4.0 LEARN AND UNDERSTAND THE CONCEPT OF SOFTWARE DESIGN, CODING & TESTING

Design:

The activities carried out during the design phase transform the SRS document into the design document.

Outcomes of a Design Process:

The following Items are designed and documented during design phase

1) **Different Module required**: The different modules in the solution should be clearly identified. Each module is a collection of functions and data shared by functions of module.
2) **Control Relationships among modules**
3) **Interface between module:** The exact data exchanged between the two modules when a one module invokes a function of other module is identified.
4) **Data Structures of the Individual Module:** Suitable Data structures for the data to be stored in a module are designed.
5) **Algorithms required to implement the Individual modules:** The algorithms required to accomplish the processing activity of various modules are clearly designed.

4.1 What is a good Software Design?

Most researchers and software engineers agree on a few desirable characteristics that every good software design for general applications must possess. The characteristics are

Correctness: A good design should first of all be correct. This is it should correctly implement all the functionalities of the system.

Understandability: A good design should be easily understandable. Unless a design solution is easily understandable, it would be difficult to implement and maintain it.

Efficiency: A good design should adequately address resources, time or cost optimization issues.

Maintainability: A good design should be easy to change. That is an important requirement, since change requests usually keep coming from the customer even after product releases.

Given that we are choosing from only correct design solutions, understand ability of a design solution is possibly the most important issue to be considered while judging the goodness of a design.

A design solution is understandable, if it is modular and the modules are arranged in layers.

1. **Modularity:** A modular design is simple words implies that the problem has been decomposed into a set of modules. Decomposition of a problem in modules facilitates taking advantage of the divide and conquer principle.
2. **Layered Design:** In layered design solution, the modules are arranged in a hierarchy of layers. A module can only invoke functions of the modules in the layer immediately below it.

4.2 Define and Classify Cohesion and Coupling

Coupling: Two modules are said to be highly coupled, in either of the following two situations arises.

- If the function calls between two modules involve passing large chucks of shared data, the modules are tightly coupled.
- If the interactions occur through some shared data, then also we say that they are highly coupled.

If two modules either do not interact with each other at all or at best interact by passing no data or only a few primitive data items, they are said to have low coupling.

Cohesion: When the functions of the module cooperate with each other for performing a single objective, the modules have good cohesion. If the functions of each module do very different things and do not cooperate with each other to perform a single piece of work, then the modules have very poor cohesion.

4.2.1 Classification of Cohesion:

Cohesiveness of a module is the degree to which different functions of the module cooperate to work towards a single objective. The cohesiveness increases from coincidental to functional cohesion. That is coincidental is the worst type of cohesion and functional is the best cohesion possible.

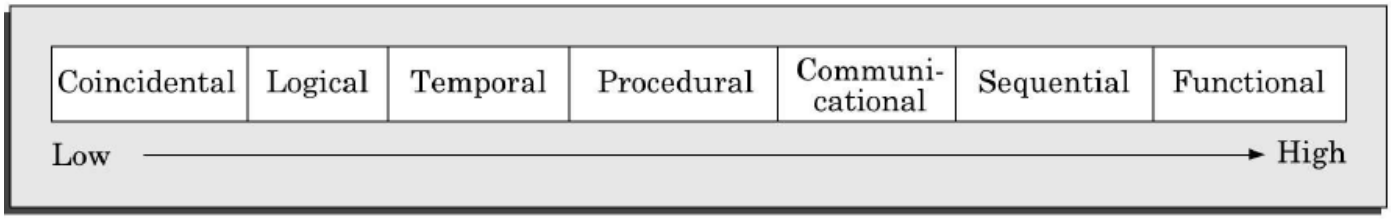
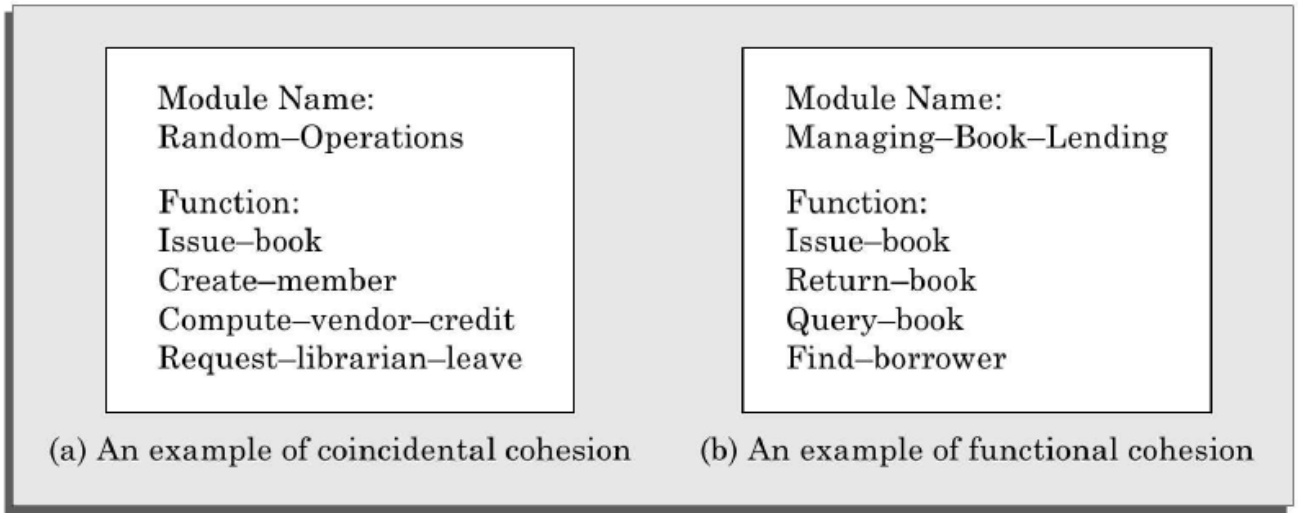


Figure : Classification of cohesion.

Coincidental Cohesion: A module is said to have coincidental cohesion, if it performs a set of tasks that relate to each other very loosely, if at all.

Functional Cohesion: A module is said to be possess functional cohesion, if different functions of the module cooperate to complete a single task.



Logical Cohesion: A module is said to be logically cohesive, if all elements of the module perform similar operations such as error handing data input, data output etc.,

Temporal Cohesion: When a module contains functions that are related by the fact that these functions are executed in the same time span, then the modules is said to possess temporal cohesion.

