Control Hierarchy

Control hierarchy, or program structure, represents the program components' organization and the hierarchy of control. Different notations are used to represent the control hierarchy, the most common of these being a tree-like diagram. Depth of a tree describes the number of levels of control and width refers to the span of control. Fan-out is the measure of the number of modules that are directly controlled by a module. Fan-in indicates the number of modules that directly control a particular module. A module being controlled is called a subordinate and the controlling module is called a superordinate.

The control hierarchy defines two other characteristics of the software architecture. Visibility indicates the set of program components that will be invoked or used directly by a component. Connectivity refers to the set of components that are invoked or used directly by a program component.

Structural Partitioning

The hierarchical style of the system allows partitioning of the program structure, both vertically and horizontally. Horizontal partitioning defines a separate branch for each major program function. Control modules are used to coordinate between the modules for communication and execution. Horizontal partitioning produces three things as a result: input, data transformation and output. Partitioning has the following advantages:

- Software becomes easier to test.
- Software is easier to maintain.
- Low side-effects propagation.
- Software becomes easy to extend.

Disadvantage of the horizontal partitioning is that it causes a lot of data to be transferred among the various modules which further complicates the control flow.

Vertical partitioning, also called factoring, separates the control and the flow in a top-down manner. Top level modules perform only control functions while the low-level modules perform the actual processing, input and output tasks. Thus, a change made in the control model has higher chances of being propagated to the subordinate modules. However, the changes in the lower level modules are not likely to be propagated to the higher level modules. Thus, the vertically partitioned programs are easily maintainable.

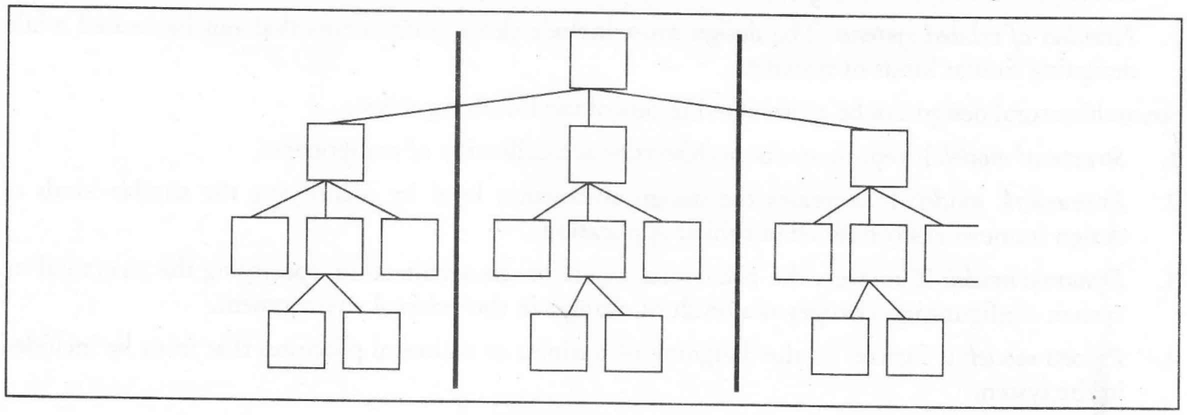

Figure 5.1(a): Horizontal Partitioning

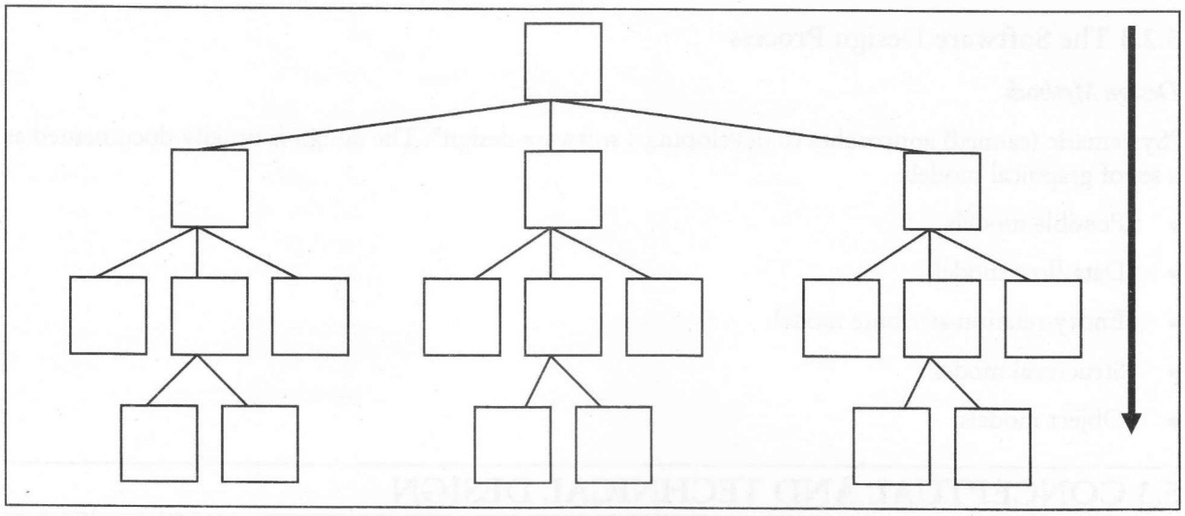

Figure 5.1(b): Vertical Partitioning

Information Hiding

In order to design best set of modules out of a single software solution, the concept of information hiding is useful. The principle of information hiding suggests that modules should be characterized by design decision that hides each one from the other. In other words, modules should be designed in such a way that the information in one module is not accessible to other modules that do not require these modules.

Hiding ensures that effective modularity is achieved by defining a set of independent modules that communicate with one another only the information that is needed for proper software functioning. Abstraction defines the procedural entities for the software while hiding defines the access constraints to procedural details and local data present in the modules.

Design Documentation

The design specification contains various aspects of the design model and is completed as the designer improves his software representation. At first, the design specification derives its scope from System specification and the analysis model.

