UNIT 6 MODULAR AND STRUCTURED DESIGN

Structure **Page Nos.**

6.0 Introduction 17
6.1 Objectives 17
6.2 Design Principles 18
6.2.1 Top Down Design
6.2.2 Bottom Up Design
6.3 Structure Charts 20
6.4 Modularity 23
6.4.1 Goals of Design
6.4.2 Coupling
6.4.3 Cohesion
6.5 Summary 29
6.6 Solutions/Answers 29
6.7 Further Readings 30

6.0 INTRODUCTION

In this unit, you will learn the process of designing system's internals. We will begin at an abstract level, taking system's processes, in the form of data flow diagrams, and convert them to structure charts. Structure charts represent design graphically. You will learn the process of drawing structure charts and how to derive them from dataflow diagrams. The structure charts you create will form the basis for the structure of the system you design and build. Decisions made at this point will influence the overall design and implementation of the information system (IS). So, you need to be aware of what constitutes a good design. You will learn about some guidelines that will help you achieve it. The primary goal behind good design is to make your system easy to read and easy to maintain. The primary way to do this is to divide the problem solutions into smaller and smaller pieces or modules. The smaller the pieces or modules the easier it is to program, to read and to revise due to changing users' requirements. Modularisation, therefore promotes ease of coding and maintenance. On the other hand, modularisation is not simply reduction of size but it is also a reflection of function that is what particular piece of a system i.e. a module supposed to do. One guideline for good design is to maximize cohesion, the extent to which a part of the system is designed to perform one and only one function or task. Modules that perform single task are easier to write and maintain than those performing different tasks. As modularisation also involves how different parts of the system work in conjunction with each other, another guideline for good design is to minimize coupling, the extent to which different parts of the system are dependent on each other.

6.1 OBJECTIVES

After going through this unit, you should be able to:

- know the meaning of design;
- learn the process of top down design and bottom up design;
- learn the process of drawing a structure chart;
- learn the goals of design;
- differentiate between five types of coupling and apply them in programs; and
- differentiate between seven types of cohesion and apply them in programs.

Design bridges the gap between specifications and coding. The design of the system is correct if the system is built according to the design that satisfies the requirements of that system.

Some of the properties of design are:

Verifiability: It is concerned with how easily the correctness of design can be argued.
Traceability: It helps in design verification. It requires that all design must be traceable to the requirements.
Completeness: The software must be complete in all respects.
Consistency: It requires that there are no inherent inconsistencies within the system.
Efficiency: It is concerned with proper usage of resources by the system.
Simplicity: It is related to simple design so that user can easily understand and use it. Simple designs are easy to maintain in the long run or through the lifecycle of a software system.

6.2 DESIGN PRINCIPLES

There are certain principles that can be used for the development of the system. These principles are meant to effectively handle the complexity of process of design. These principles are:

Problem Partitioning: It is concerned with partitioning the large problems. *Divide and Conquer* is the policy adopted here. The system is divided into modules that are self dependent. It improves the efficiency of the system. It is necessary that all modules have interaction between them.

Abstraction: It is an indispensable part of design process and is essential for problem partitioning. Abstraction is a tool that permits the designer to consider a component at an abstract level (outer view) without worrying about details of implementation of the component. Abstraction is necessary when the problem is divided into smaller parts so that one can proceed with one design process effectively and efficiently. Abstraction can be functional or data abstraction. In functional abstraction, we specify the module by the function it performs. In data abstraction, data is hidden behind functions/ operations. Data abstraction forms the basis for object-oriented design.

Design principles are necessary for efficient software design. Top down and bottom up strategies help implement these principles and achieve the objectives.

A system consists of components called modules, which have subordinate modules. A system is a hierarchy of components and the highest-level module called super ordinate module corresponds to the total system. To design such a hierarchy, there are two approaches namely top down and bottom up approaches.

The top down approach starts from the highest-level module of the hierarchy and proceeds through to lower level. On the contrary, bottom up approach starts with lower level modules and proceeds through higher levels to the top-level module.

6.2.1 Top Down Design

This approach starts by identifying major components of the system decomposing them into their own subordinate level components and interacting until the desired level of detail is achieved. Top down design methods often result in some form of stepwise refinement, starting from an abstract design, in each step, the design is refined to a more concrete level until we reach a level where no more refinement is required and the design can be implemented directly. This approach is explained by taking an example of “Library Information System” depicted in figures 6.1,6.2 and 6.3 below:

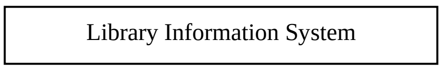
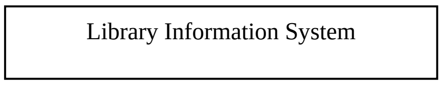


