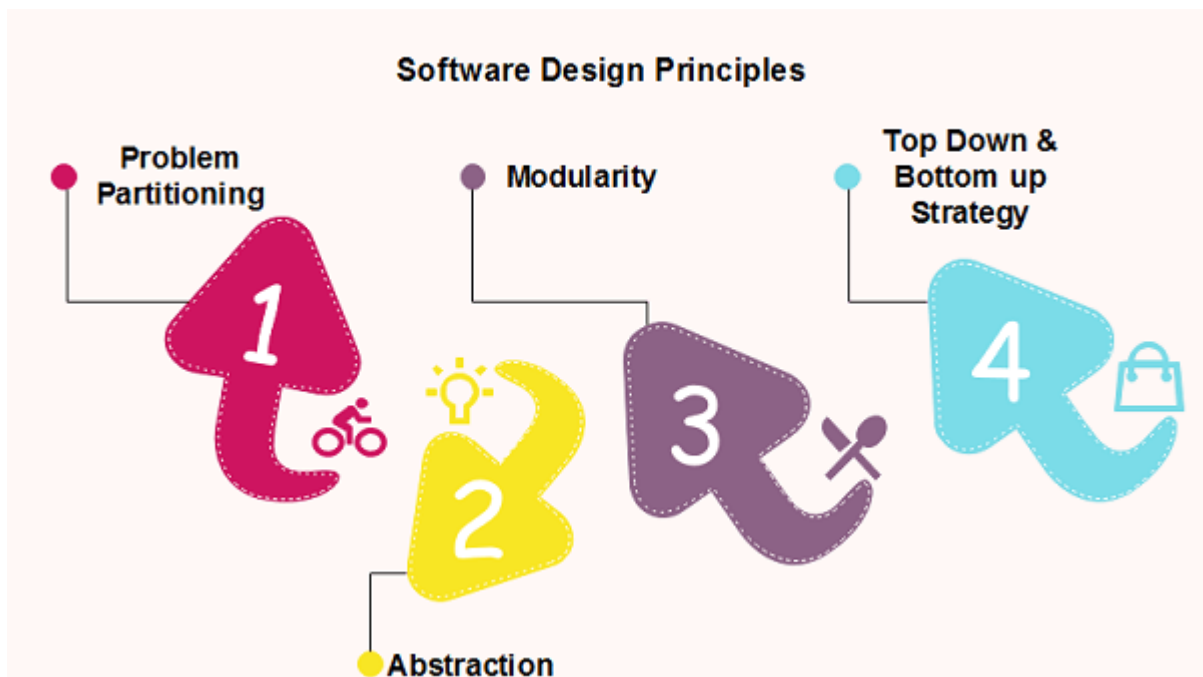


Software Design Principles

Software design principles are concerned with providing means to handle the complexity of the design process effectively. Effectively managing the complexity will not only reduce the effort needed for design but can also reduce the scope of introducing errors during design.

Following are the principles of Software Design



Problem Partitioning

For small problem, we can handle the entire problem at once but for the significant problem, divide the problems and conquer the problem it means to divide the problem into smaller pieces so that each piece can be captured separately.

For software design, the goal is to divide the problem into manageable pieces.

Benefits of Problem Partitioning

1. Software is easy to understand
2. Software becomes simple
3. Software is easy to test
4. Software is easy to modify
5. Software is easy to maintain
6. Software is easy to expand

These pieces cannot be entirely independent of each other as they together form the system. They have to cooperate and communicate to solve the problem. This communication adds complexity.

Note: As the number of partition increases = Cost of partition and complexity increases

Abstraction

An abstraction is a tool that enables a designer to consider a component at an abstract level without bothering about the internal details of the implementation. Abstraction can be used for existing element as well as the component being designed.

Here, there are two common abstraction mechanisms

1. Functional Abstraction
2. Data Abstraction

Functional Abstraction

- i. A module is specified by the method it performs.
- ii. The details of the algorithm to accomplish the functions are not visible to the user of the function.

Functional abstraction forms the basis for **Function oriented design approaches**.

Data Abstraction

Details of the data elements are not visible to the users of data. Data Abstraction forms the basis for **Object Oriented design approaches**.

Modularity

Modularity specifies to the division of software into separate modules which are differently named and addressed and are integrated later on in 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 a large number of reference variables, control paths, global variables, etc.

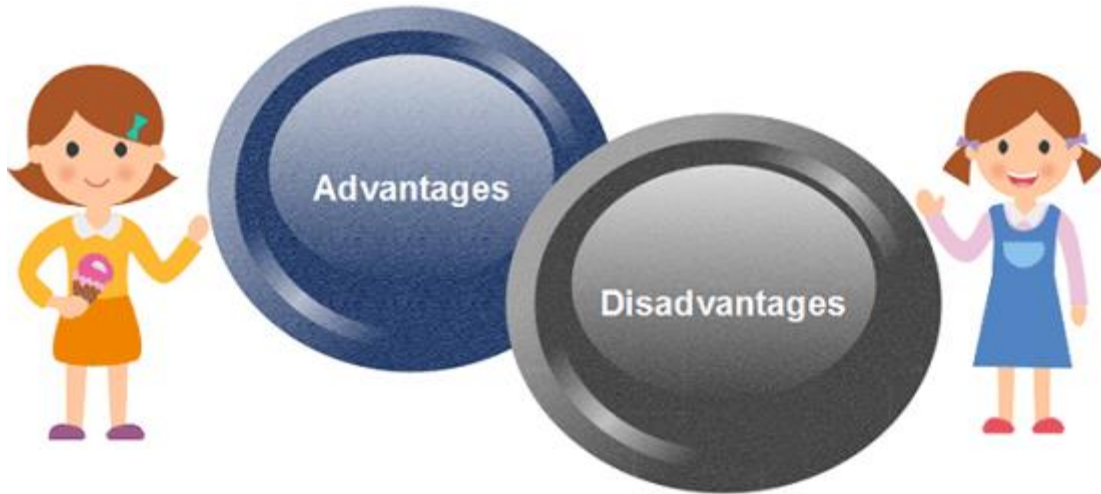
The desirable properties of a modular system are:

- Each module is a well-defined system that can be used with other applications.
- Each module has single specified objectives.
- Modules can be separately compiled and saved in the library.

- Modules should be easier to use than to build.
- Modules are simpler from outside than inside.

Advantages and Disadvantages of Modularity

In this topic, we will discuss various advantage and disadvantage of Modularity.



Advantages of Modularity

There are several advantages of Modularity

- It allows large programs to be written by several or different people
- It encourages the creation of commonly used routines to be placed in the library and used by other programs.
- It simplifies the overlay procedure of loading a large program into main storage.
- It provides more checkpoints to measure progress.
- It provides a framework for complete testing, more accessible to test
- It produced the well designed and more readable program.

Disadvantages of Modularity

There are several disadvantages of Modularity

- Execution time maybe, but not certainly, longer
- Storage size perhaps, but is not certainly, increased
- Compilation and loading time may be longer
- Inter-module communication problems may be increased
- More linkage required, run-time may be longer, more source lines must be written, and more documentation has to be done

Modular Design

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

1. Functional Independence: 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 more accessible 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:

- **Cohesion:** It measures the relative function strength of a module.
- **Coupling:** It measures the relative interdependence among modules.

2. Information hiding: The fundamental of Information hiding suggests that modules can be characterized by the design decisions that protect from the others, i.e., In other words, modules should be specified that data include within a module is inaccessible to other modules that do not need for such information.

The use of information hiding as design criteria for modular system provides the most significant benefits when modifications are required during testing's and later during software maintenance. This is because as most data and procedures are hidden from other parts of the software, inadvertent errors introduced during modifications are less likely to propagate to different locations within the software.