At the next level data design is specified. The data design includes database structure, external file structure, internal data structures and reference connecting data objects to files.

The design specification contains requirements cross reference represented as a simple matrix. The purpose of this matrix is: (i) to ensure that the design includes all the requirements and (ii) to indicate the critical components for the implementation purpose.

The design document may also contain the first stage of test documentation. After the program structure and the interfaces have been established, guidelines to test individual and integrated modules are developed.

Design constraints such as limited physical memory or special external interface can cause special design techniques to evolve which in turn will lead to changes in software package.

The final section of the design specification contains supplementary data such as algorithm description, tabular data, excerpts from other documents, etc.

5.2.3 The Software Design Process

Design Methods

"Systematic (canned) approaches to developing a software design". The design is usually documented as a set of graphical models:

- Possible models
- Data-flow model
- Entity-relation-attribute model
- Structural model
- Object models

5.3 CONCEPTUAL AND TECHNICAL DESIGN

The process of software design involves the transformation of ideas into detailed implementation descriptions, with the goal of satisfying software requirements. To transform the requirements into working systems, designer must satisfy the customers and the system builders (coding persons. The customers understand what the system is to do. At the same time the system builder must understand how the system is to work. For this reason design is really a two part, iterative process. First, we produce the conceptual design hat tells the customer exactly what the system will do. Once the customer approves the conceptual design, we translate the conceptual design into much more detailed document, the technical design that allows the system builders to understand the actual hardware and software needed to solve the customer's problems. This two part design process is shown in Figure 5.2.

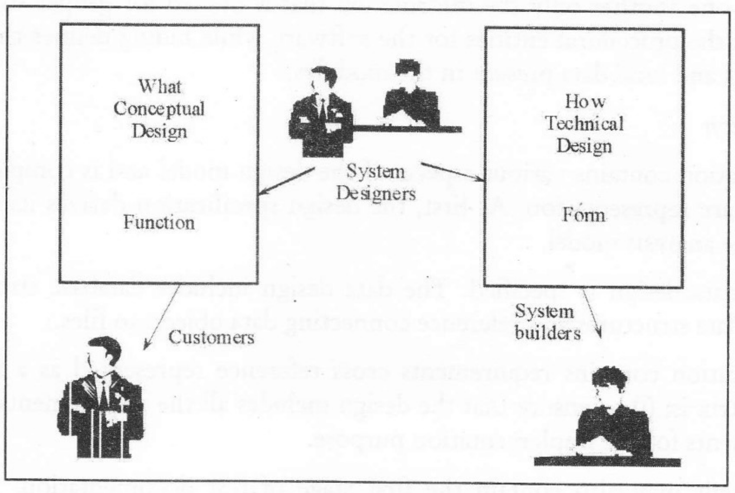

Figure 5.2: Two Part Design Process

The two design documents describes the same system, but in different ways because of the different audiences for the documents. The conceptual design answers the following questions.

- Where will the data come from?
- What will happen to the data in the system?

- How will system look to the users?
- What choices will be offered to the user?
- What is the timing of the events?
- How will the reports and screens look like?

The conceptual design describes the system in language understandable to the customers. It does not contains any technical jargons and is independent of implementations.

By contrast, technical design describes the hardware configuration, the software needs,the communication interfaces, the input output of the system, the network architecture, and anything else that translates the requirement into solution to the customer's problem.

5.4 MODULARITY

Modularity refers to the division of software into separate modules which are differently named and addressed and are integrated later on in order to obtain the completely functional software. It is the only property that allows a program to be intellectually manageable. Single large programs are difficult to understand and read due to large number of reference variables, control paths, global variables, etc. The desirable properties of a modular system are:

- Each module is a well defined system that can be used with other applications.
- Each module has a single specific purpose.
- Modules can be separately compiled and stored in a library.
- Modules can use other modules.
- Modules should be easier to use than to build.
- Modules are simpler from outside than inside.

Modularity thus enhances the clarity of design which in turn eases coding, testing, debugging, documenting and maintenance of the software product. It might seem to you that on dividing a problem into sub problems indefinitely, the effort required to develop it becomes negligibly small. However, the fact is that on dividing the program into numerous small modules, the effort associated with the integration of these modules grows. Thus, there is a number N of modules that result in the minimum development cost. However, there is no defined way to predict the value of this N.

In order to define a proper size of the modules, we need to define an effective design method to develop a modular system. Following are some of the criteria defined by Meyer for the same:

(a) ***Modular decomposability:*** The overall complexity of the program will get reduced if the design provides a systematic mechanism to decompose the problem into sub-problems and will also lead to an efficient modular design.

(b) ***Modular composability:*** If a design method involves using the existing design components while creating a new system it will lead to a solution that does not re-invent the wheel.

(c) ***Modular understandability:*** If a module can be understood as a separate standalone unit without referring to other modules it will be easier to build and edit.

(d) ***Modular continuity:*** If a change made in one module does not require changing all the modules involved in the system, the impact of change-induced side effects gets minimized.

(e) ***Modular protection:*** If an unusual event occurs affecting a module and it does not affect other modules, the impact of error-induced side effects will be minimized.

5.4.1 Modular Design

A modular design reduces the design complexity and results in easier and faster implementation by allowing parallel development of different parts of a system. We discuss the various concepts of modular design in detail in this section.

Functional Independence

This concept evolves directly from the concepts of abstraction, information hiding and modularity. Functional independence is achieved by developing functions that perform only one kind of task and do not excessively interact with other modules.

Independence is important because it makes implementation easier and faster. The independent modules are easier to maintain, test, and reduce error propagation and can be reused in other programs as well. Thus, functional independence is a good design feature which ensures software quality. It is measured using two criteria: coupling and cohesion.

Module Coupling

Coupling is the measure of degree of interdependence amongst modules. Two modules that are tightly coupled are strongly dependent on each other. However, two modules that are loosely coupled are not dependent on each other. Uncoupled modules have no interdependence at all within them. The various types of coupling techniques are depicted in Figure 5.3.

