UNIT- I

SOFTWARE PROCESS AND AGILE DEVELOPMENT

Introduction to Software Engineering, Software Process, Perspective and Specialized Process Models –Introduction to Agility – Agile process – Extreme programming – XP Process.

1.1 INTRODUCTION TO SOFTWARE ENGINEERING The Evolving Role of Software:

- Software can be considered in a dual role. It is a **product** and, a vehicle for delivering a product.
- As a product, it delivers the computing potential in material form of computer hardware.

Example

A network of computers accessible by local hardware, whether it resides within a cellular phone or operates inside a mainframe computer.

i) As the vehicle, used to deliver the product. Software delivers the most important product of our time- **Information.**

ii) Software transforms personal data, it manages business information to enhance competiveness, it provides a gateway to worldwide information networks (e.g., Internet) and provides the means for acquiring information in all of its forms.

iii) Software acts as the basis for operating systems, networks, software tools and environments.

1.1.1 software

Software Characteristics

Software is a logical rather than a physical system element. Therefore, software has characteristics that are considerably different than those of hardware:

1. *Software is developed or engineered; it is not manufactured in the classical sense*.

- Although some similarities exist between software development and hardware manufacture, the two activities are fundamentally different.
- In both activities, high quality is achieved through good design, but the manufacturing phase for hardware can introduce quality problems that are nonexistent (or easily corrected) for software.

2. Software doesn't "wear out."

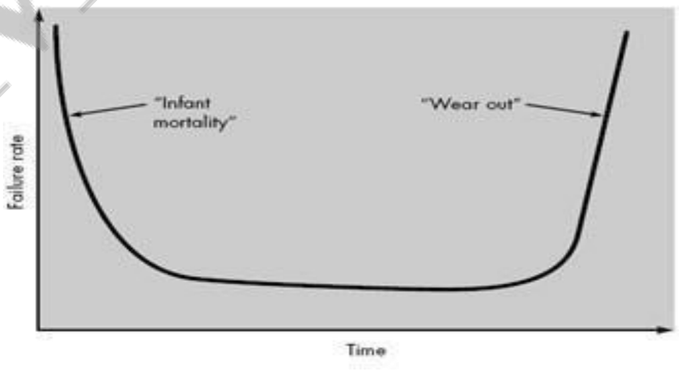


Figure 1.1 Failure curve for hardware

- Figure 1.1 depicts **failure rate as a function of time for hardware**.
- The relationship, often called the "bathtub curve", indicates that hardware exhibits relatively high failure rates early in its life (these failures are often attributable to design or manufacturing

defects); defects are corrected and the failure rate drops to a steady-state level (ideally, quite low) for some period of time.
- As time passes, however, the failure rate rises again as hardware components suffer from the cumulative effects of dust, vibration, abuse, temperature extremes, and many other environmental maladies.
- Stated simply, the hardware begins to wear out.

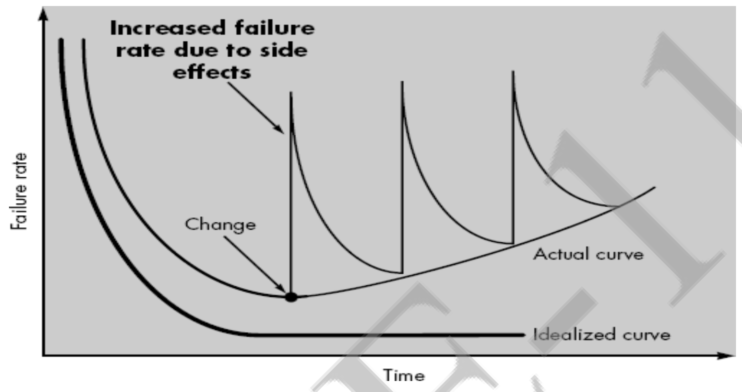


Figure 1.2 Idealized and actual failure curves for software

- The failure rate curve for software should take the form of the **"idealized curve"** shown in Fig 1.2.
- **Undiscovered defects will cause high failure rates** early in the life of a program.
- However, these are corrected (ideally, without introducing other errors) and the curve flattens as shown.
- The idealized curve is a gross oversimplification of actual failure models the implication is clear—software doesn't wear out.
- During its life, software will undergo change (maintenance).
- As changes are made, it is likely that some **new defects will be introduced, causing the failure rate curve to spike** as shown in Figure 1.2.
- Before the curve can return to the original steady-state failure rate, another change is requested, causing the curve to spike again.
- Slowly, the minimum failure rate level begins to rise—the software is deteriorating due to change.

3. Although the industry is moving toward component-based assembly, most software Continues to be custom built.

- Software component should be designed and implemented so that it can be reused in many different programs.
- For example, today's graphical user interfaces are built using reusable components that enable the creation of graphics windows, pull-down menus, and a wide variety of interaction mechanisms.
- The data structure and processing detail required to build the interface are contained with a library

of reusable components for interface construction.

1.1.2. Software Application Domains

The following categories of computer software present continuing challenges for software engineers.

a) System software:

- System software is a collection of programs written to service other programs.
- Example: compilers, editors, and file management utilities, operating system components, drivers, telecommunications processors, process largely indeterminate data.

b) Real-time software:

Elements of real-time software includes

- a data gathering component that collects and formats information from an external environment
- an analysis component that transforms information as required by the application
- a control/output component that responds to the external environment
- a monitoring component that coordinates all other components so that real-time response (typically ranging from 1 millisecond to 1 second) can be maintained.

c) Business software:

- Business information processing is the largest single software application area.
- Example: payroll, accounts receivable/payable, inventory.
- Applications in this area restructure existing data in a way that facilitates business operations or management decision making. In addition to conventional data processing application, business software applications also encompass interactive computing
- Example: point of-sale transaction processing.

d) Engineering and scientific software:

- This is the software using “number crunching" algorithms.
- Example: System simulation, computer-aided design.

e) Embedded software:

- Embedded software resides in read-only memory and is used to control products and systems for the consumer and industrial markets.
- Example: keypad control for a microwave oven
- It provides significant function and control capability
- Example: Digital functions in an automobile such as fuel control, dashboard displays, and braking systems.

f) Personal computer software:

- Word processing, spreadsheets, computer graphics, multimedia, entertainment, database management, personal and business financial applications, external network, and database access are only a few of hundreds of applications.

g) Web-based software:

- The Web pages retrieved by a browser are software that incorporates executable instructions (Ex: CGI, HTML, Perl, or Java), and data (EX: hypertext and a variety of visual and audio formats).

- Expert systems, also called knowledgebase systems, pattern recognition (image and voice), artificial neural networks, theorem proving, and game playing are representative of applications within this category.

SOFTWARE ENGINEERING

In order to build software that is ready to meet the challenges and it must recognize a few simple realities:

- It follows that a concerted effort should be made to understand the problem before a software solution is developed.
- It follows that design becomes a pivotal activity.
- It follows that software should exhibit high quality.
- It follows that software should be maintainable.

These simple realities lead to one conclusion: software in all of its forms and across all of its application domains should be engineered.

- Software engineering is the establishment and use of sound engineering principles in order to obtain economically software that is reliable and works efficiently on real machines.
- Software engineering encompasses a process, methods for managing and engineering software, and tools.

1.1.3 Software Engineering: A Layered Technology

Software engineering is a layered technology as shown in below Figure 1.3

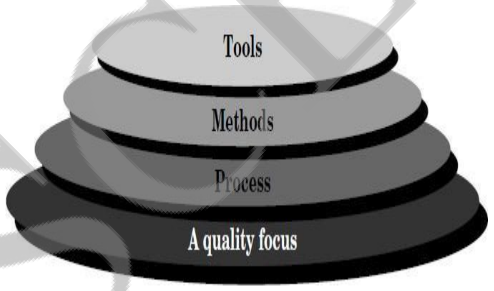


Figure1.3 Layered Technology

- The foundation for software engineering is the **process layer**.
- The software engineering process is the glue that holds the technology layers together and enables rational and timely development of computer software.
- **Process** defines a framework that must be established for effective delivery of software engineering technology.

Software process:

- The software process forms the basis for management control of software projects and establishes the context in which technical methods are applied, work products (models, documents, data, reports, forms, etc.) are produced, milestones are established, quality is ensured, and change is properly managed.

Software engineering methods:

- Software engineering methods provide the technical how-to's for building software.

- Software engineering methods rely on a set of basic principles that govern each area of the technology and include modelling activities and other descriptive techniques.

Software engineering tools:

- Software engineering tools provide automated or semi-automated support for the process and the methods.
- When tools are integrated so that information created by one tool can be used by another, a system for the support of software development, called computer-aided software engineering, is established.

1.1.4 The Software Process

- A **process** is a collection of activities, actions, and tasks that are performed when some work product is to be created.
- An **activity** strives to achieve a broad objective and is applied regardless of the application domain, size of the project, complexity of the effort, or degree of rigor with which software engineering is to be applied.
- An **action** (e.g., architectural design) encompasses a set of tasks that produce a major work product (e.g., an architectural design model).
- A **task** focuses on a small, but well-defined objective (e.g., conducting a unit test) that produces a tangible outcome.

✓ A process framework establishes the foundation for a complete software engineering process by identifying a small number of framework activities that are applicable to all software projects, regardless of their size or complexity.

✓ In addition, the process framework encompasses a set of **umbrella activities** that are applicable across the entire software process. A generic process framework for software engineering encompasses five activities:

The five generic process framework activities:

a) Communication:

- The intent is to understand stakeholders' objectives for the project and to gather requirements that help define software features and functions.

b) Planning:

- Software project plan—defines the software engineering work by describing the technical tasks to be conducted, the risks that are likely, the resources that will be required, the work products to be produced, and a work schedule.

c) Modelling:

- A software engineer does by creating models to better understand software requirements and the design that will achieve those requirements.

d) Construction:

- This activity combines code generation (either manual or automated) and the testing that is required to uncover errors in the code.

e) Deployment:

- Software engineering process framework activities are complemented by a number of umbrella activity.
- In general, umbrella activities are applied throughout a software project and help a software team manage and control progress, quality, change, and risk. Typical umbrella activities include:

i) **Software project tracking and control**—allows the software team to assess progress against the project plan and take any necessary action to maintain the schedule.

ii) **Risk management**—assesses risks that may affect the outcome of the project or the quality of the product.

iii) **Software quality assurance**—defines and conducts the activities required to ensure software quality.

iv) **Technical reviews**—access software engineering work products in an effort to uncover and remove errors before they are propagated to the next activity.

v) **Measurement**—defines and collects process, project, and product measures that assist the team in delivering software that meets stakeholders' needs; can be used in conjunction with all other framework and umbrella activities.

vi) **Software configuration management**—manages the effects of change throughout the software process.

vii) **Reusability management**—defines criteria for work product reuse(including software components) and establishes mechanisms to achieve reusable components.

viii) **Work product preparation and production**—encompasses the activities required to create work products such as models, documents, logs, forms, and lists.

1.1.5 Software Engineering Practice:

- A basic understanding of the generic concepts and principles that apply to framework activities
- The essence of problem solving, and consequently, the essence of software engineering practice:

1. Understand the problem (communication and analysis).
2. Plan a solution (modeling and software design).
3. Carry out the plan (code generation).
4. Examine the result for accuracy (testing and quality assurance).

1.1.6. Software Myths

1.Management myths.

A software manager often believes that myths will lessen the pressure

Myth: We already have a book that's full of standards and procedures for building Software, won't that provide my people with everything they need to know?

Reality: The book of standards may very well exist, but is it used? Are software Practitioners aware of its existence? Does it reflect modern software engineering practice? Is it complete? Is it streamlined to improve time to delivery while still maintaining focus on quality? In many cases, the answer to all of these questions is "no."

Myth: My people have state-of-the-art software development tools, after all, we buy them the newest computers.

Reality: It takes much more than the latest model mainframe, workstation, or PC to do high-quality software development. Computer-aided software engineering

(Sometimes called the Mongolian horde concept).

Reality: “Adding people to a late software project makes it later." As new people are added, People who were working must spend time educating the newcomers, thereby reducing the amount of time spent on productive development effort. People can be added but only in a planned and well-coordinated manner.

Myth: If I decide to outsource3 the software project to a third party, I can just relax and let that firm build it.

Reality: If an organization does not understand how to manage and control software projects internally, it will invariably struggle when it outsources software projects.

2. Customer myths. In many cases, the customer believes myths about software because Software managers and practitioners do little to correct misinformation. Myths lead to false expectations (by the customer) and ultimately, dissatisfaction with the developer.

Myth: A general statement of objectives is sufficient to begin writing programs—
We can fill in the details later.

Reality: A poor up-front definition is the major cause of failed software efforts. A formal and detailed description of the information domain, function, behavior, performance, interfaces, design constraints, and validation criteria is essential. These characteristics can be determined only after thorough communication between customer and developer.

Myth: Project requirements continually change, but change can be easily accommodated because software is flexible.

Reality: It is true that software requirements change, but the impact of change
Varies with the time at which it is introduced.

If serious attention is given to up-front definition, early requests for change can be accommodated easily. When changes are requested during software design, the cost impact grows rapidly. Change can cause upheaval that requires additional resources and major design modification, that is, additional cost. Changes in function, performance, interface, or other characteristics during implementation (code and test) have a severe impact on cost.

3. Practitioner's myths. Myths that are still believed by software practitioners have been fostered by 50 years of programming culture. During the early days of software, programming was viewed as an art form. Old ways and attitudes die hard.

Myth: Once we write the program and get it to work, our job is done.

Reality: Someone once said that "the sooner you begin 'writing code', the longer it’ll take you to get done." Industry data indicate that between 60 and 80 percent of all effort expended on software will be expended after it is delivered to the customer for the first time.

Myth: Until I get the program "running" I have no way of assessing its quality.

Reality: One of the most effective software quality assurance mechanisms can be applied from the inception of a project—the formal technical review. Software reviews are a "quality Filter" that have been found to be more effective than testing for finding certain classes of Software defects.

Myth: The only deliverable work product for a successful project is the working program.

Reality: A working program is only one part of a software configuration that includes

1.1.7 Software Engineering Paradigm:

- ✓ A software process model is an abstract representation of a process. It presents a description of a process from some particular perspective.
- ✓ Manufacturing software can be characterized by a series of steps ranging from concept exploration to final retirement; this series of steps is generally referred to as a software lifecycle.
- ✓ Steps or phases in a software lifecycle fall generally into these categories:
- Requirements
- Specification (analysis)
- Design
- Implementation
- Testing
- Integration
- Maintenance
- Retirement
- ✓ Software engineering employs a variety of methods, tools, and paradigms.
- ✓ Paradigms refer to particular approaches or philosophies for designing, building and maintaining software. Different paradigms each have their own advantages and disadvantages.
- ✓ A method (also referred to as a technique) is heavily depended on a selected paradigm and may be seen as a procedure for producing some result. Methods generally involve some formal notation and process(es).
- ✓ Tools are automated systems implementing a particular method.
- ✓ Thus, the following phases are heavily affected by selected software paradigms
- Design
- Implementation
- Integration
- Maintenance

The software development cycle involves the activities in the production of a software system. Generally the software development cycle can be divided into the following phases:

a) **Requirements analysis and specification**

- **Design**
- ✓ Preliminary design
- ✓ Detailed design
- **Implementation**
- o Component Implementation
- o Component Integration
- o System Documenting
- **Testing**
- Unit testing
- Integration testing
- System testing

- ✓ Change requirements and software upgrading

b) Verification - Validation

- Verification: "Are we building the product right"
- The software should conform to its specification
- Validation: "Are we building the right product"
- The software should do what the user really requires

1.2 SOFTWARE PROCESS A Generic Process Model

- A process was defined as a collection of work activities, actions, and tasks that are performed when some work product is to be created.
- Each of these activities, actions, and tasks reside within a framework or model that defines their relationship with the process and with one another.
- ✓ The software process is represented schematically in Figure 1.4. Referring to the figure 1.4, each framework activity is populated by a set of software engineering actions.

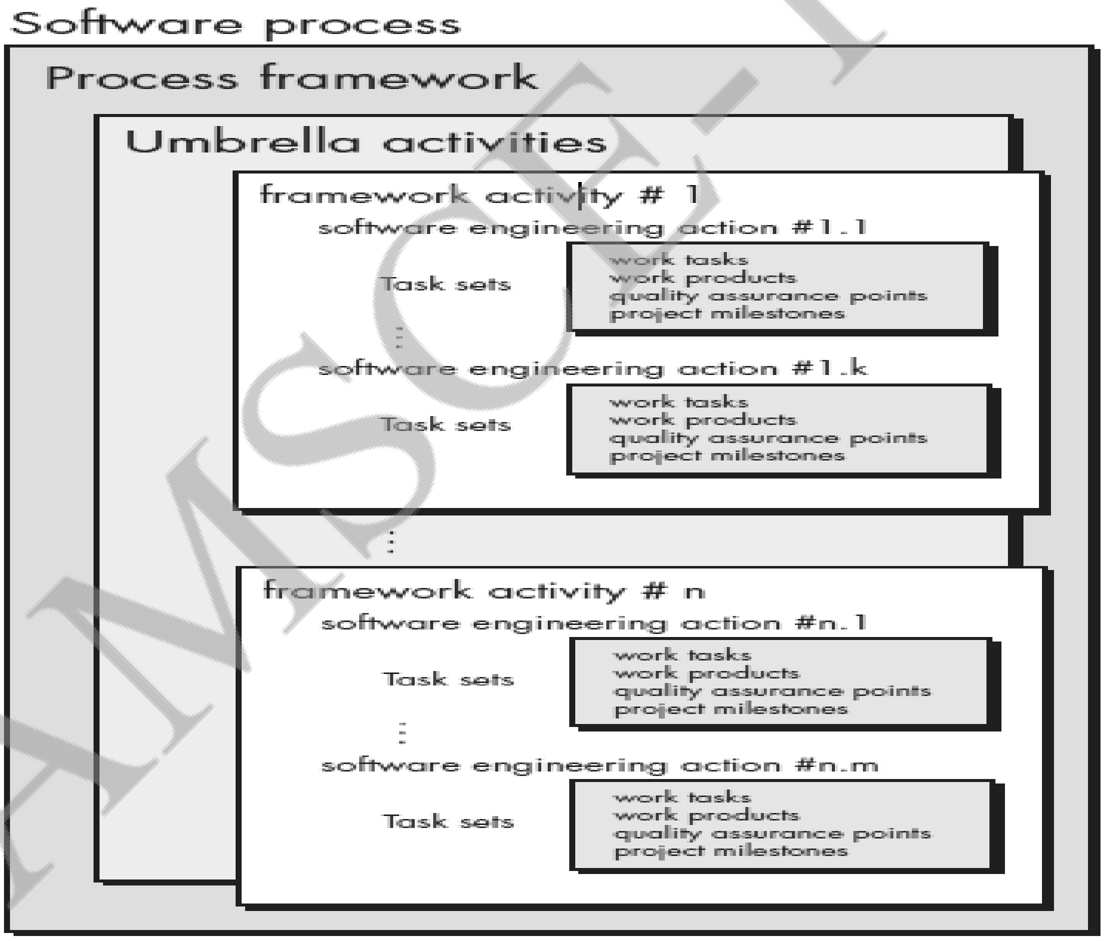
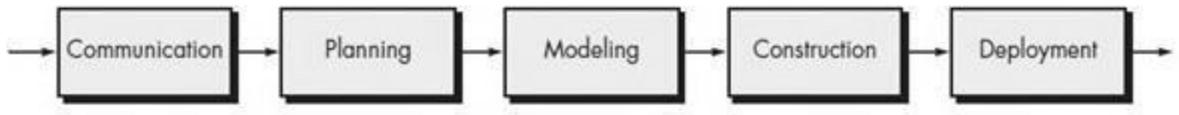


Figure 1.4 Software Process Framework

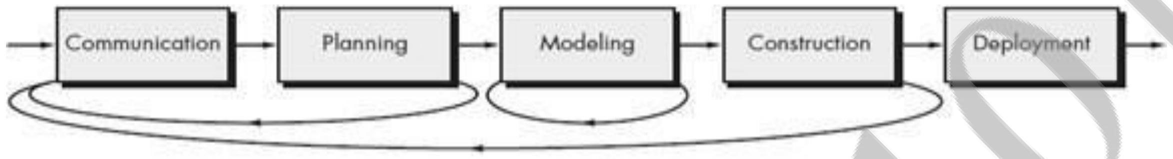
Process flow—describes how the framework activities and the actions and tasks that occur within each framework activity are organized with respect to sequence and time.