Procedural Cohesion: A module is said to possess procedural cohesion. If the set of functions of the module are executed one after the other, through these functions may work towards entirely different purpose and operate on very different data.

Communicational Cohesion: A module is said to have communicational cohesion, if all functions of the module refer to or update the same data structure.

Sequential Cohesion: A module is said to posses sequential cohesion, if the different functions of the module execute in sequence, and the output from one function is input to the next in the sequence.

4.2.2 Classification of Coupling: The coupling between two modules indicates the degree of interdependence between them. If two modules interchange large amount of data, then they are highly interdependent or coupled.

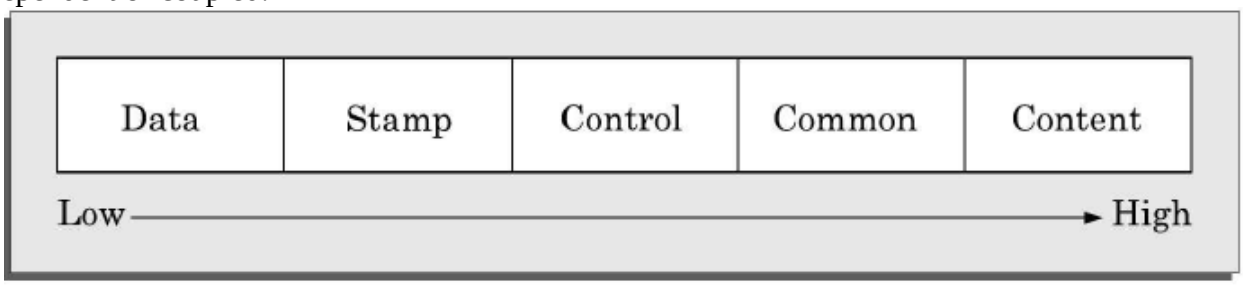


Figure : Classification of coupling.

Data coupling: Two modules are data coupled. If they communicate using am elementary data items that is passed as a parameters between the two, e.g an integer, a float, a character etc.,

Stamp Coupling: Two modules are stamp coupled, if they communicate using a composite data items such as a record in PASCAL or a structure in C.

Control Coupling: Control coupling exist between two modules, if data from one module is used to direct the order of instruction execution in another.

Common Coupling: Two modules are common coupled, if they share some global data items.

Content Coupling: Content coupling exists between two modules, if they share code. That is a jump from one module into the code of another module can occur.

4.3 Know the two approaches of Software Design:

There are two fundamentally different approaches to software design that are in use today – Function-oriented design, and Object – oriented Design. These two design approaches are radically different; they are complementary rather than competing techniques. The object oriented approach is a relatively newer technology and is still evolving. Function oriented designing is a mature technology and has a large following.

4.3.1 Function-Oriented Design:

The following are the silent features of the function-oriented design approaches.

Top-Down Decomposition: In Top-down decomposition starting at high level view of the system, each high level function is successively refined into more detailed functions.

For example, consider a function **create-new-library member** which essentially creates the records for a new records. This high level function may be refined into the following sub-functions

- Assign-membership-number
- Create-member-record
- Print-bill

Centralized system state: The system state is centralized and shared among different functions.

For example, in the library automation system, several functions such as the following share data such as member-records for reference and updation.

- Create-new-member
- Delete-member
- Update-member-record

A large number of function-oriented design approaches have been proposed in the past. A few of the well established function-oriented design approaches are

- Structured design by Constaintime and Yourdon
- Jackons Structured Design by Jackson
- Step-wise refinement by Wirth.

4.3.2 Object-Oriented Design:

In the object-oriented design (OOD) approach, a system is viewed as being made up of a collection of objects. Each object is associated with a set of functions that are called its methods. Each object contains its own data and is responsible for managing it. The data internal to an object cannot be accessed directly by other objects and only through invocation the methods of objects. The system state is decentralized since there is no globally shared data in the system and data is stored in each object.

Similar objects constitute a class. Each object is a member of some class. Classes may inherit features from a super class. Objects communicate by message passing.

4.3.3 Function-Oriented vs Object-Oriented Design

The following are some of the important differences between the function-oriented and object-oriented design.

1. Unlike function-oriented design methods, in OOD, the basic abstraction is not the services available to the users of the system.
2. In OOD, state information exists in the form of data distributed among several objects of the system. In contrast, in a procedural design, the state information is available in a centralized shared data store.
3. Functional-oriented techniques group functions together if as a group they constitute a higher level function. On the other hand, object-oriented techniques group functions together on the basis of the data they operate on.
4.

4.4 Understand the concept of User Interface Design

The user interface portion of software is responsible for all interactions with the user. The user interface part of any software product is of direct concern to the end-user. Users often judge software product based on its user interface. For many interactive applications, as much as 50% of the total development effort is spent on developing the user interface.

4.4.1 List the Characteristics of a good User Interface

1. Speed of Learning: A good user interface should be easy to learn. Speed of learning is based on syntax and semantics of commands. A good user interface should not require its users to memorize commands. The following 3 issues are crucial to enhance speed of learning.

Use of Metaphors and intuitive command names: Speed of learning an interface is greatly facilitated if these are based on some day – to -day real life examples or some physical objects with which the users are familiar with.

Consistency: User should be able to use the similar commands in different circumstances for carrying out similar actions.

Component based Interface: Users can learn an interface faster if the interaction style of the interface is very similar to the interface of other application with which the user is already familiar with.

3. **Speed of Use:** Speed of use of an interface is determined by the time and user effort to execute different commands. It indicates how fast the user can perform their intended tasks. This is achieved through careful design of the interface. Example: The most frequently used commands should have smallest length or available at the top of a menu.
4.

3. Speed of recall: Once the users learn how to use an interface, the speed with which they can recall the command issue procedure should be maximized. This is achieved with use of Metaphors, symbolic command issue procedure.

4. Error prevention: A good user interface should minimize the scope of committing errors while initiating different commands.

5. Attractive: A good user interface should be attractive to use.

6. Consistency: The commands supported by a user interface should be consistent. Consistency facilitates speed of learning, speed of recall and also helps in reduction of error rate.

7. Feedback: A good user interface must provide feedback to various user actions. Ex: if a user request takes more than few seconds of process, the user should be informed about the state of processing of his request.

