Introduction of Programming Paradigms

Paradigm can also be termed as method to solve some problem or do some task. Programming paradigm is an approach to solve problem using some programming language or also we can say it is a method to solve a problem using tools and techniques that are available to us following some approach. There are lots for programming language that are known but all of them need to follow some strategy when they are implemented and this methodology/strategy is paradigms. Apart from varieties of programming language there are lots of paradigms to fulfill each and every demand. They are discussed below:

1. Imperative programming paradigm:

It is one of the oldest programming paradigm. It features close relation to machine architecture. It is based on Von Neumann architecture. It works by changing the program state through assignment statements. It performs step by step task by changing state. The main focus is on how to achieve the goal. The paradigm consists of several statements and after execution of all the result is stored.

Advantage:

1. Very simple to implement
2. It contains loops, variables etc.

Disadvantage:

1. Complex problem cannot be solved
2. Less efficient and less productive
3. Parallel programming is not possible

Imperative programming is divided into three broad categories: Procedural, OOP and parallel processing. These paradigms are as follows:

a. Procedural programming paradigm –
This paradigm emphasizes on procedure in terms of under lying machine model. There is no difference in between procedural and imperative approach. It has the ability to reuse the code and it was boon at that time when it was in use because of its reusability.

Examples of **Procedural** programming paradigm:

C : developed by Dennis Ritchie and Ken Thompson
C++ : developed by Bjarne Stroustrup
Java : developed by James Gosling at Sun Microsystems
ColdFusion : developed by J J Allaire
Pascal : developed by Niklaus Wirth

Then comes OOP,

b. Object oriented programming –
The program is written as a collection of classes and object which are meant for communication. The smallest and basic entity is object and all kind of computation is performed on the objects only. More emphasis is on data rather procedure. It can handle almost all kind of real life problems which are today in scenario.

Advantages:

- Data security
- Inheritance
- Code reusability
- Flexible and abstraction is also present

Examples of **Object Oriented** programming paradigm:

Simula : first OOP language
Java : developed by James Gosling at Sun Microsystems
C++ : developed by Bjarne Stroustrup
Objective-C : designed by Brad Cox
Visual Basic .NET : developed by Microsoft
Python : developed by Guido van Rossum
Ruby : developed by Yukihiro Matsumoto
Smalltalk : developed by Alan Kay, Dan Ingalls, Adele

2. Declarative programming paradigm:

It is divided as Logic, Functional, Database. In computer science the *declarative programming* is a style of building programs that expresses logic of computation without talking about its control flow. It often considers programs as theories of some logic. It may simplify writing parallel programs. The focus is on what needs to be done rather how it should be done basically emphasize on what code code is actually doing. It just declare the result we want rather how it has be produced. This is the only difference between imperative (how to do) and declarative (what to do) programming paradigms. Getting into deeper we would see logic, functional and database.

Logic programming paradigms –

It can be termed as abstract model of computation. It would solve logical problems like puzzles, series etc. In logic programming we have a knowledge base which we know before and along with the question and knowledge base which is given to machine, it produces result. In normal programming languages, such concept of knowledge base is not available but while using the concept of artificial intelligence, machine learning we have some models like Perception model which is using the same mechanism.

In logical programming the main emphasize is on knowledge base and the problem. The execution of the program is very much like proof of mathematical statement, e.g., Prolog

sum of two number in prolog:

```
predicates
sumoftwonumber(integer, integer)
clauses

sum(0, 0).
```

```
sum(n, r):-
    n1=n-1,
    sum(n1, r1),
        r=r1+n
```

Functional programming paradigms –
The functional programming paradigms has its roots in mathematics and it is language independent. The key principal of this paradigms is the execution of series of mathematical functions. The central model for the abstraction is the function which are meant for some specific computation and not the data structure. Data are loosely coupled to functions.The function hide their implementation. Function can be replaced with their values without changing the meaning of the program. Some of the languages like perl, javascript mostly uses this paradigm.

Examples of **Functional** programming paradigm:

JavaScript : developed by Brendan Eich
Haskwell : developed by Lennart Augustsson, Dave Barton
Scala : developed by Martin Odersky
Erlang : developed by Joe Armstrong, Robert Virding
Lisp : developed by John Mccarthy
ML : developed by Robin Milner
Clojure : developed by Rich Hickey

The next kind of approach is of Database.

Database/Data driven programming approach –
This programming methodology is based on data and its movement. Program statements are defined by data rather than hard-coding a series of steps. A database program is the heart of a business information system and provides file creation, data entry, update, query and reporting functions. There are several programming languages that are developed mostly for database application. For example SQL. It is applied to streams of structured data, for filtering, transforming, aggregating (such as computing statistics), or calling other programs. So it has its own wide application.

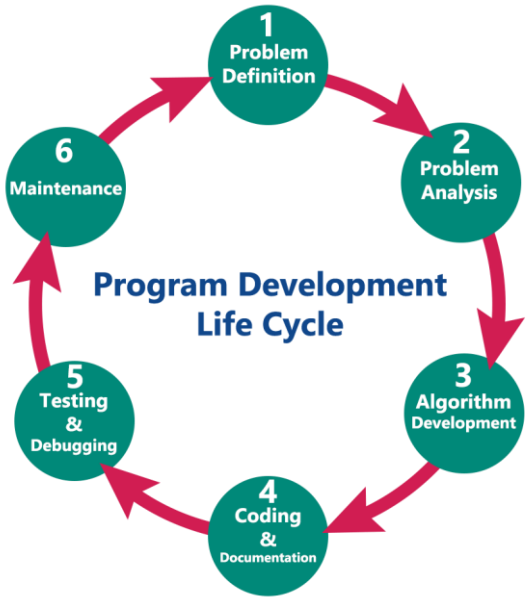
```
CREATE DATABASE databaseAddress;

CREATE TABLE Addr (
   PersonID int,
   LastName varchar(200),
   FirstName varchar(200),
   Address varchar(200),
   City varchar(200),
   State varchar(200)
    );
```

Program Development Life Cycle

When we want to develop a program using any programming language, we follow a sequence of steps. These steps are called phases in program development. The program development life cycle is a set of steps or phases that are used to develop a program in any programming language.
Generally, program development life cycle contains 6 phases, they are as follows….

- Problem Definition
- Problem Analysis
- Algorithm Development
- Coding & Documentation
- Testing & Debugging
- Maintenance



1. Problem Definition

In this phase, we define the problem statement and we decide the boundaries of the problem. In this phase we need to understand the problem statement, what is our requirement, what should be the output of the problem solution? These are defined in this first phase of the program development life cycle.

2. Problem Analysis

In phase 2, we determine the requirements like variables, functions, etc. to solve the problem. That means we gather the required resources to solve the problem defined in the problem definition phase. We also determine the bounds of the solution.

3. Algorithm Development

During this phase, we develop a step by step procedure to solve the problem using the specification given in the previous phase. This phase is very important for program development. That means we write the solution in step by step statements.

4. Coding & Documentation

This phase uses a programming language to write or implement actual programming instructions for the steps defined in the previous phase. In this phase, we construct actual program. That means we write the program to solve the given problem using programming languages like C, C++, Java etc.,

5. Testing & Debugging

During this phase, we check whether the code written in previous step is solving the specified problem or not. That means we test the program whether it is solving the problem for various input data values or not. We also test that whether it is providing the desired output or not.

6. Maintenance

During this phase, the program is actively used by the users. If any enhancements found in this phase, all the phases are to be repeated again to make the enhancements. That means in this phase, the solution (program) is used by the end user. If the user encounters any problem or wants any enhancement, then we need to repeat all the phases from the starting, so that the encountered problem is solved or enhancement is added.