(a) A **linear process flow** executes each of the five framework activities in sequence, beginning with communication and culminating with deployment.



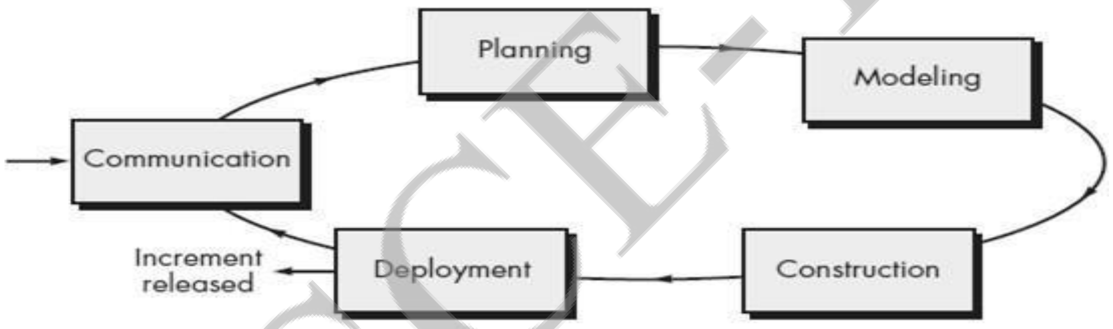
a) Linear Process Flow

(b) An **iterative process flow** repeats one or more of the activities before proceeding to the next.



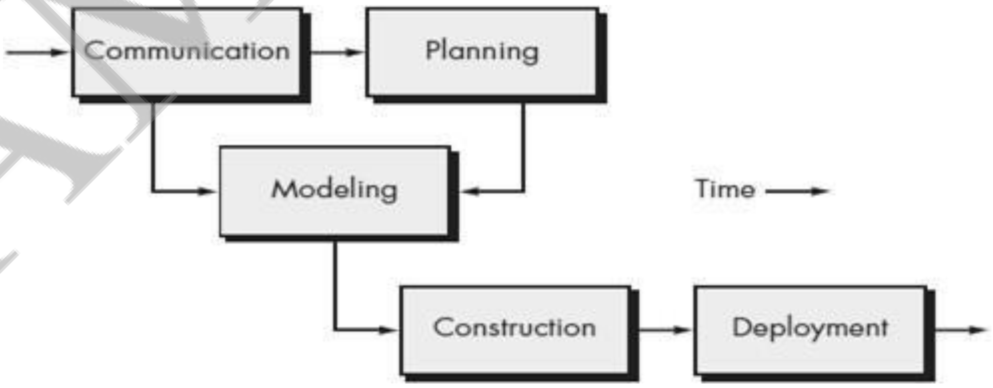
b) Iterative process flow

(c) An **evolutionary process flow** executes the activities in a "circular"manner. Each circuit through the five activities leads to a more complete version of the software.



c) Evolutionary process flow

(d) A **parallel process flow** executes one or more activities in parallel with other activities (e.g., modelling for one aspect of the software might be executed in parallel with construction of another aspect of the software).



d) Parallel process flow

Identifying a Task Set:

- A task set defines the actual work to be done to accomplish the objectives of a software engineering action.
- For example, elicitation (more commonly called “requirements gathering”) is an important software engineering action that occurs during the communication activity.
- The goal of requirements gathering is to understand what various stakeholders want from the software that is to be built.

Process Patterns:

- A process pattern describes a process-related problem that is encountered during software engineering work, identifies the environment in which the problem has been encountered, and suggests one or more proven solutions to the problem.
- Patterns can be defined at any level of abstraction. In some cases, a pattern might be used to describe a problem (and solution) associated with a complete process model (e.g., prototyping).
- In other situations, patterns can be used to describe a problem (and solution) associated with a framework activity (e.g., planning) or an action within a framework activity (e.g., project estimating).
- Ambler [Amb98] has proposed a template for describing a process pattern:

Pattern Name: The pattern is given a meaningful name describing it within the context of the software process (e.g., Technical Reviews).

Forces: The environment in which the pattern is encountered and the issues that make the problem visible and may affect its solution.

Type: The pattern type is specified. Ambler [Amb98] suggests three types:
1. Stage pattern—defines a problem associated with a framework activity for the process. Since a framework activity encompasses multiple actions and work tasks, a stage pattern incorporates multiple task patterns (see the following)that are relevant to the stage (framework activity).

✓ An example of a stage pattern might be **Establishing Communication.** This pattern would incorporate the task pattern **Requirements Gathering** and others.

2. Task pattern—defines a problem associated with a software engineering action or work task and relevant to successful software engineering practice (e.g., **Requirements Gathering** is a task pattern).

3. Phase pattern—define the sequence of framework activities that occurs within the process, even when the overall flow of activities is iterative in nature.

✓ An example of a phase pattern might be **Spiral Model or Prototyping.**

Initial context: Describes the conditions under which the pattern applies.

Prior to the initiation of the pattern:
(1) What organizational or team-related activities have already occurred?
(2) What is the entry state for the process?
(3) What software engineering information or project information already exists?
For example, the Planning pattern (a stage pattern) requires that
(1) customers and software engineers have established a collaborative communication ;

(2) successful completion of a number of task patterns [specified] for the Communication pattern has occurred; and

(3) the project scope, basic business requirements, and project constraints are known.

Problem: The specific problem to be solved by the pattern.

Solution:

- Describes how to implement the pattern successfully.
- This section describes how the initial state of the process (that exists before the pattern is implemented) is modified as a consequence of the initiation of the pattern.
- It also describes how software engineering information or project information that is available before the initiation of the pattern is transformed as a consequence of the successful execution of the pattern.

Resulting Context: Describes the conditions that will result once the pattern has been successfully implemented. Upon completion of the pattern:

(1) What organizational or team-related activities must have occurred?

(2) What is the exit state for the process?

(3) What software engineering information or project information has been developed?

Related Patterns:

i) Provide a list of all process patterns that are directly related to this one. This may be represented as a hierarchy or in some other diagrammatic form.

ii) For example, the stage pattern Communication encompasses the task patterns:

- ✓ **ProjectTeam,**
- ✓ **CollaborativeGuidelines,**
- ✓ **ScopeIsolation,**
- ✓ **RequirementsGathering,**
- ✓ **ConstraintDescription, and**
- ✓ **ScenarioCreation.**

Known Uses and Examples:

- Indicate the specific instances in which the pattern are applicable.
- For example, Communication is mandatory at the beginning of every software project, is recommended throughout the software project, and is mandatory once the deployment activity is under way.

1.3 PRESCRIPTIVE PROCESS MODELS

- ✓ Prescriptive process models were originally proposed to bring order to the chaos of software development.
- ✓ Prescriptive process models define a prescribed set of process elements anda predictable process work flow.

▪ Prescriptive Process Models

- The Waterfall Model
- Incremental Process Models
- Evolutionary Process Models

1.3.1 The Waterfall Model

The waterfall model, sometimes called the classic life cycle, suggests a systematic, sequential approach to software development that begins with customer specification of requirements and progresses through planning, modelling, construction, and deployment, culminating in ongoing support of the completed software.

- A variation in the representation of the waterfall model is called the **V-model.**
- Represented in Figure 1.5, the V-model [Buc99] depicts the relationship of quality assurance actions to the actions associated with communication, modelling, and early construction activities.

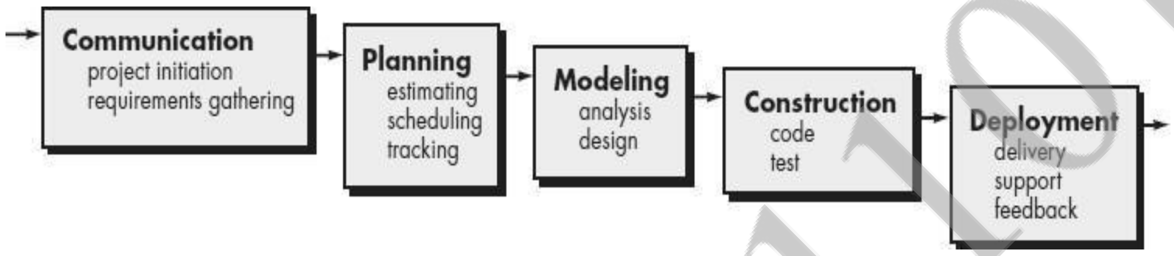
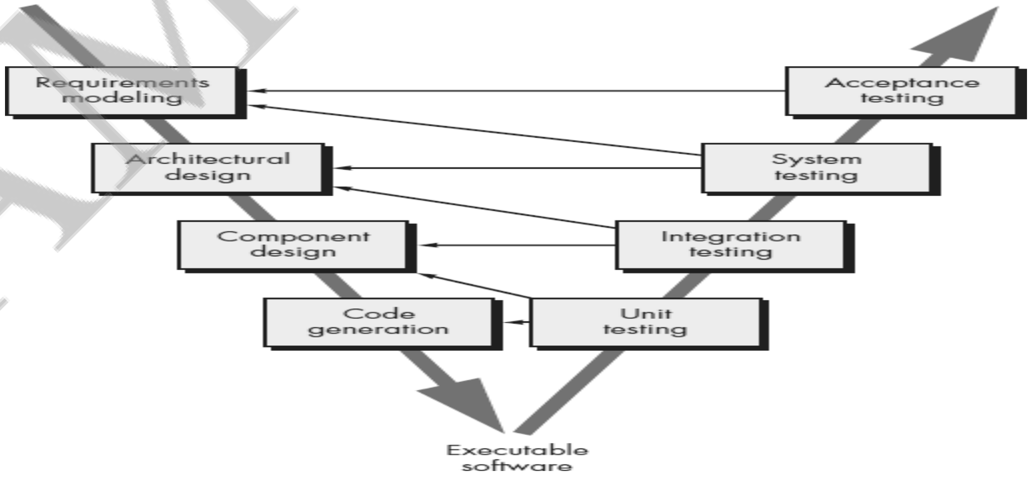


FIGURE 1.5The waterfall model

- As a software team moves down the left side of the V, basic problem requirements are refined into progressively more detailed and technical representations of the problem and its solution.
- Once code has been generated, the team moves up the right side of the V, essentially performing a series of tests (quality assurance actions) that validate each of the models created as the team moved down the left side.
- In reality, there is no fundamental difference between the classic life cycle and the V-model.
- The V-model provides a way of visualizing how verification and validation actions are applied to earlier engineering work.

Benefits of waterfall model:

- The waterfall model is simple to implement
- For implementation of small systems waterfall model is useful.



Drawbacks of waterfall model:

There are some problems that are encountered if we apply the waterfall model and those are:

- It is difficult to follow the sequential flow in software development process. If some changes are made at some phases then it may cause some confusion.
- The requirement analysis is done initially and sometimes it is not possible to state all the requirements explicitly in the beginning. This causes difficulty in the project.
- The customer can see the working model of the project only at the end. After reviewing of the working model; if the customer gets dissatisfied then it causes serious problems.
- Linear nature of waterfall model induces **blocking states**, because certain tasks may be dependent on some previous tasks. Hence it is necessary to accomplish all the dependant tasks first. It may cause long waiting time.

1.3.2 Incremental Process Models:

- The incremental model delivers a series of releases, called increments that provide progressively more functionality for the customer as each increment is delivered.
- The incremental model applies linear sequences in a staggered fashion as calendar time progresses.
- Each linear sequence produces deliverable "increments" of the software [McD93] in a manner that is similar to the increments produced by an evolutionary process flow.
- The first increment is called core product. In this release the basic requirements are implemented and then in subsequent increments new requirements are added.
- The core product is used by the customer (or undergoes detailed evaluation).
- As a result of use and/or evaluation, a plan is developed for the next increment. The plan addresses the modification of the core product to better meet the needs of the customer and the delivery of additional features and functionality. This process is repeated following the delivery of each increment, until the complete product is produced.

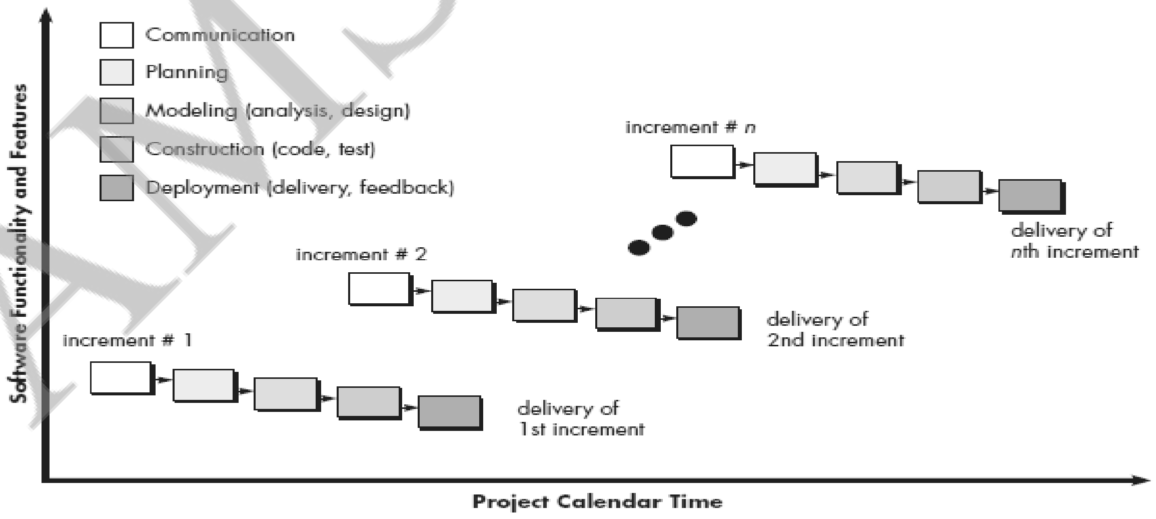

Communication
Planning
Modeling (analysis, design)
Construction (code, test)
Deployment (delivery, feedback)
increment # n
delivery of nth increment
increment # 2
delivery of 2nd increment
increment # 1
delivery of 1st increment
Software Functionality and Features
Project Calendar Time

Figure 1.7 Incremental Process Model

i) In the second increment, more sophisticated document producing and processing facilities, file management functionalities are given.

Incremental process Model advantages

1. Produces working software early during the lifecycle.
2. More flexible as scope and requirement changes can be implemented at low cost.
3. Testing and debugging is easier, as the iterations are small.
4. Low risks factors as the risks can be identified and resolved during each iteration.

Incremental process Model disadvantages

1. This model has phases that are very rigid and do not overlap.
2. Not all the requirements are gathered before starting the development; this could lead to problems related to system architecture at later iterations.

1.3.2.1 The RAD Model

- Rapid Application Development is a linear sequential software development process model that emphasizes an extremely short development cycle.
- Rapid application achieved by using a component based construction approach.
- If requirements are well understood and project scope is constrained the RAD process enables a development team to create a ―fully functional system.

RAD phases:

- Business modeling
- Data modeling
- Process modeling
- Application generation
- Testing and turnover

Business modelling:

- What information drives the business process?
- What information is generated?
- Who generates it?
- Where does the information go?
- Who processes it?

Data modelling:

- The information flow defined as part of the business modeling phase is refined into a set of data objects that are needed to support the business.
- The characteristics (called attributes) of each object are identified and the relationships between these objects are defined.

Process modelling:

- The data modelling phase are transformed to achieve the information flow necessary to implement a business function.
- Processing descriptions are created for adding, modifying, deleting, or retrieving a data object.

Application generation:

- RAD assumes the use of 4 generation techniques.

- Possible) or created reusable components (when necessary).

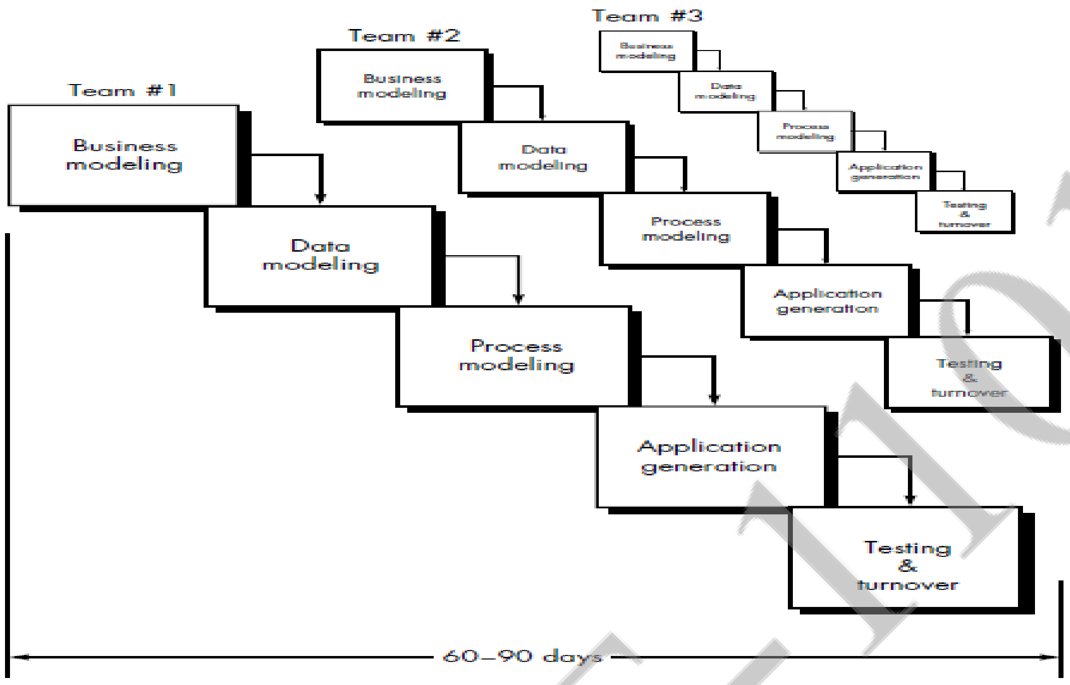


Figure1.8: RAD Process model

Testing and Turnover:

- Since the RAD process emphasizes reuse, many of the program components have already been testing.
- This reduces over all testing time.
- However, new components must be tested and all interfaces must be fully exercised.

Advantages &Disadvantages of RAD:

Advantages

- Extremely short **development** time.
- Uses component-based construction and emphasizes reuse and code generation

Disadvantages

- Large human resource requirements (to create all of the teams).
- Requires strong commitment between developers and customers for “rapid-fire” activities.
- High performance requirements can’t be met (requires tuning the components).

1.3.3 Evolutionary Process Models

- Evolutionary process models produce an increasingly more complete version of the software with each iteration.

1.3.3.1 The Prototyping Model:

- The prototyping paradigm (Figure 1.9) begins with communication.Developer and customer meet and define the overall objectives for the software, identify whatever requirements are known,

- A **quick design** focuses on a representation of those aspects of the software that will be visible to the customer/user (e.g. Input approaches and output formats). The quick design leads to the construction of a prototype.
- The prototype is evaluated by the customer/user and used to refine requirements for the software to be developed. Iteration occurs as the prototype is tuned to satisfy the needs of the customer, while at the same time enabling the developer to better understand what needs to be done.
- Ideally, the prototype serves as a mechanism for identifying software requirements.
- If a working prototype is built, the developer attempts to use existing program fragments or applies tools (e.g., report generators, window managers) that enable working programs to be generated quickly.