8. Support for Multiple skill levels: A good user interface should support multiple levels of sophistication of command issue procedure for different categories of users.

9. Error recovery (undo Facility): While issuing commands, even the expert users can commit errors. A good user interface should allow a user to undo a mistake.

10. User guidance and Online Help: Users seek guidance and online help when they either forget a command or unaware of some feature of software.

4.4.2 Understand the Basic Concepts - User Guidance and Online Help - Mode Based vs Modeless Interface -Graphical User Interface (GUI) vs Text-Based User Interface.

Online Help system: Users expect the online help messages to be tailored to the context in which they invoke the help system. A good help system should keep track of what a user is doing while invoking the help system and provide the output message in a context dependent way.

Guidance Message: The guidance messages should be carefully designed to prompt the user about the next actions he might pursue, the current state of the system, the progress so far made in processing his last command etc., A good guidance system should have different levels of sophistication for different categories of users.

Mode-Based versus Modeless Interface:

A mode is a state or collection of states in which only a subset of all user interaction tasks can be performed. In a modeless interface, the same set of commands can be invoked at any time during the running of the software. Thus a modeless interface has only a single mode and all the commands are available all the time during the operation of the software. On the other hand in a mode based interface, different sets of commands can be invoked depending on the mode in which the system is.

Graphical User Interface (GUI) Vs Text Based User Interface:

- In a GUI multiple windows with different information can simultaneously be displayed on the user screen.
- Iconic information representation and symbolic information manipulation is possible in a GUI
- A GUI usually support command selection using an attractive and user friendly menu section system
- In a GUI, a pointing device such as a mouse or a light pen can be used for issuing commands.
- On a flip side a GUI requires special terminal with graphics capabilities for running and also requires special input devices such as mouse. On the other hand a text based user interface can be implemented even on a cheap alphanumeric display terminal.

4.4.3 List the two types of User Interfaces - Command Language Based Interface - Menu Based Interface - Direct Manipulation Interfaces.

Broadly speaking user interfaces can be classified into the following three categories.
Command Language based Interface
Menu based Interface
Direct Manipulation Interface

Command Language Based Interface: Command language based Interface is based on designing a command language which the user can use to issue the commands. A simple command language based interface might simply assign unique names to the different commands. A more sophisticated command language based interface may allow users to compose complex commands by using a set of primitive commands reduces the number of command names one would have to remember.

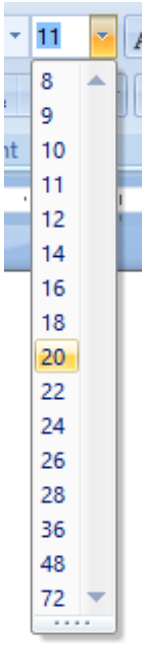
Advantages:

- Among 3 interfaces, the CLI allows for most efficient command issue procedure requiring minimal typing.
- A CLI can be implemented even on cheap alphanumeric terminals
- A CLI is easier to develop compared to a menu based or a direct manipulation interface because compiler writing techniques are well developed.

Disadvantages:

- Usually CLI are difficult to learn, and require the user to memorize the set of commands.
- Most users make errors while formulating commands in the command language and also while typing them.
- In a CLI based interface all interactions with the system is through a keyboard and cannot take advantage of effective interactive devices such as a mouse.

Menu-Based Interface: An important advantage of a menu based interface over a command language based interface is that a menu based interface does not require the users to remember the exact syntax of the command. In menu based interface the typing effort is minimal as most interactions are carried out through menu selection using a pointing device.
However experienced users may find a menu based interface to be slower than a command based interface because an experienced user can type fast and can get speed advantage by composing different commands.



The following are some of the techniques available to structure a large number of menu items.

1. **Scrolling Menus:** When a full choice list cannot be displayed within the menu area, scrolling of the menu items is required. In a scrolling menu all the commands should be highly correlated, so that the user easily locates a command that he needs. An example situation where a scrolling menu is frequently used is font size selection.
2. **Walking Menu:** Walking menu is a very commonly used to structure a large collection of menu items. In this technique, when a menu item is selected, it causes further menu items to be displayed adjacent to it in a submenu.
3. **Hierarchical Menu:** In this technique, the menu items are organised in a hierarchy or tree structure. Selecting a menu item causes menu display to be replaced by an appropriate sub menu. One can consider the menu and its various sub menu to form a hierarchical tree like structure.

Direct Manipulation Interface:
Direct manipulation interfaces present the interfaces to the user in the form of visual models (i.e Icons or Objects). For this reason, direct manipulation interfaces are sometimes called as iconic interfaces. In this type of interfaces, the user issues commands by performing actions on the visual representation of the object, ex pull an icon representing a file into an icon presenting trash box for deleting the file.
Important advantages of iconic interfaces include the fact that the icons can be recognized by the user very easily and that icons are language independent. Experienced users find direct manipulation interface difficult.

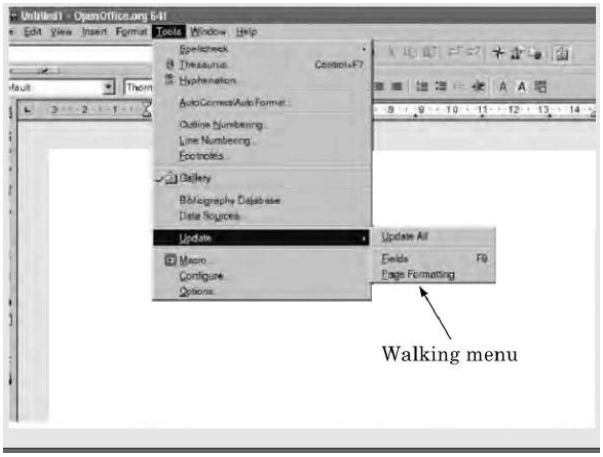


Figure Example of walking menu.

4.4.4 Know about Component Based GUI Development Window System and Types of Widgets.

The current style of user Interface development is component-based. It recognizes that every user interface can easily be built from a handful of predefined components such as menus, dialog boxes, forms etc., Besides the standard components, and the facilities to create good interfaces from them, one of the basic support available to the user interface developers is the windows system.