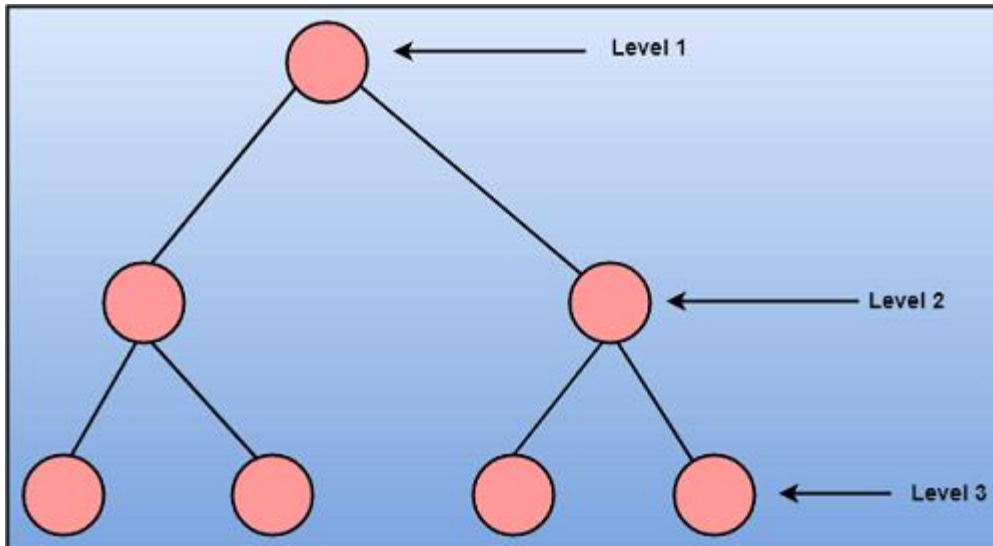
Strategy of Design

A good system design strategy is to organize the program modules in such a method that are easy to develop and latter too, change. Structured design methods help developers to deal with the size and complexity of programs. Analysts generate instructions for the developers about how code should be composed and how pieces of code should fit together to form a program.

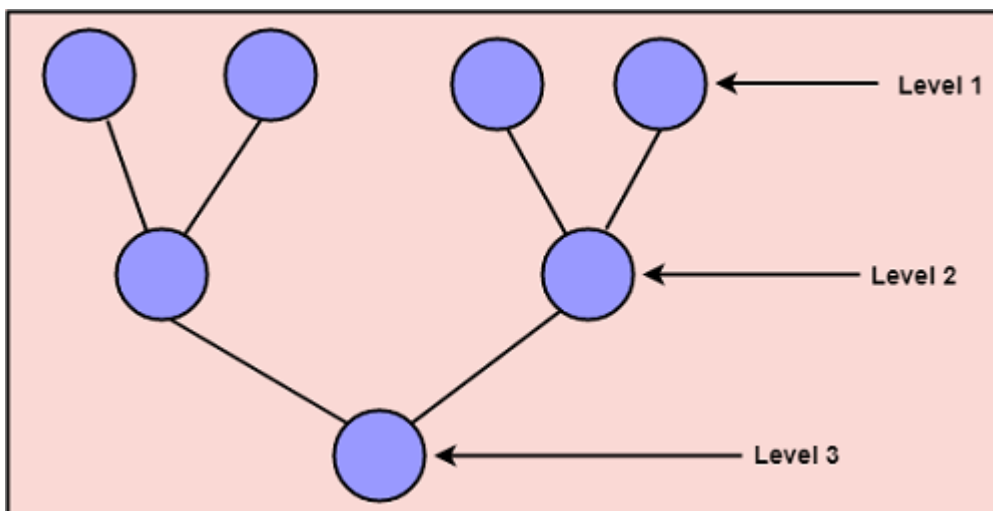
To design a system, there are two possible approaches:

1. Top-down Approach
2. Bottom-up Approach

1. Top-down Approach: This approach starts with the identification of the main components and then decomposing them into their more detailed sub-components.



2. Bottom-up Approach: A bottom-up approach begins with the lower details and moves towards up the hierarchy, as shown in fig. This approach is suitable in case of an existing system.



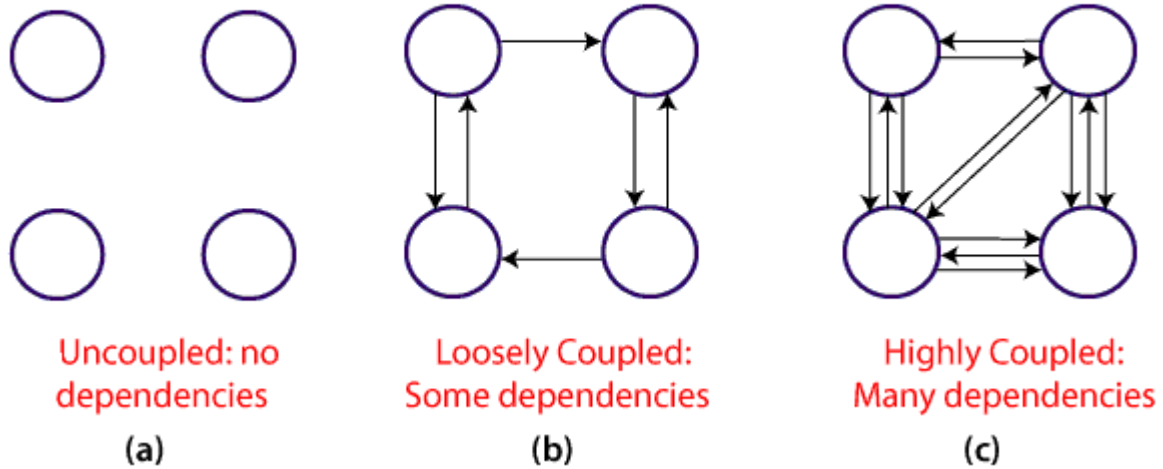
Coupling and Cohesion

Module Coupling

In software engineering, the coupling is the degree of interdependence between software 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.

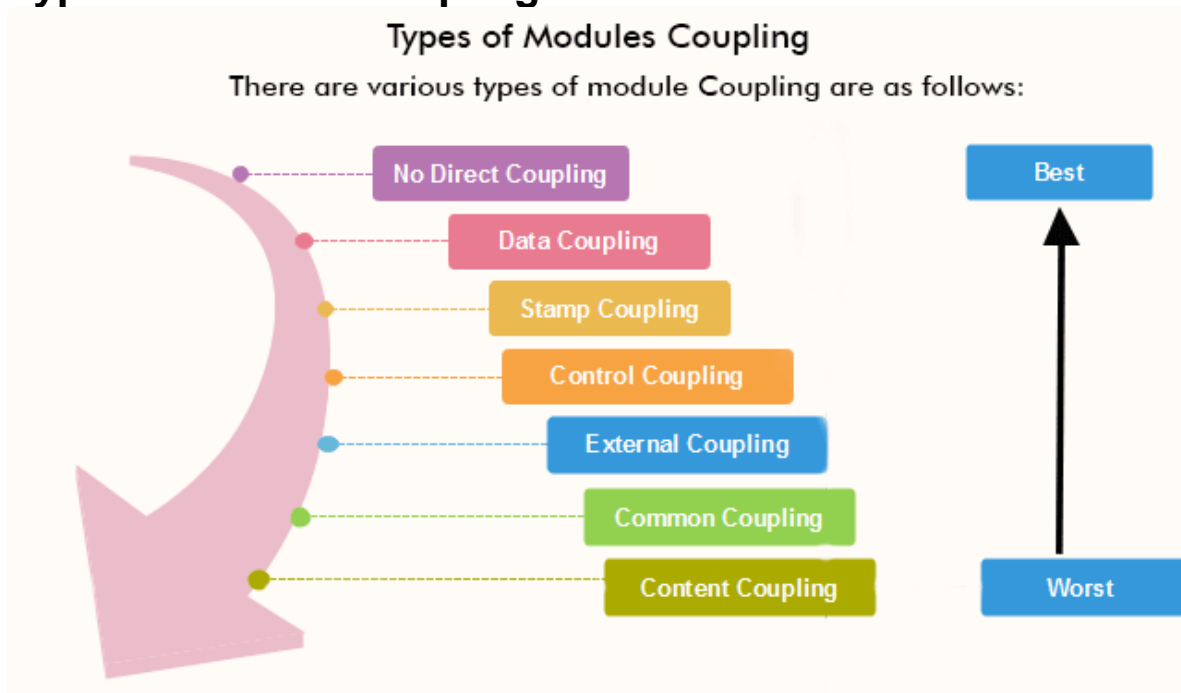
The various types of coupling techniques are shown in fig:

Module Coupling

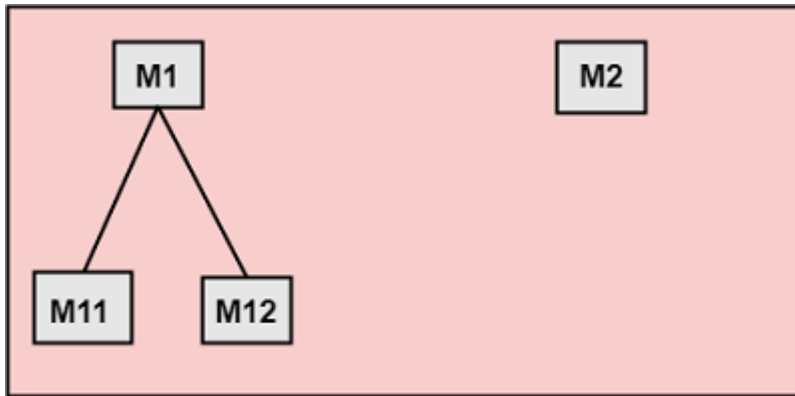


A good design is the one that has low coupling. Coupling is measured by the number of relations between the modules. That is, the coupling increases as the number of calls between modules increase or the amount of shared data is large. Thus, it can be said that a design with high coupling will have more errors.

Types of Module Coupling

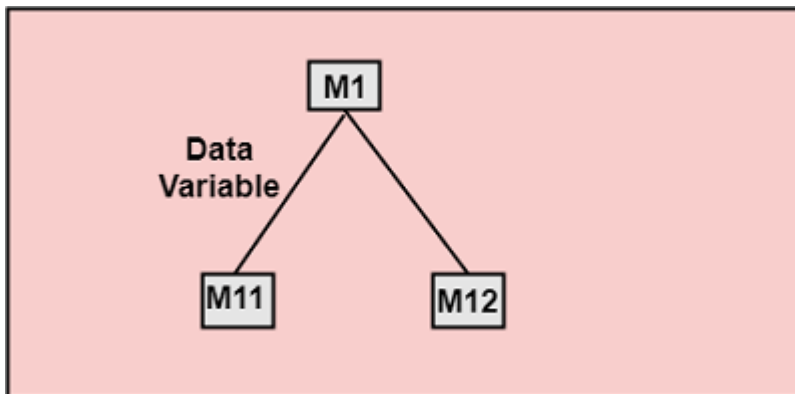


1. No Direct Coupling: There is no direct coupling between M1 and M2.



In this case, modules are subordinates to different modules. Therefore, no direct coupling.

2. Data Coupling: When data of one module is passed to another module, this is called data coupling.

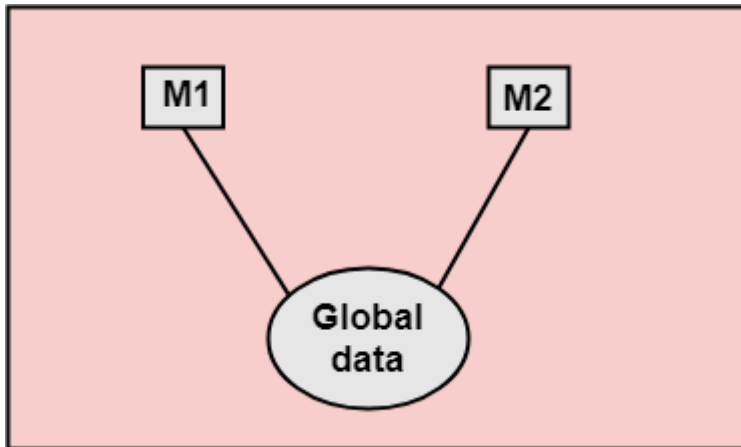


3. Stamp Coupling: Two modules are stamp coupled if they communicate using composite data items such as structure, objects, etc. When the module passes non-global data structure or entire structure to another module, they are said to be stamp coupled. For example, passing structure variable in C or object in C++ language to a module.

4. Control Coupling: Control Coupling exists among two modules if data from one module is used to direct the structure of instruction execution in another.

5. External Coupling: External Coupling arises when two modules share an externally imposed data format, communication protocols, or device interface. This is related to communication to external tools and devices.

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

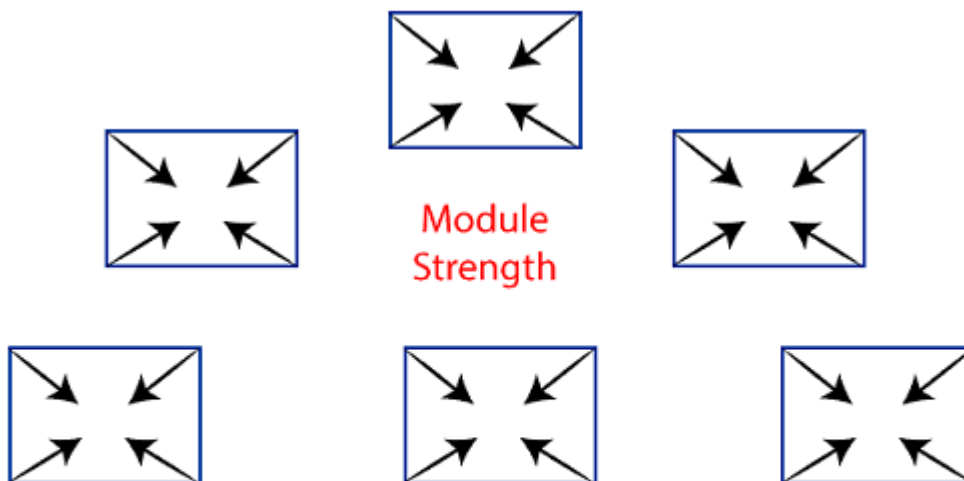


7. Content Coupling: Content Coupling exists among two modules if they share code, e.g., a branch from one module into another module.

Module Cohesion

In computer programming, cohesion defines to the degree to which the elements of a module belong together. Thus, cohesion measures the strength of relationships between pieces of functionality within a given module. For example, in highly cohesive systems, functionality is strongly related.

