualitative guidance when modularity is

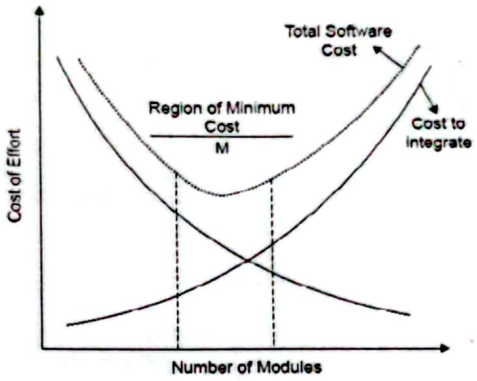


Figure 3.5: Modularity and software cost.

considered. You should modularize, but care should be taken to stay in the vicinity of M. Undermodularity or overmodularity should be avoided. But how do you know the vicinity of M? How modular should you make software? The answers to these questions require an understanding of other design concepts considered later in this chapter. **You modularize a design (and the resulting program) so that development can be more easily planned; software increments can be defined and delivered; changes can be more easily accommodated; testing and debugging can be conducted more efficiently, and long-term maintenance can be conducted without serious side effects.** A module is a work assignment for an individual programmer. A modular system consist of well defined, manageable units with defined interfaces among the units. Desirable properties of a modular system include:

(*i*) Each module is a well-defined subsystem that is potentially useful in other applications.

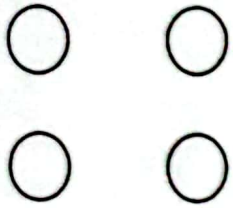
(*ii*) Each module has a single, well-defined purpose.

(*iii*) Modules can be separately compiled and stored in a library.

(*iv*) Modules can use other modules.

(*v*) Modules should be easier to use than to build.

(*vi*) Modules should be simpler from the outside than from the inside.



(Uncoupled : no dependencies)

(a)

Figure 3.6: Module coupling: Uncoupled (no dependencies)

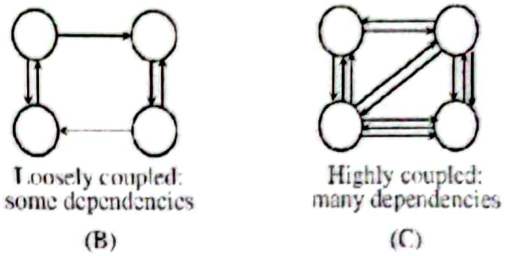
Modularity is the single attribute of software that allows a program to be intellectually manageable. It enhances design clarity, which in turn eases implementation, debugging, testing, documenting, and maintenance of the software product.

A system is considered modular if it consists of discrete components so that each component can be implemented separately and a change to one component has a minimal impact on other components. Here, one important question arises is to what extent we shall modularize. As the number of modules grows, the effort associated with integrating the module also grows. Figure 3.5 shows the relationship between cost/effort and number of modules in a software. Note that a software system cannot be made modular by simply chopping it into a set of modules. Each module needs to support a well-defined abstraction and should have a clear interface through which it can interact with other modules. Thus, it is felt that under modularity and over modularity in a software should be avoided.

3.10 Coupling

Coupling is a measure of the degree of interdependence between modules.Two modules with high coupling are strongly interconnected and thus, dependent on each other. Two modules with low coupling are not dependent on one another. 'Loosely coupled' systems are made up of modules which are relatively independent. 'Highly coupled' systems share a great deal of dependence between modules. For example, Uncoupled modules have no interconnection at all; they are completely independent as in Figure 3.6 and Figure 3.7.

A good design will have low coupling. Thus, interfaces should be carefully specified in order to keep low value of coupling. Coupling is measured by the number of interconnections between modules. For example, coupling increases as the number of



Loosely coupled:
some dependencies

(B)

Highly coupled:
many dependencies

(C)

Figure 3.7: Module coupling

calls between modules increases, or the amount of shared data increases. The hypothesis is that design with high coupling will have more errors. Loose coupling, on the other hand, minimizes the interdependence amongst modules. This can be achieved in the following ways.

(*i*) Controlling the number of parameters passed amongst modules

(*ii*) Avoid passing undesired data to calling module

(*iii*) Maintain parent/child relationship between calling and called modules

(*iv*) Pass data, not the control of information

Figure3.8 demonstrates two alternatives design for editing a student record in a 'Student Information System.' The first design demonstrates tight coupling wherein unnecessary information as student name, student address, course is passed to the calling module. Passing superfluous information unnecessary increases the overhead, rducing the system performance/efficiency.

3.10.1 Types of coupling

Different types of coupling are content, common, external, control, stamp and data. The strength of coupling from lowest coupling (best) to highest coupling (worst) is given in Figure 3.9.

Given two modules A and B, we can identify a number of ways in which they can be coupled.