Figure 6.1: The top (root) of software system



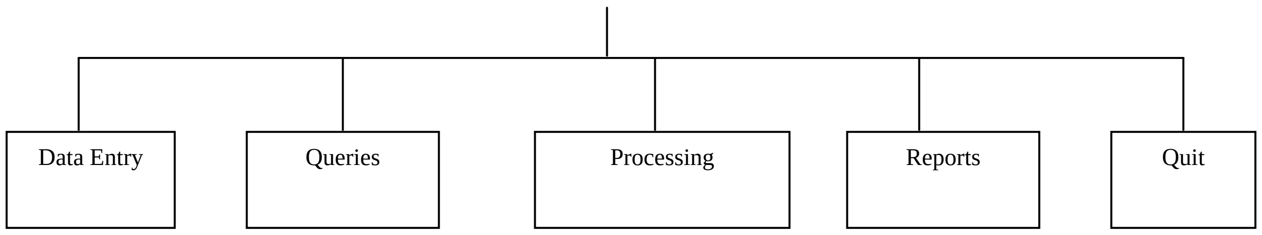


Figure 6.2: **Further decomposition of the "top" of software system**

Now, we can move further down and divide the system even further as shown in Figure 6.3.

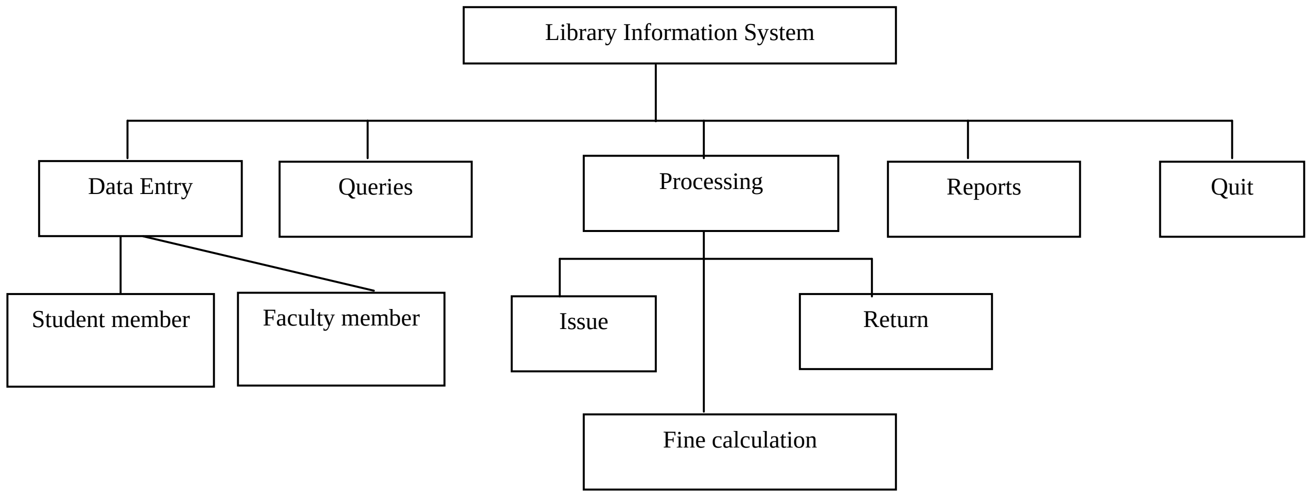


Figure 6.3: Hierarchy Chart of Library Information System

This iterative process can go on till we have reached a complete software system. A complete software system is a system that has been coded completely using any front-end tool (ex Java, Visual Basic, VC++, Power Builder etc).

Top down design strategies work very well for system that is made from the scratch. We can always start from the main menu and proceed down the hierarchy designing data entry modules, queries modules, etc.

6.2.2 Bottom Up Design

In bottom up strategy, we start from the bottom and move upwards towards the top of the software.

This approach leads to a style of design where we decide the process of combining modules to provide larger ones, to combine these to provide even larger ones and so on till we arrive at one big module. That is, the whole of the desired program.
This method has one weakness. We need to use a lot of intuition to decide the functionality that is to be provided by the module. If a system is to be built from existing system, this approach is more suitable as it starts from some existing modules.

Check Your Progress 1

1 ……..…………… is a tool that permits the designer to consider a component at an abstract level (outer view) without worrying about details of implementation of the component.

2. ………………..… starts by identifying major components of the system decomposing them into their own subordinate level components and interacting until the desired level of detail is achieved.

3. ………………..… starts from the bottom and move upwards towards the top of the software.

6.3 STRUCTURE CHARTS

A structure chart depicts the modular organization of an information system. The organization is hierarchical. A structure chart graphically shows the way the components of a program or a system are related. The relationships are shown in terms of parameter passing mechanisms applied and the basic structured programming operations namely sequence, selection and repetition. A structure chart depicts the division of a system into programs along with their internal structure. However, the internal structure of those programs which are written in third and fourth generation languages can be depicted.