Advantages:

- Requirements can be set earlier and more reliably.
- Customer sees results very quickly**.**
- Customer is educated in what is possible helping to refine requirements**.**
- Requirements can be communicated more clearly and completely.
- Between developers and clients Requirements and design options can be investigated quickly and cheaply.

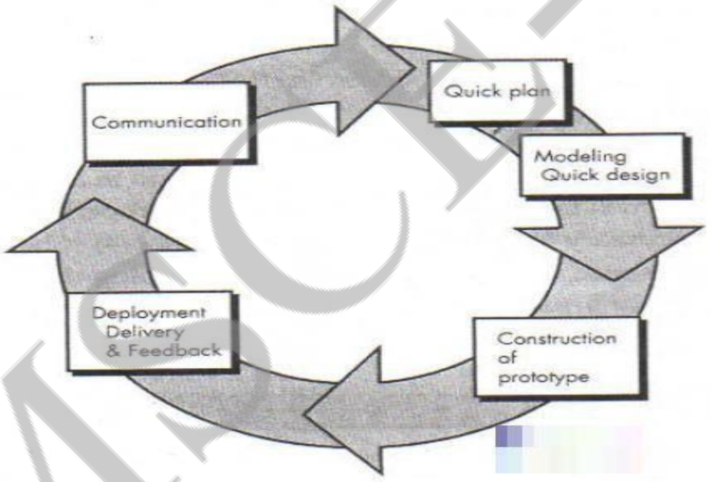


Figure 1.9: Prototyping Model

Drawbacks of prototyping:

- In the first version itself, customer often wants "few fixes" rather than rebuilding of the system whereas rebuilding of new system maintains high level of quality.
- The first version may have some compromises.
- Sometimes developer may make implementation compromises to get prototype working quickly. Later on developer may become comfortable with compromises and forget why they are inappropriate.

1.3.3.2 Spiral Model:

- The spiral model is an evolutionary software process model that couples the iterative nature of prototyping with the controlled and systematic aspects of the linear sequential model.

- The spiral development model is a risk-driven process model generator that is used to guide multi-stakeholder concurrent engineering of software intensive systems.
- It has two main distinguishing features.
- One is a cyclic approach for incrementally growing a system's degree of definition and implementation while decreasing its degree of risk.
- The other is a set of anchor point milestones for ensuring stakeholder commitment to feasible and mutually satisfactory system solutions.
- Using the spiral model, software is developed in a series of incremental releases.
- A spiral model is divided into a number of framework activities, also called **task regions.**
- Typically, there are between three and six task regions. Figure1.10depictsspiral model that contains six task regions:

  Customer communication—tasks required to establish effective communication between developer and customer.

  Planning—tasks required to define resources, timelines, and other project related information.

  Risk analysis—tasks required to assess both technical and management risks.

  Engineering—tasks required to build one or more representations of the application.

  Construction and release—tasks required to construct, test, install, and provide user support (e.g., documentation and training).

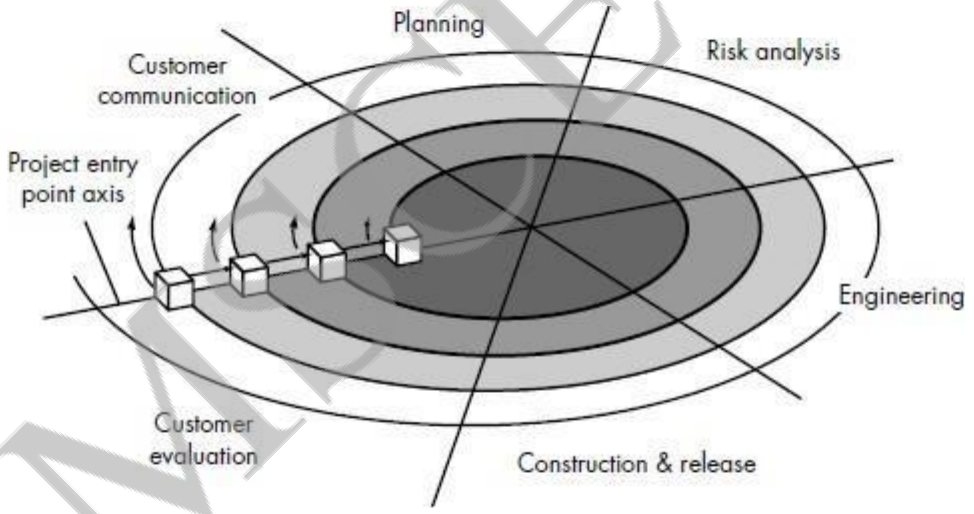


Figure 1.10: Spiral model

  Customer evaluation—tasks required to obtain customer feedback based on evaluation of the software representations created during the engineering stage and implemented during the installation stage

- As this evolutionary process begins, the software engineering team moves aroundthe spiral in a clockwise direction, beginning at the centre.
- **Anchor point milestones—**a combination of work products and conditions that are attained along the path of the spiral
- The first circuit around the spiral might result in the development of a product specification;

- Each cube placed along the axis can be used to represent the starting point for different types of projects A “concept development project” starts at the core of the spiral and will continue until concept development is complete.
- If the concept is to be developed into an actual product, the process proceeds through the next cube (new product development project entry point) and a “new development project” is initiated. The new product will evolve through a number of iterations around the spiral, following the path that bounds the core region.
- Spiral model is realistic approach to development of large-scale systems and software. Because customer and developer better understand the problem statement at each evolutionary level. Also risks can be identified or rectified at each such level.

Spiral Model Advantages:

- Requirement changes can b made at every stage.
- Risks can be identified and rectified before they get problematic.

Spiral Model disadvantages:

- It is based on **customer communication.** If the communication is not proper then the software product that gets developed will not be up to the mark.
- It demands considerable **risk assessment.** If the risk assessment is done properly then only the successful product can be obtained.

1.3.4 Concurrent Models

- The concurrent development modelis also called as **concurrent engineering**.
- It allows a software team to represent iterative and concurrent elements of any of the process models.
- In this model, the framework activities or software development tasks are represented as **states.**
- For example, the **modeling or designing** phase of software development can be in one of the states like under development, waiting for modification, under revision or under review and so on.
- All software engineering activities exist concurrently but reside in different states.
- These states make transitions. That is during **modeling,** the transition from **under development** state to **waiting for modification** state occurs.
- Customer indicates that changes in requirements must be made, the modeling activity moves from the under development state into the awaiting changes state.
- This model basically defines the **series of events** due to which the transition from one state to another state occurs. This is called **triggering.** These series of events occur for every software development activity, action or task.

Advantages:

- All types of software development can be done using concurrent development model.
- This model provides accurate picture of current state of project.
- Each activity or task can be carried out concurrently. Hence this model is an efficient process model.

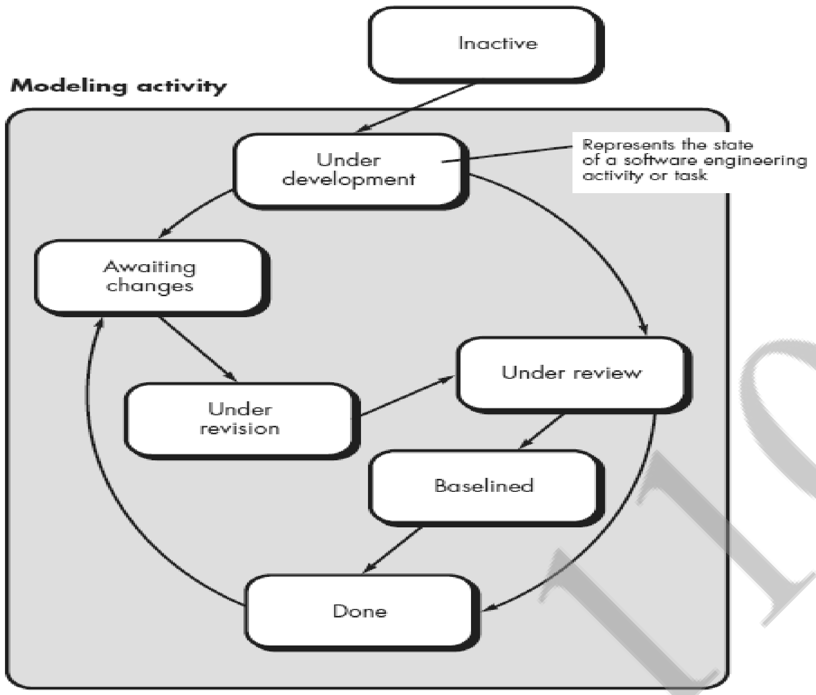


Figure 1.11 One element of the concurrent process model

1.4 SPECIALIZED PROCESS MODELS:

The specialized models are used when only collections of specialized technique or methods are expected for developing the specific software.

Various types of specialized models are-

1. Component based development
2. Formal methods model
3. Aspect oriented software development

Component based development:

- The commercial off-the-shelves components that are developed by the vendors are used during the software built.
- These components have specialized targeted functionalities and well defined interfaces. Hence it is easy to integrate these components into the existing software.
- The component-based development model incorporates many of the characteristics of the spiral model. It is evolutionary in nature.
- Modeling and construction activities begin with the identification of candidate components. These components can be designed as either conventional software modules or object-oriented classes or packages of classes

Following steps are applied for component based development

1. Available component-based products are researched and evaluated for theapplication domain in question.
2. Component integration issues are considered.
3. A software architecture is designed to accommodate the components.

- ✓ **Software reusability** is the major advantage of component based development.
- ✓ The **reusability** reduces the development cycle time and overall cost**.**

Formal methods model:

- The formal methods model encompasses a set of activities that leads to formal mathematical specification of computer software.
- Formal methods enable you to specify, develop, and verify a computer-based system by applying a rigorous, mathematical notation. A variation on this approach, called **cleanroom software engineering.**
- The **advantage** of using formal methods model is that it overcomes many problems that we encounter in traditional software process models.
- **Ambiguity, Incompletenessand Inconsistency** are those problems that can be overcome if we use formal methods model.

.

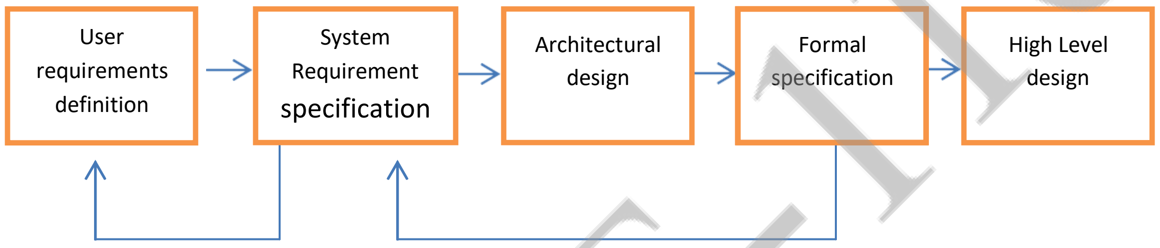

User requirements definition
System Requirement specification
Architectural design
Formal specification
High Level design

Figure 1.12 Steps involved in Formal Method Model

- The formal methods model offers detect-free software. However there are some **drawbacks of this model** which resists it from getting used widely.
- These drawbacks are
- ✓ The development of formal models is currently quite time consuming and expensive.
- ✓ Because few software developers have the necessary background to apply formal methods, extensive training is required.
- ✓ It is difficult to use the models as a communication mechanism for technically unsophisticated customers.

Aspect oriented software development:

- AOSD defines "aspects" that express customer concerns that cut across multiple system functions, features, and information.
- In traditional software development process the system is decomposed into multiple units of primary functionality.
- When concerns cut across multiple system functions, features, and information, they are often referred to as **crosscutting concerns.**
- **Aspectual requirements** define those crosscutting concerns that have an impact across the software architecture.
- **Aspect-oriented software development (AOSD),** often referred to as **aspect-oriented programming (AOP),** is a relatively new software engineering paradigm that provides a process and methodological approach for defining, specifying, designing, and constructing aspects— "mechanisms beyond subroutines and inheritance for localizing the expression of a crosscutting concern".

1.5 INTRODUCTION TO AGILITY:

1.5.1 What is Agility?

- Agility is the ability to **respond quickly to changing needs.** It encourages team structures and attitudes that make **effective communication among all stakeholders.**
- It emphasizes **rapid delivery of operational software** and de-emphasizes the importance of intermediate work products.
- It adopts the **customer as a part of the development team**.
- It helps in **organizing a team so that it is in control of the work performed.**

Yielding

- Agility results in **rapid, incremental delivery of software.**

1.5.2 Agility and the Cost of Change:

- The cost of change in software development increases nonlinearly as a project progresses (Figure 1.13, solid black curve).
- It is relatively easy to accommodate a change when software team gathered its requirements.
- The costs of doing this work are minimal, and the time required will not affect the outcome of the project.
- Cost varies quickly, and the cost and time required to ensure that the change is made without any side effects is nontrivial.
- **An agile process reduces the cost of change because software is released in increments and changes can be better controlled within an increment.**
- Agile process "flattens" the cost of change curve (Figure 1.11, shaded, solid curve), allowing a software team to accommodate changes late in a software project without dramatic cost and time impact.
- When incremental delivery is coupled with other agile practices such as continuous unit testing and pair programming, the cost of making a change is attenuated.

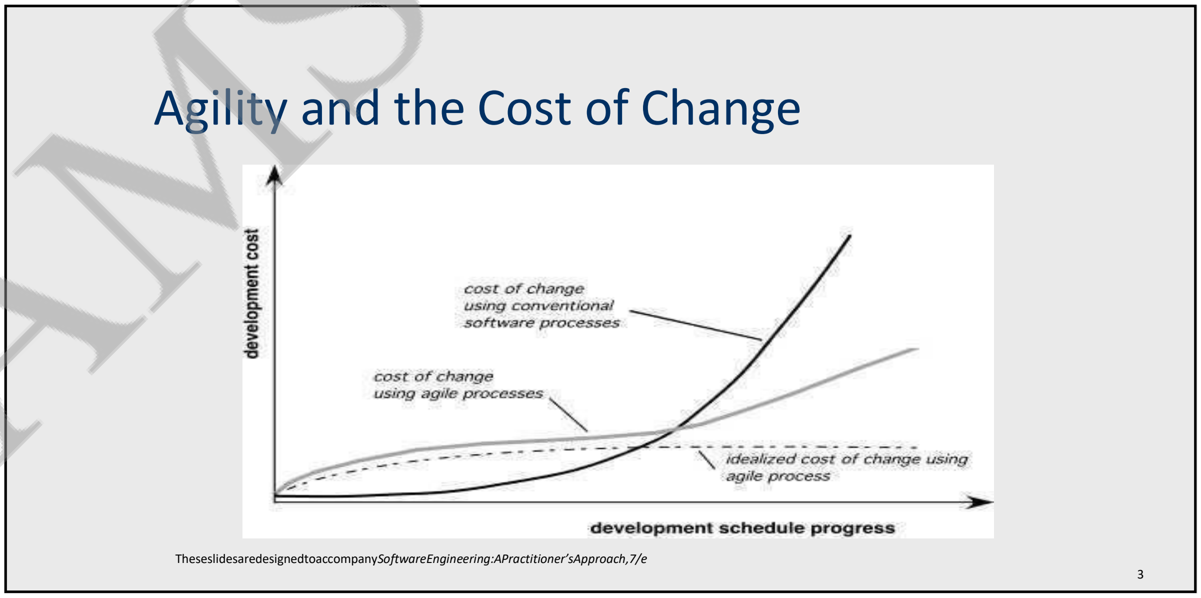


Figure 1.13Change costs as a function of time in development

1.6 AN AGILE PROCESS

An Agile Process is characterized in a manner that addresses a number of key assumptions about the majority of software project:

1. It is difficult to predict which software requirements will persist and which will change.
2. It is difficult to predict how customer priorities will change.
3. It is difficult to predict how much design is necessary before construction.
4. Analysis, design, construction, and testing are not as predictable.

1.6.1 Agility Principles:

1. To **satisfy the customer through early and continuous delivery of software**.
2. **Welcome changing requirements**, even late in development.
3. **Deliver working software frequently**, from a couple of weeks to a couple of months.
4. **'Customers and developers must work together daily** throughout the project.
5. **Build projects around motivated individuals**.
6. **Emphasis on face-to-face communication.**
7. **Working software is the primary measure of progress**.
8. Agile processes promote **sustainable development**.
9. Continuous attention to **technical excellence and good design enhances agility**.
10. **Simplicity**--the art of maximizing the amount of work not done--**is essential**.
11. **Self-organizing teams produce the best architectures/requirements/design**.
12. **The team reflects on how to become more effective** at regular intervals.

1.6.2 Human Factors:

- Agile development focuses on **the talents and skills of individuals, molding the process to specific people and teams.**
- ***The process molds to the needs of the people and team***, not the other wayaround.
- A number of key traits must exist among the people on an agile team and the team itself:
- ✓ Competence.
- ✓ Commonfocus.
- ✓ Collaboration.
- ✓ Decision-makingability.
- ✓ Fuzzy problem-solvingability.
- ✓ Mutual trust andrespect.
- ✓ Self-organization**.**

1.7 EXTREME PROGRAMMING (XP):

- The best-known and a very influential agile method, Extreme Programming (XP) takes an 'extreme' approach to iterative development.
- ✓ New versions may be built several times per day;
- ✓ Increments are delivered to customers every 2 weeks;
- ✓ All tests must be run for every build and the build is only accepted if tests run successfully.

This is how XP supports **agile principles**:

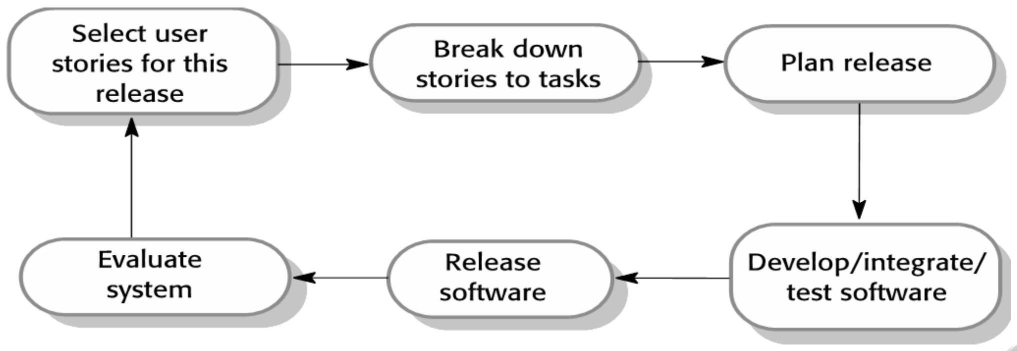

Select user stories for this release
Break down stories to tasks
Plan release
Develop/integrate/ test software
Release software
Evaluate system

Figure 1.14The extreme programming release cycle

- People not process through **pair programming, collective ownership** and a process that avoids long working hours.
- Change supported through **regular system releases**.
- Maintaining simplicity through **constant refactoring of code.**

1.7.1 XP values:

- ✓ XP is comprised of five values such as:

i. Communication
ii. Simplicity
iii. Feedback
iv. Courage
v. Respect.

- ✓ Each of these values is used as a driver for specific XP activities, actions, and task.
- ✓ In order to achieve effective **communication** between **software engineers and other stakeholders**, XP emphasizes close, yet informal(verbal) collaboration between customers and developers, the establishment of effective metaphors for communicating important concepts, continuous feedback, and the avoidance of voluminous documentation as a communication medium.
- ✓ To consider **simplicity**, XP restricts developers to design only for immediate needs, rather than future needs.
- ✓ **Feedback** is derived from three sources: the **software, the customer and other team members**.
- ✓ By designing and implementing an effective testing strategy, the software provides the agile team with feedback.
- ✓ The team develops a **unit test** for each class being developed, to exercise each operation according to its specified functionality.
- ✓ The **user stories or use cases** are implemented by the increments being used as a basis for acceptance tests. The degree to which software implements the **output, function, and behavior of the test case** is a form of feedback.
- ✓ An agile XP team must have the courage (discipline) to design for today, recognizing that future requirements may change dramatically, thereby demanding substantial rework of the design and implemented code.
- ✓ For example, there is often significant pressure to design for future requirements.