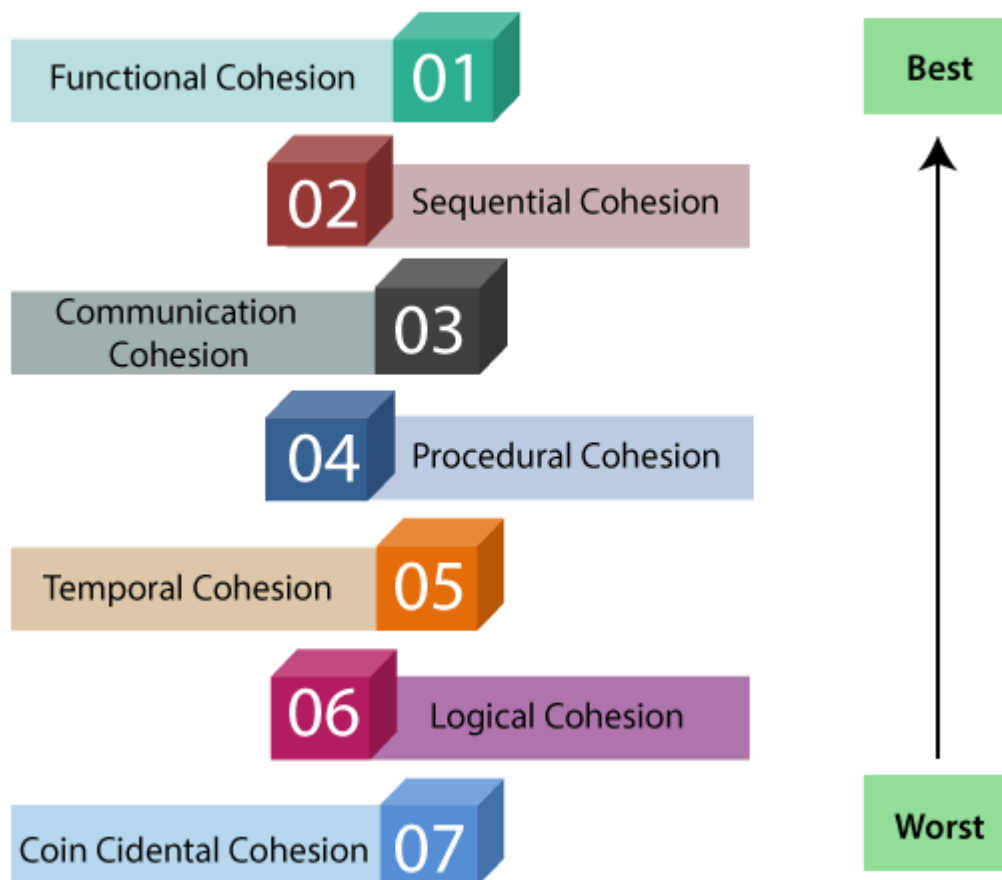
Cohesion is an **ordinal** type of measurement and is generally described as "high cohesion" or "low cohesion."



Cohesion= Strength of relations within Modules

Types of Modules Cohesion

Types of Modules Cohesion



1. **Functional Cohesion:** Functional Cohesion is said to exist if the different elements of a module, cooperate to achieve a single function.
2. **Sequential Cohesion:** A module is said to possess sequential cohesion if the element of a module form the components of the sequence, where the output from one component of the sequence is input to the next.
3. **Communicational Cohesion:** A module is said to have communicational cohesion, if all tasks of the module refer to or update the same data structure, e.g., the set of functions defined on an array or a stack.
4. **Procedural Cohesion:** A module is said to be procedural cohesion if the set of purpose of the module are all parts of a procedure in which particular sequence of steps has to be carried out for achieving a goal, e.g., the algorithm for decoding a message.
5. **Temporal Cohesion:** When a module includes functions that are associated by the fact that all the methods must be executed in the same time, the module is said to exhibit temporal cohesion.

6. **Logical Cohesion:** A module is said to be logically cohesive if all the elements of the module perform a similar operation. For example Error handling, data input and data output, etc.
7. **Coincidental Cohesion:** A module is said to have coincidental cohesion if it performs a set of tasks that are associated with each other very loosely, if at all.

Differentiate between Coupling and Cohesion

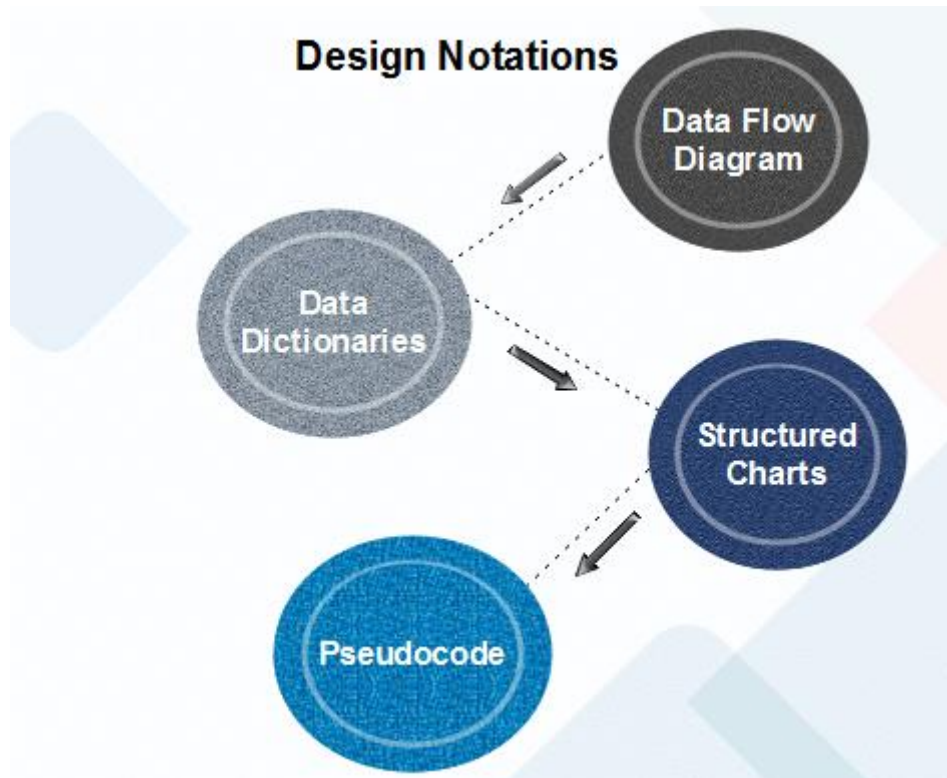
Coupling	Cohesion
Coupling is also called Inter-Module Binding.	Cohesion is also called Intra-Module Binding.
Coupling shows the relationships between modules.	Cohesion shows the relationship within the module.
Coupling shows the relative independence between the modules.	Cohesion shows the module's relative functional strength.
While creating, you should aim for low coupling, i.e., dependency among modules should be less.	While creating you should aim for high cohesion, i.e., a cohesive component/ module focuses on a single function (i.e., single-mindedness) with little interaction with other modules of the system.
In coupling, modules are linked to the other modules.	In cohesion, the module focuses on a single thing.

Function Oriented Design

Function Oriented design is a method to software design where the model is decomposed into a set of interacting units or modules where each unit or module has a clearly defined function. Thus, the system is designed from a functional viewpoint.

Design Notations

Design Notations are primarily meant to be used during the process of design and are used to represent design or design decisions. For a function-oriented design, the design can be represented graphically or mathematically by the following:







Data Flow Diagram

Data-flow design is concerned with designing a series of functional transformations that convert system inputs into the required outputs. The design is described as data-flow diagrams. These diagrams show how data flows through a system and how the output is derived from the input through a series of functional transformations.