A structure chart depicts various modules across different levels of the hierarchical organisation. Always, it has one coordinating module at the top. The modules at the next level are called by the coordinating module. If a menu based system is concerned, the main menu may be considered as the coordinating module and the options in it may be considered as subordinate modules. Even, the calling mechanism is hierarchical. The coordinating module at the top calls the modules at the next level and the modules at this level call the modules at the next level to them. A module calls a subordinate module whenever there is a need for the operation to be performed by the subordinate module. Now, the following question arises: What about those modules at the lowest level? Whom do they call, as there are no modules after that level? The answer is: Modules at lower level perform various tasks. They don't call any other module.

Consider Figure 6.4. It depicts a structure chart. *System* is the top module. It's subordinate modules are *Get X* and *Make Y*. The subordinate modules of *Get X* are *Get W* and *Make X*. There are no subordinate modules for *Make Y*. It is very important that the function of a module can be easily grasped from its name. So, naming of the modules is critical to understand the system as a whole. Since a module should not perform more than one task, there will be no use of *and* in the name of a module as it means that the module is performing multiple tasks. But, it is common to find modules which perform multiple tasks and this can be easily realised from the conjunctions used in their names. In the case of such modules, it is advisable to divide them into multiple modules with each module performing a single task. These modules can be executed from left to right.

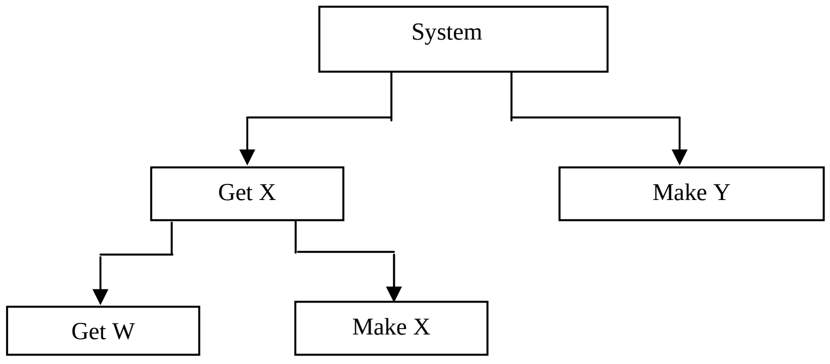


Figure 6.4: Hierarchy of a Structure Chart

The communication between various modules in a Structure chart takes place by passing the requisite data items as parameters. Data is represented as data couples and flags. A *data couple* is a symbol which consists of an empty circle with an arrow coming out of it. The direction of the arrow indicates the direction of data communication. A control flag is indicated by using the same symbol as that of a data couple except that the circle is filled. A control flag gives information about the data being communicated. It may indicate EOF etc. Figure 6.5 shows the symbols for data couples and Figure 6.6 show the symbols for control flag.

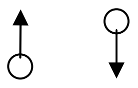


Figure 6.5: Data couples

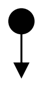


Figure 6.6: Control Flag

A module is indicated by a rectangle. The name of the module may be indicated with in the module. Figure 6.7 shows the symbol for module.

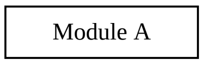


Figure 6.7: Symbol for Module

Figure 6.8 depicts a set of superordinate and subordinate modules. Here A is superordinate module and B is subordinate module. So, A will call B whenever necessary.

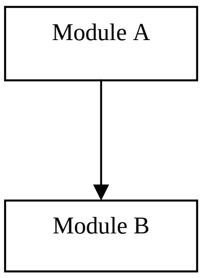


Figure 6.8: Symbols for superordinate module A and subordinate module B

Figure 6.9 depicts a total of three modules namely A, B and C. A is the superordinate module and B, C are subordinate modules. The curved arrow over the two communication lines connecting module A to B and C indicates that B and C are repeatedly called by A.

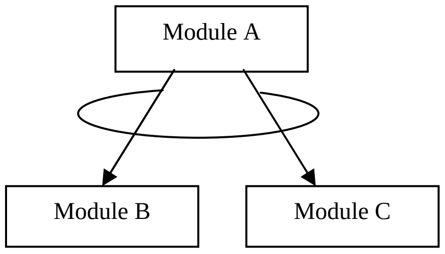


Figure 6.9: Repeated calls of Modules B and C by Module A

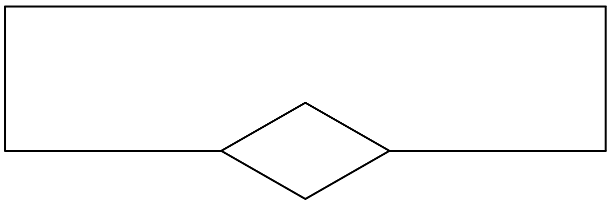


Figure 6.10: Subordinate modules are called on condition

Figure 6.10 depicts the diamond symbol. It indicates the subordinate module to be called on the result of the execution of a conditional statement. Though, there can be a number of subordinate modules for a superordinate module, not all of them are called when a diamond symbol exists.