1.8 The XP Process:

- Extreme programming uses an **object-oriented approach**for software development.
- There are four framework activities involved in XP Process are shown in Figure 1.15.

1. Planning
2. Designing
3. Coding
4. Testing

1. Planning:

- Begins with the creation of a set of stories (also called user stories).
- Each story is written by the customer and is placed on an index card.
- The customer assigns a value (i.e. a priority) to the story.
- Agile team assesses each story and assigns a cost.
- Stories are grouped to for a deliverable increment.

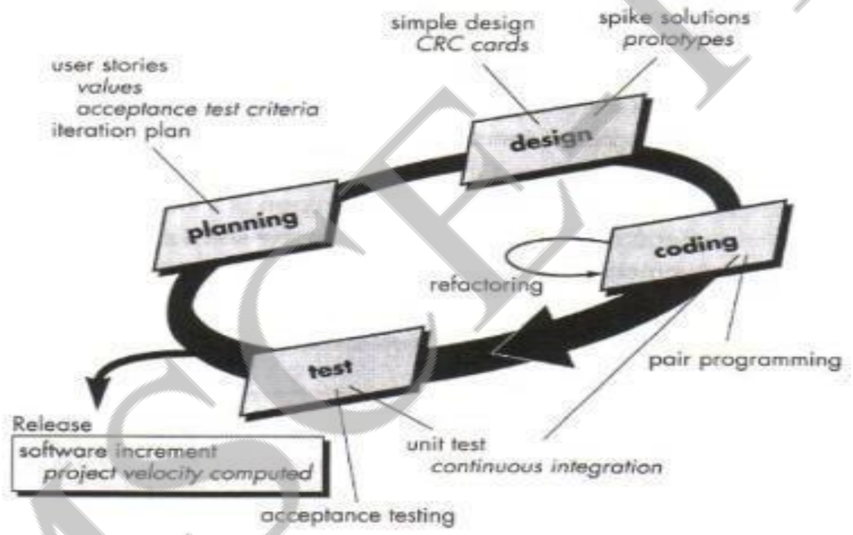


Figure 1.15The Extreme Programming Process

- A commitment is made on delivery date.
- After the first increment "project velocity" is used to help define subsequent delivery dates for other increments.

2. Design:

- Follows the keep it simple principle.
- Encourage the use of CRC (class-responsibility-collaborator) cards.
- For difficult design problems, suggests the creation of "spike solutions"—a design prototype.
- Encourages "refactoring"—an iterative refinement of the internal program design
- Design occurs both before and after coding commences.

3. Coding:

- Recommends the construction of a series of unit tests for each of the stories before coding

- Encourages “pair programming”
  - Developers work in pairs, checking each other's work and providing the support to always do a good job.
  - Mechanism for real-time problem solving and real-time quality assurance.
  - Keeps the developers focused on the problem at hand.
- Needs continuous integration with other portions (stories) of the s/w, which provides a “smoke testing” environment.

4. Testing:

- The creation of unit test before coding is the key element of the XP approach.
- The unit tests that are created should be implemented using a framework that enables them to be automated.
- This encourages regression testing strategy whenever code is modified.
- Individual unit tests are organize into a “Universal Testing Suit”, integration and validation testing of the system can occur on daily basis. This provides the XP team with a continual indication of progress and also can raise warning flags early if things are going away.
- XP acceptance test, also called customer test, are specified by the customer and focus on the overall system feature and functionality that are visible and reviewable by the customer.

1.8.1 Industrial XP:

i) IXP is an organic evolution of XP.

ii) It is imbued with XP’s minimalist, customer –centric, test-driven spirit.IXP differs most from the original XP in its greater inclusion of management, its expanded role for customers, and its upgraded technical practices.

iii) IXP incorporates six new practices that are designed to help ensure that an XP project works successfully for significant projects within a large organization.

Readiness assessment: The organization should conduct a readiness assessment prior to the initiation of an IXP project. The assessment ascertains whether

i) an appropriate development environment exists

ii) the team will be populated by the proper set of stakeholders.

iii) the organization has a distinct quality program and supports continuous improvement.

iv) the organizational culture will support the new values of an agile team, and

v) the broader project community will be populated appropriately.

Project community:

i) People on the team must be well-trained, adaptable and skilled, and have the proper temperament to contribute to a self-organizing team.

ii) When XP is to be applied for a significant project in a large organization, the concept of the “team” should morph into that of a community.

iii) A community may have a technologist and customers who are central to the success of a project as well as many other stakeholders may play important roles on the project.

Project chartering:

i) The IXP team assess the project itself to determine whether the project exists and whether the project will further the overall goals and objectives of the organization.

ii) It also determines how it complements, extends, or replaces existing systems or process.

i) Test driven management establishes a series of measurable “destinations” and then defines mechanisms for determining whether or not these destinations have been reached. **Retrospectives:**

i) An IXP team conducts a technical review after software increment is delivered called retrospective.

ii) This review examines “issues,events,and lessons-learned” across a software increment and/or the entire software release.

iii) The intent is to improve the IXP process.

Continuous learning:

i) Learning is a vital product of continuous process improvement, members of the XP team are encouraged to learn new methods and techniques that can lead to a higher quality product.

ii) In addition to these six new practices, IXP modifies a number of existing XP practices.

✓ **Story-driven development (SDD)** insists that stories for acceptance tests be written before a single line of code is developed.

✓ **Domain-driven design (DDD):**

i) It is an improvement on the “system metaphor” concept used in XP.

ii) It suggests the creation of a domain model that accurately represents how domain experts think about their subject.

- **Pairing** extends the XP pair-programming concept to include managers and their stakeholders. The intent is to improve knowledge sharing among XP team members who may not be directly involved in technical development.
- **Iterative usability** discourages front-loaded interface design in favor of usability design that evolves as software increments are delivered and users’ interaction with the software.

1.8.2 The XP Debate:

- Extreme Programming has done heated debate for both new process models and methods.
- This examines the efficacy of XP, but Stephens and Rosenberg argue that many XP practices are worthwhile, but others have been overhyped, and a few are problematic.
- The authors suggest that the codependent natures of XP practices are both its strength and its weakness.
- Because many organizations adopt only a subset of XP practices, they weaken the efficacy of the entire process.
- Proponents counter that XP is continuously evolving and that many of the issues raised by critics have been addressed as XP practice matures.
- Among the issues that continue to trouble some critics of XP are:
- ***Requirements volatility*.**

✓ Because the customer is an active member of the XP team, changes to requirements are requested informally.

✓ As a consequence, the scope of the project can change and earlier work may have to be modified to accommodate current needs**.**

- ***Conflicting customer needs***.

✓ Many projects have multiple customers, each withhisown set of needs.Requirements are expressed informally.

- ✓ Critics argue that amore formal model or specification is often needed to ensure that omissions,inconsistencies, and errors are uncovered before the system is built.
- *Lack of formal design.*
- ✓ XP deemphasizes the need for architectural design andin many instances, suggests that design of all kinds should be relativelyinformal.
- ✓ Critics argue that when complex systems are built, design must beemphasized to ensure that the overall structure of the software will exhibit quality and maintainability.
- ✓ XP proponents suggest that the incremental nature of the XP process limits complexity (simplicity is a core value) and therefore reduces the need for extensive design.

1.8.3 Other Agile Process Models

1. Adaptive Software Development (ASD)
2. Dynamic Systems Development Method (DSDM)
3. Scrum
4. Crystal
5. Feature Driven Development (FDD)
6. Agile Modeling (AM)
7. Lean Software Development (LSD)
8. Agile Unified Process (AUP)

1.8.3.1 Adaptive Software Development (ASD)

- Adaptive Software Development (ASD) is a technique for building complex software and ASD incorporates three phases Speculation, Collaboration, and Learning systems
- ASD focus on human collaboration and team self-organization.

Speculation:

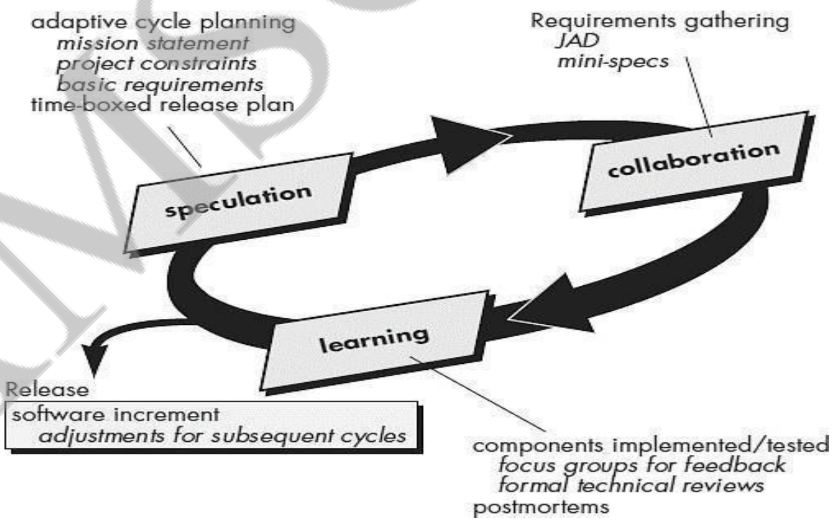


Figure 1.16Adaptive Software Development

- “Speculate” refers to the planning paradox—outcomes are unpredictable, therefore, endless suppositions on a product’s look and feel are not likely to lead to any business value.
- The big idea behind speculate is when we plan a product to its smallest detail as in a requirements

- In the ASD mindset, planning is to speculation as intention is to need.

Collaboration:

- Collaboration represents a balance between managing the doing and creating and maintaining the collaborative environment.
- Speculation says we can't predict outcomes. If we can't predict outcomes, we can't plan. If we can't plan, traditional project management theory suffers.
- Collaboration weights speculation in that a project manager plans the work between the predictable parts of the environment and adapts to the uncertainties of various factors—stakeholders, requirements, software vendors, technology, etc.

Learning:

- "Learning" cycles challenge all stakeholders and project team members.
- Based on short iterations of design, build and testing, knowledge accumulates from the small mistakes we make due to false assumptions, poorly stated or ambiguous requirements or misunderstanding the stakeholders' needs.
- Correcting those mistakes through shared learning cycles leads to greater positive experience and eventual mastery of the problem domain**.**

1.8.3.2 Dynamic Systems Development Methods (DSDM)

- The Dynamic Systems Development Method is an agile software development approach that "provides a framework for building and maintaining systems which meet tight time constraints through the use of incremental prototyping in a controlled project environment".
- DSDM is an iterative software process in which each iteration follows the 80 percent rule.
- That is, only enough work is required for each increment to facilitate movement to the next increment.
- The remaining detail can be completed later when more business requirements are known or changes have been requested and accommodated.
- DSDM life cycle that defines three different iterative cycles, preceded by two additional life cycle activities:

Feasibility study—establishes the basic business requirements and constraints associated with the application to be built and then assesses whether the application is a viable candidate for the DSDM process.

Business study—establishes the functional and information requirements that will allow the application to provide business value; also, defines the basic application architecture and identifies the maintainability requirements for the application.

Functional model iteration—produces a set of incremental prototypes that demonstrate functionality for the customer.

Design and build iteration—revisits prototypes built during functional model iteration to ensure that each has been engineered in a manner that will enable it to
provide operational business value for end users.

Implementation—places the latest software increment into the operational environment.

- DSDM can be combined with XP to provide a combination approach that defines a solid process model (the DSDM life cycle) with the nuts and bolts practices (XP) that are required to build combined process model.

1.8.3.3 Scrum

- Scrum principles are consistent with the agile manifesto and are used to guide development activities within a process that incorporates the five framework activities: requirements, analysis, design, evolution, and delivery.
- Within each framework activity, work tasks occur within a process pattern called a sprint
- The work conducted within a sprint (the number of sprints required for each framework activity will vary depending on product complexity and size) is adapted to the problem at hand and is defined and often modified in real time by the Scrum team.
- Scrum emphasizes the use of a set of software process patterns that have proven effective for projects with tight timelines, changing requirements, and business criticality.
- Each of these process patterns defines a set of development actions: Backlog—a prioritized list of project requirements or features that provide business value for the customer.
- Items can be added to the backlog at any time (this is how changes are introduced).
- The product manager assesses the backlog and updates priorities as required.

1.8.3.4 Crystal

- The Crystal methodology is one of the most lightweight, adaptable approaches to software development. Crystal is actually comprised of a family of agile methodologies such as Crystal Clear, Crystal Yellow, Crystal Orange and others, whose unique characteristics are driven by several factors such as team size, system criticality, and project priorities.
- This Crystal family addresses the realization that each project may require a slightly tailored set of policies, practices, and processes in order to meet the project‘s unique characteristics.
- Several of the key tenets of Crystal include teamwork, communication, and simplicity, as well as reflection to frequently adjust and improve the process.
- Like other agile process methodologies, Crystal promotes early, frequent delivery of working software, high user involvement, adaptability, and the removal of bureaucracy or distractions.

1.8.3.5 Feature Driven Development(FDD)

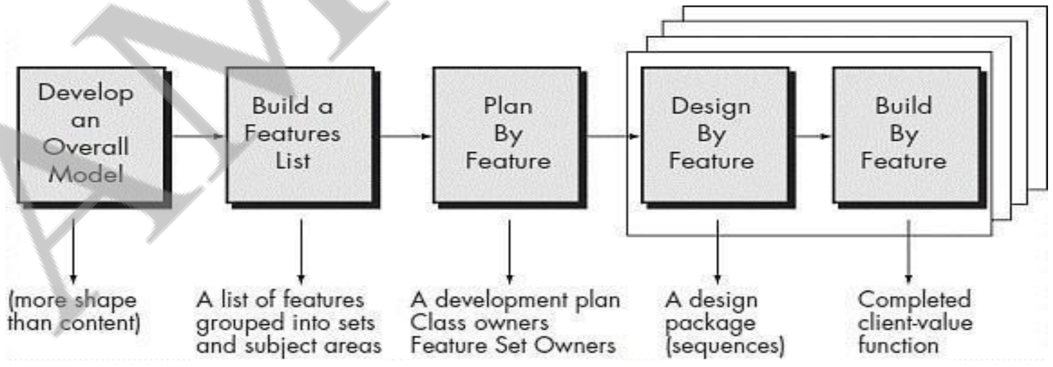

Develop an Overall Model
Build a Features List
Plan By Feature
Design By Feature
Build By Feature
(more shape than content)
A list of features grouped into sets and subject areas
A development plan Class owners Feature Set Owners
A design package (sequences)
Completed client-value function

Figure1.17: Feature Driven Development Model

- FDD is a model-driven, short-iterationprocess.

 The features are small, “useful in the eyes of the client” results.
 FDD designs the rest of the development process around feature delivery using the following eight practices:

- ✓ Domain Object Modelling
- ✓ Developing by Feature
- ✓ Component/Class Ownership
- ✓ Feature Teams
- ✓ Inspections
- ✓ Configuration Management
- ✓ Regular Builds
- ✓ Visibility of progress and results

 FDD recommends specific programmer practices such as “Regular Builds” and “Component/Class Ownership”.
 Unlike other agile methods, FDD describes specific, very short phases of work, which are to be accomplished separately per feature.
 These include Domain Walkthrough, Design, Design Inspection, Code, Code Inspection, and Promote to Build.

1.8.3.6 Agile Modelling (AM)

 Agile Modeling (AM) is a practice-based methodology for effective modeling and documentation of software-based systems.
 Simply put, Agile Modeling (AM) is a collection of values, principles, and practices for modeling software that can be applied on a software development project in an effective and light-weight manner.
 Although AM suggests a wide array of “core” and “supplementary” modeling principles, those that make AM unique are:

Use multiple models.

- ✓ There are many different models and notations that can be used to describe software.
- ✓ AM suggests that to provide needed insight, each model should present a different aspect of the system and only those models that provide value to their intended audience should be used.

Travel light.

- ✓ As software engineering work proceeds, keep only those models that will provide long-term value and jettison the rest.

Content is more important than representation.

- ✓ Modeling should impart information to its intended audience.
- ✓ A syntactically perfect model that imparts little useful content is not as valuable as a model with flawed notation that nevertheless provides valuable content for its audience.

Know the models and the tools you use to create them.

- ✓ Understand the strengths and weaknesses of each model and the tools that are used to create it.

Adapt locally.

- ✓ The modelling approach should be adapted to the needs of the agile team.

1.8.3.7 Lean Software Development (LSD):

- Lean Software Development (LSD) has adapted the principles of lean manufacturing to the world of software engineering.
- The lean principles that inspire the LSD process can be summarized as ***eliminate waste, build quality in, create knowledge, defer commitment, deliver fast, respect people, and optimize the whole.***
- For example, eliminate waste within the context of an agile software project can be interpreted to mean

✓ Adding no extraneous features or functions
✓ Assessing the cost and schedule impact of any newly requested requirement
✓ Removing any superfluous process steps
✓ Establishing mechanisms to improve the way team members find information
✓ Ensuring the testing finds as many errors as possible,
✓ Reducing the time required to request and get a decision that affects the software or the process that is applied to create it,
✓ Streamlining the manner in which information is transmitted to all stakeholders involved in the process.

1.8.3.8 Agile Unified Process (AUP):

- AUP adopts a "serial in the large" an "iterative in the small" philosophy for building computer-based systems.
- Byadopting the classic UP phased activities –**inception, elaboration, construction, andtransition.**
- It enables a team to visualize the overall process flow for a software project.
- Each AUP iteration addresses the following activities:

(i) **Modelling**. It represents the business and problem domains.
(ii) **Implementation**. Models translated into source code.
(iii) **Testing**. Executes a series of tests to uncover errors and ensures that the source code meets its requirements.
(iv) **Deployment**. Focus on the delivery of software increment and the acquisition of feedback
(v) from end users.
(vi) **Configuration and project management**. Configuration management addresses change management, risk management, andthecontrolofanypersistent workproductsthat areproducedbytheteam.
(vii) **Environment management**. It coordinates a process infrastructure that includes standards, tools, and other support technology available to the team.

Agile Methods Applicability:

- Product development where a software company is developing a small or medium-sized product.
- Custom system development within an organization, where there is a clear commitment from the customer to become involved in the development process and where there are not a lot of external rules and regulations that affect the software.
- Because oftheir focus on small, tightly-integrated teams, there are problems in scaling agile methods to large systems.

Problems with agile methods:

- It can be difficult to keep the interest of customers who are involved in the process.

- Maintaining simplicity requires extra work.
- Contracts may be a problem as with other approaches to iterative development.

Important questions:

Compare and Contrast the different life cycle models.

Waterfall model	**Spiral model**	**Prototyping model**	**Incremental model**
Requirements must be clearly understood and defined at the beginning only.	The Requirements analysis and gathering can be done in iterations because requirements get changed quite often.	Requirement analysis can be made in the later stages of development cycle, because requirements get changed quite often.	Requirement analysis can be made in the later stages of development cycle
The development team having the adequate experience of working on the similar project is chosen to work on this type of process model.	The development team having the adequate experience of working on the similar project is allowed in this process model.	The development team having the adequate experience of working on the similar project is allowed in this process model.	The development team having the adequate experience of working on the similar project is chosen to work on this type of process model.
There is no user involvement in all the phases of development process.	There is no user involvement in all the phases of development process.	There is user involvement in all the phases of development process.	There is user involvement in all the phases of development process.
When the requirements are reasonably well defined and the development effort suggests a purely linear effort then the **waterfall model is chosen.**	Due to iterative nature of this model the risk identification and rectification is done before they get problematic. Hence for handling real time problems the **spiral model is chosen.**	When developer is unsure about the efficiency of an algorithm or the adaptability of an operating system then **the Prototyping model is chosen.**	When the requirements are reasonably wel l defined and the development effort suggests a purely linear effort and when limited set of software functionality is needed quickly then the **incremental model is chosen.**

Compare and Contrast waterfall model with spiral model.