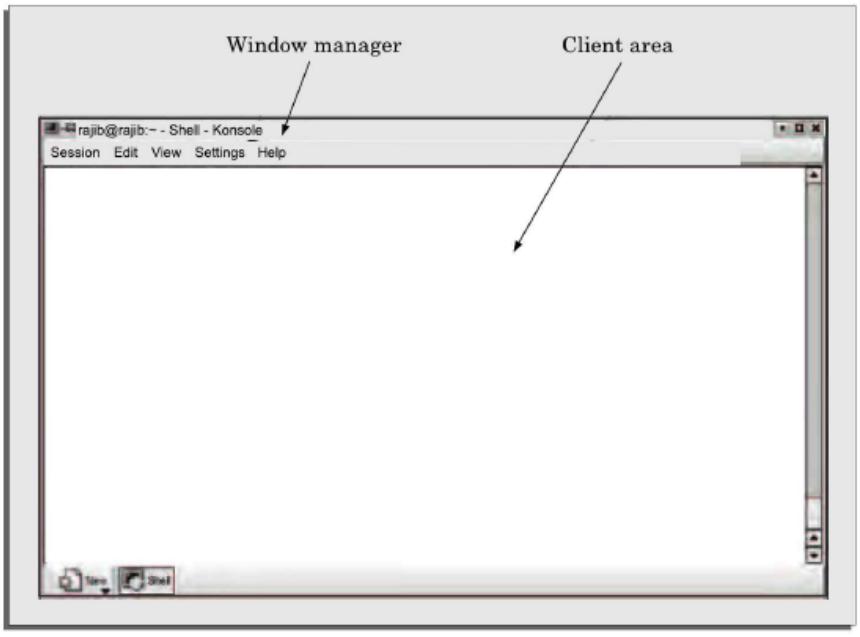


Figure : Window with client and user areas marked.

Window: A window can be divided into two parts – Client part, and non-client part. The client area makes up the whole of the window, except for the border and scroll bars. The client area is the area available to a client application for display. The non-client part of the window determines the

look and feel of the window. The window manager is responsible for managing and maintaining the non-client area of a window.

Window Management system (WMS):
A GUI consists of a large number of windows. It is necessary to have some way to manage these windows. Most GUI do this though Window Management system (WMS).

A WMS simplifies the task of GUI designer to a great extent by providing the basic behaviour of various windows such as move, resize, iconify etc., and provides routines to manipulate the windows from application programs such as creating, destroying, change attributes etc.,

A WMS consists of two parts

- A Window Manager
- A window system

Window manger and Window System:
Window manager is the component of WMS with which the end user interacts to do various window related operations such as window repositioning, window resizing, iconification etc.,

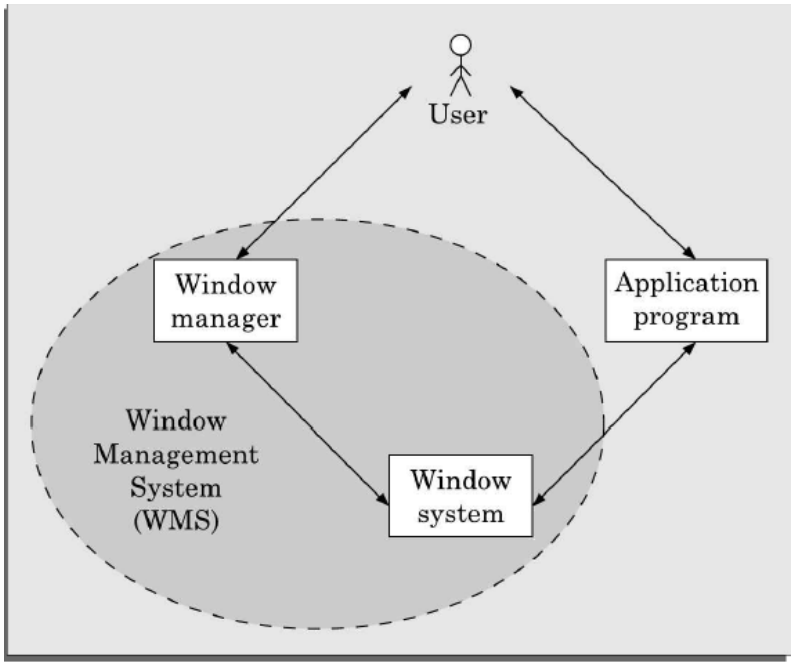


Figure : Window management system.

Window manager is built on top of window system in the sense that it makes use of various services provided by window system. The end user can either interact with the application itself or with the window manager and both the application and the window manager invoke services of the window system.

Most user interface development systems usually provide a high-level abstraction called **widgets** for user interface development. A widget is the short form of a window object. Widgets are the standard user interface components. A user interface us usually made up by integrating several widgets.

Component based development: A development style based on widgets is called component based GUI development style. The advantages are
They help users learn an interface fast.
It reduces the application programmer's work significantly.

Types of Widgets

1. **Label Widget:** A label widget does nothing except to display a label.
2. **Container widget:** It is used to contain other widgets. Other widgets are created as children of the container widget.
3. **Pop-Up Menu:** A popup menu appears upon pressing the mouse button irrespective of mouse position.
4. **Pull-Down Menu:** You have to move the cursor to a specific location and pull down this type of menu.
5. **Dialog Box:** Dialog boxes ask you to enter information, some dialog boxes are merely informative and then Clock ok to dismiss the box.
6. **Push Button:** A push button contains key words or a picture that describes the action that is triggered when you activate the button.
7. **Radio Button:** A set of radio buttons are used when only one option has to be selected out of many options.
8. **Combo Boxes:** A combo box looks like a button until the user interacts with it. When the user presses or clicks it, the combo box displays a menu of items to choose from.

4.5 Understand the concept of Software Coding and Testing

Coding is undertaken once the design phase is complete and the design document has been successfully reviewed. After all the modules of a system have been coded and unit tested, the integration and system testing phase is undertaken. The objective of the coding phase is to transfer the design of a system into code in a high-level language, and then to unit test this code.

4.5.1 Coding Standards and Guidelines - Code Review- Code Walk Throughs - Code Inspection.

Coding Standards and Guidelines:

Good software development organisations require their programmers to follow some well defined and standard style of coding standard. The main advantages of following coding standards are

1. Coding standard gives a uniform appearance to the codes written by different engineers.
2. It facilitates code understanding
3. It promotes good programming practices.

Good software organizations usually develop their own coding standards and guidelines what suits their organisations best. We shall list some general coding standards and guidelines which are commonly adopted by many software companies.