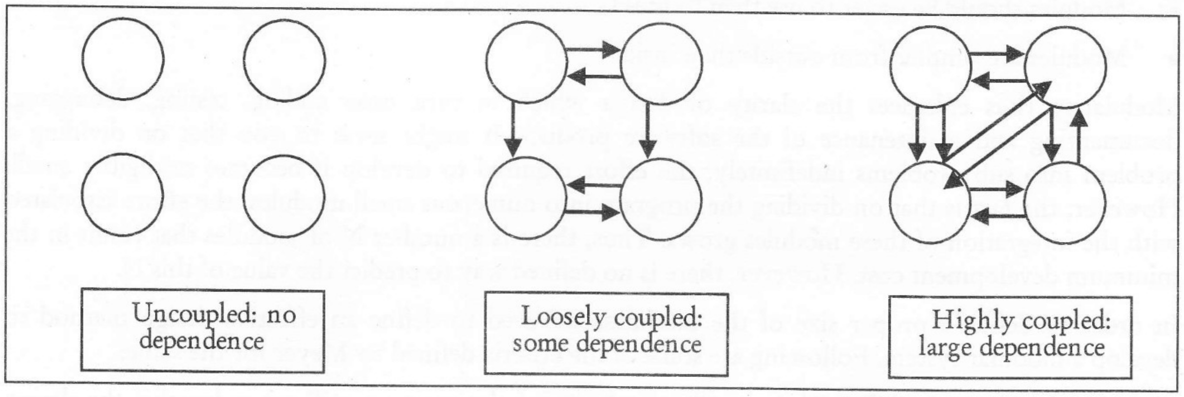
Uncoupled: no dependence
Loosely coupled: some dependence
Highly coupled: large dependence

Figure 5.3: Module Coupling

A good design is the one that has low coupling. Coupling is measured by the number of interconnections between the modules. I.e. the coupling increases as the number of calls between modules increase or the amount of shared data is large. Thus, it can be said that a design with high coupling will have more errors. Different types of coupling are content, common, external, control, stamp and data. The level of coupling in each of these types is given in Figure 5.4.

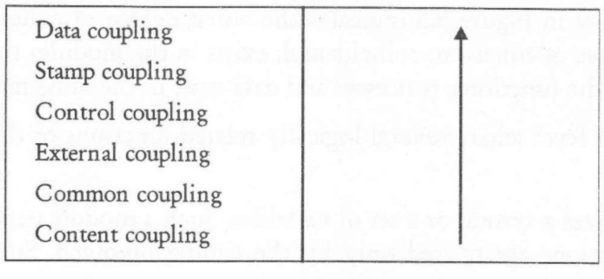

Figure 5.4: Types of Modules Coupling

The direction of the arrow in Figure 5.4 points from the lowest coupling to highest coupling. The strength of coupling is influenced by the complexity of the modules, the type of connection and the type of communication. Modifications done in the data of one block may require changes in other block of a different module which is coupled to the former module. However, if the communication takes place in the form of parameters then the internal details of the modules are no required to be modified while making changes in the related module.

Given two procedures X and Y, the type of coupling can be identified in them.

1. *Data coupling:* When X and Y communicates by passing parameters to one another and not unnecessary data. Thus, if a procedure needs a part of a data structure, it should be passed just that and not the complete thing.
2. *Stamp Coupling:* Although X and Y make use of the same data type but perform different operations on them.
3. *Control Coupling (activating):* X transfers control to Y through procedure calls.
4. *Common Coupling:* Both X and Y use some shared data e.g. global variables. This is the most undesirable, because if we wish to change the shared data, all the procedures accessing this shared data will need to be modified.
5. *Content Coupling:* When X modifies Y either by branching in the middle of Y or by changing the local data values or instructions of Y.

Cohesion

Cohesion is the measure of the degree of functional dependence of modules. A strongly cohesive module implements functionality that interacts little with the other modules. Thus, in order to achieve higher interaction amongst modules a higher cohesion is desired. Different types of cohesion are listed in Figure 5.5.

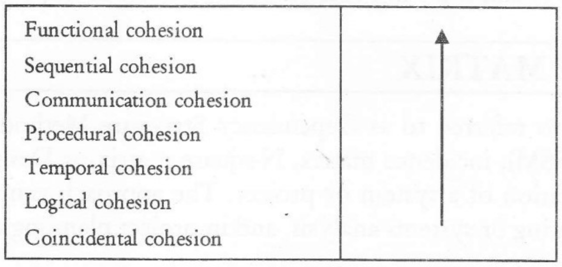

Figure 5.5: Types of Module Cohesion

The direction of the arrow in Figure 5.5 indicates the worst degree of cohesion to the best degree of cohesion. The worst degree of cohesion, coincidental, exists in the modules that are not related to each other in any way. So, all the functions, processes and data exist in the same module.

Logical is the next higher level where several logically related functions or data are placed in the same module.

At times a module initializes a system or a set of variables. Such a module performs several functions in sequence, but these functions are related only by the timing involved. Such cohesion is said to be temporal.

When the functions are grouped in a module to ensure that they are performed in a particular order, the module is said to be procedurally cohesive. Alternatively, some functions that use the same data set and produce the same output are combined in one module. This is known as communicational cohesion.

If the output of one part of a module acts as an input to the next part, the module is said to have sequential cohesion. Because the module is in the process of construction, it is possible that it does not contain all the processes of a function. The most ideal of all module cohesion techniques is functional cohesion. Here, every processing element is important to the function and all the important functions are contained in the same module. A functionally cohesive function performs only one kind of function and only that kind of function.