S.No	Waterfall model	Spiral model
	It requires well understanding of requirements and familiar technology.	It is developed in iterations. Hence the requirement can be identified at new iterations.
	Difficult to accommodate changes after the process has started.	The required changes can be made at every stage of new version.
	Can accommodate iteration but indirectly.	It is iterative model.
	Risks can be identified at the end which may cause failure to the product.	Risks can be identified and reduced before they get problematic.
	The customer can see the working model of the project only at the end. After reviewing of the working model, if the customer gets dissatisfied then it causes serious problems.	The customer can see the working product at certain stages of iterations.
	Customers prefer this model.	Developers prefer this model.
	This model is good for small systems.	This model is good for large systems
	It has sequential nature.	It has evolutionary nature.

Unit -II - SOFTWARE REQUIREMENT SPECIFICATION

Requirement analysis and specification – Requirements gathering and analysis – Software Requirement Specification – Formal system specification – Finite State Machines – Petri nets – Object modelling using UML – Use case Model–Class diagrams – Interaction diagrams–Activity diagrams–State chart diagrams –Functional modelling–Data Flow Diagram.

2.1 Requirement analysis and specification

The **Requirement Analysis** and **Specification** phase starts after the feasibility study stage is complete and the project is financially viable and technically feasible. This phase ends when the requirements specification document has been developed and reviewed. It is usually called the **Software Requirements Specification (SRS)** Document.

These activities are usually carried out by a few experienced members of the development team and it normally requires them to spend some time at the customer site. The engineers who gather and analyse customer requirements and then write the requirements specification document are known as **system analysts** in the software industry. System analysts collect data about the product to be developed and analyse the collected data to conceptualize what exactly needs to be done. After understanding the precise user requirements, the analysts analyse the requirements to weed out inconsistencies, anomalies and incompleteness.

We can conceptually divide the requirements gathering and analysis activity into two separate tasks:

- Requirements gathering
- Requirements analysis

2.1.1 Requirements Gathering:

It is also known as **Requirements Elicitation**. The primary objective of the requirements-gathering task is to collect the requirements from the stakeholders. A **stakeholder** is a source of the requirements and is usually a person or a group of persons who either directly or indirectly are concerned with the software.

1. Studying existing documentation: The analyst usually studies all the available documents regarding the system to be developed before visiting the customer site. Customers usually provide a statement of purpose (SoP) document to the developers. Typically, these documents might discuss issues such as the context in which the software is required.

2. Interview: Typically, there are many different categories of users of the software. Each category of users typically requires a different set of features from the software. Therefore, it is important for the analyst to first identify the different categories of users and then determine the requirements of each.

3. Task analysis: The users usually have a black-box view of software and consider the software as something that provides a set of services. A service supported by the software is also called a **task**. We can therefore say that the software performs various tasks for the users. In this context, the analyst tries to identify and understand the different tasks to be performed by the software. For each identified task, the analyst tries to formulate the different steps necessary to realise the required functionality in consultation with the users.

4. Scenario analysis: A task can have many scenarios of operation. The different scenarios of a task may take place when the task is invoked under different situations. For different types of scenarios of a task, the behaviour of the software can be different.

5. Form analysis: Form analysis is an important and effective requirement-gathering activity that is undertaken by the analyst when the project involves automating an existing manual system. During the operation of a manual system, normally several forms are required to be filled up by the stakeholders, and in turn, they receive several notifications. In form analysis, the existing forms and the formats of the notifications produced are analysed to determine the data input to the system and the data that are output from the system.

2.1.2 Requirement Analysis and Specification:

After requirements gathering is complete, the analyst analyses the gathered requirements to form a clear understanding of the exact customer requirements and to weed out any problems in the gathered requirements. It is natural to expect the data collected from various stakeholders to contain several contradictions, ambiguities, and incompleteness.

The main purpose of the requirements analysis activity is to analyse the gathered requirements to remove all ambiguities, incompleteness, and inconsistencies from the gathered customer requirements and to obtain a clear understanding of the software to be developed.

During requirements analysis, the analyst needs to identify and resolve three main types of problems in the requirements:

- Anomaly
- Inconsistency
- Incompleteness

Anomaly: It is an anomaly is an ambiguity in a requirement. When a requirement is anomalous, several interpretations of that requirement are possible. Any anomaly in any of the requirements can lead to the development of an incorrect system, since an anomalous requirement can be interpreted in several ways during development.

Inconsistency: Two requirements are said to be inconsistent if one of the requirements contradicts the other.

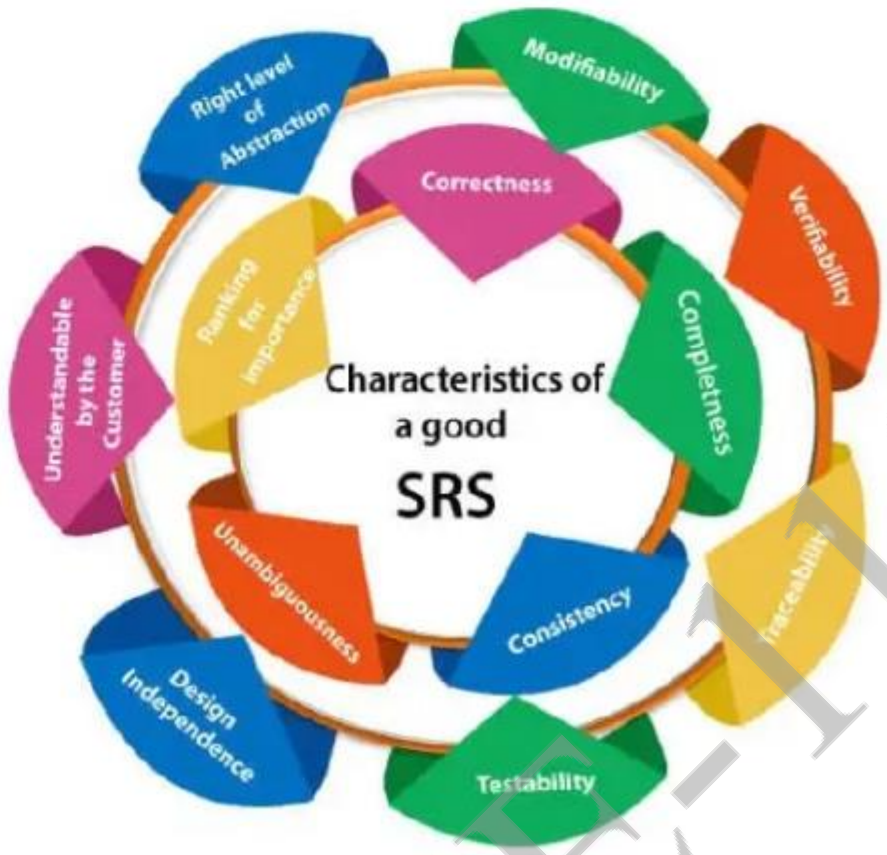
Incompleteness: An incomplete set of requirements is one in which some requirements have been overlooked. The lack of these features would be felt by the customer much later, possibly while using the software. Often, incompleteness is caused by the inability of the customer to visualise the system that is to be developed and to anticipate all the features that would be required.

2.2 Software Requirement Specification

The production of the requirements stage of the software development process is Software Requirements Specifications (SRS) (also called a requirements document). This report lays a foundation for software engineering activities and is constructing when entire requirements are elicited and analyzed. SRS is a formal report, which acts as a representation of software that enables the customers to review whether it (SRS) is according to their requirements. Also, it comprises user requirements for a system as well as detailed specifications of the system requirements.

The SRS is a specification for a specific software product, program, or set of applications that perform particular functions in a specific environment. It serves several goals depending on who is writing it. First, the SRS could be written by the client of a system. Second, the SRS could be written by a developer of the system. The two methods create entirely various situations and establish different purposes for the document altogether. The first case, SRS, is used to define the needs and expectation of the users. The second case, SRS, is written for various purposes and serves as a contract document between customer and developer.

2.2.1 Characteristics of good SRS



Following are the features of a good SRS document:

1. Correctness: User review is used to provide the accuracy of requirements stated in the SRS. SRS is said to be perfect if it covers all the needs that are truly expected from the system.

2. Completeness: The SRS is complete if, and only if, it includes the following elements:

(1). All essential requirements, whether relating to functionality, performance, design, constraints, attributes, or external interfaces.

(2). Definition of their responses of the software to all realizable classes of input data in all available categories of situations.

(3). Full labels and references to all figures, tables, and diagrams in the SRS and definitions of all terms and units of measure.

3. Consistency: The SRS is consistent if, and only if, no subset of individual requirements described in its conflict. There are three types of possible conflict in the SRS:

(1). The specified characteristics of real-world objects may conflicts. For example,

(a) The format of an output report may be described in one requirement as tabular but in another as textual.

(b) One condition may state that all lights shall be green while another states that all lights shall be blue.

(2). There may be a reasonable or temporal conflict between the two specified actions. For example,

(a) One requirement may determine that the program will add two inputs, and another may determine that the program will multiply them.

(b) One condition may state that "A" must always follow "B," while other requires that "A and B" co-occurs.

(3). Two or more requirements may define the same real-world object but use different terms for that object. For example, a program's request for user input may be called a "prompt" in one requirement's and a "cue" in another. The use of standard terminology and descriptions promotes consistency.

4. Unambiguousness: SRS is unambiguous when every fixed requirement has only one interpretation. This suggests that each element is uniquely interpreted. In case there is a method used with multiple definitions, the requirements report should determine the implications in the SRS so that it is clear and simple to understand.

5. Ranking for importance and stability: The SRS is ranked for importance and stability if each requirement in it has an identifier to indicate either the significance or stability of that particular requirement.

Typically, all requirements are not equally important. Some prerequisites may be essential, especially for life-critical applications, while others may be desirable. Each element should be identified to make these differences clear and explicit. Another way to rank requirements is to distinguish classes of items as essential, conditional, and optional.

6. Modifiability: SRS should be made as modifiable as likely and should be capable of quickly obtain changes to the system to some extent. Modifications should be perfectly indexed and cross-referenced.

7. Verifiability: SRS is correct when the specified requirements can be verified with a cost-effective system to check whether the final software meets those requirements. The requirements are verified with the help of reviews.

8. Traceability: The SRS is traceable if the origin of each of the requirements is clear and if it facilitates the referencing of each condition in future development or enhancement documentation.

2.2.2 There are two types of Traceability:

1. Backward Traceability: This depends upon each requirement explicitly referencing its source in earlier documents.

2. Forward Traceability: This depends upon each element in the SRS having a unique name or reference number.

The forward traceability of the SRS is especially crucial when the software product enters the operation and maintenance phase. As code and design document is modified, it is necessary to be able to ascertain the complete set of requirements that may be concerned by those modifications.

9. Design Independence: There should be an option to select from multiple design alternatives for the final system. More specifically, the SRS should not contain any implementation details.

10. Testability: An SRS should be written in such a method that it is simple to generate test cases and test plans from the report.

11. Understandable by the customer: An end user may be an expert in his/her explicit domain but might not be trained in computer science. Hence, the purpose of formal notations and symbols should be avoided too as much extent as possible. The language should be kept simple and clear.

12. The right level of abstraction: If the SRS is written for the requirements stage, the details should be explained explicitly. Whereas,for a feasibility study, fewer analysis can be used. Hence, the level of abstraction modifies according to the objective of the SRS.

2.2.3 Properties of a good SRS document

The essential properties of a good SRS document are the following:

Concise: The SRS report should be concise and at the same time, unambiguous, consistent, and complete. Verbose and irrelevant descriptions decrease readability and also increase error possibilities.

Structured: It should be well-structured. A well-structured document is simple to understand and modify. In practice, the SRS document undergoes several revisions to cope up with the user requirements. Often, user requirements evolve over a period of time. Therefore, to make the modifications to the SRS document easy, it is vital to make the report well-structured.

Black-box view: It should only define what the system should do and refrain from stating how to do these. This means that the SRS document should define the external behavior of the system and not discuss the implementation issues. The SRS report should view the system to be developed as a black box and should define the externally visible behavior of the system. For this reason, the SRS report is also known as the black-box specification of a system.

Conceptual integrity: It should show conceptual integrity so that the reader can merely understand it. Response to undesired events: It should characterize acceptable responses to unwanted events. These are called system response to exceptional conditions.

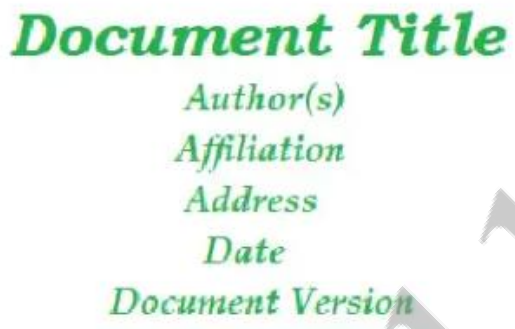
Verifiable: All requirements of the system, as documented in the SRS document, should be correct. This means that it should be possible to decide whether or not requirements have been met in an implementation.

2.2.4 Software Requirement Specification (SRS) Format

In order to form a good SRS, here you will see some points which can be used and should be considered to form a structure of good SRS. These are as follows : 1. Introduction

- **(i)** Purpose of this document
- **(ii)** Scope of this document
- **(iii)** Overview

2. General description 3. Functional Requirements 4. Interface Requirements 5. Performance Requirements 6. Design Constraints 7. Non-Functional Attributes 8. Preliminary Schedule and Budget 9. Appendices **Software Requirement Specification (SRS) Format** as name suggests, is complete specification and description of requirements of software that needs to be fulfilled for successful development of software system. These requirements can be functional as well as non-functional depending upon type of requirement. The interaction between different customers and contractor is done because its necessary to fully understand


Document Title

Author(s)

Affiliation

Address

Date

Document Version

needs of customers. Depending upon information gathered after interaction, SRS is developed which describes requirements of software that may include changes and modifications that is needed to be done to increase quality of product and to satisfy customer's demand.

1. **Introduction :**
   - **(i) Purpose of this Document –** At first, main aim of why this document is necessary and what's purpose of document is explained and described.
   - **(ii) Scope of this document –** In this, overall working and main objective of document and what value it will provide to customer is described and explained. It also includes a description of development cost and time required.
   - **(iii) Overview –** In this, description of product is explained. It's simply summary or overall review of product.
2. **General description :** In this, general functions of product which includes objective of user, a user characteristic, features, benefits, about why its importance is mentioned. It also describes features of user community.
3. **Functional Requirements :** In this, possible outcome of software system which includes effects due to operation of program is fully explained. All functional requirements which may include calculations, data processing, etc. are placed in a ranked order.
4. **Interface Requirements :** In this, software interfaces which mean how software program communicates with each other or users either in form of any language, code, or message are fully described and explained. Examples can be shared memory, data streams, etc.
5. **Performance Requirements :** In this, how a software system performs desired functions under specific condition is explained. It also explains required time, required memory, maximum error rate, etc.
6. **Design Constraints :** In this, constraints which simply means limitation or restriction are specified and explained for design team. Examples may include use of a particular algorithm, hardware and software limitations, etc.
7. **Non-Functional Attributes :** In this, non-functional attributes are explained that are required by software system for better performance. An example may include Security, Portability, Reliability, Reusability, Application compatibility, Data integrity, Scalability capacity, etc.

8. **Preliminary Schedule and Budget :** In this, initial version and budget of project plan are explained which include overall time duration required and overall cost required for development of project.
9. **Appendices :** In this, additional information like references from where information is gathered, definitions of some specific terms, acronyms, abbreviations, etc. are given and explained.

2.3 Formal system specification

Formal system specification in software engineering refers to the process of precisely describing the behavior, properties, and requirements of a software system using formal methods and mathematical notations. It aims to eliminate ambiguity, clarify system requirements, and enable formal analysis and verification of system properties.

Here are some key aspects of formal system specification:

Formal Methods: Formal methods are mathematical techniques used to specify, model, and reason about software systems. They provide a rigorous and precise approach to system specification, ensuring clarity and unambiguous representation. Common formal methods used for system specification include Z notation, B method, and Specification and Description Language (SDL).

Mathematical Notations: Formal system specification often involves using mathematical notations to express system requirements, constraints, and behaviors. These notations can include predicate logic, set theory, algebraic expressions, temporal logic, and process calculi, depending on the chosen formal method.

Language Constructs: Formal specification languages provide constructs and syntax for expressing system properties. For example, Z notation includes constructs for defining data types, predicates, schema definitions, and invariants. These language constructs allow for a systematic and structured representation of the system.

Requirements Capture: Formal system specification helps in capturing and refining system requirements. It enables the identification of inconsistencies, ambiguities, and gaps in **requirements** early in the development process. Formal methods facilitate the translation of informal requirements into precise and unambiguous specifications.

Verification and Validation: Formal system specification enables formal analysis, verification, and validation of system properties. Formal methods support the use of automated tools and theorem provers to check the consistency, correctness, and completeness of system specifications. This helps in identifying design flaws, detecting logical inconsistencies, and verifying that the system meets its intended requirements.

Refinement: Formal specification allows for stepwise refinement of system specifications. System requirements can be progressively refined into more detailed specifications, ensuring that the design and implementation faithfully adhere to the original requirements.

Documentation: Formal system specifications serve as a valuable source of documentation. They provide a clear and unambiguous representation of system requirements, constraints, and behavior, which aids in system understanding, maintenance, and future enhancements.

Formal system specification plays a crucial role in software engineering by enhancing the quality, reliability, and maintainability of software systems. It supports rigorous analysis, verification, and validation, leading to more dependable and trustworthy software designs.

2.4 Finite State Machine

- A state machine is a software computation model. It's just a model to solve a complex application, and it comprises a finite number of states. Hence it is also called a Finite State Machine. States are nothing but situations of your application (different situations).
- Since states are finite, there is a finite number of transitions among the states. Transitions are triggered by the incidents or input events fed to the state machine. An FSM is an event-driven reactive system.
- A state machine also produces an output. The output produced depends on the current state of the state machine sometimes, and sometimes it also depends on the input events fed to the state machine.

Benefits of using state machines:

- It is used to describe situations or scenarios of your application (Modelling the lifecycle of a reactive object through interconnections of states.
- FSMs are helpful to model complex applications that involve lots of decision-making, producing different outputs, and processing various events.
- State machines are visualized through state machine diagrams in the form of state charts, which helps to communicate between non-developers and developers.
- FSM makes it easier to visualize and implement the changes to the behavior of the project.

- Complex application can be visualized as a collection of different states processing a fixed set of events and producing a fixed set of outputs.
- Loose coupling: An application can be divided into multiple behaviors or state machines, and each unit can be tested separately or could be reused in other applications.
- Easy debugging and easy code maintenance.
- Scalable
- Narrow down the whole application completely to state-level complexity, analyze and implement.

Different types of state machines:

1. Mealy machine
2. Moore machine
3. Harel state charts
4. UML state machines

Some of these state machines are used for software engineering, and some state machines are still being used in digital electronics, VLSI design, etc.

UML Modelling tool and code generator

- Rhapsody® by IBM®
- QM™ Model-based design tool by Quantum Leaps, LLC
- Visual State by IAR
- Yakindu state chart tools by Itemis AG

Some are paid tools, and some are open-source. These tools can interpret your model and can auto-generate the code.

Types of Finite State Machines

Mealy machines are a type of simple state machine where the next state is determined by the current state and the given input.