System Development Life Cycle (SDLC)

SDLC, Software Development Life Cycle is a process used by software industry to design, develop and test high quality software. The SDLC aims to produce a high quality software that meets or exceeds customer expectations, reaches completion within times and cost estimates.

- SDLC is the acronym of Software Development Life Cycle.
- It is also called as Software development process.
- The software development life cycle (SDLC) is a framework defining tasks performed at each step in the software development process.
- It aims to be the standard that defines all the tasks required for developing and maintaining software.

SDLC is a process followed for a software project, within a software organization. It consists of a detailed plan describing how to develop, maintain, replace and alter or enhance specific software. The life cycle defines a methodology for improving the quality of software and the overall development process.

Importance and the necessity of SDLC

SDLC is important due to the following reasons:

- It breaks down the entire life cycle of software
- Development makes easier to evaluate each part of software development
- It also makes easier for programmer to work concurrently on each phase
- Provide a rough time when software will be available for use

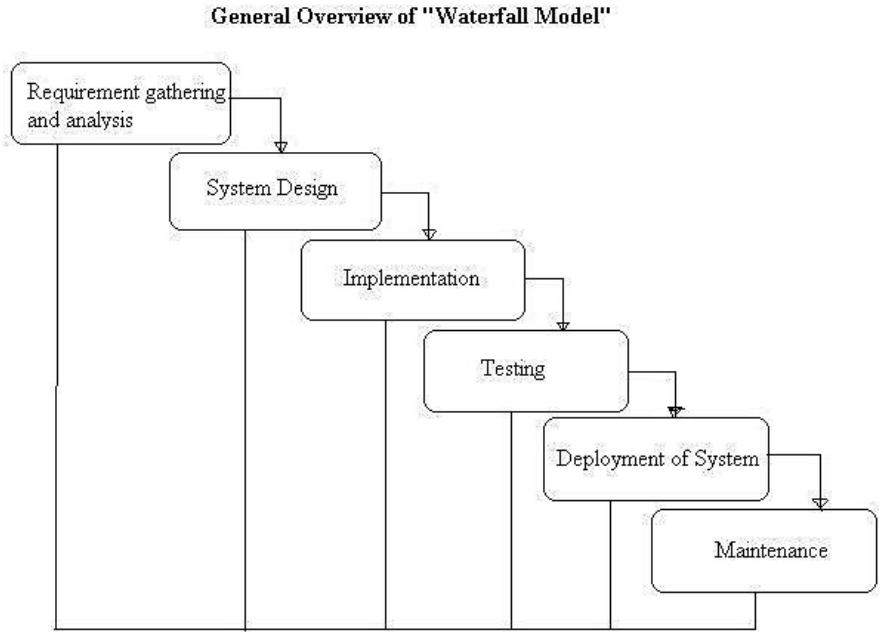
System Development Models:

The different models of SDLC are as follows:

a) Waterfall Model

The Waterfall Model was first Process Model to be introduced. It is also referred to as a **linear-sequential life cycle model**. It is very simple to understand and use. In a waterfall model, each phase must be completed fully before the next phase can begin. This type of model is basically used for the for the project which is small and there are no uncertain requirements. At the end of each phase, a review takes place to determine if the project is on the right path and whether or not to continue or discard the project. In this model the testing starts only after the development is complete. In **waterfall model phases** do not overlap.

Diagram of Waterfall-model:



Advantages of waterfall model:

- This model is simple and easy to understand and use.
- It is easy to manage due to the rigidity of the model – each phase has specific deliverables and a review process.
- In this model phases are processed and completed one at a time. Phases do not overlap.
- Waterfall model works well for smaller projects where requirements are very well understood.

Disadvantages of waterfall model:

- Once an application is in the testing stage, it is very difficult to go back and change something that was not well-thought out in the concept stage.

- No working software is produced until late during the life cycle.
- High amounts of risk and uncertainty.
- Not a good model for complex and object-oriented projects.
- Poor model for long and ongoing projects.
- Not suitable for the projects where requirements are at a moderate to high risk of changing.

When to use the waterfall model:

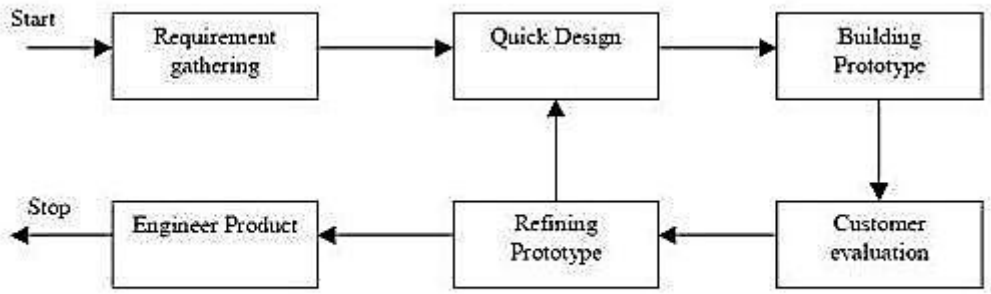
- This model is used only when the requirements are very well known, clear and fixed.
- Product definition is stable.
- Technology is understood.
- There are no ambiguous requirements
- Ample resources with required expertise are available freely
- The project is short.

b) Prototype model

The basic idea here is that instead of freezing the requirements before a design or coding can proceed, a throwaway prototype is built to understand the requirements. This prototype is developed based on the currently known requirements. By using this prototype, the client can get an "actual feel" of the system, since the interactions with prototype can enable the client to better understand the requirements of the desired system. Prototyping is an attractive idea for complicated and large systems for which there is no manual process or existing system to help determining the requirements.

The prototypes are usually not complete systems and many of the details are not built in the prototype. The goal is to provide a system with overall functionality.

Diagram of Prototype model:



Prototyping Model

Advantages of Prototype model:

- Users are actively involved in the development
- Since in this methodology a working model of the system is provided, the users get a better understanding of the system being developed.
- Errors can be detected much earlier.
- Quicker user feedback is available leading to better solutions.
- Missing functionality can be identified easily

- Confusing or difficult functions can be identifiedRequirements validation, Quick implementation of, incomplete, butfunctional, application.

Disadvantages of Prototype model:

- Leads to implementing and then repairing way of building systems.
- Practically, this methodology may increase the complexity of the system as scope of the system may expand beyond original plans.
- Incomplete application may cause application not to be used as thefull system was designedIncomplete or inadequate problem analysis.

When to use Prototype model:

- Prototype model should be used when the desired system needs to have a lot of interaction with the end users.
- Typically, online systems, web interfaces have a very high amount of interaction with end users, are best suited for Prototype model. It might take a while for a system to be built that allows ease of use and needs minimal training for the end user.
- Prototyping ensures that the end users constantly work with the system and provide a feedback which is incorporated in the prototype to result in a useable system. They are excellent for designing good human computer interface systems.

c) Spiral model

The spiral model is similar to the **incremental model**, with more emphasis placed on risk analysis. The spiral model has four phases: Planning, Risk Analysis, Development and Verification. A software project repeatedly passes through these phases in iterations (called Spirals in this model). The baseline spiral, starting in the planning phase, requirements are gathered and risk is assessed.

It has four phases

Planning Phase:

Requirements are gathered during the planning phase. Requirements like 'BRS' that is 'Bussiness Requirement Specifications' and 'SRS' that is 'System Requirement specifications'.