Given a procedure carrying out operations A and B, we can identify various forms of cohesion between A and B:

1. *Functional Cohesion:* A and B are part of one function and hence, are contained in the same procedure.
2. *Sequential Cohesion:* A produces an output that is used as input to B. Thus they can be a part of the same procedure.
3. *Communicational Cohesion:* A and B take the same input or generate the same output. They can be parts of different procedures.
4. *Procedural Cohesion:* A and B are structured in the similar manner. Thus, it is not advisable to put them both in the same procedure.
5. *Temporal Cohesion:* Both A and B are performed at moreover the same time. Thus, they cannot necessarily be put in the same procedure because they may or may not be performed at once.
6. *Logical Cohesion:* A and B perform logically similar operations.
7. *Coincidental Cohesion:* A and B are not conceptually related but share the same code.

5.5 DEPENDENCE MATRIX

A **Dependency Matrix** also referred to as Dependency Structure Method, Design Structure Matrix, Problem Solving Matrix (PSM), incidence matrix, N-square matrix or Design Precedence Matrix), is a compact, matrix representation of a system or project. The approach can be used to model complex systems in systems engineering or systems analysis, and in project planning and project management.

A dependency matrix lists all constituent subsystems/activities and the corresponding information exchange and dependency patterns. In other words, it details what pieces of information are needed to start a particular activity, and shows where the information generated by that activity leads. In this way, one can quickly recognize which other tasks are reliant upon information outputs generated by each activity.

DM analysis provides insights into how to manage complex systems or projects, highlighting information flows, task sequences and iteration. It can help teams to streamline their processes based on the optimal flow of information between different interdependent activities.

DM analysis can also be used to manage the effects of change. For example, if the specification for a component had to be changed, it would be possible to quickly identify all processes or activities which had been dependent on that specification, reducing the risk that work continues based on out-of-date information.

DM is used in the Analytical Design Planning Technique, an engineering design planning and control approach.

A DM represents dependencies amongst the *design parameters* of a design. A design parameter represents the range of choices that can be made about an aspect of a design. Typical software design parameters include data structures, algorithms, and procedure and type signatures. Others might include graphical interface look-and-feel, interoperability, and performance characteristics. The rows and columns of a DSM are labeled by the design parameters. Dependence between two parameters is represented by a *mark* (X). A mark in row B, column A means that an efficacious choice for B depends on the choice for A. Parameters that require mutual consistency—algorithm and data structures go hand-in-hand, for example—are *interdependent*, resulting in symmetric (A,B) and (B,A) marks. When one parameter naturally precedes the other—there is no GUI lookand- feel unless there is a GUI—the parameters are called *hierarchically dependent*. Finally, *independent* design parameters may be determined and changed individually. Choosing the design parameters to model and the values they finally take on is the task of the designer. In exploring the value of a design, the marks in a DM represent the believed likelihood of deriving a benefit from recognizing and allowing parameter dependencies. DM's are typically derived from experience with prior versions of the product or similar products. In early versions of a product, there may be relatively few marks in the matrix, capturing little more than the dependencies necessary to deploy a correctly functioning product. A DM for a subsequent version might include marks for subtle but powerful dependencies that were discovered through use of the product, additions to knowledge, and innovation. Likewise, new design parameters will be recognized and added to the DM over time. In these respects a DM represents not just abstract design dependencies, but concrete requirements for communication.

Check Your Progress

1. A good design must be understandable only by the designers and not developers and testers. (True/False)
2. The design should be traceable to the analysis model. (True/False)
3. At the highest level of abstraction the solution is mentioned at a very narrow level in terms of the problem. (True/False)
4. .... is the measure of the number of modules that are directly controlled by a module.

5.6 LET US SUM UP

Design is the technical kernel of software engineering. Design produces representation of software that can be assessed for quality. During the design process continuous refinement of data structures, architecture, interfaces, etc of software are developed, documented and reviewed.

A large number of design principles and concepts have developed over the last few decades. Modularity and abstraction allow the designers to simplify the software and make it reusable. Information hiding and functional independence provide heuristics for achieving efficient modularity.

5.7 KEYWORDS

STD: State Transition Diagram

CSPEC: Control Specification

PSPEC: Process Specification

DM: Dependence Matrix

5.8 QUESTIONS FOR DISCUSSION

1. What is modularity? List the important properties of a modular system.
2. Describe information hiding with appropriate examples.
3. Discuss how structural partitioning makes a program maintainable.
4. Discuss the relation between information hiding and module independence.
5. What is dependence matrix?
6. What is the difference between conceptual and technical design?

Check Your Progress: Model Answers

1. False
2. True
3. False
4. Fan-out

5.9 SUGGESTED READINGS

R.S. Pressman, *Software Engineering- A Practitioner's Approach*, 5th Edition, Tata McGraw Hill Higher education.

Belady L, *Foreword to Software Design: Methods and Techniques*, Yourdon Press, 1981.

Bass, P Clements and R Kazman, *Software Architecture in Practice*, Addison-Wesley, 1998.

Sommerville, *Software Engineering*, Addison-Wesley, 3rd Edition, 1989.

LESSON

6

STRATEGY OF DESIGN

CONTENTS

6.0 AIMS AND OBJECTIVES

After studying this lesson, you should be able to:

- Explain various design strategies
- Define function oriented design
- Define object oriented design
- Compare the various design techniques

6.1 INTRODUCTION

To transform requirements into a working system, designers must satisfy both the customers and the builders. The customer tells what the system is intended to do whereas the builder tells how the system will work. Thus, the design is split into two parts: Conceptual design and technical design. At first, the designer comes up with the conceptual design that tells the customer what exactly the system will do. On approval of this design from the customer it is converted to a detailed design, i.e. the technical design which allows the developers to understand the actual hardware and software requirements.

We have understood the conceptual techniques in the previous lesson and will discuss the various technical design techniques in detail here.