Figure 6.11 depicts a module A flanked by two vertical bars. Presence of these bars indicate that the module is predefined. It is analogous to predefined functions or built in functions (Of course, not always). These modules exist at the bottom level of a Structure chart.

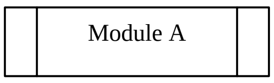


Figure 6.11: Predefined module A

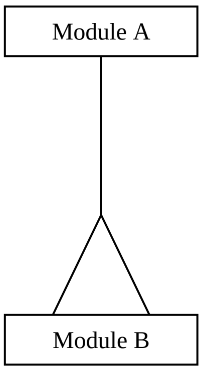


Figure 6.12: Module B is embedded in Module A

Figure 6.12 depicts the *hat* symbol for an embedded module. Though, B is logically shown separately from A, since A and B are connected by the embedded module symbol *hat*, it means that B is in fact a part of A and physically the module does not exist separately. This is often done due to the size of such modules which is very smaller and don't fit to be called a separate module. In such cases, they become part of superordinate module.

Consider the structure chart of Figure 6.13. The system module calls Get Marks A module. This module in turn calls Read Marks A . This module sends the requisite marks to Get Marks A. Then, Get Marks A calls Validate Marks A and also sends Marks A to it for validation. Validated marks A are sent back from Validate Marks A to Get Marks A. The process repeats again in the case of Marks B also. After obtaining validated marks in A and B, the system module calls Make Result R to compute the result. It sends marks in A and B to it. Make Result R sends the result R to system. Finally, the system module calls Put Result R module and passes the result to it.

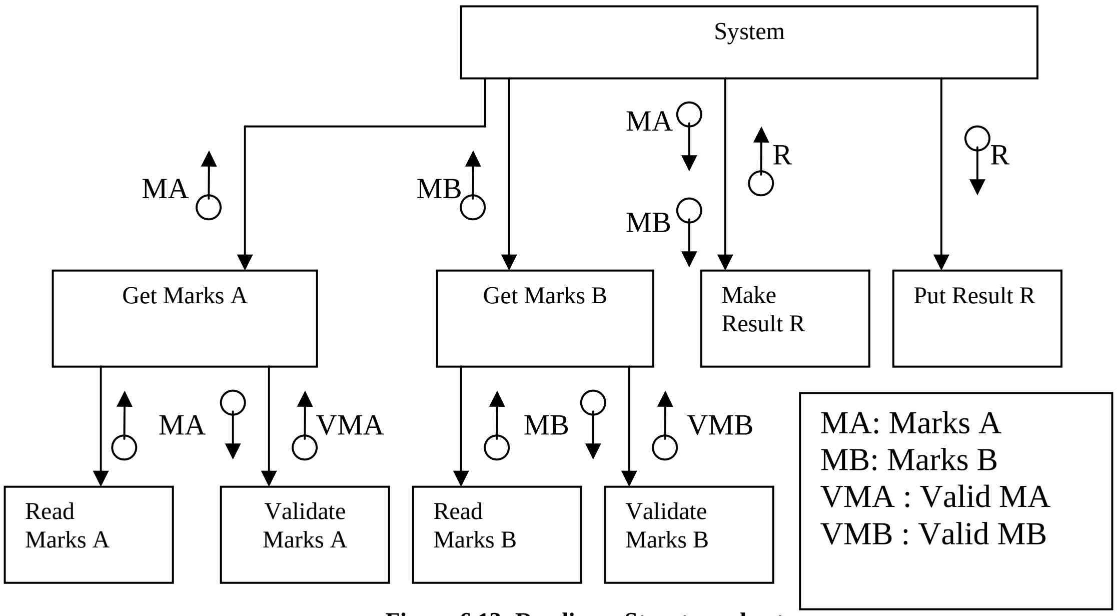


Figure 6.13: Reading a Structure chart

Check Your Progress 2

1. A ………………..…… depicts the modular organization of an information system.
2. If a menu based system is concerned, the main menu may be considered as the …………….. and the menu options in it may be considered as subordinate modules.
3. Modules at ……….. don't call any other modules.

6.4 MODULARITY

According to C. Mayers, *Modularity* is a single attribute of software that allows a program to be intellectually manageable". It increases the design clarity that results in easy implementation, testing, debugging, documentation and maintenance of software product. Modularity means "decomposing a system into smaller components that can be coded separately". Modularity does not mean simply chopping off system into smaller components but certain concepts like coupling and cohesion needs to be followed while breaking a system into different modules.