Risk Analysis:

In the **risk analysis phase**, a process is undertaken to identify risk and alternate solutions. A prototype is produced at the end of the risk analysis phase. If any risk is found during the risk analysis then alternate solutions are suggested and implemented.

Development Phase: In this phase software is **developed**, along with **testing** at the end of the phase. Hence in this phase the development and testing is done.

Verification phase: This phase allows the customer to evaluate the output of the project to date before the project continues to the next spiral.

Diagram of Spiral model:

Advantages of Spiral model:

- High amount of risk analysis hence, avoidance of Risk is enhanced.
- Good for large and mission-critical projects.
- Strong approval and documentation control.
- Additional Functionality can be added at a later date.
- Software is produced early in the software life cycle.

Disadvantages of Spiral model:

- Can be a costly model to use.
- Risk analysis requires highly specific expertise.
- Project's success is highly dependent on the risk analysis phase.
- Doesn't work well for smaller projects.

When to use Spiral model:

- When costs and risk evaluation is important
- For medium to high-risk projects
- Long-term project commitment unwise because of potential changes to economic priorities
- Users are unsure of their needs
- Requirements are complex
- New product line
- Significant changes are expected (research and exploration)

System Development Phase

System Development Life Cycle (SDLC) is a series of six main phases to create a hardware system only, a software system only or a combination of both to meet or exceed customer's expectations. Software Development Life Cycle is a limited term that explains the phases of creating a software component that integrates with other software components to create the whole system.

Below we'll take a general look on System Development Life Cycle phases, bearing in mind that each system is different from the other in terms of complexity, required components and expected solutions and functionalities:

System Development Life Cycle Phases:

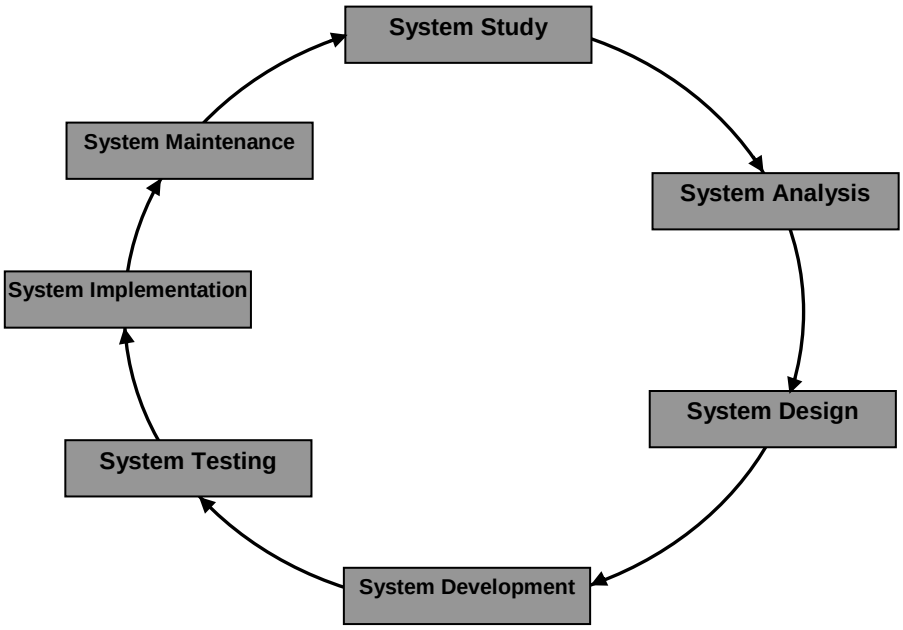


Fig: Stages of SDLC

System Study

The Study phase is the most crucial step in creating a successful system.During this phase you decide exactly what you want to do and the problems you're trying to solve by defining the problems, the objectives and the resources such as personnel and costs.

Studying the ability of proposing alternative solutions after meeting with clients, suppliers, consultants and employees. Studying how to make your product better than your competitors'. After analyzing this data you will have three choices: develop a new system, improve the current system or leave the system as it is.

System Analysis

The end-user's requirements should be determined and documented, what their expectations are for the system, and how it will perform. A feasibility study will be made for the project as well, involving determining whether it's organizationally, economically, socially, technologically feasible. it's very important to maintain strong communication level with the clients to make sure you have a clear vision of the finished product and its function.

Feasibility Study

Feasibility is defined as the practical extent to which a project can be performed successfully. To evaluate feasibility, a feasibility study is performed, which determines whether the solution considered to accomplish the requirements is practical and workable in the software. Information such as resource availability, cost estimation for software development, benefits of the software to the organization after it is developed and cost to be incurred on its maintenance are considered during the feasibility study. The objective of the feasibility study is to establish the reasons for developing the software that is acceptable to users, adaptable to change and conformable to established standards.

A well-designed feasibility study should provide a historical background of the business or project, a description of the product or service, accounting statements, details of the operations

and management, marketing research and policies, financial data, legal requirements and tax obligations. Generally, feasibility studies precede technical development and project implementation.

Various other objectives of feasibility study are listed below.

- To analyze whether the software will meet organizational requirements
- To determine whether the software can be implemented using the current technology and within the specified budget and schedule
- To determine whether the software can be integrated with other existing software.

Types of Feasibility

1. Technical feasibility:

Technical feasibility assesses the current resources (such as hardware and software) and technology, which are required to accomplish user requirements in the software within the allocated time and budget. For this, the software development team ascertains whether the current resources and technology can be upgraded or added in the software to accomplish specified user requirements. Technical feasibility also performs the following tasks.

- Analyzes the technical skills and capabilities of the software development team members
- Determines whether the relevant technology is stable and established
- Ascertains that the technology chosen for software development has a large number of users so that they can be consulted when problems arise or improvements are required.

2. Operational feasibility:

Operational feasibility assesses the extent to which the required software performs a series of steps to solve business problems and user requirements. This feasibility is dependent on human resources (software development team) and involves visualizing whether the software will operate after it is developed and be operative once it is installed. Operational feasibility also performs the following tasks.

- Determines whether the problems anticipated in user requirements are of high priority
- Determines whether the solution suggested by the software development team is acceptable
- Analyzes whether users will adapt to a new software
- Determines whether the organization is satisfied by the alternative solutions proposed by the software development team.

3. Economic feasibility

Economic feasibility determines whether the required software is capable of generating financial gains for an organization. It involves the cost incurred on the software development team, estimated cost of hardware and software, cost of performing feasibility study, and so on. For this, it is essential to consider expenses made on purchases (such as hardware purchase) and activities required to carry out software development. In addition, it is necessary to consider the benefits that can be achieved by developing the software. Software is said to be economically feasible if it focuses on the issues listed below.

- Cost incurred on software development to produce long-term gains for an organization
- Cost required to conduct full software investigation (such as requirements elicitation and requirements analysis)
- Cost of hardware, software, development team, and training.

4. **Legal feasibility**

Legal feasibility Determines whether the proposed system conflicts with legal requirements, e.g. a data processing system must comply with the local data protection regulations and if the proposed venture is acceptable in accordance to the laws of the land.

1. Schedule feasibility

A project will fail if it takes too long to be completed before it is useful. Typically this means estimating how long the system will take to develop, and if it can be completed in a given time period using some methods like payback period. Schedule feasibility is a measure of how reasonable the project timetable is. Given our technical expertise, are the project deadlines reasonable? Some projects are initiated with specific deadlines. It is necessary to determine whether the deadlines are mandatory or desirable.

6. **Resource Feasibility**

Do you have enough resources, what resources will be required, what facilities will be required for the project, etc.

System Design

The design phase comes after a good understanding of customer's requirements, this phase defines the elements of a system, the components, the security level, modules, architecture and the different interfaces and type of data that goes through the system.