6.2 STRATEGY OF DESIGN

In order to reduce the time required to come up with the code from the design, a number of designing techniques have been developed. These work at a relatively higher level and permit multiple levels of design. Some of the design strategies are:

Bottom-up design: This approach works by first identifying the various modules that are required by many programs. These modules are grouped together in the form of a library. Thus we need to decide about the approach to combine these modules to produce larger and even larger modules till the time the entire program is obtained. Since the design progresses from the bottom layers toward the top, it is called the bottom-up design. The chances of re-coding a module are higher in this technique of design if we start the coding soon after the design. But the coded module is tested and design can be verified comparatively sooner than the modules which have not yet been designed. This method, however, involves a lot of intuition work to be able to think about the total functionality of the module at an early stage. In case of a mistake at a higher level the rework needs to be done right from the lower level. However, this technique is more suitable for the system that is being developed from some existing modules.

Top-down design: The main idea behind this technique is that the specification is viewed as a black-box program and the designer needs to decide the inner functioning of the black boxes from smaller black boxes. This process is repeated till the time the black boxes can be directly coded. The approach begins by identifying the major modules, decomposing them into smaller modules and iterating till the desired level is achieved. This methodology is suitable if all the specifications are clear in the beginning and the development is from the scratch.

Hybrid design: Pure top-down and bottom-up approaches are not practical as both the approaches have their own shortcomings as discussed above. This has lead to the evolution of a hybrid approach of design based upon the reusability of the modules.

6.3 FUNCTION ORIENTED DESIGN

Function oriented design is an approach of software design where the design is divided into a number of interacting units where each unit has a clearly defined purpose. Thus, a system is designed from functional viewpoint.

This approach was advocated by Niklaus Wirth who called it a stepwise refinement. It is a top-down approach. We begin with a high level design of what the program does and then later on, this is refined at each stage giving grater details of each of these parts.

It works fine for smaller programs. Because it is not easy to know what a large program does, it's not useful for large programs.

The refinement of modules continues till the point they can be easily coded in a programming language. Because the program is being developed in a top-down manner, the modules are highly specialized. Each module is being used by at most one other modules i.e. its parent. For a module to be reusable it must conform to the design reusable structure given in Figure 6.1.

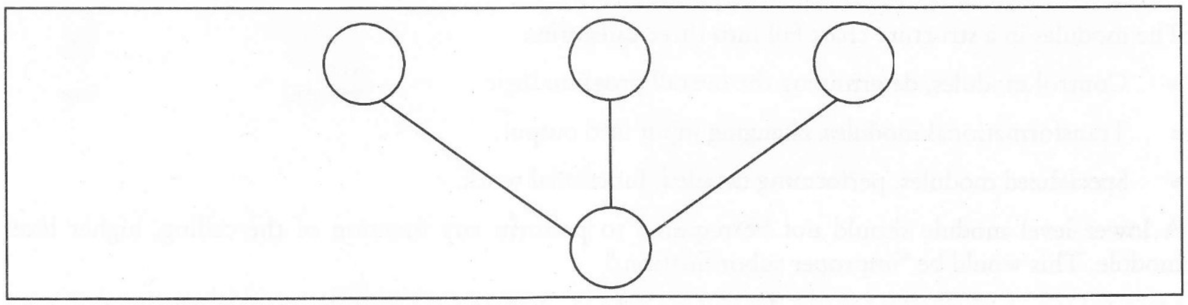

Figure 6.1: Design Reusability Structure

Although the program is function-oriented, it should not necessarily be created in a top-down manner. If we wish to delay the decision of what the program is supposed to do, a better choice is to structure the program around the data rather than the actions taken by the program.

6.3.1 Design Notations

Design notations are meant to be used during the design phase to represent design and design decisions. A function-oriented design can be represented graphically or mathematically using the following:

- Data flow diagrams
- Data dictionaries
- Structure charts
- Pseudo code

The first two techniques have been discussed in detail in the lesson 4 and the other two are discussed below:

Structure Charts

A structure chart is a diagram consisting of rectangular boxes representing modules and connecting arrows. Structure charts encourage top-down structured design and modularity. Top-down structured design deals with the size and complexity of an application by breaking it up into a hierarchy of modules that result in an application that is easier to implement and maintain.

Top-down design allows the systems analyst to judge the overall organizational objectives and how they can be met in an overall system. Then, the analyst moves to dividing that system into subsystems and their requirements. The modular programming concept is useful for the top-down approach: once the top-down approach is taken, the whole system is broken into logical, manageable-sized modules, or subprograms.

Modular design is the decomposition of a program or application into modules. A module is a group of executable instructions (code) with a single point of entry and a single point of exit. A module could be a subroutine, subprogram, or main program. It also could be a smaller unit of measure such as a paragraph in a COBOL program.

Data passed between structure chart modules has either Data Coupling where only the data required by the module is passed or Stamp Coupling where more data than necessary is passed between modules.

The modules in a structure chart fall into three categories:

- Control modules, determining the overall program logic.
- Transformational modules, changing input into output.
- Specialized modules, performing detailed, functional work.

A lower level module should not be required to perform any function of the calling, higher level module. This would be "improper subordination."

Modules are represented by rectangles or boxes that include the name of the module. The highest level module is called the system, root, supervisor, or executive module. It calls the modules directly beneath it which in turn call the modules beneath them.

A connection is represented by an arrow and denotes the calling of one module by another. The arrow points from the calling (higher) module to the called (subordinate) module.

Note: The structure charts drawn in the Kendall and Kendall text book do not include the arrowhead on the connections between modules. Kendall and Kendall draw plain lines between module boxes.

A data couple indicates that a data field is passed from one module to another for operation and is depicted by an arrow with an open circle at the end.

A flag, or control couple, is a data field (message, control parameter) that is passed from one module to another and tested to determine some condition. Control flags are depicted by an arrow with a darkened circle at the end. Sometimes a distinction is made between a control switch (which may have two values, e.g., yes-no, on-off, one-zero, etc.) and a control flag (which may have more than two values).

Modules that perform input are referred to as afferent while those that produce output are called efferent.