Representative Coding standards

1. **Rules for limiting the use of Globals:** These rules list what types of data can be declared global and what cannot.
2. **Standard Header to precede the code of different modules:** The information contained in the headers of different modules should be standard for an organisation. The following is an example of header format adopted by some companies.
    - **a.** Name of the Module
    - **b.** Date on which the module was created
    - **c.** Authors Name
    - **d.** Modification history
    - **e.** Synopsis of the module
    - **f.** Different functions supported in the module, along with their input/ output parameters
    - **g.** Global variables accessed / modified by the module.
3. **Naming Conventions for Global variables, local variables and constant identifiers:** A popular naming conventions is that variables are named using mixed case lettering. Global variables names would always start with a capital letter (Ex GlobalData) and local variables names start with small letter (ex:localData).Constant names should be formed using capital letters only (ex: CONSTDATA)
4. **Conventions regarding error return values and exception handling mechanisms:** The way error conditions are reported by different functions in a program should be standard within an organisation. For example, all functions while encountering an error condition should either return 0 or -1.

Representative coding guidelines:

1. **Do not use a coding style that is too clever or too difficult to understand:** Code should be easy to understand. Many inexperienced engineers take pride by writing clever coding. Clever coding reduces understanding as a result maintenance and debugging difficult and expensive.
2. **Avoid obscure side effects:** The side effects of function call include modifications to the parameters passed by reference, modification of global variables, and I/O operations. An obscure side effect is one that is not understandable from the casual examination of the code. Obscure side effect makes it difficult to understand a piece of code.
3. **Do not use an identifier for multiple purposes:** Programmers often use the same identifier to denote several temporary entities. Example programmers use temporary loop variable for also computing and storing final result. Some of the problems caused by use of a variable for multiple purpose are
    - **a.** Each variable should be given a descriptive name indicating its purpose. This is not possible if an identifier is used for multiple purposes.

b. It makes further enhancement more difficult. For example changing the final computed result from integer to float type, the programmer might notice that temporary loop variable cannot be a float type.

4. **The code should be well documented:** As a rule of thumb, there should be at least one comment line on the average for every three source lines of code.
5. **The length of any function should not exceed 10 source lines:** A lengthy function is usually very difficult to understand as it probably has a large number of variables and carries out many different types of computations.
6. **Do Not use GO TO statements:** Use of GOTO statement makes a program unstructured, thereby making it very difficult to understand, debug and maintain the program.

Code Review:

Code review for a module is undertaken after the module successfully compiles. That is all the syntax errors have been eliminated from the module. Code reviews are extremely cost effective strategies for eliminating coding errors and for producing high quality code. The following two types of reviews are carried out on the code of a module.

Code inspection
Code Walkthrough.

Code Walkthrough: Code walkthrough is an informal code analysis technique. In this, a module is taken for review after the module has been coded, successfully compiled and all syntax errors have been eliminated. A few members of the development team are given the code a couple of days before the walkthrough meeting. Each member selects some test cases and simulates execution of the code by hand. The main objective of code walkthrough is to discover the algorithmic and logical errors in the code.

Even Code walkthrough is a informal technique, several guidelines have evolved over years.

- The Team performing Code walkthrough should not be either too big or too small. Ideally it should consist of between 3 to 7 members.
- Discussions should focus on discovery of errors and avoid discussions on how to fix the discovered errors.
- Managers should not attend the code walkthrough meeting.

Code Inspection: The principal aim of code inspection is to check for the presence of some common types of errors that usually creep into code due to programmer oversight and to check whether coding standards have been stick to. Following is a list of some classical programming errors which can be checked during code inspection.

- Use of uninitialized variables
- Jumps into loops
- Non-Terminating loops
- Incompatible assignments
- Array indices out of bound
- Improper storage allocation and de allocation
- Mismatches between actual and formal parameters in procedure call
- Use of incorrect logical operators or incorrect precedence among operators.
- Improper modification of loop variables.
- Comparison of equality of floating point values etc,

4.5.2 Clean Room Testing - Software Documentation- Software Testing

Clean Room Testing: Clean Room testing was pioneered by IBM. This type of testing relies heavily on walkthroughs, inspection and formal verification. The programmers are not allowed to test any of their code by executing the code other than doing some syntax testing using a compiler.

This technique reportedly produces documentation and code that is more reliable and maintainable than other development methods relying heavily on code execution based testing. The main problem with this approach is that testing effort is increased as walkthrough, inspection and verification are time consuming for detecting all simple errors.

Software Documentation: When a software product is developed, in addition to the executable files and the source code, several kinds of documents such as user manuals, software requirements specification (SRS), design document, test document, installation manual etc are developed as part of software engineering process. Good documents are very useful and serve the following purpose.

- Good document help enhance understand ability of a software product.
- Document helps the user to understand and effectively use the system.
- Good document help in effectively tackling the manpower turnover problem. When engineers leave the organization and new engineer comes in he can built up the required knowledge.
- Production of good documents helps the manager to effectively track the progress of the project.

Different types of documents can broadly be classified as
Internal Documentation: These are provided in the source code itself.
External Documentation: These are the supporting documents that usually accompany a software product.

Internal Documentation: Internal documentation can be provided in the code in several forms. The important types of internal documentation are the following

- Comment embedded in the source code
- Use of meaningful variables names
- Module and function headers
- Code indentation
- Code structuring (Code decomposed into modules and functions)
- Use of enumerated types.
- Use of constant identifiers
- Use of User-defined data types.

Careful experiments suggest that out of all types of internal documentation, meaningful variable names are most useful while trying to understand a piece of code.

External Documentation: External documentation is provided through various types of supporting documents such as user manuals, software requirements specification document, design document, test document etc.
An important feature of good external documentation is consistency with the code. If the different documents are not consistent, a lot of confusion is created for somebody trying to understand the problem.
All documents developed for a product should be up to date.
Another important feature of External documents is proper understand ability by the category of users for whom the document is designed.

Software Testing

The aim of testing is to identify all defects in a program. However, in practice, even after satisfactory completion of testing phase it is not possible to guarantee that a program is error free. This is because the input data domain of most programs is very large. Testing a program involves providing the program with a set of test inputs (or test cases) and observing if the program behaves as expected. If the program fails to behave as expected, then the input data and the conditions under which the failure occurs are noted for later debugging and error condition.
The following are some commonly used terms associated with testing.
An **error** is a mistake committed by the development team during any of the development phases.