6.4.1 Goals of Design

If there is a system which can be read easily, code easily and maintain easily, then we can come to a conclusion that the design is fine. Any design which achieves the goals given below can be termed as good design:

1. The design of the system should be module based. It means there are modules which together make up the system and the organization of these modules is hierarchical.
2. Each module controls the functions of a suitable number of subordinate modules at the next hierarchical level.
3. One of the important features of good design is that the modules, which make up the system don't communicate intensively. The communication should be kept at minimum level. The reason for this imposition is that modules should be independent of each other to the maximum extent possible. Independence means, "one module's functionality should not be dependent on the internal functions of other module".
4. The size of module should be appropriate as required for the features it should possess like being relatively independent of other modules etc. Basically, no specific size of range of size can be defined on modules though it is done occasionally. The size varies from module to module and from project to project.
5. A module should not be assigned the duty of performing more than one function.
6. The coding of modules should be generic. It enables the system to use the module as frequently as possible.

Based on the above listed goals, a set of guidelines for good design can be arrived at. They are given below:

- A system should be divided into as many relatively independent modules as possible. This is known as *factoring*.
- A superordinate module should control not more than seven subordinate modules. Of course, this guideline is not strict and varies from system to system.
- The dependency levels between modules should be minimum. This automatically leads to the design of modules, which don't communicate, frequently with each other. Also, the communication between modules should be through parameters. Of course, Boolean variables or flags can be used for the purpose of communication. This is called *coupling*.
- Usually, a module if of not more than 100 lines. It may be a minimum of 50 lines. But, these sizes are not to be strictly followed and they may vary from system to system. It is notable here that lesser the lines of code, easier to read.
- A module should not perform more than one function. There should be no line in the code of the module, which is concerned with a function that is not the objective of that particular module. One easy check for this conformance is that the module's function should be describable easily in a few words. This is called *cohesion*.
- Modules at the lower level of the design are called by more than one superordinate module. It means that multiple superordinate modules use most of the modules at the lower level.

6.4.2 Coupling

The dependency levels between modules should be minimum. This automatically leads to the design of modules that don't communicate frequently with each other.

Also, the communication between modules should be through parameters. Of course, Boolean variables or flags can be used for the purpose of communication. This is called *coupling*.

The coupling between the modules should be minimum. The reason for stressing the need for minimum dependence between modules is that, if a module-1 is largely dependent on another module-2, then, any error in module-2 will affect the functionality of module-1. This is the case of two modules that are largely dependent on each other. But, in the case of multiple modules being largely dependent among themselves, the consequences of errors in one or more modules will be drastic. The other problem with the dependency of one module on another module is related to maintenance. If a programmer has to change the functionality of a module then he should also make necessary changes to the internals of the modules on which the module in question is largely dependent. It automatically leads to the disturbance of the entire system. Such modular design usually leads to the need for development of the system from the scratch which is going to have significant implications in terms of efforts to be put, amount to be spent etc. If modules are independent to the extent possible then it will become easy for the programmers to make changes in a particular module with out making any changes in other modules. Also, it leads to a greater reuse of the modules in multiple projects wherever the functionality of the module is needed. Though it is desirable, it is highly possible to minimize coupling among the modules.

There are five types of coupling. They are explained below:

Data Coupling: In this type of coupling, the communication between the modules is through passing of data as parameters. The other alternative in this type of coupling is the usage of flags. So, one module will not be and need not be aware of the internal structure of the module with which it is communicating.

Consider the Figure 6.14. **Prepare the salary of employee** is the superordinate module. **Calculate Salary** is the subordinate module. It is coupled with **Calculate Salary**. But, **Prepare the salary of employee need** not be aware of the internal structure of **Calculate Salary** . **Calculate Salary** needs to know the data being passed to it for doing the requisite task and the data that has to be returned by it to the superordinate module.

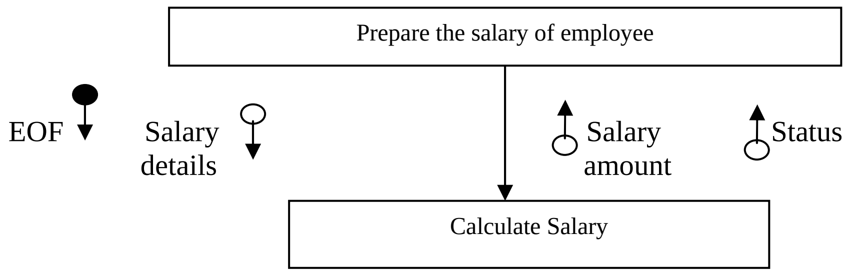


Figure 6.14: Example of Data Coupling

Stamp Coupling: The mechanism of communication in Stamp Coupling is achieved by passing data structures . Alternatively, records consisting of requisite data are sent. The problem with this type of coupling is that any changes in the data structure will lead to a chain reaction and all the modules that use this data structure have to be change. Sending data as stated in the technique of data coupling is ideal. Stamp coupling increases the dependency levels among the modules. All the modules, which are using the same data structure, should be aware of the internal functions of each other. This is required to avoid errors due to the usage of the same data structure. Since the same data structure is being used, the entire data is passed to the subordinate module, which leads to redundant increased communication and more scope for data corruption.