A general system design can be done with a pen and a piece of paper to determine how the system will look like and how it will function, and then a detailed and expanded system design is produced, and it will meet all functional and technical requirements, logically and physically.

System Development

System Development is the process of defining, designing, testing and implementing a software application. This includes the internal development of customized systems as well as the acquisition of software developed by third parties. System development is also referred to as software development, software engineering or application development. System development includes the management of the entire process of the development of computer software.

Consider the example of a large organization that wants to streamline purchase orders. Right now every department has its own systems in place, which includes a variety of approaches that have been developed over the years. After a few serious errors, senior management has decided to centralize the ordering process into a single system that every department will have access to.

System Testing

Bringing different components and subsystems together to create the whole integrated system, and then introducing the system to different inputs to obtain and analyze its outputs and behavior and the way it functions. Testing is becoming more and more important to ensure customer's satisfaction, and it requires no knowledge in coding, hardware configuration or design.

Testing can be performed by real users, or by a team of specialized personnel, it can also be systematic and automated to ensure that the actual outcomes are compared and equal to the predicted and desired outcomes.

System Implementation

This phase comes after a complete understanding of system requirements and specifications.It's the actual construction process after having a complete and illustrated design for the requested system.

In the Software Development Life Cycle, the actual code is written here, and if the system contains hardware, then the implementation phase will contain configuration and fine-tuning for the hardware to meet certain requirements and functions.

In this phase, the system is ready to be deployed and installed in customer's premises, ready to become running, live and productive, training may be required for end users to make sure they know how to use the system and to get familiar with it, the implementation phase may take a long time and that depends on the complexity of the system and the solution it presents.

System Maintenance

In this phase, periodic maintenance for the system will be carried out to make sure that the system won't become obsolete, this will include replacing the old hardware and continuously evaluating system's performance, it also includes providing latest updates for certain components to make sure it meets the right standards and the latest technologies to face current security threats.

These are the main six phases of the System Development Life Cycle, and it's an iterative process for each project. It's important to mention that excellent communication level should be maintained with the customer, and Prototypes are very important and helpful when it comes to meeting the requirements. By building the system in short iterations; we can guarantee meeting the customer's requirements before we build the whole system.

Concept of System Design Tools:

Tools used to develop a program is known as System Design Tools.

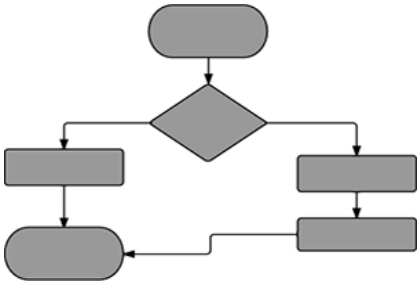
The different types of System Design Tools are

1) Algorithm

When you are telling the computer *what* to do, you also get to choose *how* it's going to do it. That's where **computer algorithms** come in. The algorithm is the basic technique used to get the job done. Let's follow an example to help get an understanding of the algorithm concept.

An algorithm is a formula or set of steps for solving a particular problem. To be an algorithm, a set of rules must be unambiguous and have a clear stopping point. Algorithms can be expressed in any language from natural language like English or French to programming language like FORTAN.

2) Flowchart



A flowchart is a formalized graphic representation of a logic sequence, work or manufacturing process, organization chart, or similar formalized structure. The purpose of a flow chart is to provide people with a common language or reference point when dealing with a project or process.

Flowcharts use simple geometric symbols and arrows to define relationships. In programming, for instance, the beginning or end of a program is represented by an oval. A process is represented by a rectangle, a decision is represented by a diamond and an I/O process is represented by a parallelogram.

A flowchart is a diagram that represents a process or algorithm. The steps are represented by a series of boxes or other specialized symbols, then connected with arrows.

3) Context Diagram

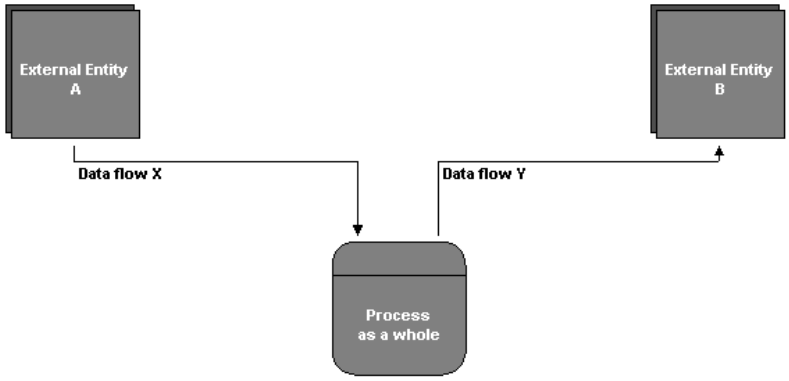
A context diagram is a data flow diagram, with only one massive central process that subsumes everything inside the scope of the system. It shows how the system will receive and send data flows to the external entities involved

Context diagrams are a subset of data flow diagrams, which often require a high level of technical knowledge to make and understand.

Here's a theoretical example:

Components

A Context diagram can be assembled from the followingthree components:



Process: A process is a logical activity that can be regarded as a black box with major data flowing in and out of it. A rounded rectangle (or circle) represents a process under study. Each process is labelled inside its rectangle to describe its function or purpose.

External entity: An external entity sits outside the domain of interest and supplies data to or receives data from the domain. An external entity is referred to as an external source or sink (destination) for data flowing in and out of the domain. A rectangle defines an external entity and is labelled with a noun phrase inside it to describe an organization, process, machine or person (i.e. a thing) that is outside the domain under analysis. Examples of naming an external entity are 'Payment company', 'Store locator', 'Mainframe server' and 'Customer'.

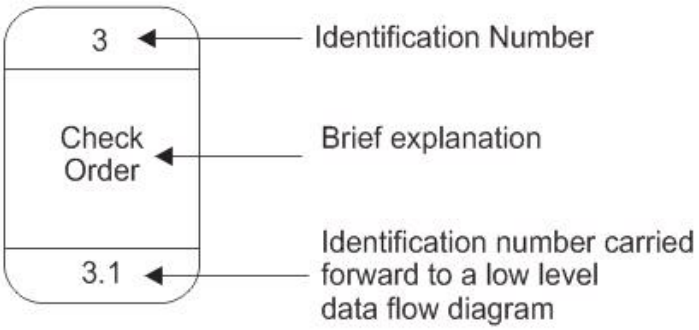
Data flow: A data flow represents the path of data moving through the domain under analysis. A data flow shows the movement of data between a process and an external entity. An arrow is

the symbol used to connect a process with external entities. Each arrow should be labelled appropriately to describe the data being passed, e.g. 'Customer details', 'Rejected order'