3.10.2 Data coupling

The dependency between module A and B is said to be data coupled if their dependency is based on the fact they communicated by passing of data. By ensuring that modules

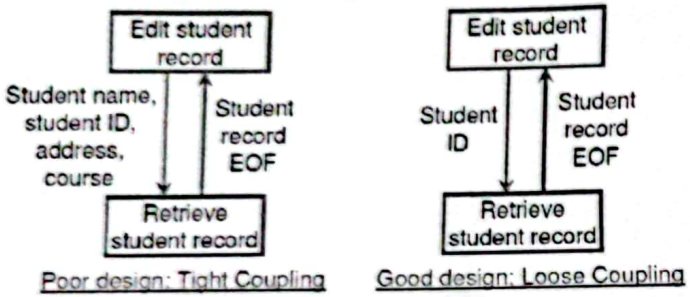


Figure 3.8: Example of coupling.

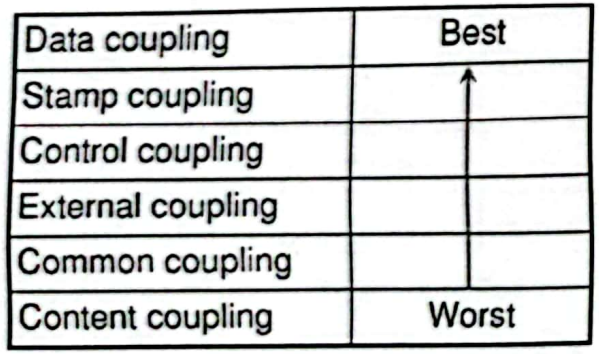


Figure 3.9: Types of module coupling.

communicate only necessary data, module dependency is minimized.

3.10.3 Stamp coupling

Stamp coupling occurs between module A and module B when complete data structure is passed from one module to another. Since not all data making up the structure are usually necessary in communication between the modules, stamp coupling typically involves tramp data. If one procedure only needs a part of a data structure, calling module should pass just that part, not the complete data structure.

3.10.4 Control coupling

Module A and module B are said to be control coupled if they communicate by passing of control information. This is usually accomplished by means of flags that are set by one module and reacted upon by the dependent module.

3.10.5 External coupling

A form of coupling in which a module has a dependency to other module, external to the software being developed or to a particular type of hardware. This is basically related to the communication to external tools and devices.

3.10.6 Common coupling

With common coupling, module A and module B have shared data. Global data areas are commonly found in programming languages. Making a change to the common data means tracing back to all the modules which access that data to evaluate the effect of change. With common coupling, it can be difficult to determine which module is responsible for having set a variable to a particular value. Figure 3.10 shows how common coupling works.

3.10.7 Content coupling

Content coupling occurs when module A changes data of module B or when control is passed from one module to the middle of another. Figure 3.11 shows module B branching into D, even though D is supposed to be under the control of C.

3.11 Module cohesion

Cohesion is a measure of the degree to which the elements of a module are functionally related. A strongly cohesive module implements functionality that is related to one feature of the solution and requires little or no interaction with other modules. This is shown in Figure 3.12. Cohesion may be viewed as a glue that keeps the module together. Thus, we

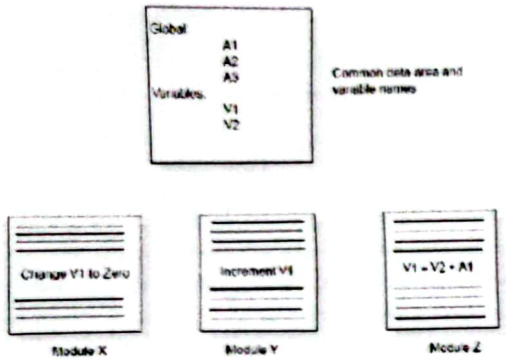


Figure 3.10: Example of common coupling.

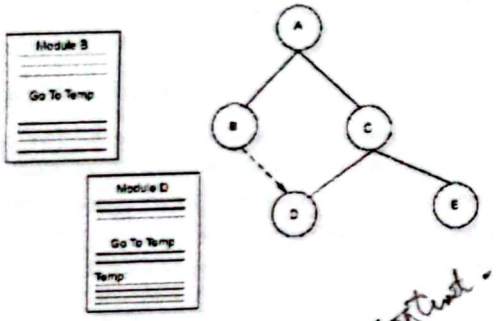


Figure 3.11: Example of common coupling.

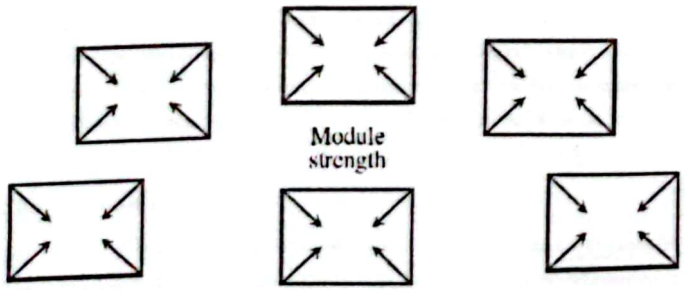


Figure 3.12: Cohesion=Strength of relations within modules.