A structure chart is a graphic tool that shows the hierarchy of program modules and interfaces between them. Structure charts include annotations for data flowing between modules. An arrow from a module P to module Q represents that module P invokes module Q. Q is called the subordinate of P and P is called the super ordinate of Q. The arrow is labeled by the parameters receives as inputs by Q and the parameters returned as output by Q with the appropriate direction of flow. E.g.

```
void main()
{
    avg=calculate_average(a,n);
    product=calculate_product(x,y);
}

float calculate_average( float *x, int size)
{
    float sum=0.0;
    int I;
    for (i=0;i<size;i++)
    sum = sum + x[i];
    return (sum/size);
}
int calculate_product( int a, int b)
{
    return(a * b);
}
```

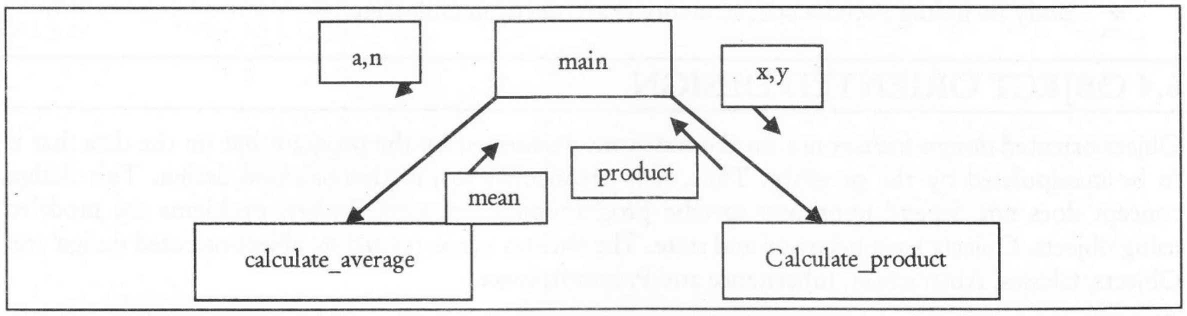

Figure 6.2: Structure Chart of the Program Above

Pseudocode

Pseudocode notation can be used both at the preliminary stage as well as the advanced stage. It can be used at any abstraction level. In a Pseudocode, a designer describes system characteristics using short English language phrases with the help of keywords like while, if..the..else, End, etc. Keywords and indentation describe the flow of control while the English phrases describe the processing action that is being taken. Using the top-down strategy, each English phrase is expanded into a more detailed Pseudocode till the point it reaches the level of implementation.

Functional Procedural Layers

- Functions are built in layers. Layers are used to depict additional information.
- *Level 0:*
  - ❖ Function or procedure name
  - ❖ Relationship with other systems
  - ❖ Description of the function purpose
  - ❖ Author, date
- *Level 1:*
  - ❖ Functional parameters
  - ❖ Global variables
  - ❖ Routines called by functions
  - ❖ Side effects
  - ❖ Input/output assertions

- *Level 2:*
  - Local data structures, variables, etc
  - Timing constraints
  - Exception handling
  - Any other limitations
- *Level 3:*
  - Body including Pseudocode, structure chart, decision tables, etc.

6.4 OBJECT ORIENTED DESIGN

Object-oriented design focuses not on the functions performed by the program but on the data that is to be manipulated by the program. Thus, it is orthogonal to function-oriented design. This design concept does not depend upon any specific programming language. Rather, problems are modeled using objects. Objects have behavior and state. The various terms related to object-oriented design are: Objects, Classes, Abstraction, Inheritance and Polymorphism.

- *Objects:* An object is an entity which has a state and a defined set of operations which operate on that state. The state is represented as a collection of attributes. The operations on these objects are performed when requested from some other object. Objects communicate by passing messages to each other which initiate operations on objects.
- *Messages:* Objects communicate through messages. A message comprises of the identity of the target object, the name of the operation and any operator needed to perform this operation. In a distributed system, messages are in the form of text which is exchanged among the objects. Messages are implemented as procedures or function calls.
- *Abstraction:* Abstraction is the elimination of the irrelevant information and the amplification of the essentials. It is discussed in detail in lesson 5.
- *Class:* It is a set of objects that share a common structure and behavior. They act as blueprint for objects. E.g. a circle class has attributes: center and radius. It has two operations: area and circumference.
- *Inheritance:* The property of extending the features of one class to another class is called inheritance. The low level classes are called subclasses or derived classes and the high level class is called the super or base class. The derived class inherits state and behavior from the super class. E.g. A shape class is a super class for two derived classes: circle and triangle.
- *Polymorphism:* When we abstract only the interface of an operation and leave the implementation to subclasses, it is called polymorphic operation and the process is called polymorphism.

Why is Object Orientation better?

Object oriented technology produces better solution than its structures counterpart. This is because these systems are easier to adapt to changing requirements, easier to maintain, promote design and code re-use to a larger extent. The reasons behind this are:

- Object orientation works at a higher level of abstraction.

- The object oriented software life cycle does not require vaulting between phases.
- The data is more stable than the functionality it supports.
- It encourages good programming techniques.
- It promotes code and design re-use.

6.4.1 Comparison of Design Notations

Design notations must lead to representation of the procedures that re easy to understand and review. It must also increase the "to code" ability as a result of the design. Finally, the design must be easily maintainable.

The following attributes of design notations have been developed in the general context of notations described previously:

Modularity: Design notations must support modular development of software and must provide for interface specification.

Overall simplicity: They should be simple to learn, use and read..

Ease of editing: The procedural design should be easily editable further facilitating each software engineering task.

Maintainability: Because software maintenance is the most important and costliest of all the phases, it must be easy to maintain the software because it ultimately means maintenance of software design.

Data representation: The ability to represent local and global data in an essential element of component level design. Ideally, a design notation should represent such data directly.

Logic verifiability: Verification of design logic is the foremost goal during software testing to improve testing adequacy.