Figure 6.15 demonstrates Stamp coupling. Obviously, the entire **Employee record** will contain more data than data required by the **Calculate Total Salary** module. The process **Format Payslip** then uses data structure **Employee record**. Once again **Employee record** contains too much data for this process. It would be better for both superordinate, subordinate modules and for the system as a whole if only the relevant data elements were passed instead of the entire record.

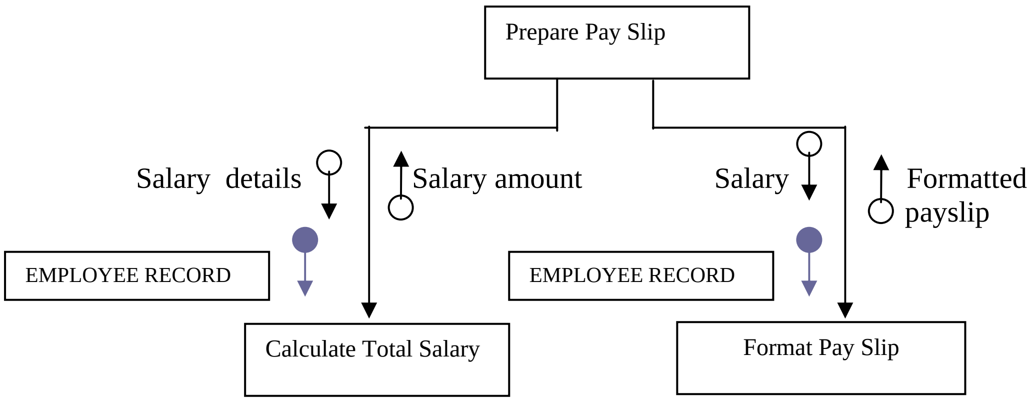


Figure 6.15: Example of Stamp Coupling

Control Coupling: In this technique of coupling, the superordinate module communicates with subordinate module by passing control information. The control information conveys the functions to the subordinate module that are to be performed by it. In this type of coupling, interdependence between the superordinate and subordinate modules is high as the superordinate module should definitely know the internal functions of the subordinate module to invoke it for a particular task. Figure 6.16 depicts an example of Control coupling. The signal that control information is being passed is that the label of the flag starts with the verb **Prepare Payslip**. It is to be noted that, in some cases, control information may be passed from the subordinate module to superordinate module. But, this rarely occurs.

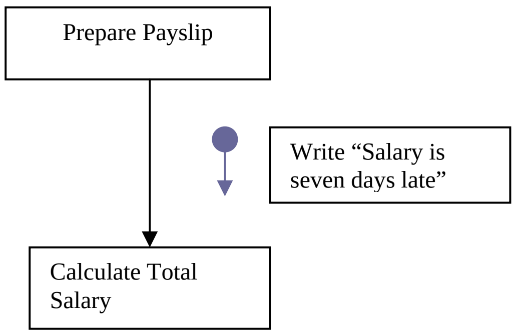


Figure 6.16: Example of Control Coupling

Common Coupling: In this technique of coupling, global data areas are used by the multiple modules. Usage of global variables is permitted in most of the High level languages. A variable can be declared at appropriate level so that it is treated as global variable. All modules which use this data area will be accessing the data in that area that is present there at that point of time. If any module performs invalid operation on the data in the global data area then the data area holds the resultant wrong value. But, this wrong value will be used by the module which subsequently accesses this global data area resulting in further invalid processing. In this type of coupling, interdependence among the modules is very high as they are sharing the data area and any wrong doing by any of the modules on this global area is going to impact the processing of all the subsequent modules which use the data in this global data area.

Content Coupling: This is the technique of coupling which has to be used when none of the above are possible. The major drawback of this technique is that one module can access the data inside another module and alter it. Also, it is possible to change the code of one module by another module. This is the technique of coupling in which independence among the modules is not even slightly visible. Fortunately, most of the High level languages don't support content coupling.

All the coupling techniques that are discussed above rate from top to bottom in terms of priority. In other words, data coupling should be most sought after technique of coupling followed by stamp coupling and then control coupling followed by common coupling and lastly content coupling.

6.4.3 Cohesion

Cohesion reflects the degree to which a module conforms itself to the performance of a single task. A simple way to check if a module is cohesive or not, is to examine each instruction in it. If every instruction is related to the performance of a single task, then the module is said to be cohesive. Modules should be highly cohesive. Two objectives can be achieved if we strive to make a module cohesive. First is that the module will perform single task. It leads to a larger degree of portability and we can directly plug in the module in an application which requires the performance of this task. The second is that module will be loosely coupled. Since the module is performing the single task, it will accept the data from a superordinate module , does the requisite function and returns the results. So, there is no need or minimum need to know the internal function of any other module.