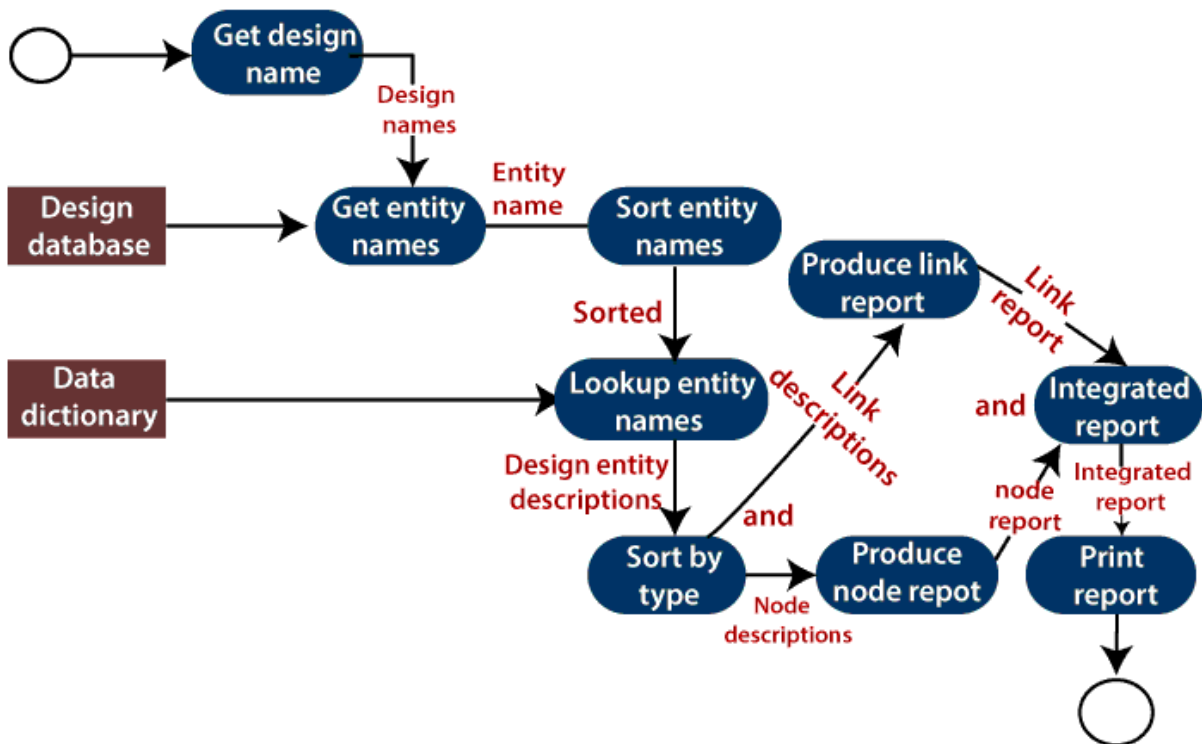
Data-flow diagrams are a useful and intuitive way of describing a system. They are generally understandable without specialized training, notably if control information is excluded. They show end-to-end processing. That is the flow of processing from when data enters the system to where it leaves the system can be traced.

Data-flow design is an integral part of several design methods, and most CASE tools support data-flow diagram creation. Different ways may use different icons to represent data-flow diagram entities, but their meanings are similar.

The notation which is used is based on the following symbols:

Symbol	Name	Meaning
	Rounded Rectangle	It represents functions which transforms input to output. The transformation name indicates its function.
	Rectangle	It represents data stores. Again, they should give a descriptive name.
	Circle	It represents user interactions with the system that provides input or receives output.
	Arrows	It shows the direction of data flow. Their name describes the data flowing along the path.
"and" and "or"	Keywords	The keywords "and" and "or". These have their usual meanings in boolean expressions. They are used to link data flows when more than one data flow may be input or output from a transformation.

Data flow diagram of a design report generator



The report generator produces a report which describes all of the named entities in a data-flow diagram. The user inputs the name of the design represented by the diagram. The report generator then finds all the names used in the data-flow diagram. It looks up a data dictionary and retrieves information about each name. This is then collated into a report which is output by the system.

Data Dictionaries

A data dictionary lists all data elements appearing in the DFD model of a system. The data items listed contain all data flows and the contents of all data stores looking on the DFDs in the DFD model of a system.

A data dictionary lists the objective of all data items and the definition of all composite data elements in terms of their component data items. For example, a data dictionary entry may contain that the data *grossPay* consists of the parts *regularPay* and *overtimePay*.

$$\text{grossPay} = \text{regularPay} + \text{overtimePay}$$

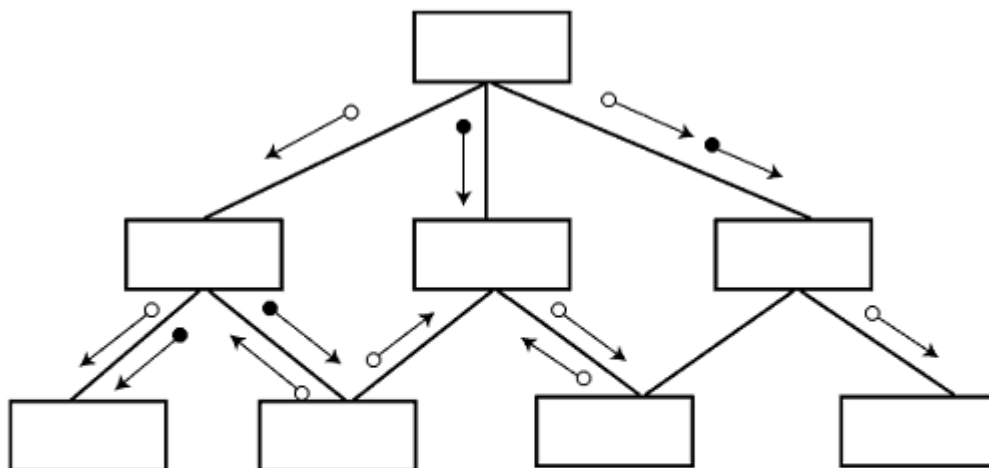
For the smallest units of data elements, the data dictionary lists their name and their type.

A data dictionary plays a significant role in any software development process because of the following reasons:

- A Data dictionary provides a standard language for all relevant information for use by engineers working in a project. A consistent vocabulary for data items is essential since, in large projects, different engineers of the project tend to use different terms to refer to the same data, which unnecessarily causes confusion.
- The data dictionary provides the analyst with a means to determine the definition of various data structures in terms of their component elements.

Structured Charts

It partitions a system into block boxes. A Black box system that functionality is known to the user without the knowledge of internal design.





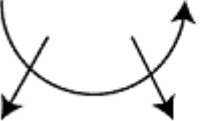
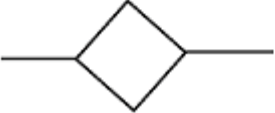


Hierarchical format of a structure chart

Structured Chart is a graphical representation which shows:

- System partitions into modules
- Hierarchy of component modules
- The relation between processing modules
- Interaction between modules
- Information passed between modules

The following notations are used in structured chart:

SYMBOL	DESCRIPTION
	Module
	Arrow
	Data couple
	Control Flag
	Loop
	Decision

Pseudo-code

Pseudo-code notations can be used in both the preliminary and detailed design phases. Using pseudo-code, the designer describes system characteristics using short, concise, English Language phrases that are structured by keywords such as If-Then-Else, While-Do, and End.

Coding

The coding is the process of transforming the design of a system into a computer language format. This coding phase of software development is concerned with software translating design specification into the source code. It is necessary to write source code & internal documentation so that conformance of the code to its specification can be easily verified.

Coding is done by the coder or programmers who are independent people than the designer. The goal is not to reduce the effort and cost of the coding phase, but to cut to the cost of a later stage. The cost of testing and maintenance can be significantly reduced with efficient coding.

Goals of Coding

1. **To translate the design of system into a computer language format:** The coding is the process of transforming the design of a system into a computer language format, which can be executed by a computer and that perform tasks as specified by the design of operation during the design phase.
2. **To reduce the cost of later phases:** The cost of testing and maintenance can be significantly reduced with efficient coding.
3. **Making the program more readable:** Program should be easy to read and understand. It increases code understanding having readability and understandability as a clear objective of the coding activity can itself help in producing more maintainable software.

For implementing our design into code, we require a high-level functional language. A programming language should have the following characteristics:

Characteristics of Programming Language

Following are the characteristics of Programming Language:

Characteristics of Programming Language



Readability: A good high-level language will allow programs to be written in some methods that resemble a quite-English description of the underlying functions. The coding may be done in an essentially self-documenting way.