4) Data flow diagram

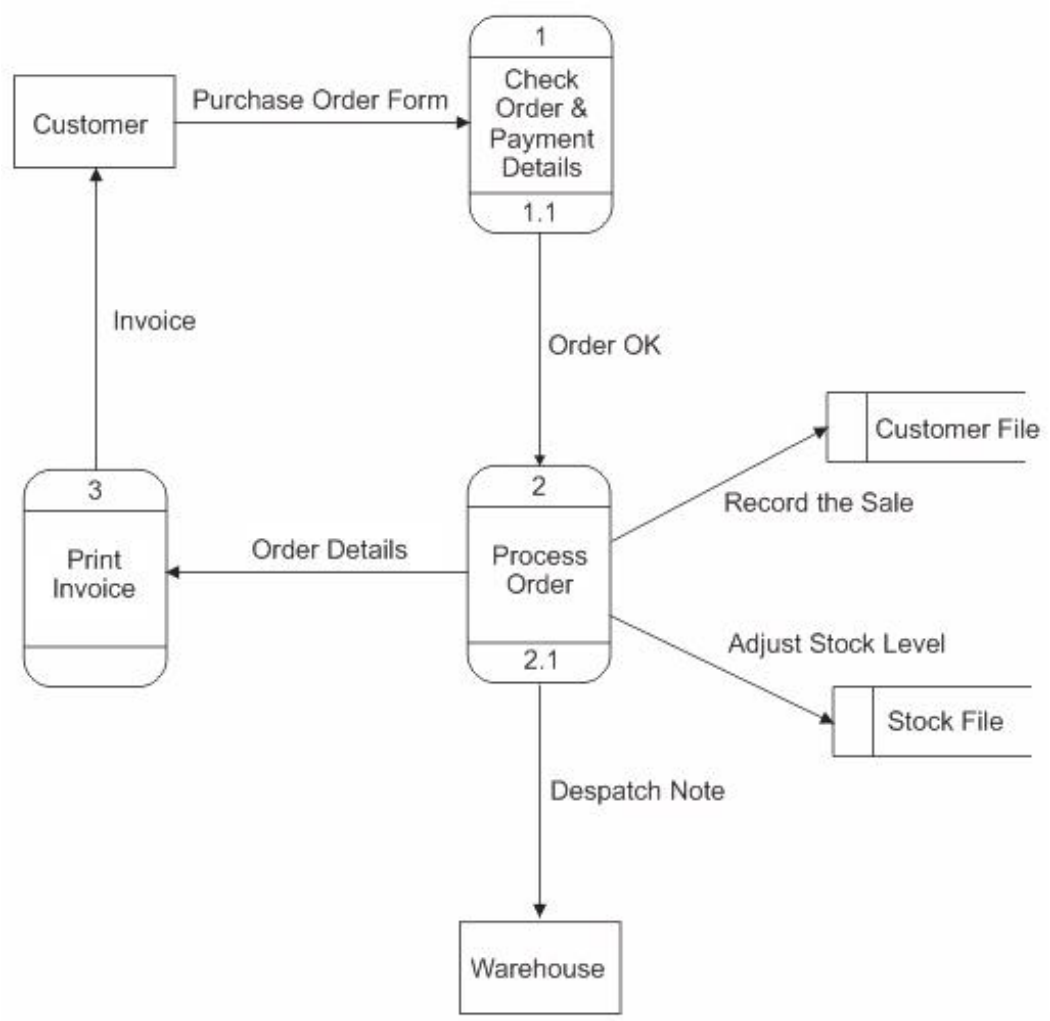
A data flow diagram (DFD) illustrates how data is processed by a system in terms of inputs and outputs. As its name indicates its focus is on the flow of information, where data comes from, where it goes and how it gets stored. DFDs show the flow of data from external entities into the system

The following symbols are used in a data flow diagram:

Symbol	Meaning	Example
	An **entity**. A source of data or a destination for data.	
	A **process** or task that is performed by the system.	
	A **data store**, a place where data is held between processes.	
	A **data flow**.	

		Invoice →

The diagram below shows a data flow diagram for an ICT system that is used to manage the sale, supply and payment for goods in a business.



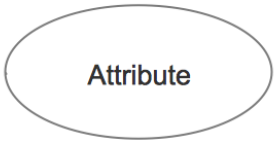
5) E-R Diagram

An *entity relationship model*, also called an *entity-relationship (ER) diagram*, is a graphical representation of entities and their relationships to each other, typically used in computing in

regard to the organization of data within databases or information systems. An entity is a piece of data-an object or concept about which data is stored.

An entity-relationship diagram (ERD) is a graphical representation of an information system that shows the relationship between people, objects, places, concepts or events within that system. An ERD is a data modeling technique that can help define business processes and can be used as the foundation for a relational database.

The symbols used in E-R diagram are

Symbol	Shape Name	Symbol Description
	Entity	An entity is represented by a rectangle which contains the entity's name.
	Attribute	Each attribute is represented by an oval containing attribute's name
	relationship	A relationship where entity is existence-independent of other entities.

2. Decision Table

A **Decision table** is a good way to deal with combinations of things (e.g. inputs). This technique is sometimes also referred to as a 'cause-effect' table. The reason for this is that there is an associated logic diagramming technique called 'cause-effect graphing' which was sometimes used to help derive the decision table .. However, most people find it more useful just to use the

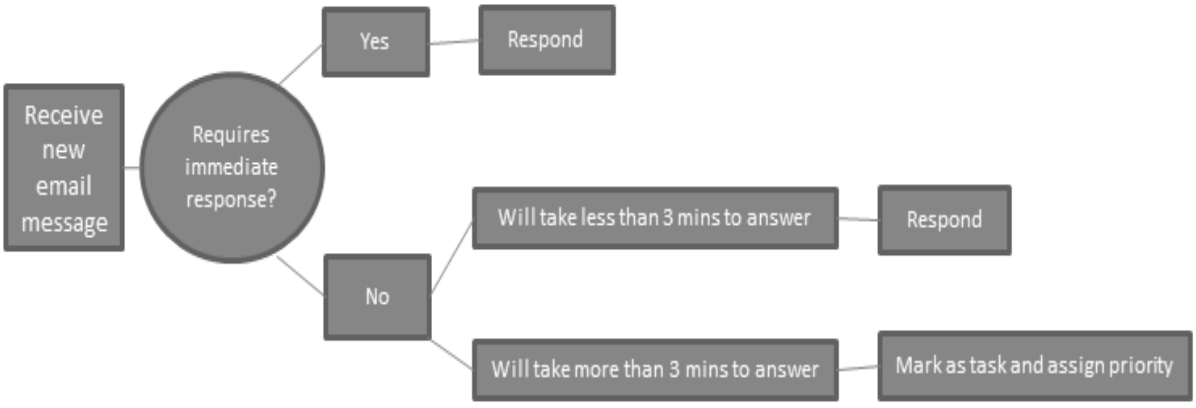
- Decision tables provide a systematic way of stating complex business rules, which is useful for developers as well as for testers.
- Decision tables can be used in test design whether or not they are used in specifications, as they help testers explore the effects of combinations of different inputs and other software states that must correctly implement business rules.
- It helps the developers to do a better job can also lead to better relationships with them. Testing combinations can be a challenge, as the number of combinations can

often be huge. Testing all combinations may be impractical if not impossible. We have to be satisfied with testing just a small subset of combinations but making the choice of which combinations to test and which to leave out is also important. If you do not have a systematic way of selecting combinations, an arbitrary subset will be used and this may well result in an ineffective test effort.

Table: Decision table for credit card example

Conditions	Rule 1	Rule 2	Rule 3	Rule 4	Rule 5	Rule 6	Rule 7	Rule 8
New customer (15%)	T	T	T	T	F	F	F	F
Loyalty card (10%)	T	T	F	F	T	T	F	F
Coupon (20%)	T	F	T	F	T	F	T	F
Actions								
Discount (%)	X	X	20	15	30	10	20	0

3. Decision Tree



A decision tree is a graph that uses a branching method to illustrate every possible outcome of a decision.