There are seven types of cohesion. They are explained below:

Functional Cohesion: A module is functionally cohesive if every instruction in the module is related to a single task. One easy way to know whether the module is functionally cohesive or not is to examine its name. The name of the module will usually indicate the task that is performed by it. For example, Print Grade Cards, Generate payslips etc. are names of modules that perform a single function.

Sequential Cohesion: In this type of cohesion, all instructions in the module are related to each other through the data that is passed to the module. If each instruction is examined individually, it is difficult to know whether the module is performing single function or not. But, if the module is simulated and instruction wise simulation is examined, then we can conclude that the module is sequentially cohesive if each instruction's input data is the output data of the previous immediate instruction. In other words, the concept of sequential cohesion is similar to the concept of pipeline processing. So, in sequential cohesion, sequencing of instructions plays a major role in the cohesiveness of the module.

Consider the following example of sequential cohesion:

Produce purchase order,
Prepare shipping order,
Update inventory,
Update accounts.

Purchase order is the initial input for this set of instructions. Produce purchase order is the input to the second instruction where the shipping order is prepared. This instruction will serve as an input to the third instruction to update inventory and this will serve as input to update accounts. So, in this way, the output of first instruction has become the input for the second instruction, the output of the second instruction has become the input for the third instruction and so on.

Communicational cohesion: This type of cohesion shares an analogy with sequential cohesion regarding the aspect that all instructions in the module are related by the data used by the module. But, at the same time, it differs from the sequential cohesion with no restriction on the sequencing pattern of the instructions. So, in communicational cohesion, the ordering of instructions is irrelevant. The most important thing is that, input data for each instruction is same.

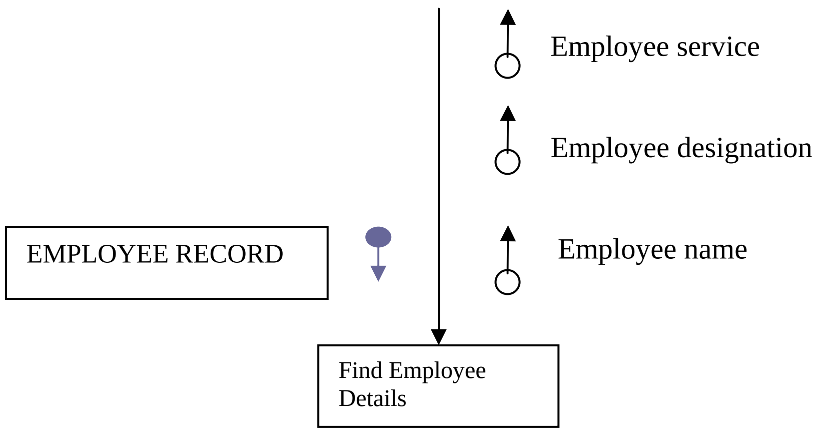


Figure 6.17: An example of a Communicational Cohesion module

Figure 6.17 depicts an example of Communicational Cohesion. **Find Employee Details** is a subordinate module which receives an *Employee* record as input from the superordinate module and finds the employee's name, designation and service. To find each of name, designation and service, *Employee* record is used. So, the input for all the three instructions is same. Also, sequencing does not matter here because any of the name, designation and service details can be found at any point in the order and the input to any of these instructions is not dependent on the output of the other instructions.

It is also possible to use two functionally cohesive modules than one communicational cohesive module. Figure 6.18 depicts two functionally cohesive modules instead of one communicational cohesive module in Figure 6.17.