The next state is determined by checking which input generates the next state with the current state. Imagine a button that only works when you're logged in. That button is a state machine with login as the current state.

The new state is determined by logging in. Once you're logged in, the next state is determined by the current state which is logged in.

Moore Machines

A Moore State Machine is a type of state machine for modeling systems with a high degree of uncertainty. In these systems, predicting the exact sequence of future events is difficult. As a result, it is difficult to determine the next event.

A Moore model captures this uncertainty by allowing the system to move from one state to another based on the outcome of a random event.

A Moore model has many applications in both industry and academia. For example, it can be used to predict when a system will fail or when certain events will occur with a high probability (e.g., when there will be an earthquake).

It can also be used as part of an optimization algorithm when dealing with uncertain inputs (e.g., produce only 1% more product than standard).

In addition, Moore models are often used as rules for automatic control systems (e.g., medical equipment) that need to respond quickly and accurately without human intervention.

Turing Machine

The Turing Machine consists of an input tape (with symbols on it), an internal tape (which corresponds to memory), and an output tape (which contains the result).

A Turing Machine operates through a series of steps: it scans its input tape, reads out one symbol at a time from its internal tape, and then applies this symbol as a command (or decision) to its output tape. For example: "If you see 'X' on the input tape, then print 'Y' on the output tape."

The input tape can be considered a finite set of symbols, while the internal and output tapes are infinite. The Turing Machine must read an entire symbol from its internal tape before it can move its head to the next symbol on the input tape.

Once it has moved its head to the next symbol, it can read that symbol out of its internal tape and then move to the next symbol on its input tape.

State Machine Use Cases

State machines are helpful for a variety of purposes. They can be used to model the flow of logic within a program, represent the states of a system, or for modeling the flow of events in a business process.

There are many different types of state machines, ranging from simple to highly complex. A few common use cases include:

Modeling Business Workflow Processes

State machines are ideal for modeling business workflows. This includes account setup flows, order completion, or a hiring process.

These things have a beginning and an end and follow some sort of sequential order. State machines are also great for modeling tasks that involve conditional logic.

Business Decision-Making

Companies pair FSMs with their data strategy to explore the cause and effect of business scenarios to make informed business decisions.

Business scenarios are often complex and unpredictable. There are many possible outcomes, and each one impacts the business differently.

A simulation allows you to try different business scenarios and see how each plays out. You can then assess the risk and determine the best course of action.

2.4 Petri nets

Petri nets are a mathematical modeling tool used in software engineering to analyze and describe the behavior of concurrent systems. They were introduced by Carl Adam Petri in the 1960s and have since been widely used in various fields, including software engineering.

Petri nets provide a graphical representation of a system's state and the transitions between those states. They consist of places, transitions, and arcs. Places represent states, transitions represent events or actions, and arcs represent the flow of tokens (also called markings) between places and transitions.

In software engineering, Petri nets can be used for various purposes, including:

System Modeling: Petri nets are used to model the behavior of complex software systems, especially concurrent and distributed systems. They help in understanding and visualizing how different components of the system interact and how their states change over time.

Specification and Verification: Petri nets can be used to specify the desired behavior of a software system. By modeling the system using Petri nets, it becomes possible to formally verify properties such as safety, liveness, reachability, and deadlock-freeness. Verification techniques based on Petri nets help in identifying design flaws, potential deadlocks, or other issues early in the development process.

Performance Analysis: Petri nets can be used to analyze the performance of software systems, including their throughput, response time, and resource utilization. By modeling the system's behavior and introducing timing annotations, it becomes possible to perform quantitative analysis and predict system performance under different conditions.

Workflow Modeling: Petri nets are often used to model business processes and workflows. In software engineering, this is particularly useful for modeling the flow of tasks and activities in software development processes, such as agile methodologies or continuous integration/continuous deployment (CI/CD) pipelines.

Software Testing: Petri nets can be utilized in software testing to generate test cases and verify the correctness of the system. By modeling the system's behavior and different scenarios, it becomes possible to systematically generate test cases that cover various paths and states, helping in identifying potential bugs and ensuring the software's reliability.

2.5 Object modelling using UML

UML (Unified Modeling Language) can be called a **graphical modeling language** that is used in the field of software engineering. It specifies, visualizes, builds, and documents the software system's artifacts (main elements). It is a visual language, not a programming language. UML diagrams are used to depict the behavior and structure of a system. UML facilitates modeling, design, and analysis for software developers, business owners, and system architects. A picture is worth a thousand words.

The Benefits of Using UML

- Because it is a general-purpose modeling language, it can be used by any modeler. UML is a straightforward modeling approach utilized to describe all practical systems.
- Because no standard methods were available then, UML was developed to systemize and condense object-oriented programming. Complex applications demand collaboration and preparation from numerous teams, necessitating a clear and straightforward means of communication among them.
- The **UML diagrams** are designed for customers, developers, laypeople, or anyone who wants to understand a system, whether software or non-software. Businesspeople do not understand code. As a result, UML becomes vital for communicating the system's essential requirements, features, and procedures to non-programmers. Technical details became easier to understand by non-technical people through an introduction to UML.
- When teams can visualize processes, user interactions, and the system's static structure, they save a lot of time in the long run.

Characteristics of UML

The UML has the following characteristics:

- It is a modeling language that has been generalized for various use cases.
- It is not a programming language; instead, it is a graphical modeling language that uses diagrams that can be understood by non-programmers as well.
- It has a close connection to object-oriented analysis and design.
- It is used to visualize the system's workflow.

Conceptual Modeling

Before moving ahead with the concept of UML, we should first understand the basics of the conceptual model.

A conceptual model is composed of several interrelated concepts. It makes it easy to understand the objects and how they interact with each other. This is the first step before drawing UML diagrams.

Following are some object-oriented concepts that are needed to begin with UML:

- **Object:** An object is a real world entity. There are many objects present within a single system. It is a fundamental building block of UML.
- **Class:** A class is a software blueprint for objects, which means that it defines the variables and methods common to all the objects of a particular type.
- **Abstraction:** Abstraction is the process of portraying the essential characteristics of an object to the users while hiding the irrelevant information. Basically, it is used to envision the functioning of an object.
- **Inheritance:** Inheritance is the process of deriving a new class from the existing ones.
- **Polymorphism:** It is a mechanism of representing objects having multiple forms used for different purposes.
- **Encapsulation:** It binds the data and the object together as a single unit, enabling tight coupling between them.

2.5.1 Use Case Diagram

A use case diagram is used to represent the dynamic behavior of a system. It encapsulates the system's functionality by incorporating use cases, actors, and their relationships. It models the tasks, services, and functions required by a system/subsystem of an application. It depicts the high-level functionality of a system and also tells how the user handles a system.

Purpose of Use Case Diagrams

The main purpose of a use case diagram is to portray the dynamic aspect of a system. It accumulates the system's requirement, which includes both internal as well as external influences. It invokes persons, use cases, and several things that invoke the actors and elements accountable for the implementation of use case diagrams. It represents how an entity from the external environment can interact with a part of the system.

Following are the purposes of a use case diagram given below:

1. It gathers the system's needs.
2. It depicts the external view of the system.
3. It recognizes the internal as well as external factors that influence the system.
4. It represents the interaction between the actors.

How to draw a Use Case diagram?

It is essential to analyze the whole system before starting with drawing a use case diagram, and then the system's functionalities are found. And once every single functionality is identified, they are then transformed into the use cases to be used in the use case diagram.

After that, we will enlist the actors that will interact with the system. The actors are the person or a thing that invokes the functionality of a system. It may be a system or a private entity, such that it requires an entity to be pertinent to the functionalities of the system to which it is going to interact.

Once both the actors and use cases are enlisted, the relation between the actor and use case/ system is inspected. It identifies the no of times an actor communicates with the system. Basically, an actor can interact multiple times with a use case or system at a particular instance of time.

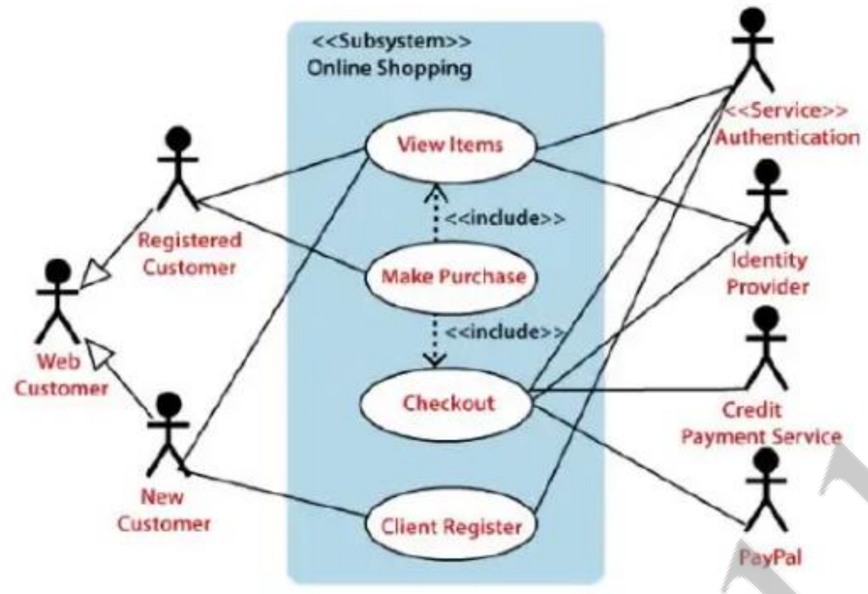
Following are some rules that must be followed while drawing a use case diagram:

1. A pertinent and meaningful name should be assigned to the actor or a use case of a system.
2. The communication of an actor with a use case must be defined in an understandable way.
3. Specified notations to be used as and when required.
4. The most significant interactions should be represented among the multiple no of interactions between the use case and actors.

Example of a Use Case Diagram

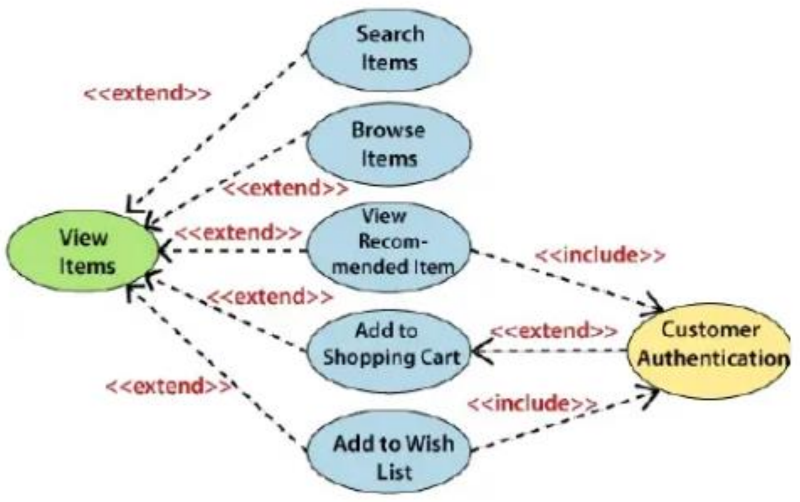
A use case diagram depicting the Online Shopping website is given below.

Here the Web Customer actor makes use of any online shopping website to purchase online. The top-level uses are as follows; View Items, Make Purchase, Checkout, Client Register. The **View Items** use case is utilized by the customer who searches and view products. The **Client Register** use case allows the customer to register itself with the website for availing gift vouchers, coupons, or getting a private sale invitation. It is to be noted that the **Checkout** is an included use case, which is part of **Making Purchase,** and it is not available by itself.



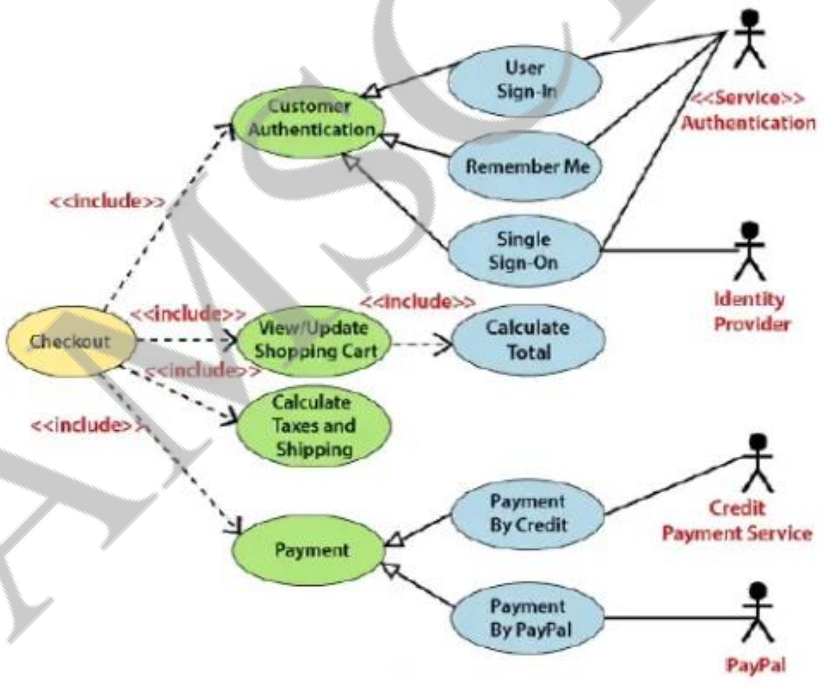
The **View Items** is further extended by several use cases such as; Search Items, Browse Items, View Recommended Items, Add to Shopping Cart, Add to Wish list. All of these extended use cases provide some functions to customers, which allows them to search for an item. The View Items is further extended by several use cases such as; Search Items, Browse Items, View Recommended Items, Add to Shopping Cart, Add to Wish list. All of these extended use cases provide some functions to customers, which allows them to search for an item.

Both **View Recommended Item** and **Add to Wish List** include the Customer Authentication use case, as they necessitate authenticated customers, and simultaneously item can be added to the shopping cart without any user authentication.



Similarly, the **Checkout** use case also includes the following use cases, as shown below. It requires an authenticated Web Customer, which can be done by login page, user authentication cookie ("Remember me"), or Single Sign-On (SSO). SSO needs an external identity provider's participation, while Web site authentication service is utilized in all these use cases.

The Checkout use case involves Payment use case that can be done either by the credit card and external credit payment services or with PayPal.



Important tips for drawing a Use Case diagram

Following are some important tips that are to be kept in mind while drawing a use case diagram:

1. A simple and complete use case diagram should be articulated.
2. A use case diagram should represent the most significant interaction among the multiple interactions.
3. At least one module of a system should be represented by the use case diagram.
4. If the use case diagram is large and more complex, then it should be drawn more generalized.

2.5.2 Class diagram

The class diagram depicts a static view of an application. It represents the types of objects residing in the system and the relationships between them. A class consists of its objects, and also it may inherit from other classes. A class diagram is used to visualize, describe, document various different aspects of the system, and also construct executable software code.

It shows the attributes, classes, functions, and relationships to give an overview of the software system. It constitutes class names, attributes, and functions in a separate compartment that helps in software development. Since it is a collection of classes, interfaces, associations, collaborations, and constraints, it is termed as a structural diagram.

Purpose of Class Diagrams

The main purpose of class diagrams is to build a static view of an application. It is the only diagram that is widely used for construction, and it can be mapped with object-oriented languages. It is one of the most popular UML diagrams. Following are the purpose of class diagrams given below:

1. It analyses and designs a static view of an application.
2. It describes the major responsibilities of a system.
3. It is a base for component and deployment diagrams.
4. It incorporates forward and reverse engineering.

Benefits of Class Diagrams

1. It can represent the object model for complex systems.
2. It reduces the maintenance time by providing an overview of how an application is structured before coding.
3. It provides a general schematic of an application for better understanding.
4. It represents a detailed chart by highlighting the desired code, which is to be programmed.
5. It is helpful for the stakeholders and the developers.

Vital components of a Class Diagram

The class diagram is made up of three sections:

- **Upper Section:** The upper section encompasses the name of the class. A class is a representation of similar objects that shares the same relationships, attributes, operations, and semantics. Some of the following rules that should be taken into account while representing a class are given below:

  a. Capitalize the initial letter of the class name.
  b. Place the class name in the center of the upper section.
  c. A class name must be written in bold format.
  d. The name of the abstract class should be written in italics format.

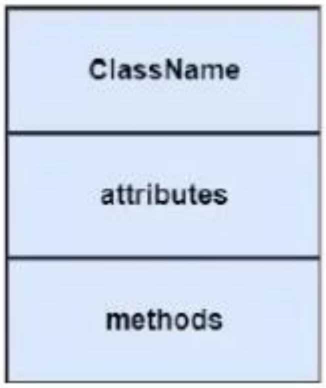
- **Middle Section:** The middle section constitutes the attributes, which describe the quality of the class. The attributes have the following characteristics:

a. The attributes are written along with its visibility factors, which are public (+), private (-), protected (#), and package (~).

b. The accessibility of an attribute class is illustrated by the visibility factors.

c. A meaningful name should be assigned to the attribute, which will explain its usage inside the class.

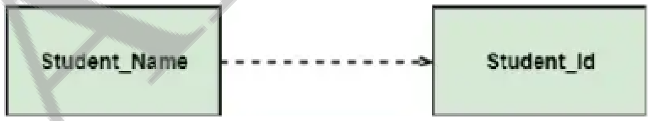
- **Lower Section:** The lower section contain methods or operations. The methods are represented in the form of a list, where each method is written in a single line. It demonstrates how a class interacts with data.



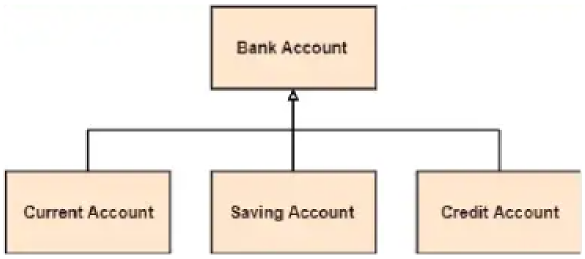
Relationships

In UML, relationships are of three types:

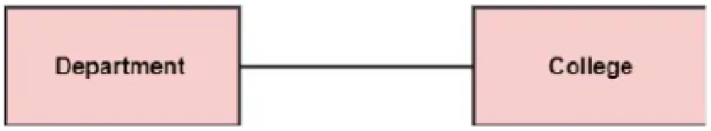
- **Dependency:** A dependency is a semantic relationship between two or more classes where a change in one class cause changes in another class. It forms a weaker relationship.
  In the following example, Student_Name is dependent on the Student_Id.



- **Generalization:** A generalization is a relationship between a parent class (superclass) and a child class (subclass). In this, the child class is inherited from the parent class. For example, The Current Account, Saving Account, and Credit Account are the generalized form of Bank Account.

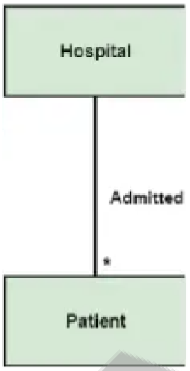


- **Association:** It describes a static or physical connection between two or more objects. It depicts how many objects are there in the relationship. For example, a department is associated with the college.



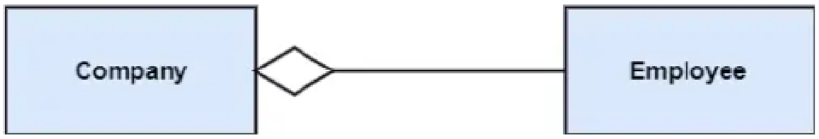
Multiplicity: It defines a specific range of allowable instances of attributes. In case if a range is not specified, one is considered as a default multiplicity.

For example, multiple patients are admitted to one hospital.