A **Failure** is a symptom of an **error (or defect or bug).**

A **test case** is a tiplet [I,S,O] where I is the data input to the system, S is the state of the system at which the data is input, and O is the expected output of the system.

A **test suite** is the set of all test cases with which a given software product is tested.

4.5.3 Know What is Testing? Testing involves performing the following main activities.

1. Test suite design: The set of test cases using which a program is to be tested is designed using different test case design techniques.

2. Running test cases and checking the results to detect failures: Each test case is run and result are compared with the expected results. A mismatch between the actual result and expected results indicates a failure. The test cases for which the system fails are noted down for later debugging.

3. Debugging: For each failure observed during the previous activity, debugging is carried out to identify the statements that are in error.

4. Error Correction: After the error is located in the previous activity, the code is appropriately changed to correct the error.

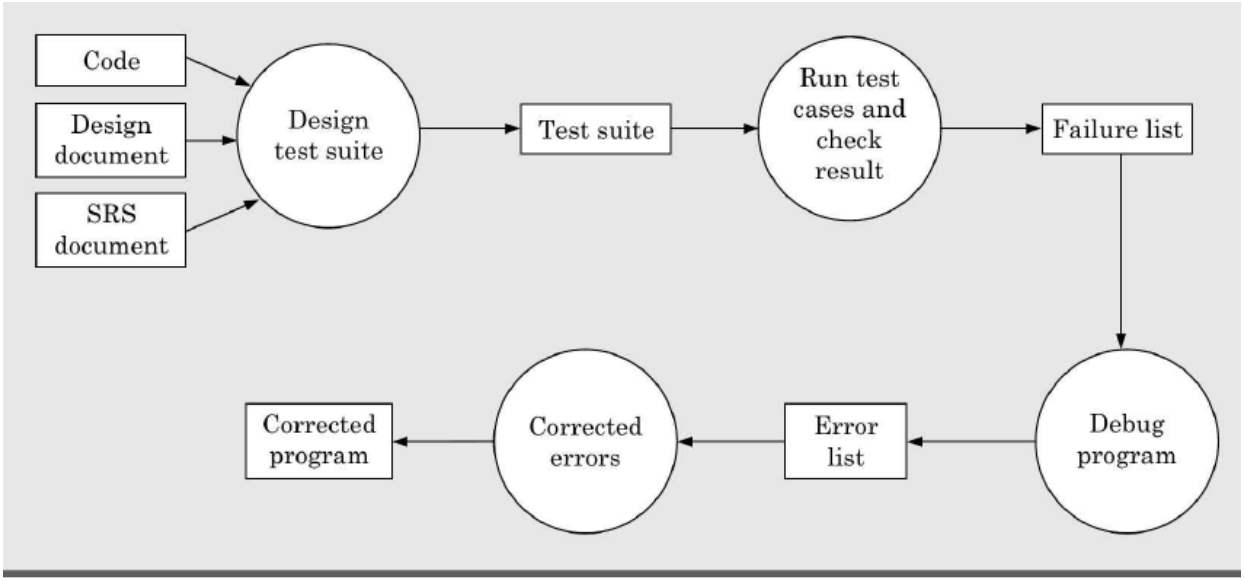


Figure : The testing process.

4.5.4 Differentiate Verification and Validation :

Verification is the process of determining whether the output of one phase of software development conforms to that of its previous phase, whereas validation is the process of determining whether a fully developed system conforms to its requirements specification. Alternatively, we can state the different between verification and validation in the following words.

"While verification is concerned with phase containment of errors, the aim of validation is to make the final product error free''.

4.5.5 List 2 Designs of Test Cases

When test cases are designed based on random input data, many of the test cases don not help detect any additional defects not already being detected by other test cases in the suite.

A minimal test suite is a carefully designed set of test cases such that each test case helps detect different errors. This is in contrast to testing using some random input values.

There are 2 main approaches to design test cases

1. Black Box approach.
2. White Box (or Glass box) approach

In Black Box approach test cases are designed using only the functional specification of software. Test cases are designed based on an analysis of the input / output and does not require any knowledge of the internal structure of the program. For this reason block box testing is also called functional testing.

On the other hand, white box test cases requires a thorough knowledge of the internal structure of a program, and therefore white box testing is also called structural testing.

Two approaches to test case design are complementary. Program has to be tested using the test cases designed by both the approaches.

4.5.6 Differentiate Testing in the Large vs Testing in the Small:

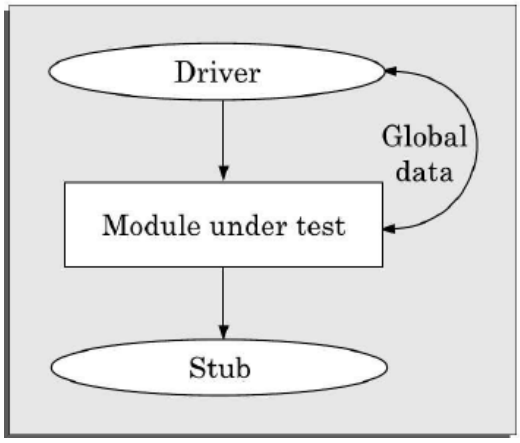
A Software product is normally tested in three levels or stages:
Unit Testing
Integration Testing
System Testing
Unit testing is referred to as testing in the small, whereas integration and system testing are referred to as testing in the large.
After testing all the units individually, the units are slowly integrated and tested after each step of integration. Finally the fully integrated system is tested. Integration and system testing are known as testing in the large.

4.5.7 Understand Unit Testing - Driver and Stub Modules

Unit testing is undertaken after a module has been coded and reviewed. Before carrying out unit testing, the unit test cases have to be designed and the test environment for the unit under test has to be developed.

Drive and Stub Modules: In order to test a single module, we need a complete environment to provide all relevant code that is necessary for execution of the module.

- The procedures belonging to other modules that the module under test calls.
- Nonlocal data structures that the modules accesses.
- A procedure to call the functions of the module under test with appropriate parameters.



Stub: The role of stub and drive modules is pictorially shown in figure. A stub procedure is a dummy procedure that has the same I/O parameters as the given procedure, but has a highly simplified behaviour.