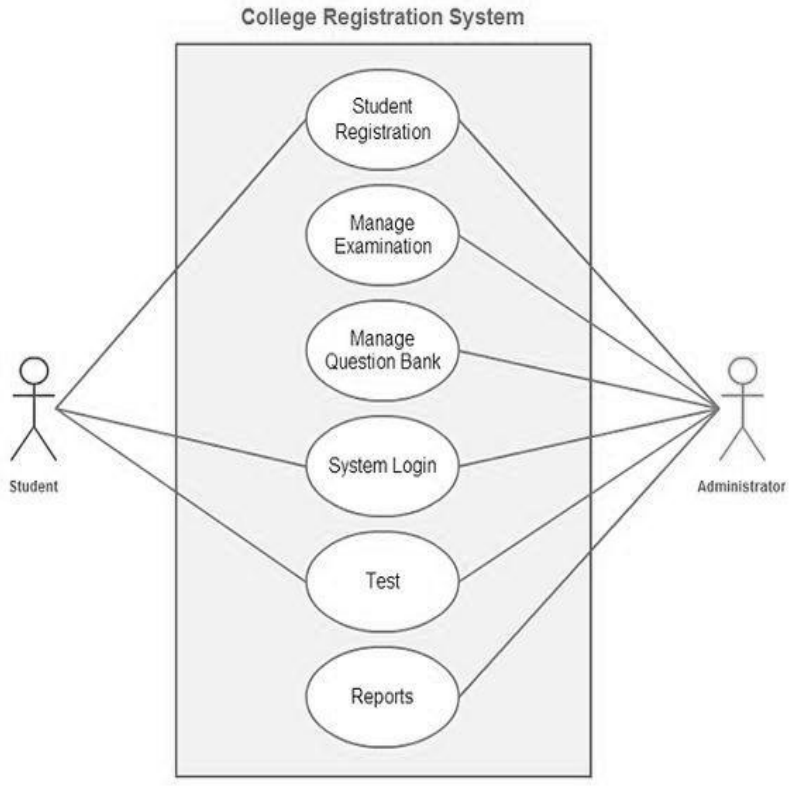
A decision tree is a schematic tree-shaped diagram used to determine a course of action or show a statistical probability. Each branch of the decision tree represents a possible decision or occurrence. The tree structure shows how one choice leads to the next, and the use of branches indicates that each option is mutually exclusive.

Decision trees can be drawn by hand or created with a graphics program or specialized software. Informally, decision trees are useful for focusing discussion when a group must make a decision. Programmatically, they can be used to assign monetary/time or other values to possible outcomes so that decisions can be automated.

Here's a simple example: An email management decision tree might begin with a box labeled "Receive new message." From that, one branch leading off might lead to "Requires immediate

response." From there, a "Yes" box leads to a single decision: "Respond." A "No" box leads to "Will take less than three minutes to answer" or "Will take more than three minutes to answer." From the first box, a box leads to "Respond" and from the second box, a branch leads to "Mark as task and assign priority." The branches might converge after that to "Email responded to? File or delete message."

8) Use Case



A Use Case diagram is a graphic depiction of the interactions among the elements of a system. A use case is a methodology used in system analysis to identify, clarify, and organize system requirements. In this context, the term "system" refers to something being developed or operated, such as a mail-order product sales and service Web site. Use case diagrams are employed in UML (Unified Modeling Language), a standard notation for the modeling of real-world objects and systems.

System objectives can include planning overall requirements, validating a hardware design, testing and debugging a software product under development, creating an online help reference, or performing a consumer-service-oriented task. For example, use cases in a product sales environment would include item ordering, catalog updating, payment processing, and customer relations. A use case diagram contains four components.

- The boundary, which defines the system of interest in relation to the world around it.
- The actors, usually individuals involved with the system defined according to their roles.
- The use cases, which are the specific roles played by the actors within and around the system.
- The relationships between and among the actors and the use cases.

9. UML (Unified Modeling Language)

UML (Unified Modeling Language) is a standard notation for the modeling of real-world objects as a first step in developing an object-oriented design methodology. Its notation is derived from and unifies the notations of three object-oriented design and analysis methodologies:

Unified Modeling language (UML) is a standardized modeling language enabling developers to specify, visualize, construct and document artifacts of a software system. Thus, UML makes these artifacts scalable, secure and robust in execution. UML is an important aspect involved in object-oriented software development. It uses graphic notation to create visual models of software systems.

UML is designed to enable users to develop an expressive, ready to use visual modeling language. In addition, it supports high level development concepts such as frameworks, patterns and collaborations. UML includes a collection of elements such as:

- Programming Language Statements
- Actors: specify a role played by a user or any other system interacting with the subject.
- Activities: These are tasks, which must take place in order to fulfill an operation contract. They are represented in activity diagrams.
- Business Process: includes a collection of tasks producing a specific service for customers and is visualized with a flowchart as a sequence of activities.
- Logical and Reusable Software Components.

Documentation

The term is derived from the idea that engineers and programmers "document" their products in formal writing. The earliest computer users were sometimes simply handed the engineers' or programmers' "documentation."

Documentation is a set of documents provided on paper, or online, or on digital or analog media, such as audio tape or CDs. Examples are user guides, white papers, on-line help, quick-reference guides. It is becoming less common to see paper (hard-copy) documentation. Documentation is distributed via websites, software products, and other on-line applications.

Types

User Documentation

Also known as software manuals, user documentation is intended for end users and aims to help them use software properly. It is usually arranged in a book-style and typically also features table of contents, index and of course, the body which can be arranged in different ways, depending on whom the software is intended for. For example, if the software is intended for beginners, it usually uses a tutorial approach and guides the user step-by-step. Software manuals which are intended for intermediate users, on the other hand, are typically arranged thematically, while manuals for advanced users follow reference style. Besides printed version, user documentation can also be available in an online version or PDF format. Often, it is also accompanied by additional documentation such as video tutorials, knowledge based articles, videos, etc.

Requirements Documentation

Requirements documentation, also referred to simply as requirements explains what a software does and shall be able to do. Several types of requirements exist which may or may not be included in documentation, depending on purpose and complexity of the system. For example,

applications that don't have any safety implications and aren't intended to be used for a longer period of time may be accompanied by little or no requirements documentation at all. Those that can affect human safety or/and are created to be used over a longer period of time, on the other hand, come with an exhausting documentation.

Architecture Documentation

Also referred to as software architecture description, architecture documentation either analyses software architectures or communicates the results of the latter (work product). It mainly deals with technical issues including online marketing and services but it also covers non-technical issues in order to provide guidance to system developers, maintenance technicians and others involved in the development or use of architecture including end users. Architecture documentation is usually arranged into architectural models which in turn may be organized into different views, each of which deals with specific issues.

Comparison document is closely related to architecture documentation. It addresses current situation and proposes alternative solutions with an aim to identify the best possible outcome. In order to be able to do that, it requires an extensive research.

Technical Documentation

Technical documentation is a very important part of software documentation and many programmers use both terms interchangeably despite the fact that technical documentation is only one of several types of software documentation. It describes codes but it also addresses algorithms, interfaces and other technical aspects of software development and application.

Programming Techniques

Software designing is very anesthetic phase of *software development cycle*. The beauty of heart, skill of mind and practical thinking is mixed with system objective to implement design.

The *designing process* is not simple, but complex, cumbersome and frustrating with many curves in the way of successful design.

Here are some approaches:

- Top Down Designing
- Bottom Up Designing

The objective of Program design is:

(i) **Replace old system**: The new system is used to replace old system because maintenance cost is high in old system and efficiency level low.

(ii) **Demand of Organization**: The new system is developed and installed on the demand of organization and working groups.

(iii) **Productivity**: The new system is installed to increase productivity of company or organization.

(iv) **Competition**: The new system is a matter of status also. In the age of roaring competition, if organization does not cope with modem technology failed to face competitions.