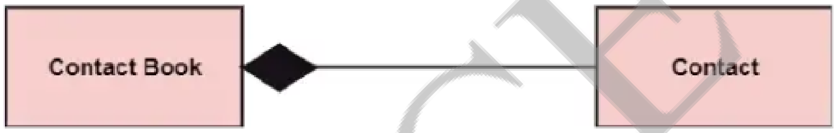
Aggregation: An aggregation is a subset of association, which represents has a relationship. It is more specific then association. It defines a part-whole or part-of relationship. In this kind of relationship, the child class can exist independently of its parent class.

The company encompasses a number of employees, and even if one employee resigns, the company still exists.



Composition: The composition is a subset of aggregation. It portrays the dependency between the parent and its child, which means if one part is deleted, then the other part also gets discarded. It represents a whole-part relationship.

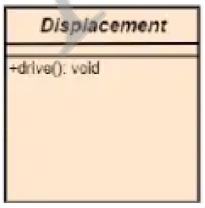
A contact book consists of multiple contacts, and if you delete the contact book, all the contacts will be lost.



Abstract Classes

In the abstract class, no objects can be a direct entity of the abstract class. The abstract class can neither be declared nor be instantiated. It is used to find the functionalities across the classes. The notation of the abstract class is similar to that of class; the only difference is that the name of the class is written in italics. Since it does not involve any implementation for a given function, it is best to use the abstract class with multiple objects.

Let us assume that we have an abstract class named **displacement** with a method declared inside it, and that method will be called as a **drive ()**. Now, this abstract class method can be implemented by any object, for example, car, bike, scooter, cycle, etc.



How to draw a Class Diagram?

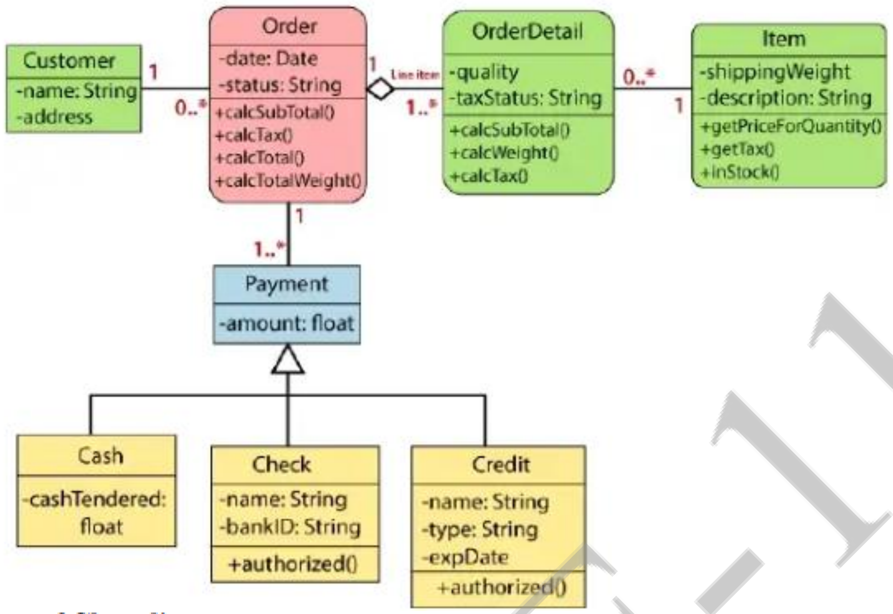
The class diagram is used most widely to construct software applications. It not only represents a static view of the system but also all the major aspects of an application. A collection of class diagrams as a whole represents a system.

Some key points that are needed to keep in mind while drawing a class diagram are given below:

1. To describe a complete aspect of the system, it is suggested to give a meaningful name to the class diagram.
2. The objects and their relationships should be acknowledged in advance.
3. The attributes and methods (responsibilities) of each class must be known.
4. A minimum number of desired properties should be specified as more number of the unwanted property will lead to a complex diagram.
5. Notes can be used as and when required by the developer to describe the aspects of a diagram.
6. The diagrams should be redrawn and reworked as many times to make it correct before producing its final version.

Class Diagram Example

A class diagram describing the sales order system is given below.



Usage of Class diagrams

The class diagram is used to represent a static view of the system. It plays an essential role in the establishment of the component and deployment diagrams. It helps to construct an executable code to perform forward and backward engineering for any system, or we can say it is mainly used for construction. It represents the mapping with object-oriented languages that are C++, Java, etc. Class diagrams can be used for the following purposes:

1. To describe the static view of a system.
2. To show the collaboration among every instance in the static view.
3. To describe the functionalities performed by the system.
4. To construct the software application using object-oriented languages.

2.5.3 Interaction diagram

As the name suggests, the interaction diagram portrays the interactions between distinct entities present in the model. It amalgamates both the activity and sequence diagrams. The communication is nothing but units of the behaviour of a classifier that provides context for interactions.

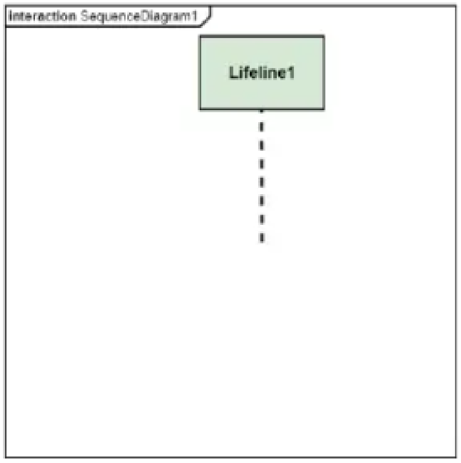
A set of messages that are interchanged between the entities to achieve certain specified tasks in the system is termed as interaction. It may incorporate any feature of the classifier of which it has access. In the interaction diagram, the critical component is the messages and the lifeline.

In UML, the interaction overview diagram initiates the interaction between the objects utilizing message passing. While drawing an interaction diagram, the entire focus is to represent the relationship among different objects which are available within the system boundary and the message exchanged by them to communicate with each other.
The message exchanged among objects is either to pass some information or to request some information. And based on the information, the interaction diagram is categorized into the sequence diagram, collaboration diagram, and timing diagram.

The sequence diagram envisions the order of the flow of messages inside the system by depicting the communication between two lifelines, just like a time-ordered sequence of events.

The collaboration diagram, which is also known as the communication diagram, represents how lifelines connect within the system, whereas the timing diagram focuses on that instant when a message is passed from one element to the other.

Notation of an Interaction Diagram



Purpose of an Interaction Diagram

The interaction diagram helps to envision the interactive (dynamic) behavior of any system. It portrays how objects residing in the system communicates and connects to each other. It also provides us with a context of communication between the lifelines inside the system.

Following are the purpose of an interaction diagram given below:

1. To visualize the dynamic behavior of the system.
2. To envision the interaction and the message flow in the system.
3. To portray the structural aspects of the entities within the system.
4. To represent the order of the sequenced interaction in the system.
5. To visualize the real-time data and represent the architecture of an object-oriented system.

How to draw an Interaction Diagram?

Since the main purpose of an interaction diagram is to visualize the dynamic behavior of the system, it is important to understand what a dynamic aspect really is and how we can visualize it. The dynamic aspect is nothing but a screenshot of the system at the run time.

Before drawing an interaction diagram, the first step is to discover the scenario for which the diagram will be made. Next, we will identify various lifelines that will be invoked in the communication, and then we will classify each lifeline. After that, the connections are investigated and how the lifelines are interrelated to each other.

Following are some things that are needed:

1. A total no of lifeline which will take part in the communication.
2. The sequence of the message flow among several entities within the system.
3. No operators used to ease out the functionality of the diagram.
4. Several distinct messages that depict the interactions in a precise and clear way.
5. The organization and structure of a system.
6. The order of the sequence of the flow of messages.
7. Total no of time constructs of an object.

Use of an Interaction Diagram

The interaction diagram can be used for:

1. The sequence diagram is employed to investigate a new application.
2. The interaction diagram explores and compares the use of the collaboration diagram sequence diagram and the timing diagram.
3. The interaction diagram represents the interactive (dynamic) behaviour of the system.
4. The sequence diagram portrays the order of control flow from one element to the other elements inside the system, whereas the collaboration diagrams are employed to get an overview of the object architecture of the system.
5. The interaction diagram models the system as a time-ordered sequence of a system.
6. The interaction diagram models the system as a time-ordered sequence of a system.
7. The interaction diagram systemizes the structure of the interactive elements.

2.5.4 Activity diagram

In UML, the activity diagram is used to demonstrate the flow of control within the system rather than the implementation. It models the concurrent and sequential activities.

The activity diagram helps in envisioning the workflow from one activity to another. It put emphasis on the condition of flow and the order in which it occurs. The flow can be sequential, branched, or concurrent, and to deal with such kinds of flows, the activity diagram has come up with a fork, join, etc.

It is also termed as an object-oriented flowchart. It encompasses activities composed of a set of actions or operations that are applied to model the behavioral diagram.

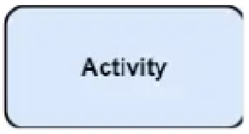
Components of an Activity Diagram

Following are the component of an activity diagram:

Activities

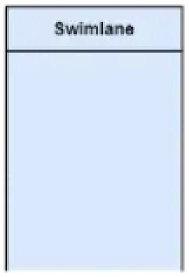
The categorization of behavior into one or more actions is termed as an activity. In other words, it can be said that an activity is a network of nodes that are connected by edges. The edges depict the flow of execution. It may contain action nodes, control nodes, or object nodes.

The control flow of activity is represented by control nodes and object nodes that illustrates the objects used within an activity. The activities are initiated at the initial node and are terminated at the final node.



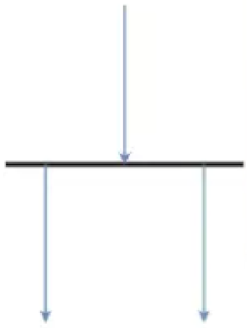
Activity partition /swimlane

The swimlane is used to cluster all the related activities in one column or one row. It can be either vertical or horizontal. It used to add modularity to the activity diagram. It is not necessary to incorporate swimlane in the activity diagram. But it is used to add more transparency to the activity diagram.



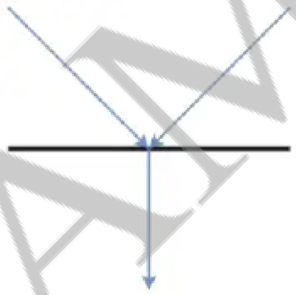
Forks

Forks and join nodes generate the concurrent flow inside the activity. A fork node consists of one inward edge and several outward edges. It is the same as that of various decision parameters. Whenever a data is received at an inward edge, it gets copied and split crossways various outward edges. It split a single inward flow into multiple parallel flows.



Join Nodes

Join nodes are the opposite of fork nodes. A Logical AND operation is performed on all of the inward edges as it synchronizes the flow of input across one single output (outward) edge.



Pins

It is a small rectangle, which is attached to the action rectangle. It clears out all the messy and complicated thing to manage the execution flow of activities. It is an object node that precisely represents one input to or output from the action.

Notation of an Activity diagram

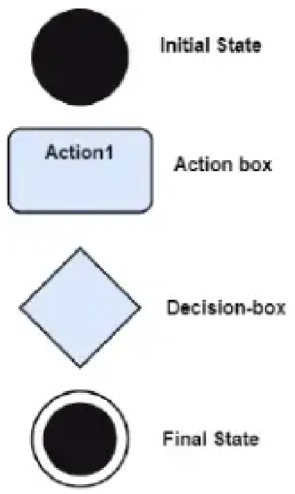
Activity diagram constitutes following notations:

Initial State: It depicts the initial stage or beginning of the set of actions.

Final State: It is the stage where all the control flows and object flows end.

Decision Box: It makes sure that the control flow or object flow will follow only one path.

Action Box: It represents the set of actions that are to be performed.



Why use Activity Diagram?

An event is created as an activity diagram encompassing a group of nodes associated with edges. To model the behavior of activities, they can be attached to any modeling element. It can model use cases, classes, interfaces, components, and collaborations.

It mainly models processes and workflows. It envisions the dynamic behavior of the system as well as constructs a runnable system that incorporates forward and reverse engineering. It does not include the message part, which means message flow is not represented in an activity diagram.

It is the same as that of a flowchart but not exactly a flowchart itself. It is used to depict the flow between several activities.

How to draw an Activity Diagram?

An activity diagram is a flowchart of activities, as it represents the workflow among various activities. They are identical to the flowcharts, but they themself are not exactly the flowchart. In other words, it can be said that an activity diagram is an enhancement of the flowchart, which encompasses several unique skills.

Since it incorporates swimlanes, branching, parallel flows, join nodes, control nodes, and forks, it supports exception handling. A system must be explored as a whole before drawing an activity diagram to provide a clearer view of the user. All of the activities are explored after they are properly analyzed for finding out the constraints applied to the activities. Each and every activity, condition, and association must be recognized.

After gathering all the essential information, an abstract or a prototype is built, which is then transformed into the actual diagram.

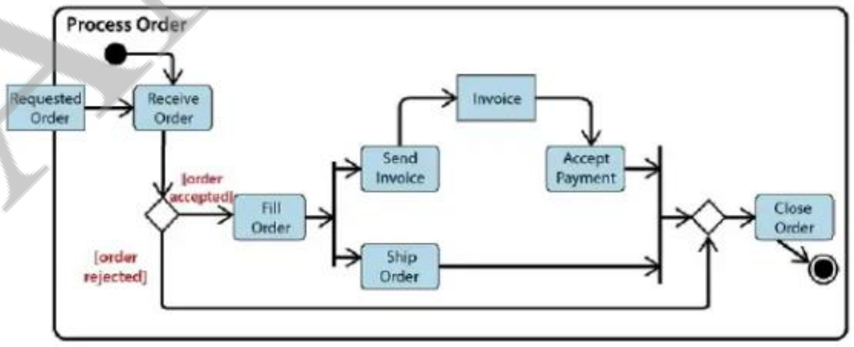
Following are the rules that are to be followed for drawing an activity diagram:

1. A meaningful name should be given to each and every activity.
2. Identify all of the constraints.
3. Acknowledge the activity associations.

Example of an Activity Diagram

An example of an activity diagram showing the business flow activity of order processing is given below.

Here the input parameter is the Requested order, and once the order is accepted, all of the required information is then filled, payment is also accepted, and then the order is shipped. It permits order shipment before an invoice is sent or payment is completed.



When to use an Activity Diagram?

An activity diagram can be used to portray business processes and workflows. Also, it used for modeling business as well as the software. An activity diagram is utilized for the followings:

1. To graphically model the workflow in an easier and understandable way.
2. To model the execution flow among several activities.
3. To model comprehensive information of a function or an algorithm employed within the system.
4. To model the business process and its workflow.
5. To envision the dynamic aspect of a system.
6. To generate the top-level flowcharts for representing the workflow of an application.
7. To represent a high-level view of a distributed or an object-oriented system.

2.5.5 State chart diagram

A **state diagram** is used to represent the condition of the system or part of the system at finite instances of time. It's a **behavioral** diagram and it represents the behavior using finite state transitions. State diagrams are also referred to as **State machines** and **State-chart Diagrams**. These terms are often used interchangeably. So simply, a state diagram is used to model the dynamic behavior of a class in response to time and changing external stimuli. We can say that each and every class has a state but we don't model every class using State diagrams. We prefer to model the states with three or more states.

Uses of statechart diagram –

- We use it to state the events responsible for change in state (we do not show what processes cause those events).
- We use it to model the dynamic behavior of the system .
- To understand the reaction of objects/classes to internal or external stimuli.

Firstly let us understand what are **Behavior diagrams**? There are two types of diagrams in UML :

1. **Structure Diagrams –** Used to model the static structure of a system, for example- class diagram, package diagram, object diagram, deployment diagram etc.
2. **Behavior diagram –** Used to model the dynamic change in the system over time. They are used to model and construct the functionality of a system. So, a behavior diagram simply guides us through the functionality of the system using Use case diagrams, Interaction diagrams, Activity diagrams and State diagrams.

Basic components of a statechart diagram –

1. **Initial state –** We use a black filled circle represent the initial state of a System or a class.

Figure – initial state notation

2. **Transition –** We use a solid arrow to represent the transition or change of control from one state to another. The arrow is labelled with the event which causes the change in state.

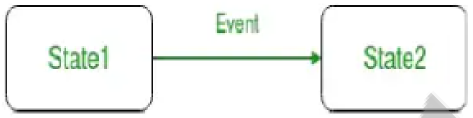


Figure – transition

3. **State –** We use a rounded rectangle to represent a state. A state represents the conditions or circumstances of an object of a class at an instant of time.

Figure – state notation

4. **Fork –** We use a rounded solid rectangular bar to represent a Fork notation with incoming arrow from the parent state and outgoing arrows towards the newly created states. We use the fork notation to represent a state splitting into two or more concurrent states.

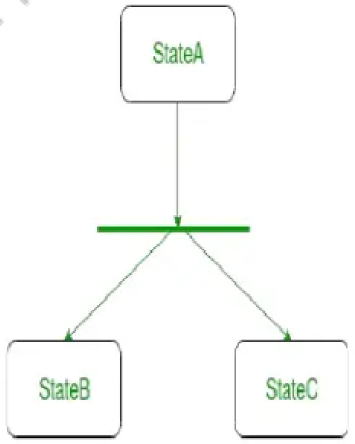


Figure – a diagram using the fork notation

5. **Join –** We use a rounded solid rectangular bar to represent a Join notation with incoming arrows from the joining states and outgoing arrow towards the common goal state. We use the join notation when two or more states concurrently converge into one on the occurrence of an event or events.

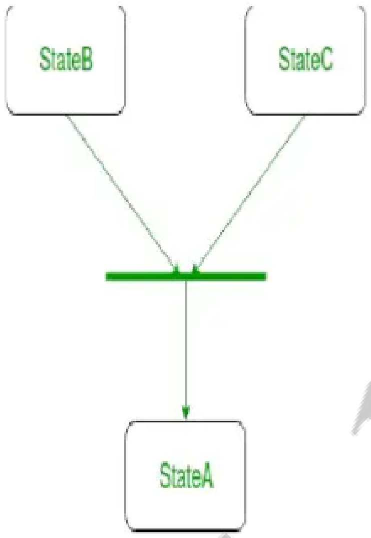


Figure – a diagram using join notation

6. **Self transition –** We use a solid arrow pointing back to the state itself to represent a self transition. There might be scenarios when the state of the object does not change upon the occurrence of an event. We use self transitions to represent such cases.

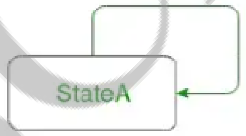


Figure – self transition notation

7. **Composite state –** We use a rounded rectangle to represent a composite state also. We represent a state with internal activities using a composite state.

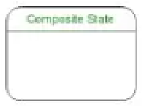


Figure – a state with internal activities

8. **Final state –** We use a filled circle within a circle notation to represent the final state in a state machine diagram.

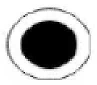


Figure – final state notation

Steps to draw a state diagram –

1. Identify the initial state and the final terminating states.
2. Identify the possible states in which the object can exist (boundary values corresponding to different attributes guide us in identifying different states).
3. Label the events which trigger these transitions.