Portability: High-level languages, being virtually machine-independent, should be easy to develop portable software.

Generality: Most high-level languages allow the writing of a vast collection of programs, thus relieving the programmer of the need to develop into an expert in many diverse languages.

Brevity: Language should have the ability to implement the algorithm with less amount of code. Programs mean in high-level languages are often significantly shorter than their low-level equivalents.

Error checking: A programmer is likely to make many errors in the development of a computer program. Many high-level languages invoke a lot of bugs checking both at compile-time and run-time.

Cost: The ultimate cost of a programming language is a task of many of its characteristics.

Quick translation: It should permit quick translation.

Efficiency: It should authorize the creation of an efficient object code.

Modularity: It is desirable that programs can be developed in the language as several separately compiled modules, with the appropriate structure for ensuring self-consistency among these modules.

Widely available: Language should be widely available, and it should be feasible to provide translators for all the major machines and all the primary operating systems.

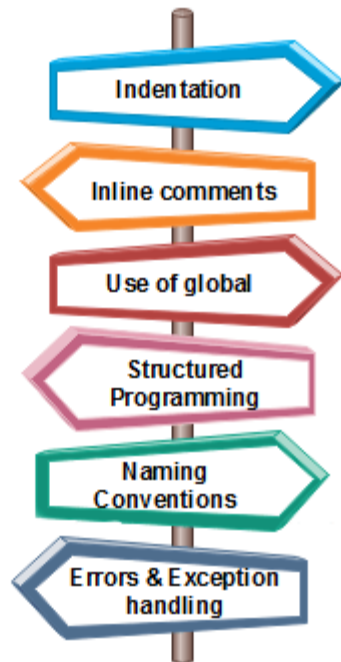
A coding standard lists several rules to be followed during coding, such as the way variables are to be named, the way the code is to be laid out, error return conventions, etc.

Coding Standards

General coding standards refers to how the developer writes code, so here we will discuss some essential standards regardless of the programming language being used.

The following are some representative coding standards:

Coding Standards

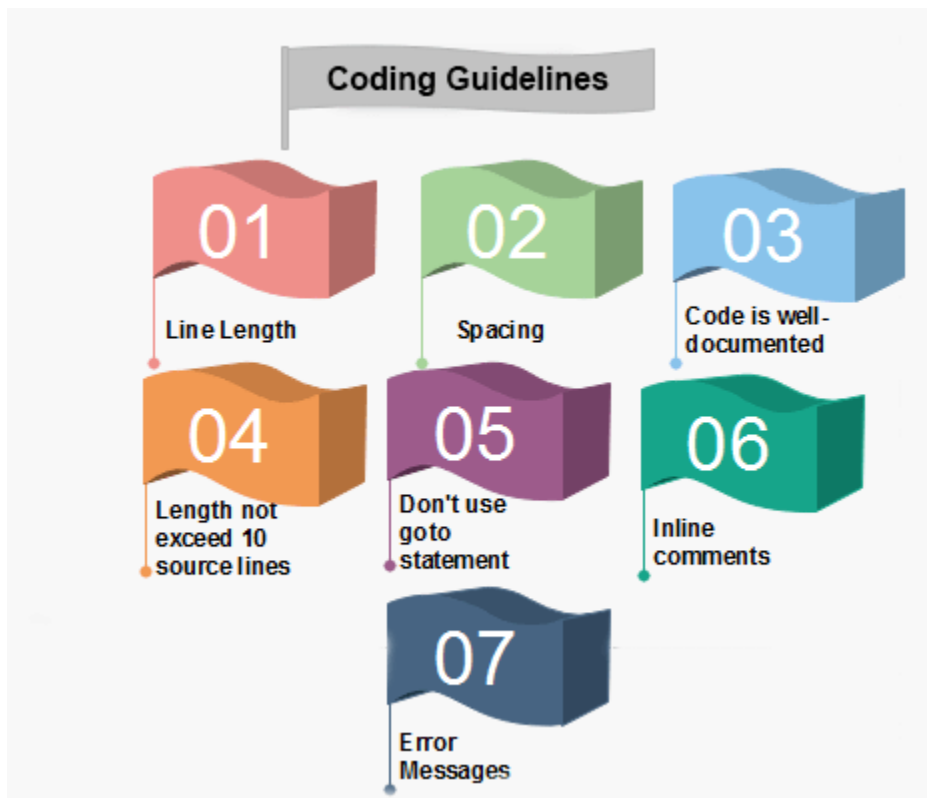


1. **Indentation:** Proper and consistent indentation is essential in producing easy to read and maintainable programs. Indentation should be used to:
 - Emphasize the body of a control structure such as a loop or a select statement.
 - Emphasize the body of a conditional statement
 - Emphasize a new scope block
2. **Inline comments:** Inline comments analyze the functioning of the subroutine, or key aspects of the algorithm shall be frequently used.
3. **Rules for limiting the use of global:** These rules file what types of data can be declared global and what cannot.
4. **Structured Programming:** Structured (or Modular) Programming methods shall be used. "GOTO" statements shall not be used as they lead to "spaghetti" code, which is hard to read and maintain, except as outlined line in the FORTRAN Standards and Guidelines.
5. **Naming conventions for global variables, local variables, and constant identifiers:** A possible naming convention can be that global variable names always begin with a capital letter, local variable names are made of small letters, and constant names are always capital letters.
6. **Error return conventions and exception handling system:** Different functions in a program report the way error conditions are handled should be standard within an organization. For example, different tasks while encountering an error condition should either return a 0 or 1 consistently.

Coding Guidelines

General coding guidelines provide the programmer with a set of the best methods which can be used to make programs more comfortable to read and maintain. Most of the examples use the C language syntax, but the guidelines can be tested to all languages.

The following are some representative coding guidelines recommended by many software development organizations.



1. Line Length: It is considered a good practice to keep the length of source code lines at or below 80 characters. Lines longer than this may not be visible properly on some terminals and tools. Some printers will truncate lines longer than 80 columns.

2. Spacing: The appropriate use of spaces within a line of code can improve readability.

Example:

Bad:

```
cost=price+(price*sales_tax)
fprintf(stdout,"The total cost is %5.2f\n",cost);
```

Better:

```
cost = price + ( price * sales_tax )
fprintf (stdout,"The total cost is %5.2f\n",cost);
```

3. The code should be well-documented: As a rule of thumb, there must be at least one comment line on the average for every three-source line.

4. The length of any function should not exceed 10 source lines: A very lengthy function is generally very difficult to understand as it possibly carries out many various

functions. For the same reason, lengthy functions are possible to have a disproportionately larger number of bugs.