Drive: A drive module should contain the non-local data structures accessed by the module under test. Additionally it should also have the code to call the different functions of the module under test with appropriate parameters values for testing.

4.5.8 Understand Black - box Testing and White Box Testing.

Black-Box testing: In a black box testing, test cases are designed from an examination of the input/ output values only and no knowledge of design and code is required. The following are the two approaches available to design black-box test cases.

- Equivalence Class partitioning
- Boundary Value Analysis.

Equivalence Class Partitioning: In the equivalence class partitioning approach, the domain of input values to the program under test is partitioned into a set of equivalence classes. The partitioning is done such that for every input data belonging to the same equivalence class, the program behaves similarly. Therefore testing the code with any one value belonging to an equivalence class is as good as testing the code with any other value belonging to the same equivalence class.
Example: For a software that computers the square root of an input integer that can assume values in the range 0 to 5000. Determine the equivalence class test suite.
Ans: There are three equivalence classes- the set of negative integer, the set of integers in the range of 0 to 5000, and the set of integers larger than 5000. A possible test suite can be {-5,500,6000}.

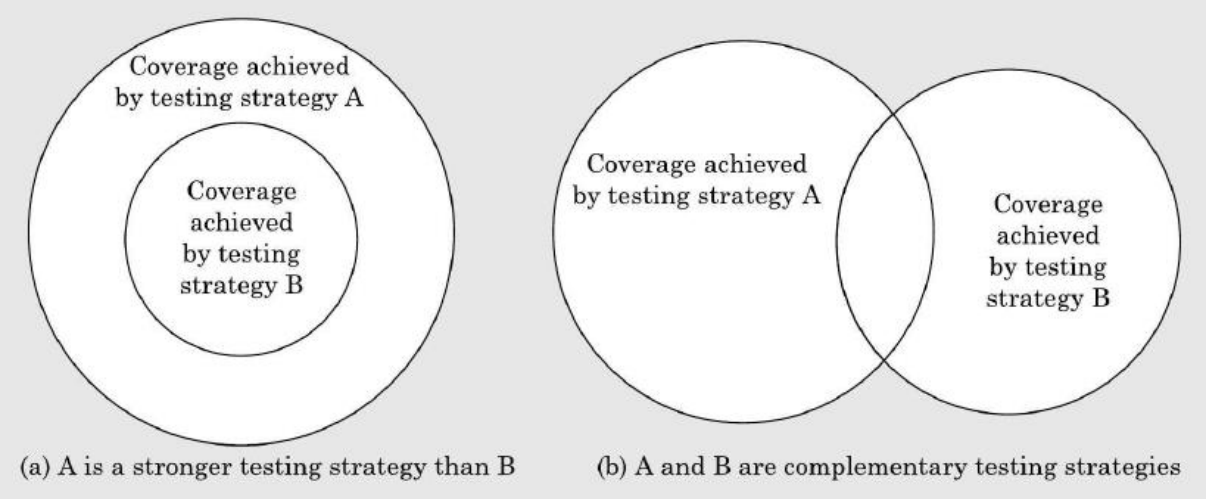
Boundary Value Analysis: Boundary value analysis – based test suite design involves designing test cases using the values at the boundaries of different equivalence classes.
Example: For software that computers the square root of an input integer that can assume values in the range 0 to 5000. Determine the boundary value test suite.
Ans: There are three equivalence classes. The set of negative integers, the set of integers in the range of 0 to 5000, and the set of integers larger than 5000. The boundary value based test suite is: {0,-1,5000,5001}

White Box Testing:

White Box testing is an important type of Unit Testing. A large number of white box testing strategies exist. Each testing strategy essentially designs test cases based on analysis of some aspect of source code.

A white box testing strategy can be either coverage-based or fault based.
Fault-Based Testing: A fault based testing strategy targets to detect certain types of faults.
Coverage-Based Testing: A coverage based testing strategy attempts to execute (ie cover) certain program elements for discovering failure.

A white box testing strategy is said to be stronger than another strategy, if all types of program elements covered by the second testing strategy are also covered by the first testing strategy and the first strategy additionally covers some more types of elements not covered by the second strategy. If a stronger testing has been performed then a weak testing need not be carried out.



(a) A is a stronger testing strategy than B (b) A and B are complementary testing strategies

Statement Coverage: The statement coverage based strategy aims to design test cases so as to execute every statement in a program at least once. The principal idea governing the statement coverage strategy is that unless a statement is executed, there is no way to determine whether an error exist in that statement.
Branch Coverage: Branch coverage based testing requires test cases to be designed so as to make each branch condition in the program to assume true and false value in turn. Brach testing is also called edge testing.
Condition Coverage: In condition coverage based testing; test cases are designed to make each to make each component of a composite conditional expression to assume both true and false values.
Path Coverage: Path Coverage bases testing strategy requires designing test cases such that all linearly independent paths in the program are executed at least once.
Mutation Testing: The idea behind mutation testing is to make a few arbitrary changes to a program at a time. Each time the program is changed, it is called a **mutated program** and the change effected is called a **mutant.** A mutated program is tested against the original test suite of the program. If there exist at least one test case in the test suite for which a mutated program yields an incorrect result, then the mutant is said to be dead. If a mutant remains alive even after all the test cases have been exhausted, the test suite is enhanced to kill the mutant.

4.6 Explain the concept of Debugging

After a failure has been detected, it is necessary to first identify the program statements that are in error and are responsible for the failure, the error can then be fixed.

4.6.1 Explain the Debugging Approaches:

The following are some of the approaches that are popularly adopted by the programmers for debugging.

Brute Force Method: This is most common method of debugging, but is the least efficient method. In this approach, print statements are inserted though out the program to print intermediate values with the hope that some of the printed values will help to identify the statement in error.

This approach become more systematic with the use of source code debugger, because values of different variables can be easily checked and break points and watch points can be easily set.

Backtracking: In this approach, beginning from the statement at which an error symptom has been observed, the source code is traced backwards until the error is discovered.

Casual elimination method: In this approach, once a failure is observed, the symptoms of the failure are noted. Based on the failure symptoms the causes which could possibly have contributed to the symptom is developed and tests are conducted to eliminate each.

Program Slicing: This technique is similar to back tracking. However, the search space is recued by defining slices.

4.6.2 List the Debugging Guidelines:

Debugging is often carried out by programmers based on their creativity and experience. The following are some general guidelines for effective debugging.