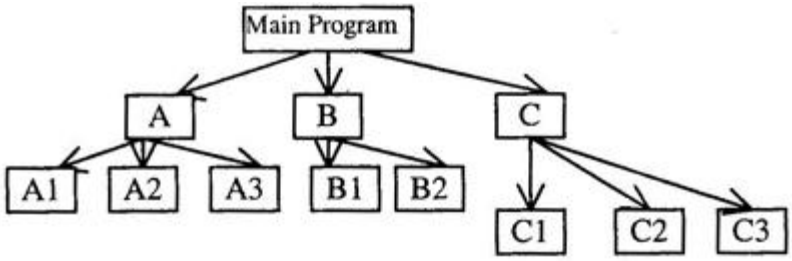
(v) **Maintenance**: The new system is needed to maintain organization status.

Top down Approach

(a) The large program is divided into many small **module** or subprogram or function or procedure from top to bottom.

(b) At first supervisor program is identified to control other sub modules. Main modules are divided into sub modules, sub-modules into sub- sub- modules. The decomposition of modules is continuing whenever desired module level is not obtained.

(c) Top module is tested first, and then sub-modules are combined one by one and tested.



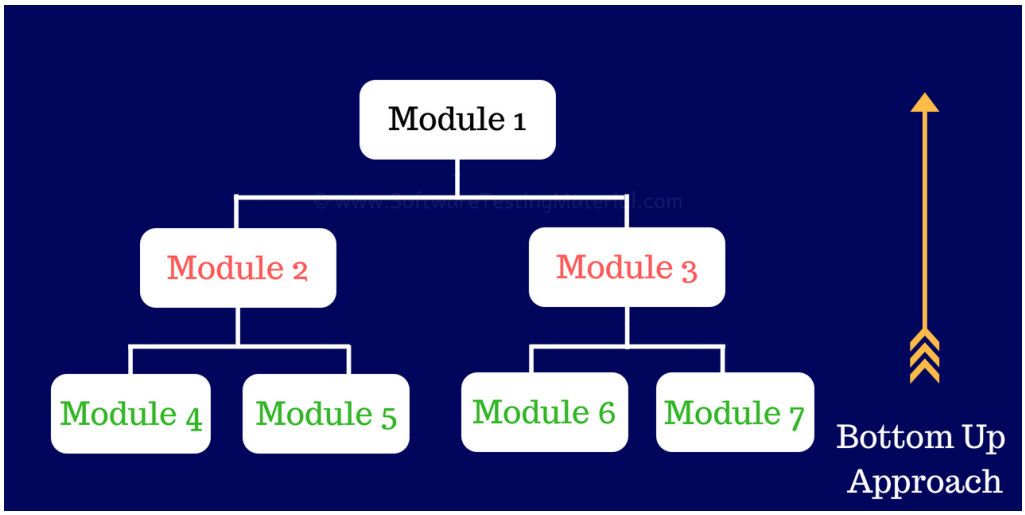
Top down Approach

Example: The main program is divided into sub-program A, B, and C. The A is divided into subprogram A1, A2 and A3.The B is into B1, and B2. Just like these subprograms, C is also divided into three subprogram C1, C2 and C3. The solution of Main *program* is obtained from sub program A, B and C.

Bottom up Approach

- In this approach designing is started from bottom and advanced stepwise to top. So, this approach is called Bottom up approach.
- At first bottom layer modules are designed and tested, second layer modules are designed and combined with bottom layer and combined modules are tested. In this way, designing and testing progressed from bottom to top.

- In software designing, only pure top down or Bottom up approach is not used. The hybrid type of approach is recommended by many designers in which top down and bottom up, both approaches are utilized.



Difference between Top-down and Bottom-up Approach

Top-Down Approach	Bottom-Up Approach
Divides a problem into smaller units and then solve it.	Starts from solving small modules and adding them up together.
This approach contains redundant information.	Redundancy can easily be eliminated.
A well-established communication is not required.	Communication among steps is mandatory.
The individual modules are thoroughly analyzed.	Works on the concept of data-hiding and encapsulation.
Structured programming languages such as C	OOP languages like C++ and Java, etc. uses

use top-down approach.	bottom-up mechanism.
Relation among modules is not always required.	The modules must be related for better communication and work flow.
Primarily used in code implementation, test case generation, debugging and module documentation.	Finds use primarily in testing.

COHESION
Measure of how well module fits together.

A component should implement a single logical function or single logical entity. All the parts should contribute to the implementation.

Many levels of cohesion:

1. Coincidental cohesion: the parts of a component are not related but simply bundled into a single component and harder to understand and not reusable.
2. Logical association: similar functions such as input, error handling, etc. put together. Functions fall in same logical class. May pass a flag to determine which ones executed. Interface difficult to understand. Code for more than one function may be intertwined, leading to severe maintenance problems. Difficult to reuse
3. Temporal cohesion: all of statements activated at a single time, such as start up or shut down, are brought together. Initialization, clean up.

   Functions weakly related to one another, but more strongly related to functions in other modules so may need to change lots of modules when do maintenance.

4. Procedural cohesion: a single control sequence, e.g., a loop or sequence of decision statements. Often cuts across functional lines. May contain only part of a complete function or parts of several functions. Functions still weakly connected, and again unlikely to be reusable in another product.
5. Communicational cohesion: operate on same input data or produce same output data. May be performing more than one function. Generally acceptable if alternate structures with higher cohesion cannot be easily identified. Still problems with reusability.
6. Sequential cohesion: output from one part serves as input for another part. May contain several functions or parts of different functions.

7. Informational cohesion: performs a number of functions, each with its own entry point, with independent code for each function, all performed on same data structure. Different than logical cohesion because functions not intertwined.
8. Functional cohesion: each part necessary for execution of a single function. e.g., compute square root or sort the array. Usually reusable in other contexts. Maintenance easier.
9. Type cohesion: modules that support a data abstraction.

Coupling

In software engineering, **coupling** is the degree of interdependence between software modules; a measure of how closely connected two routines or modules are; the strength of the relationships between modules.

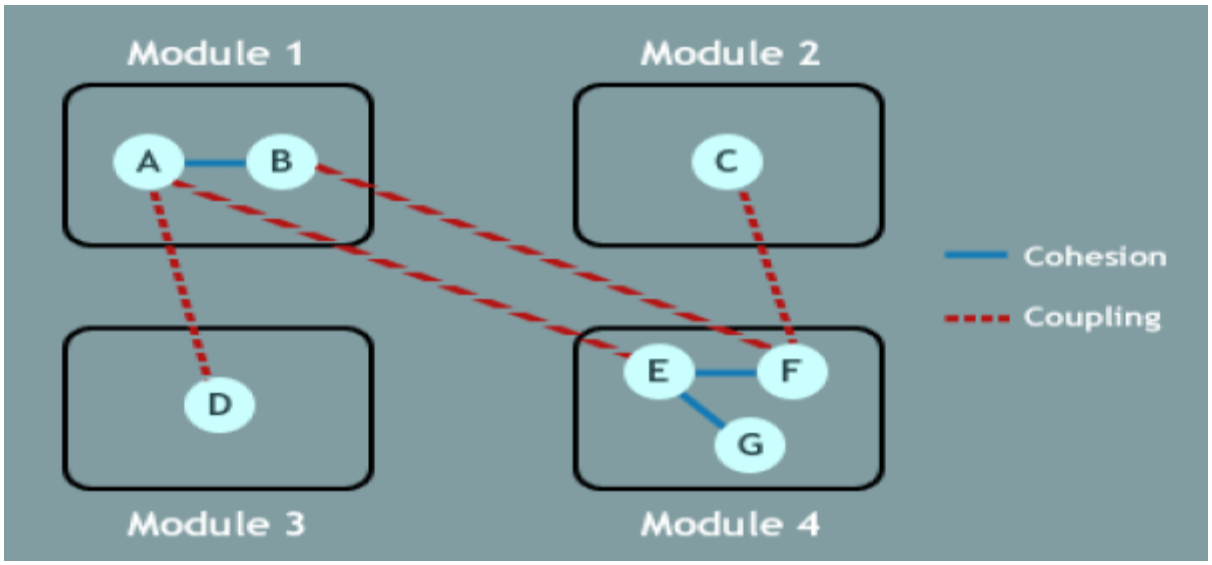
Coupling is usually contrasted with cohesion. Low coupling often correlates with high cohesion, and vice versa. Low coupling is often a sign of a well-structured computer system and a good design, and when combined with high cohesion, supports the general goals of high readability and maintainability.