5. Do not use goto statements: Use of goto statements makes a program unstructured and very tough to understand.

6. Inline Comments: Inline comments promote readability.

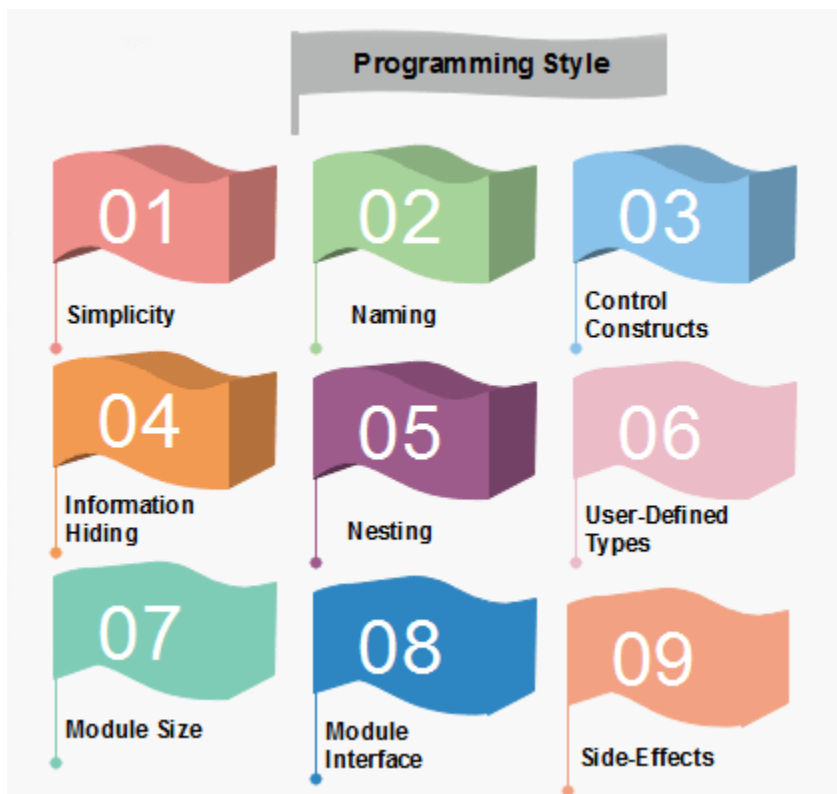
7. Error Messages: Error handling is an essential aspect of computer programming. This does not only include adding the necessary logic to test for and handle errors but also involves making error messages meaningful.

Programming Style

Programming style refers to the technique used in writing the source code for a computer program. Most programming styles are designed to help programmers quickly read and understands the program as well as avoid making errors. (Older programming styles also focused on conserving screen space.) A good coding style can overcome the many deficiencies of a first programming language, while poor style can defeat the intent of an excellent language.

The goal of good programming style is to provide understandable, straightforward, elegant code. The programming style used in a various program may be derived from the coding standards or code conventions of a company or other computing organization, as well as the preferences of the actual programmer.

Some general rules or guidelines in respect of programming style:



1. Clarity and simplicity of Expression: The programs should be designed in such a manner so that the objectives of the program is clear.

2. Naming: In a program, you are required to name the module, processes, and variable, and so on. Care should be taken that the naming style should not be cryptic and non-representative.

For Example: $a = 3.14 * r * r$
area of circle = 3.14 * radius * radius;

3. Control Constructs: It is desirable that as much as a possible single entry and single exit constructs used.

4. Information hiding: The information secure in the data structures should be hidden from the rest of the system where possible. Information hiding can decrease the coupling between modules and make the system more maintainable.

5. Nesting: Deep nesting of loops and conditions greatly harm the static and dynamic behavior of a program. It also becomes difficult to understand the program logic, so it is desirable to avoid deep nesting.

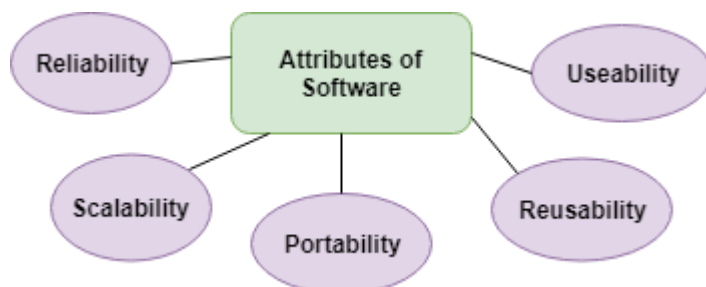
6. User-defined types: Make heavy use of user-defined data types like enum, class, structure, and union. These data types make your program code easy to write and easy to understand.

7. Module size: The module size should be uniform. The size of the module should not be too big or too small. If the module size is too large, it is not generally functionally cohesive. If the module size is too small, it leads to unnecessary overheads.

8. Module Interface: A module with a complex interface should be carefully examined.

9. Side-effects: When a module is invoked, it sometimes has a side effect of modifying the program state. Such side-effect should be avoided where as possible.

What is Software Testing



Software testing is a process of identifying the correctness of a software by considering its all attributes (Reliability, Scalability, Portability, Re-usability, Usability) and evaluating the execution of software components to find the software bugs or errors or defects.

Software testing provides an independent view and objective of the software and gives surety of fitness of the software. It involves testing of all components under the required services to confirm that whether it is satisfying the specified requirements or not. The process is also providing the client with information about the quality of the software.

Testing is mandatory because it will be a dangerous situation if the software fails any of time due to lack of testing. So, without testing software cannot be deployed to the end user.

What is Testing

Testing is a group of techniques to determine the correctness of the application under the predefined script but, testing cannot find all the defect of application. The main intent of testing is to detect failures of the application so that failures can be discovered and corrected. It does not demonstrate that a product functions properly under all conditions but only that it is not working in some specific conditions.

Testing furnishes comparison that compares the behavior and state of software against mechanisms because the problem can be recognized by the mechanism. The mechanism may include past versions of the same specified product, comparable products, and interfaces of expected purpose, relevant standards, or other criteria but not limited up to these.

Testing includes an examination of code and also the execution of code in various environments, conditions as well as all the examining aspects of the code. In the current scenario of software development, a testing team may be separate from the development team so that Information derived from testing can be used to correct the process of software development.

The success of software depends upon acceptance of its targeted audience, easy graphical user interface, strong functionality load test, etc. For example, the audience of banking is totally different from the audience of a video game. Therefore, when an organization develops a software product, it can assess whether the software product will be beneficial to its purchasers and other audience.

Manual Testing

Manual testing is a software testing process in which test cases are executed manually without using any automated tool. All test cases executed by the tester manually according to the end user's perspective. It ensures whether the application is working as mentioned in the requirement document or not. Test cases are planned and implemented to complete almost 100 percent of the software application. Test case reports are also generated manually.

Manual Testing is one of the most fundamental testing processes as it can find both visible and hidden defects of the software. The difference between expected output and output, given by the software is defined as a defect. The developer fixed the defects and handed it to the tester for retesting.

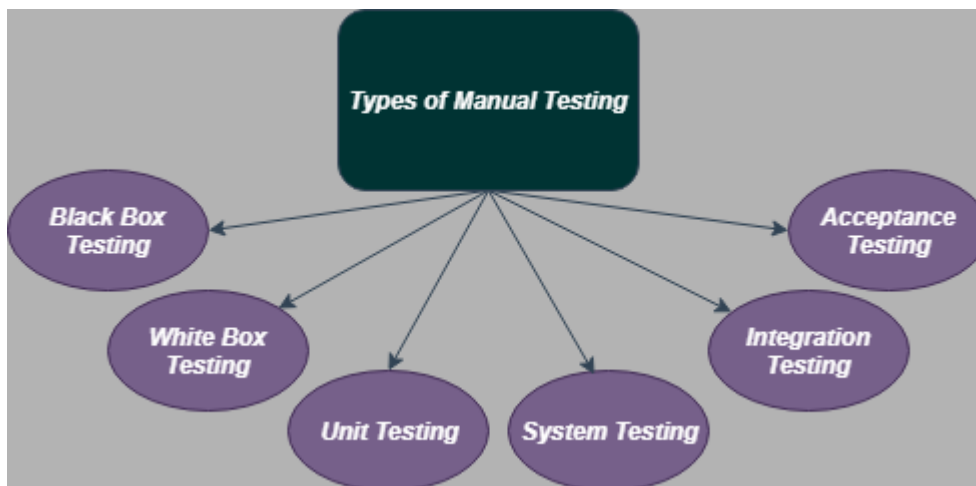
Manual testing is mandatory for every newly developed software before automated testing. This testing requires great efforts and time, but it gives the surety of bug-free software. Manual Testing requires knowledge of manual testing techniques but not of any automated testing tool.

Manual testing is essential because one of the software testing fundamentals is "100% automation is not possible."