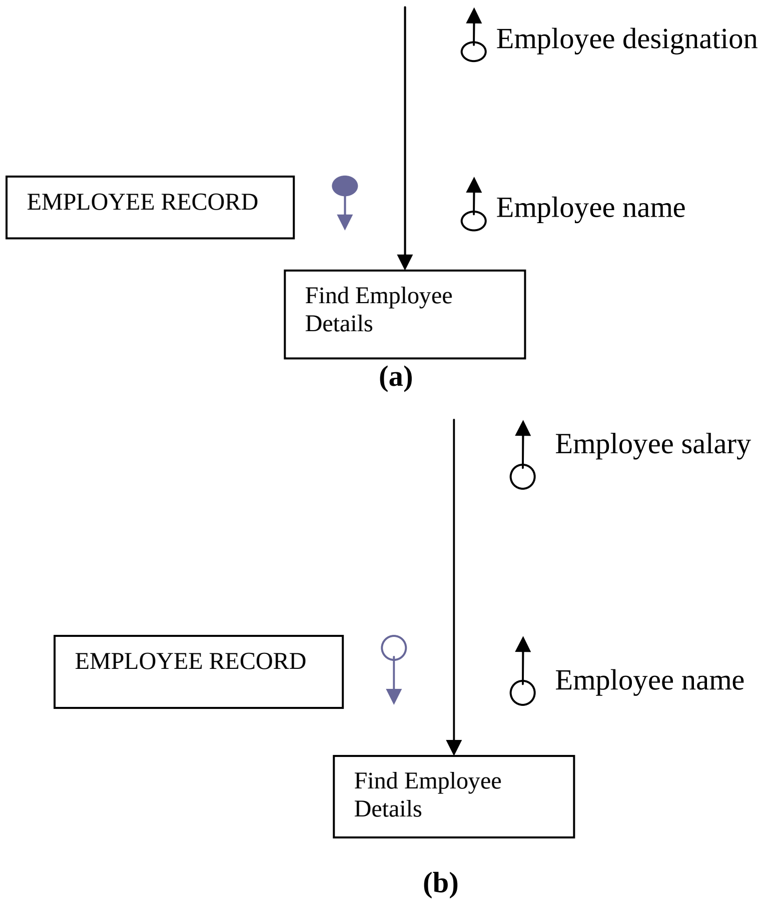


Figure 6.18 (a) & (b): An example of two functionally cohesive modules which resulted due to the split of one communicational cohesive module

Procedural Cohesion: Any module which is not functionally cohesive is difficult to maintain. In this type of cohesion, the sequence of instructions is important and they are related to each other by the control flow. It is possible to make a change in sequence, but it cannot be arbitrarily done. The execution of instructions in the module which is procedurally cohesive usually leads to the calls to other modules. So, the instructions in a procedurally cohesive module are related to the instructions in other modules.

Consider the following example:

Pick the course material from MPDD,
Check the enrolment number on the Hall ticket,
Attend counselling sessions at Study Centre,
Submit assignments at Study Centre.

In the above example, instructions are not related to each other in terms of sequence. In some cases, sequence may be of importance. But, at the same time, each instruction is separate in functionality and leads to the execution of instructions in other modules.

Temporal Cohesion: In this type of cohesion, instructions in a module are related to each other only by flow of control and are totally unrelated to their sequence. In a temporally cohesive module, execution of all the instructions may take place at a time.

Consider the following example:

Delete duplicate data from inventory file,
Reindex inventory file,
Backup of inventory file.

All these operations have nothing in common except that they are end of the day clean up activities.

Logical Cohesion: In this type of cohesion, the relation between various instructions in the module is either nil or at a bare minimum. A logically cohesive module consists of instructions in the form of sets. So, execution takes place in terms of set of instructions rather than individual instructions. But, the superordinate module which calls the logically cohesive subordinate module will determine the set of instructions to be executed. This mechanism is handled with the help of a flag which is passed to the subordinate module by the superordinate module indicating the set of instructions to be executed.
Consider the following example:

Study in home,
Study in library,
Study in garden.

This is bad type of cohesion. It is very difficult to maintain logically cohesive modules.

Coincidental Cohesion: In this type of cohesion, there is no relationship between the instructions. This is worse type of cohesion among the discussed. The reason for placing such type of totally unrelated instructions may be to save time from programming, to fix errors in the existing modules etc.

The priority of all the seven types of cohesion discussed moves from top to bottom. So, Functional cohesion is the best and Coincidental cohesion is the worse type of cohesions. The order of priority is as follows: Functional cohesion, Sequential cohesion, Communicational cohesion, Procedural cohesion, Temporal cohesion, Logical cohesion and Coincidental cohesion.

Check Your Progress 3

1. A module should not be assigned the duty of performing more than ……. function.
2. The coupling between the modules should be ………..
3. ……………. is the best and ………… is the worse type of cohesions.

6.5 SUMMARY

This unit focussed on the requirements of a good design. We have studies various principles of good design. The two design techniques, namely, Top Down Design and Bottom Up Design have been explained. The process of depicting the modular organization of a system has been explained using Structure charts. Different symbols used in Structure charts have been discussed. An example structure chart for reading the marks in various courses and computing the result has been demonstrated. The issue of the degree of communication that should be present between various modules has been discussed through Coupling and Cohesion. There are 5 types of coupling namely Data coupling, Stamp coupling, Control coupling, Common coupling and Content coupling. There are a total of 7 types of cohesion namely, Functional cohesion, Sequential cohesion, Communicational cohesion, Procedural cohesion, Temporal cohesion, Logical cohesion and Coincidental cohesion.

6.6 SOLUTIONS/ANSWERS

Check Your Progress 1

1. Abstraction
2. Top Down Design
3. Bottom Up Design

Check Your Progress 2

1. Structure Charts
2. Coordinating module , Subordinate modules
3. Lower level

Check Your Progress 3

1. One
2. Minimum
3. Functional cohesion, Coincidental cohesion

6.7 FURTHER READINGS

Joey George, J. Hoffer and Joseph Valacich; *Modern Systems Analysis and Design*; Pearson Education;Third Edition;2001.

Alan Dennis, Barbara Haley Wixom; *Systems Analysis and Design;*John Wiley & Sons;2002.

Reference Websites

http://www.rspa.com