Types of Coupling

1. **Content coupling**: if one directly references the contents of the other.

   When one module modifies local data values or instructions in another module. (Can happen in assembly language) if one refers to local data in another module. if one branches into a local label of another.

2. **Common coupling**: access to global data. Modules bound together by global data structures.
3. **Control coupling**: passing control flags (as parameters or globals) so that one module controls the sequence of processing steps in another module.
4. **Stamp coupling**: similar to common coupling except that global variables are shared selectively among routines that require the data. E.g., packages in Ada. More desirable than common coupling because fewer modules will have to be modified if a shared data structure is modified. Pass entire data structure but need only parts of it.
5. **Data coupling**: use of parameter lists to pass data items between routines.



Differentiate between Cohesion and Coupling

Cohesion	Coupling
Cohesion is the indication of the relationship within module.	Coupling is the indication of the relationships between modules.
Cohesion shows the module's relative functional strength.	Coupling shows the relative independence among the modules.
Cohesion is a degree (quality) to which a component / module focuses on the single thing.	Coupling is a degree to which a component / module is connected to the other modules.
While designing you should strive for high cohesion i.e. a cohesive component/ module focus on a single task (i.e., single-mindedness) with little interaction with other modules of the system.	While designing you should strive for low coupling i.e. dependency between modules should be less.
Cohesion is the kind of natural extension of data hiding for example, class having all members visible with a package having default visibility.	Making private fields, private methods and non public classes provides loose coupling.
Cohesion is Intra – Module Concept.	Coupling is Inter -Module Concept.

Structured Programming

Structured Programming Approach, as the word suggests, can be defined as a programming approach in which the program is made as a single structure. It means that the code will execute the instruction by instruction one after the other. It doesn't support the possibility of jumping from one instruction to some other with the help of any statement like GOTO, etc. Therefore, the instructions in this approach will be executed in a serial and structured manner. The languages that support Structured programming approach are:

- C
- C++
- Java
- C# etc..

Advantages

1. Easier to read and understand
2. User Friendly
3. Easier to Maintain
4. Mainly problem based instead of being machine based
5. Development is easier as it requires less effort and time
6. Easier to Debug
7. Machine-Independent, mostly.

Disadvantages

1. Since it is Machine-Independent, So it takes time to convert into machine code.
2. The converted machine code is not the same as for assembly language.
3. The program depends upon changeable factors like data-types. Therefore it needs to be updated with the need on the go.
4. Usually the development in this approach takes longer time as it is language-dependent. Whereas in the case of assembly language, the development takes lesser time as it is fixed for the machine.

Difference between Deterministic and Non-deterministic Algorithms/Techniques

In **deterministic algorithm**, for a given particular input, the computer will always produce the same output going through the same states but in case of **non-deterministic algorithm**, for the same input, the compiler may produce different output in different runs. In fact non-deterministic algorithms can't solve the problem in polynomial time and can't determine what the next step is. The non-deterministic algorithms can show different behaviors for the same input on different execution and there is a degree of randomness to it.

DETERMINISTIC ALGORITHM	NON-DETERMINISTIC ALGORITHM
For a particular input the computer will	For a particular input the computer will give

DETERMINISTIC ALGORITHM	NON-DETERMINISTIC ALGORITHM
give always same output.	different output on different execution.
Can solve the problem in polynomial time.	Can't solve the problem in polynomial time.
Can determine the next step of execution.	Cannot determine the next step of execution due to more than one path the algorithm can take.

Iterative and Recursive logic

BASIS FOR COMPARISON	RECURSION	ITERATION
Basic	The statement in a body of function calls the function itself.	Allows the set of instructions to be repeatedly executed.
Format	In recursive function, only termination condition (base case) is specified.	Iteration includes initialization, condition, execution of statement within loop and update (increments and decrements) the control variable.
Termination	A conditional statement is included in the body of the function to force the function to return without recursion call being executed.	The iteration statement is repeatedly executed until a certain condition is reached.

Condition	If the function does not converge to some condition called (base case), it leads to infinite recursion.	If the control condition in the iteration statement never become false, it leads to infinite iteration.
Infinite Repetition	Infinite recursion can crash the system.	Infinite loop uses CPU cycles repeatedly.
Applied	Recursion is always applied to functions.	Iteration is applied to iteration statements or "loops".
Stack	The stack is used to store the set of new local variables and parameters each time the function is called.	Does not uses stack.
Overhead	Recursion possesses the overhead of repeated function calls.	No overhead of repeated function call.
Speed	Slow in execution.	Fast in execution.
Size of Code	Recursion reduces the size of the code.	Iteration makes the code longer.

Modular Designing and Programming

Modular programming also called as **stepwise refinement** or **top-down design** is a programming approach that breaks down program functions into modules.

Modular programming is a technique of developing software by separating the functionality of a program into independent, interchangeable modules that are combined together to get the final working software.

Modular programming is the process of subdividing a computer program into separate sub-programs. A module is a separate software component. It can often be used in a variety of applications and functions with other components of the system.

- Some programs might have thousands or millions of lines and to manage such programs it becomes quite difficult as there might be too many of syntax errors or logical errors present in the program, so to manage such type of programs concept of **modular programming** approached.
- Each sub-module contains something necessary to execute only one aspect of the desired functionality.
- Modular programming emphasis on breaking of large programs into small problems to increase the maintainability, readability of the code and to make the program handy to make any changes in future or to correct the errors.

  Programming Languages that support the concept of modular programming are Ada, Algol, COBOL, Component Pascal, D, Erlang, F, Fortran, Haskell, HyperTalk, IBM/360 Assembler, IBM RPG, Java (packages are considered as modules), MATLAB, ML, Pascal, Perl, PL/I, Python, Ruby, SmallTalk etc.
  Modular programming is a solution to the problem of very large programs that are difficult to debug and maintain. Libraries of components built from separately compiled modules can be combined into a whole by using a programming tool called linker.

Advantages of Modular Programming

- **Ease of Use :**This approach allows simplicity, as rather than focusing on the entire thousands and millions of lines code in one go we can access it in the form of modules. This allows ease in debugging the code and prone to less error.
- **Reusability :**It allows the user to reuse the functionality with a different interface without typing the whole program again.
- **Ease of Maintenance :** It helps in less collision at the time of working on modules, helping a team to work with proper collaboration while working on a large application.
- Faster development.
- Several programmers can work on individual programs at the same time.
- Easy debugging and maintenance.
- Easy to understand as each module works independently to another module.
- Less code has to be written.
- The scoping of variables can easily be controlled.
- Modules can be re-used, eliminating the need to retype the code many times.