Allows designers to think in terms of interacting objects that maintain their own state and provide operations on that state instead of a set of functions operating on shared data.

Objects hide information about the representation of state and hence limit access to it. Objects may be distributed and may work sequentially or in parallel.

A design strategy based on "information hiding"...

- A useful definition of information hiding:
- Potentially changeable design decisions are isolated (i.e., "hidden") to minimize the impact of change.

—David Parnas

Advantages of OOD

- Objects may be understood as stand-alone entities.
- Objects are appropriate reusable components.

For some systems, there is an obvious mapping from real world entities to system objects.

Check Your Progress

1. The property of extending the features of one class to another class is called ....
2. Modules that perform input are referred to .... as while those that produce output are called ....
3. Object oriented technology produces systems that are easier to adapt to changing requirements. (True/False)

6.5 LET US SUM UP

The design process comprises of the activities that reduce the level of abstraction. Structured programming allows the designer to define algorithms which are less complex, easier to read, test and maintain.

Function oriented design is an approach of software design where the design is divided into a number of interacting units where each unit has a clearly defined purpose. Thus, a system is designed from functional viewpoint. Object-oriented design focuses not on the functions performed by the program but on the data that is to be manipulated by the program. Thus, it is orthogonal to function-oriented design. This design concept does not depend upon any specific programming language. Rather, problems are modeled using objects. Objects have behavior and state. The various terms related to object-oriented design are: Objects, Classes, Abstraction, Inheritance and Polymorphism.

6.6 KEYWORDS

OOD: Object Oriented Design

FOD: Function Oriented Design

Bottom-up Design: This approach works by first identifying the various modules that are required by many programs.

Top-down Design: The main idea behind this technique is that the specification is viewed as a black-box program.

6.7 QUESTIONS FOR DISCUSSION

1. Describe the various design strategies. Which is the most popular and practical?
2. If some existing modules are to be re-used in creating a new system, which design strategy is used and why?
3. What is the difference between a flowchart and a structure chart?

Check Your Progress: Model Answers

1. Inheritance
2. Afferent, efferent
3. True

6.8 SUGGESTED READINGS

R.S. Pressman, *Software Engineering-A Practitioner's Approach*, 5th Edition Tata McGraw Hill Higher education.

Belady L, *Foreword to Software Design: Methods and Techniques*, Yourdon Press, 1981.

R Fairly, *Software Engineering Concepts*, Tata McGraw Hill Pub., 2000.

Sommerville, *Software Engineering*, Addison-Wesley, 3rd Edition, 1989.

Paul Feld, *An Introduction to Object Oriented Design*, Paulfield@dial.pipex.com, 2001.

UNIT IV

LESSON

7

SOFTWARE RELIABILITY

CONTENTS

7.0 AIMS AND OBJECTIVES

After studying this lesson, you should be able to:

- Explain the importance of software reliability, hardware reliability
- Discuss failures and faults related to software reliability
- Define software reliability concepts, models and allocation

7.1 INTRODUCTION

Software reliability is defined as the probability of failure-free software operation for a specified period of time in a specified environment. Software reliability is an attribute and key factor in software quality. It is also a system dependability concept.

Software Reliability Engineering (SRE) is defined as the quantitative study of the operational behavior of software-based systems with respect to user requirements concerning reliability. As such it encompasses all aspects of the software development process. SRE employs proven best practice to ensure that product reliability meets user needs, to speed products to market faster, reduce product cost, improve customer satisfaction, and increase tester and developer productivity. The essential components of SRE are categorized as follows:

- Establish reliability goals
- Develop operational profile
- Plan and execute tests
- Use test results to drive decisions

These components are sequential. Although this START addresses these SRE elements in isolation the reader should note that in reality they are integrated within the software development process. Reliability goals are part of the requirement definition process. Development of the operational profile occurs in parallel with software design and coding. Testing for reliability is a testing approach which is included the overall test plan.

7.2 IMPORTANCE OF SOFTWARE RELIABILITY

Importance of reliability describes the customer's expectation of satisfactory performance of the software in terms that are meaningful to the customer. This description may be significantly different from the theoretical definition of reliability but the customer has no need for the theoretical definition. The customer must tell you the circumstances under which they will "trust" the system you build. For example, someone who purchases a fax machine wants assurance that 99 out of every 100 faxes received will print properly. The fact that the machine must run for 25 hours without failure in order to demonstrate the specified degree of reliability is irrelevant to the customer. In order to test for reliability we often need to translate expressions that are meaningful to the customer into equivalent time units, such as execution time, but the goal remains as the customer perceives it.

What is important is that these customer needs, or expectations, are described in a quantifiable manner using the customer's terminology. They do not have to be statements of probability in order to be useful for determining product reliability.

Some examples of quantified reliability importance are:

The system will be considered sufficiently reliable if 10 (or less) errors result from 10,000 transactions.

The customer can tolerate no more than one class 2 operational failure per release and no class 1 failures per release for a software maintenance effort. All participants in the development (or maintenance) process need to be aware of these reliability goals and their prioritization by the customer. If possible this awareness should come from direct contact with the customer during the requirements gathering phase. This helps to cement the team around common goals. Ideally, reliability goals should be determined up front before design begins; however defining them at any point in the life cycle is better than not having them.

7.3 SOFTWARE AND HARDWARE RELAIBITY

The bathtub curve for Software Reliability.

Over time, hardware shows the failure characteristics shown in Figure 7.1, known as the bathtub curve. Period A, B and C stands for burn-in phase, useful life phase and end-of-life phase.