Functional Cohesion	Best (high)
Sequential Cohesion	
Communicational Cohesion	
Procedural Cohesion	
Temporal Cohesion	
Logical Cohesion	
Coincidental Cohesion	Worst (low)

Figure 3.13: Types of module cohesion.

want to maximize the interaction within a module. Hence, an important design objectives is to maximize the module cohesion and minimize the module coupling.

3.12 Types of Cohesion

There are several types or levels of cohesion and are shown in Figure 3.13. Given a procedure that carries out operations X and Y, we can describe various forms of cohesion between X and Y.

3.12.1 Functional cohesion

X and Y are part of a single functional task. This is very good reason for them to be contained in the same procedure. Such a module often transformed a single input datum into a single output datum. The mathematical subroutines such as 'calculate current GPA' or 'cumulative GPA' are typical examples of functional cohesion.

3.12.2 Sequential cohesion

X outputs some data which forms the input to Y. This is the reason for them to be contained in the same procedure. For example, addition of marks of individual subjects into a specific format is used to calculate the GPA as input for preparing the result of the students.

A component is made of parts that need to communicate/exchange data from one source for different functional purposes. They are together in a component for communicational convenience. For example calculate current and cumulative GPA uses the 'student grade record' as input.

3.12.3 Communicational cohesion

X and Y both operate on the same input data or contribute towards the same output data. This is okay, but we might consider making them separate procedures.

3.12.4 Procedural cohesion

X and Y are both structured in the same way. This is a poor reason for putting them in the same procedure. Thus, procedural cohesion occurs in modules whose instructions although accomplish different tasks yet have been combined because there is a specific order in which the tasks are to be completed. These types of modules are typically the result of first flow charting the solution to a program and then selecting a sequence of instructions to serve as a module. Since these modules consist of instructions that accomplish several tasks that are virtually unrelated these types of modules tend to be less maintainable. For example, if a report module of an examination system includes the following 'calculate student GPA, print student record, calculate cumulative GPA, print cumulative GPA' is a case of procedural cohesion.

3.12.5 Temporal cohesion

X and Y both must perform around the same time. So, module exhibits temporal cohesion when it contains tasks that are related by the fact that all tasks must be executed in the same time-span. The set of functions responsible for initialization, start up activities such as setting program counters or control flags associated with programs exhibit temporal cohesion. This is not a good reason to put them in same procedure.

3.12.6 Logical cohesion

X and Y perform logically similar operations. Therefore, logical cohesion occurs in modules that contain instructions that appear to be related because they fall into the same logical class of functions. Considerable duplication can exist in the logical strength level. For example, more than one data item in an input transaction may be a date. Separate code would be written to check that each such date is a valid date. A better way to construct a DATECHECK module and call this module whenever a date check is necessary.

3.12.7 Coincidental cohesion

X and Y have no conceptual relationship other than shared code. Hence, coincidental cohesion exists in modules that contain instructions that have little or no relationship to one another. That is, instead of creating two components, each of one part, only one

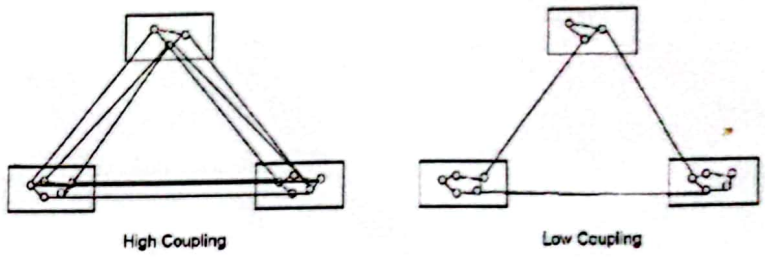


Figure 3.14: View of cohesion and coupling

component is made with two unrelated parts. For example, check validity and print is a single component with two parts. Coincidental cohesion is to be avoided as far as possible.

3.13 Relationship between cohesion and coupling

The essence of the design process is that the system is decomposed into parts to facilitate the capability of understanding and modifying a system. Projects gets into trouble because of massive requirement changes. These changes can be properly recognised and properly reviewed.

If the software is not properly modularized, a host of seemingly trivial enhancement or changes will result into death of the project. Therefore, a good software design professes clean decomposition of a problem into modules and the arrangement of these modules in a neat hierarchy. Therefore, a software engineer must design the modules with goal of high cohesion and low coupling.

A good example of a system that has high cohesion and low coupling is the 'plug and play' feature of the computer system. Various slots in the motherboard of the system simply facilitate to add or remove the various services/functionalities without affecting the entire system. This is because the add on components provide the services in highly cohesive manner. Figure 3.14 provides a graphical review of cohesion and coupling. Module design with high cohesion and low coupling characterises a module as black box when the entire structure of the system is described. Each module can be dealt separately when the module functionality is described.