There are various methods used for manual testing. Each method is used according to its testing criteria. Types of manual testing are given below:

Types of Manual Testing:

1. Black Box Testing
2. White Box Testing
3. Unit Testing
4. System Testing
5. Integration Testing
6. Acceptance Testing



How to perform Manual Testing

- First, tester examines all documents related to software, to select testing areas.
- Tester analyses requirement document to cover all requirements stated by the customer.
- Tester develops the test cases according to the requirement document.
- All test cases are executed manually by using Black box testing and white box testing.
- If bugs occurred then the testing team informs to the development team.
- Development team fixes bugs and handed software to the testing team for retesting.

Advantages of Manual Testing

- It does not require programming knowledge while using the Black box method.

- It is used to test dynamically changing GUI designs.
- Tester interacts with software as a real user so that they are able to discover usability and user interface issues.
- It ensures that the software is a hundred percent bug-free.
- It is cost effective.
- Easy to learn for new testers.

Disadvantages of Manual Testing

- It requires a large number of human resources.
- It is very time-consuming.
- Tester develops test cases based on their skills and experience. There is no evidence that they have covered all functions or not.
- Test cases cannot be used again. Need to develop separate test cases for each new software.
- It does not provide testing on all aspects of testing.
- Since two teams work together, sometimes it is difficult to understand each other's motives, it can mislead the process.

Manual testing tools

Selenium

Selenium is used to test the Web Application.

Appium

Appium is used to test the mobile application.

TestLink

TestLink is used for test management.

Postman

Postman is used for API testing.

Firebug

Firebug is an online debugger.

JMeter

JMeter is used for load testing of any application.

Mantis

Mantis is used for bug tracking.

Automation Testing

When the testing case suites are performed by using automated testing tools is known as Automation Testing. The testing process is done by using special automation tools to control the execution of test cases and compare the actual result with the expected result. Automation testing requires a pretty huge investment of resources and money.

Generally, repetitive actions are tested in automated testing such as regression tests. The testing tools used in automation testing are used not only for regression testing but also for automated GUI interaction, data set up generation, defect logging, and product installation.

The goal of automation testing is to reduce manual test cases but not to eliminate any of them. Test suits can be recorded by using the automation tools, and tester can play these suits again as per the requirement. Automated testing suites do not require any human intervention.

Advantages of Automation Testing

- Automation testing takes less time than manual testing.
- A tester can test the response of the software if the execution of the same operation is repeated several times.
- Automation Testing provides re-usability of test cases on testing of different versions of the same software.
- Automation testing is reliable as it eliminates hidden errors by executing test cases again in the same way.
- Automation Testing is comprehensive as test cases cover each and every feature of the application.
- It does not require many human resources, instead of writing test cases and testing them manually, they need an automation testing engineer to run them.
- The cost of automation testing is less than manual testing because it requires a few human resources.

Disadvantages of Automation Testing

- Automation Testing requires high-level skilled testers.
- It requires high-quality testing tools.
- When it encounters an unsuccessful test case, the analysis of the whole event is complicated.
- Test maintenance is expensive because high fee license testing equipment is necessary.

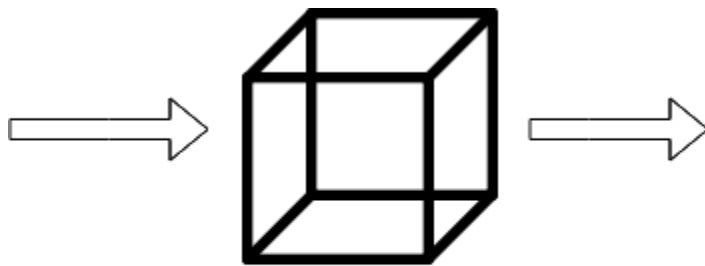
- Debugging is mandatory if a less effective error has not been solved, it can lead to fatal results.

White Box Testing

The box testing approach of software testing consists of black box testing and white box testing. We are discussing here white box testing which also known as glass box is **testing, structural testing, clear box testing, open box testing and transparent box testing**. It tests internal coding and infrastructure of a software focus on checking of predefined inputs against expected and desired outputs. It is based on inner workings of an application and revolves around internal structure testing. In this type of testing programming skills are required to design test cases. The primary goal of white box testing is to focus on the flow of inputs and outputs through the software and strengthening the security of the software.

The term 'white box' is used because of the internal perspective of the system. The clear box or white box or transparent box name denote the ability to see through the software's outer shell into its inner workings.

Test cases for white box testing are derived from the design phase of the software development lifecycle. Data flow testing, control flow testing, path testing, branch testing, statement and decision coverage all these techniques used by white box testing as a guideline to create an error-free software.



Whitebox Testing

White box testing follows some working steps to make testing manageable and easy to understand what the next task to do. There are some basic steps to perform white box testing.

Generic steps of white box testing

- Design all test scenarios, test cases and prioritize them according to high priority number.
- This step involves the study of code at runtime to examine the resource utilization, not accessed areas of the code, time taken by various methods and operations and so on.
- In this step testing of internal subroutines takes place. Internal subroutines such as nonpublic methods, interfaces are able to handle all types of data appropriately or not.

- This step focuses on testing of control statements like loops and conditional statements to check the efficiency and accuracy for different data inputs.
- In the last step white box testing includes security testing to check all possible security loopholes by looking at how the code handles security.

Reasons for white box testing

- It identifies internal security holes.
- To check the way of input inside the code.
- Check the functionality of conditional loops.
- To test function, object, and statement at an individual level.

Advantages of White box testing

- White box testing optimizes code so hidden errors can be identified.
- Test cases of white box testing can be easily automated.
- This testing is more thorough than other testing approaches as it covers all code paths.
- It can be started in the SDLC phase even without GUI.

Disadvantages of White box testing

- White box testing is too much time consuming when it comes to large-scale programming applications.
- White box testing is much expensive and complex.
- It can lead to production error because it is not detailed by the developers.
- White box testing needs professional programmers who have a detailed knowledge and understanding of programming language and implementation.

Black box testing

Black box testing is a technique of software testing which examines the functionality of software without peering into its internal structure or coding. The primary source of black box testing is a specification of requirements that is stated by the customer.

In this method, tester selects a function and gives input value to examine its functionality, and checks whether the function is giving expected output or not. If the function produces correct output, then it is passed in testing, otherwise failed. The test team reports the result to the development team and then tests the next function. After completing testing of all functions if there are severe problems, then it is given back to the development team for correction.



Generic steps of black box testing

- The black box test is based on the specification of requirements, so it is examined in the beginning.
- In the second step, the tester creates a positive test scenario and an adverse test scenario by selecting valid and invalid input values to check that the software is processing them correctly or incorrectly.
- In the third step, the tester develops various test cases such as decision table, all pairs test, equivalent division, error estimation, cause-effect graph, etc.
- The fourth phase includes the execution of all test cases.
- In the fifth step, the tester compares the expected output against the actual output.
- In the sixth and final step, if there is any flaw in the software, then it is cured and tested again.

Test procedure

The test procedure of black box testing is a kind of process in which the tester has specific knowledge about the software's work, and it develops test cases to check the accuracy of the software's functionality.

It does not require programming knowledge of the software. All test cases are designed by considering the input and output of a particular function. A tester knows about the definite output of a particular input, but not about how the result is arising. There are various techniques used in black box testing for testing like decision table technique, boundary value analysis technique, state transition, All-pair testing, cause-effect graph technique, equivalence partitioning technique, error guessing technique, use case technique and user story technique. All these techniques have been explained in detail within the tutorial.

Test cases

Test cases are created considering the specification of the requirements. These test cases are generally created from working descriptions of the software including requirements, design parameters, and other specifications. For the testing, the test designer selects both positive test scenario by taking valid input values and adverse test scenario by taking invalid input values to determine the correct output. Test cases are mainly designed for functional testing but can also be used for non-functional testing. Test cases are designed by the testing team, there is not any involvement of the development team of software.