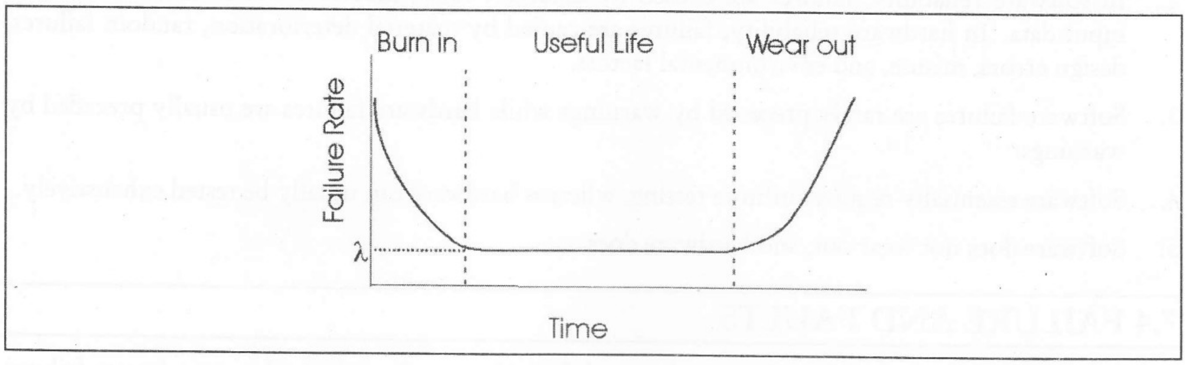

Figure 7.1: Bathtub Curve for Hardware Reliability

Software reliability, however, does not show the same characteristics similar as hardware. A possible curve is shown in Figure 7.2 if we projected software reliability on the same axes. There are two major differences between hardware and software curves. One difference is that in the last phase, software does not have an increasing failure rate as hardware does. In this phase, software is approaching obsolescence; there is no motivation for any upgrades or changes to the software. Therefore, the failure rate will not change. The second difference is that in the useful-life phase, software will experience a drastic increase in failure rate each time an upgrade is made. The failure rate levels off gradually, partly because of the defects found and fixed after the upgrades.

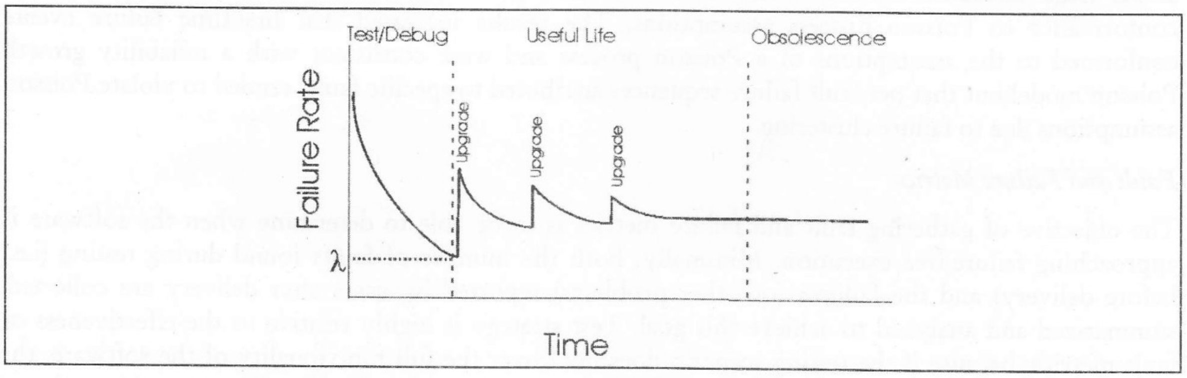

Figure 7.2: Revised Bathtub Curve for Software Reliability

The upgrades in Figure 7.2 imply feature upgrades, not upgrades for reliability. For feature upgrades, the complexity of software is likely to be increased, since the functionality of software is enhanced. Even bug fixes may be a reason for more software failures, if the bug fix induces other defects into software. For reliability upgrades, it is possible to incur a drop in software failure rate, if the goal of the upgrade is enhancing software reliability, such as a redesign or reimplementation of some modules using better engineering approaches, such as clean-room method.

Differences between Software and Hardware Reliability

Some of the important differences between software and hardware reliability are:

1. Failure does not occur if the software is not used. However in hardware reliability, material deterioration can cause failure even when the system is not in use.
2. In software reliability, failures are caused by incorrect logic, incorrect statements, or incorrect input data. In hardware reliability, failures are caused by material deterioration, random failures, design errors, misuse, and environmental factors.
3. Software failures are rarely preceded by warnings while hardware failures are usually preceded by warnings.
4. Software essentially requires infinite testing, whereas hardware can usually be tested exhaustively.
5. Software does not wear out, and hardware does.

7.4 FAILURE AND FAULTS

Software systems mainly contain design and code defects that manifest themselves as software failures at various points during program execution. These software faults can be viewed as causing failures to occur according to some chance mechanism that is often taken to be a Poisson process. Individual per-fault failure sequences (resulting from delayed fault detection and software correction) during test execution were analyzed to determine whether they satisfied Poisson process assumptions. The failure events, defined as severe, system-wide affecting events attributed to software faults, typically occurred following many cycles of interacting system features in the expected user mode. The system analyzed, a large, complex, industrial software system, was executed under a defined operational profile corresponding to a controlled command and autonomous event mix characteristic of the system's actual usage. Execution times between failure events were calculated and then statistically analyzed for conformance to Poisson process assumptions. The results indicated that first-time failure events conformed to the assumptions of a Poisson process and were consistent with a reliability growth Poisson model but that per-fault failure sequences attributed to specific faults tended to violate Poisson assumptions due to failure clustering.

Fault and Failure Metrics

The objective of gathering fault and failure metrics is to be able to determine when the software is approaching failure-free execution. Minimally, both the number of faults found during testing (i.e., before delivery) and the failures (or other problems) reported by users after delivery are collected, summarized and analyzed to achieve this goal. Test strategy is highly relative to the effectiveness of fault metrics, because if the testing scenario does not cover the full functionality of the software, the software may pass all tests and yet be prone to failure once delivered. Usually, failure metrics are based upon customer information regarding failures found after release of the software. The failure data collected is therefore used to calculate failure density, Mean Time Between Failures (MTBF) or other parameters to measure or predict software reliability.