- Many times debugging requires a thorough understanding of the program design. Trying to debug based on a partial understanding of the program design may require an inordinate amount of effort to be put into debugging even for simple problems.
- Debugging may sometimes even require full redesign of the system.
- One must be beware of the possibility that an error correction may introduce new errors. Therefore after every round of error fixing, regression testing must be carried.

4.6.3 Program Analysis Tools - Static Analysis Tools - Dynamic Analysis Tools.

A program analysis tool usually is an automated tool that takes either the source code or the executable code of a program as input and produces reports regarding several important characteristics of the program such as its size, complexity, sufficiency of commenting, adherence to programming standards, sufficiency to testing etc., We can classify various program analysis tools into the following two broad categories.

- Static Analysis Tool
- Dynamic Analysis Tools

Static Analysis Tool: Static Program Analysis Tool assesses and computes various characteristics of a program without executing it. Code review techniques such as code walkthrough and code inspection might be considered as static analysis methods since that target to detect error based on analysing the source code.

A major practical limitation of the static analysis tool lies in their inability to analyse run time information such as dynamic memory references (pointer variables and pointer arithmetic's etc).

Static analysis tool often summarize the results of analysis of very function in a polar chart known as Kiviat chart. A Kiviat chart typically shows the analysed values for cyclomatic complexity, number of source lines, percentage of comment lines, Halstead metrics etc.,

Dynamic Analysis Tool: Dynamic program analysis tools can be used to evaluate several program characteristics based on analysis of the run time behaviour of a program.
The code when executed, records the behaviour of the software for different test cases. The dynamic analysis tool can report the statement, branch, and path coverage achieved by the test suite. If the coverage achieved is not satisfactory more test cases can be designed, added to the test suite and run.

Normally the dynamic analysis results are reported in the form of a histogram or pie chart to describe the structural coverage achieved for different modules of the program.

4.6.4 List and Explain the four Integration Testings - Phases vs Incremental Integration Testing- System Testing - Performance Testing.

List and Explain the four Integration Testings: Integration testing is carried out after all the modules have been unit tested. Successful completion of unit testing, to a large extent ensures that the unit as a whole works satisfactorily. The objective of integration testing is to check whether different modules of a program interface with each other properly.
During integration testing, different modules of a system are integrated in a planned manner using an integration plan. The integration plan specifies the order in which modules are combined to realize the full system. After each integration step, the partially integrated system is tested.

Any one (or a mixture) of the following approaches can be used to develop the test plan.

Big- Band Integration testing: Big-band testing is the most obvious approach to integration testing. In this approach, all the modules making up a system are integrated in a single step. In simple words all modules of the system are simply linked together and tested. However this technique can be used only for very small systems. The main problem with this approach is it is very difficult to localize error as the error may lie in any of the modules.

Bottom-Up Integration Testing: Large software products are often made up of several subsystems. In bottom-up approach integration testing, first the modules for the each subsystem are integrated. Thus the subsystem can be integrated separately and independently.

Top – Down Integration Testing: Top-Down integration testing starts with the root module and one or more subordinate modules in the system. After the top-Level skeleton has been tested, the modules that are at the immediately lower layer of the skeleton are combined with it and tested.

Mixed Integration Testing: The mixed (also called Sandwiched) integration testing follows a combination of top-down and bottom-up testing approach. In top-down approach, testing can start only after the top-level modules have been coded and unit tested. Bottom-up testing can start only after the bottom level modules are ready. The mixed approach overcomes this shortcoming testing can start as and when modules become available after the unit testing.

Phases vs Incremental Integration Testing: Big-bang integration testing is carried out in a single step of integration. In contrast, in the other strategies, integration is carried out over several steps. In these later strategies, modules can be integrated either in phased or incremental manner. A comparison of these two strategies is as follows.

- In incremental Integration testing, only one new module is added to the partially integrated system each time.
- In phased integration, a group of related modules are added to the partial system each time.

Phased integration requires less number of integration steps compared to the incremental integration approach. However, when failures are detected, it is easier to debug the system while using the incremental testing approach.

System Testing: After all the units of a program have been integrated together and tested, system testing is taken up. System tests are designed to validate a fully developed system to assure that it meets its requirements. The test cases are therefore designed solely based on the SRS document.
There are essentially three main kinds of system testing.

1. **Alpha Testing:** Alpha testing refers to the system testing carried out by the test team within the developing organisation.
2. **Beta Testing:** Beta testing is the system testing performed by a select group of friendly customers.
3. **Acceptance testing:** Acceptance testing is the system testing performed by the customer to determine whether he should accept the delivery of the system.

The system test cases can be classified into functionality and performance test cases.
The functionality tests are designed to check whether the software satisfies the functional requirements as documented in the SRS document. The performance tests on the other hand test the conformance of the system with the non-functional requirements of the system.

Performance Testing: Performance testing is carried out to check whether the system meets the non-functional requirements identified in the SRS document. All performance tests can be considered as black-box tests.

1. **Stress Testing:** Stress testing is also known as endurance testing (survival Testing). Stress testing evaluates system performance when it is stressed for short periods of time. Stress testing is especially important for systems that usually operate below their maximum capacity but may be stressed at some peak demand hours.

2. **Volume Testing;** Volume testing checks whether the data structures (buffers, arrays, queues, stack etc) have been designed to successfully handle extraordinary situations.

3. **Configuration Testing:** Configuration testing is used to test system behaviour in various hardware and software configurations specified in the requirements.

4. **Compatibility Testing:** This type of testing is required when the system interfaces with external systems (ex: databases, servers, etc).

5. **Regression Testing:** This type of testing is required when the system being tested in an up gradation of an already existing system to fix some bugs or enhance functionality, performance etc.,

6. **Recovery Testing:** Recovery testing tests the response of the system to the presence of faults, or less of power, devices, services, data etc.

7. **Maintenance Testing:** This addresses testing the diagnostic programs, and other procedures that are required to help maintenance of the system.

8. **Documentation Testing:** It is checked whether the required user manual, maintenance manuals, and technical manuals exist and are consistent.

9. **Usability testing:** Usability testing concerns checking the user interface to see if it meets all user requirements concerning the user interface.

10. **Security testing:** Security testing is essential for software products that process confidential data. In needs to be tested whether the system is fool-proof from security attacks such as intrusion by hackers.