Example – state diagram for an online order –

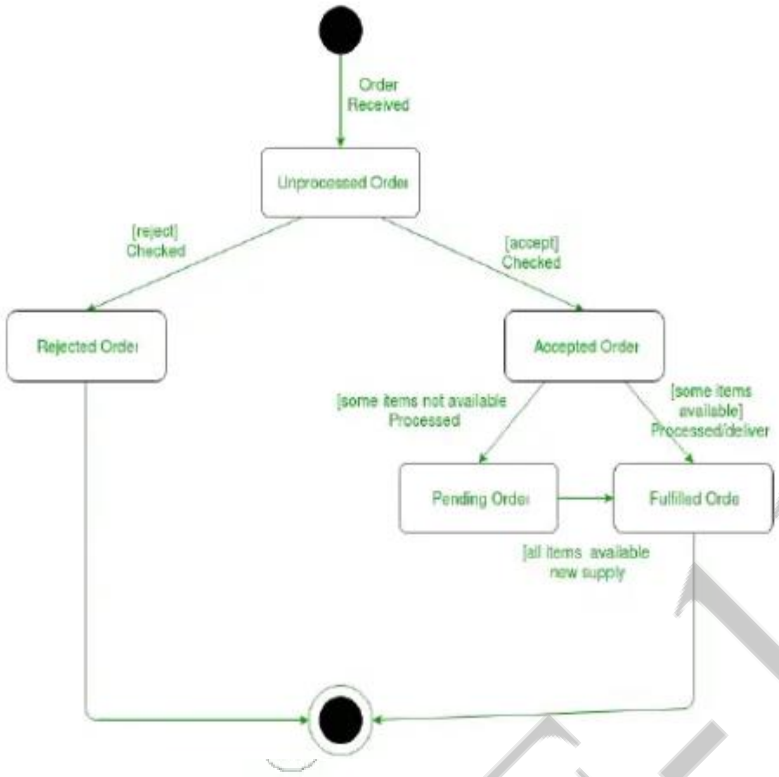


Figure – state diagram for an online order

The UMl diagrams we draw depend on the system we aim to represent. Here is just an example of how an online ordering system might look like :

1. On the event of an order being received, we transit from our initial state to Unprocessed order state.
2. The unprocessed order is then checked.
3. If the order is rejected, we transit to the Rejected Order state.
4. If the order is accepted and we have the items available we transit to the fulfilled order state.
5. However if the items are not available we transit to the Pending Order state.
6. After the order is fulfilled, we transit to the final state. In this example, we merge the two states i.e. Fulfilled order and Rejected order into one final state.

2.6 Functional Modelling

Functional Modelling gives the process perspective of the object-oriented analysis model and an overview of what the system is supposed to do. It defines the function of the internal processes in the system with the aid of Data Flow Diagrams (DFDs). It depicts the functional derivation of the data values without indicating how they are derived when they are computed, or why they need to be computed.

2.6.1 Data Flow Diagrams

Functional Modelling is represented through a hierarchy of DFDs. The DFD is a graphical representation of a system that shows the inputs to the system, the processing upon the inputs, the outputs of the system as well as the internal data stores. DFDs illustrate the series of transformations or computations performed on the objects or the system, and the external controls and objects that affect the transformation.

Rumbaugh et al. have defined DFD as, "A data flow diagram is a graph which shows the flow of data values from their sources in objects through processes that transform them to their destinations on other objects."

The four main parts of a DFD are −

- Processes,
- Data Flows,
- Actors, and
- Data Stores.

The other parts of a DFD are −

- Constraints, and
- Control Flows.

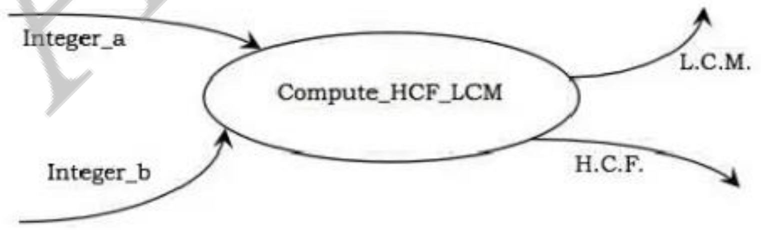
Features of a DFD

Processes

Processes are the computational activities that transform data values. A whole system can be visualized as a high-level process. A process may be further divided into smaller components. The lowest-level process may be a simple function.

Representation in DFD − A process is represented as an ellipse with its name written inside it and contains a fixed number of input and output data values.

Example − The following figure shows a process Compute_HCF_LCM that accepts two integers as inputs and outputs their HCF (highest common factor) and LCM (least common multiple).



Data Flows

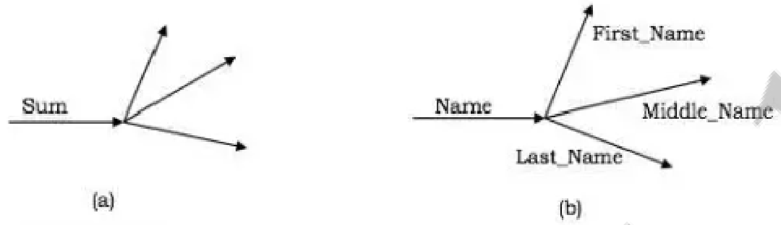
Data flow represents the flow of data between two processes. It could be between an actor and a process, or between a data store and a process. A data flow denotes the value of a data item at some point of the computation. This value is not changed by the data flow.

Representation in DFD − A data flow is represented by a directed arc or an arrow, labelled with the name of the data item that it carries.

In the above figure, Integer_a and Integer_b represent the input data flows to the process, while L.C.M. and H.C.F. are the output data flows.

A data flow may be forked in the following cases −

- The output value is sent to several places as shown in the following figure. Here, the output arrows are unlabelled as they denote the same value.
- The data flow contains an aggregate value, and each of the components is sent to different places as shown in the following figure. Here, each of the forked components is labelled.

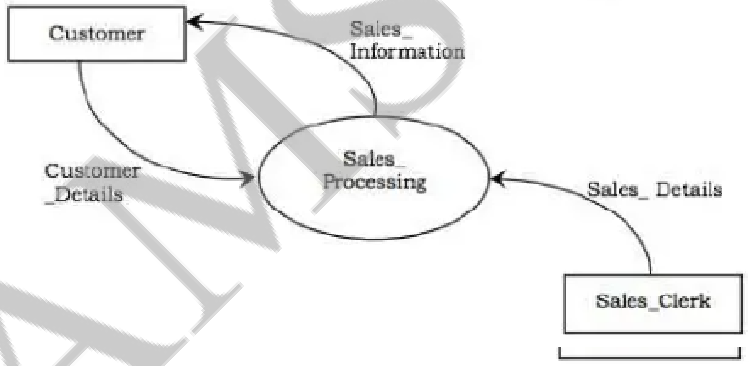


Actors

Actors are the active objects that interact with the system by either producing data and inputting them to the system, or consuming data produced by the system. In other words, actors serve as the sources and the sinks of data.

Representation in DFD − An actor is represented by a rectangle. Actors are connected to the inputs and outputs and lie on the boundary of the DFD.

Example − The following figure shows the actors, namely, Customer and Sales_Clerk in a counter sales system.

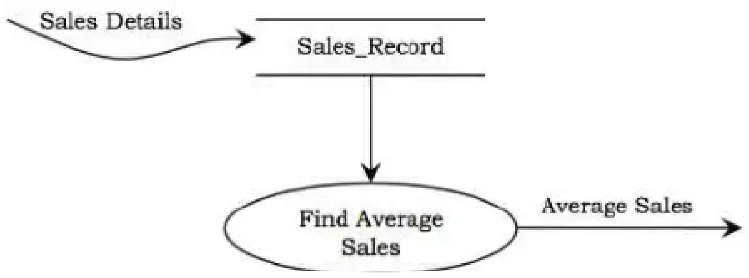


Data Stores

Data stores are the passive objects that act as a repository of data. Unlike actors, they cannot perform any operations. They are used to store data and retrieve the stored data. They represent a data structure, a disk file, or a table in a database.

Representation in DFD − A data store is represented by two parallel lines containing the name of the data store. Each data store is connected to at least one process. Input arrows contain information to modify the contents of the data store, while output arrows contain information retrieved from the data store. When a part of the information is to be retrieved, the output arrow is labelled. An unlabelled arrow denotes full data retrieval. A two-way arrow implies both retrieval and update.

Example − The following figure shows a data store, Sales_Record, that stores the details of all sales. Input to the data store comprises of details of sales such as item, billing amount, date, etc. To find the average sales, the process retrieves the sales records and computes the average.



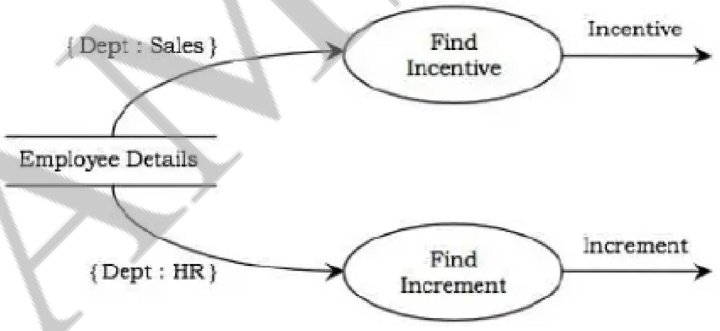
Constraints

Constraints specify the conditions or restrictions that need to be satisfied over time. They allow adding new rules or modifying existing ones. Constraints can appear in all the three models of object-oriented analysis.

- In Object Modelling, the constraints define the relationship between objects. They may also define the relationship between the different values that an object may take at different times.
- In Dynamic Modelling, the constraints define the relationship between the states and events of different objects.
- In Functional Modelling, the constraints define the restrictions on the transformations and computations.

Representation − A constraint is rendered as a string within braces.

Example − The following figure shows a portion of DFD for computing the salary of employees of a company that has decided to give incentives to all employees of the sales department and increment the salary of all employees of the HR department. It can be seen that the constraint {Dept:Sales} causes incentive to be calculated only if the department is sales and the constraint {Dept:HR} causes increment to be computed only if the department is HR.

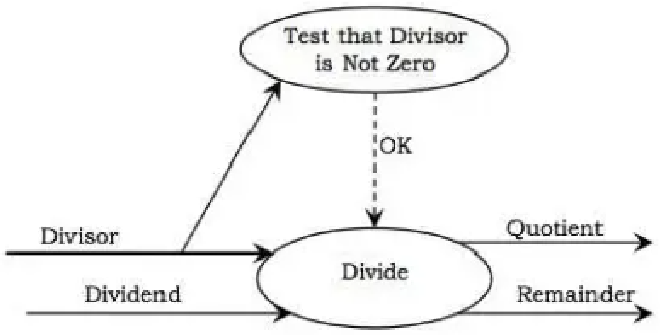


Control flow

A process may be associated with a certain Boolean value and is evaluated only if the value is true, though it is not a direct input to the process. These Boolean values are called the control flows.

Representation in DFD − Control flows are represented by a dotted arc from the process producing the Boolean value to the process controlled by them.

Example − The following figure represents a DFD for arithmetic division. The Divisor is tested for non-zero. If it is not zero, the control flow OK has a value True and subsequently the Divide process computes the Quotient and the Remainder.



Developing the DFD Model of a System

In order to develop the DFD model of a system, a hierarchy of DFDs are constructed. The top-level DFD comprises of a single process and the actors interacting with it.

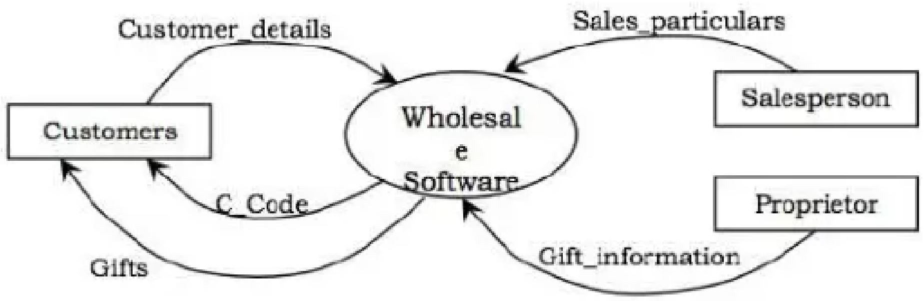
At each successive lower level, further details are gradually included. A process is decomposed into sub-processes, the data flows among the sub-processes are identified, the control flows are determined, and the data stores are defined. While decomposing a process, the data flow into or out of the process should match the data flow at the next level of DFD.

Example − Let us consider a software system, Wholesaler Software, that automates the transactions of a wholesale shop. The shop sells in bulks and has a clientele comprising of merchants and retail shop owners. Each customer is asked to register with his/her particulars and is given a unique customer code, C_Code. Once a sale is done, the shop registers its details and sends the goods for dispatch. Each year, the shop distributes Christmas gifts to its customers, which comprise of a silver coin or a gold coin depending upon the total sales and the decision of the proprietor.

The functional model for the Wholesale Software is given below. The figure below shows the top-level DFD. It shows the software as a single process and the actors that interact with it.

The actors in the system are −

- Customers
- Salesperson
- Proprietor



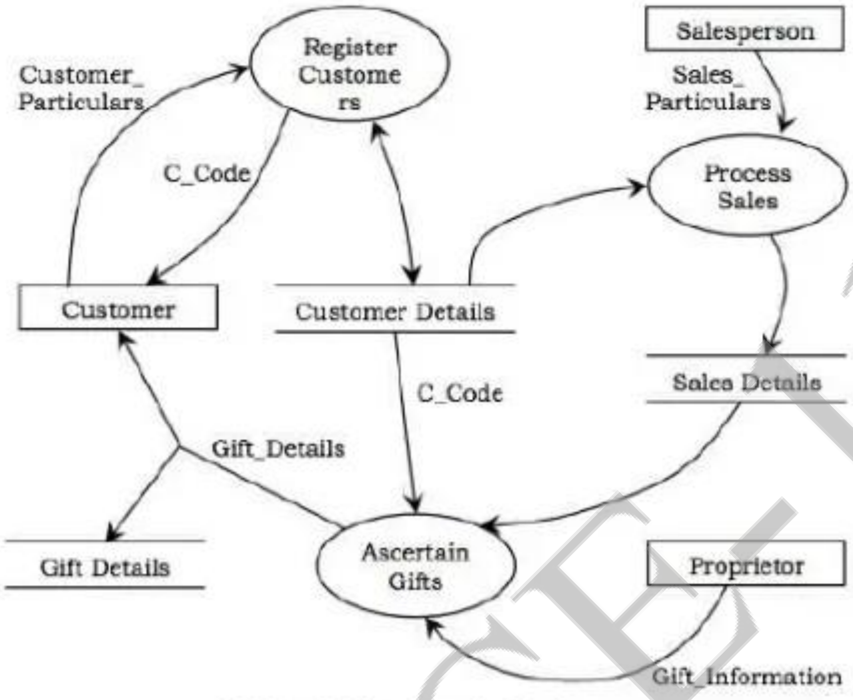
In the next level DFD, as shown in the following figure, the major processes of the system are identified, the data stores are defined and the interaction of the processes with the actors, and the data stores are established.

In the system, three processes can be identified, which are −

- Register Customers
- Process Sales
- Ascertain Gifts

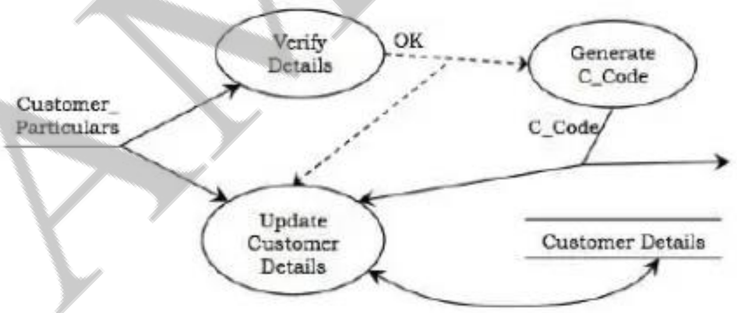
The data stores that will be required are –

- Customer Details
- Sales Details
- Gift Details



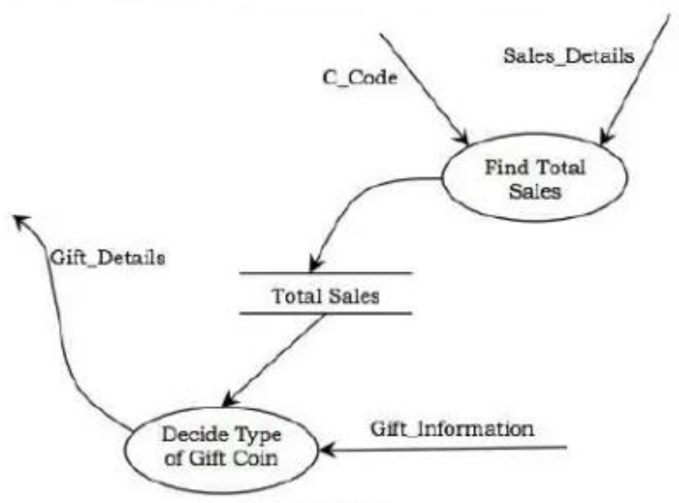
DFD of Wholesale Software

The following figure shows the details of the process Register Customer. There are three processes in it, Verify Details, Generate C_Code, and Update Customer Details. When the details of the customer are entered, they are verified. If the data is correct, C_Code is generated and the data store Customer Details is updated.



DFD of Customer Process

The following figure shows the expansion of the process Ascertain Gifts. It has two processes in it, Find Total Sales and Decide Type of Gift Coin. The Find Total Sales process computes the yearly total sales corresponding to each customer and records the data. Taking this record and the decision of the proprietor as inputs, the gift coins are allotted through Decide Type of Gift Coin process.



DFD of Gift Process

DFD of Gift Process

Advantages and Disadvantages of DFD

Advantages	Disadvantages
DFDs depict the boundaries of a system and hence are helpful in portraying the relationship between the external objects and the processes within the system.	DFDs take a long time to create, which may not be feasible for practical purposes.
They help the users to have a knowledge about the system.	DFDs do not provide any information about the time-dependent behavior, i.e., they do not specify when the transformations are done.
The graphical representation serves as a blueprint for the programmers to develop a	They do not throw any light on the frequency of computations or the reasons for computations.

system.	
DFDs provide detailed information about the system processes.	The preparation of DFDs is a complex process that needs considerable expertise. Also, it is difficult for a non-technical person to understand.
They are used as a part of the system documentation.	The method of preparation is subjective and leaves ample scope to be imprecise.

Relationship between Object, Dynamic, and Functional Models

The Object Model, the Dynamic Model, and the Functional Model are complementary to each other for a complete Object-Oriented Analysis.

- Object modelling develops the static structure of the software system in terms of objects. Thus it shows the "doers" of a system.
- Dynamic Modelling develops the temporal behavior of the objects in response to external events. It shows the sequences of operations performed on the objects.
- Functional model gives an overview of what the system should do.

Functional Model and Object Model

The four main parts of a Functional Model in terms of object model are −

- **Process** − Processes imply the methods of the objects that need to be implemented.
- **Actors** − Actors are the objects in the object model.
- **Data Stores** − These are either objects in the object model or attributes of objects.
- **Data Flows** − Data flows to or from actors represent operations on or by objects. Data flows to or from data stores represent queries or updates.

Functional Model and Dynamic Model

The dynamic model states when the operations are performed, while the functional model states how they are performed and which arguments are needed. As actors are active objects, the dynamic model has to specify when it acts. The data stores are passive objects and they only respond to updates and queries; therefore, the dynamic model need not specify when they act.

Object Model and Dynamic Model

The dynamic model shows the status of the objects and the operations performed on the occurrences of events and the subsequent changes in states. The state of the object as a result of the changes is shown in the object model.

UNIT III SOFTWARE DESIGN 9

Software design – Design process – Design concepts – Coupling – Cohesion – Functional independence – Design patterns – Model-view-controller – Publish subscribe – Adapter – Command – Strategy – Observer – Proxy – Facade – Architectural styles – Layered - Client Server - Tiered - Pipe and filter- User interface design-Case Study.

Software Design is the process of transforming user requirements into a suitable form, which helps the programmer in software coding and implementation. During the software design phase, the design document is produced, based on the customer requirements as documented in the SRS document. Hence, this phase aims to transform the SRS document into a design document.

The following items are designed and documented during the design phase:

1. Different modules are required.
2. Control relationships among modules.
3. Interface among different modules.
4. Data structure among the different modules.
5. Algorithms are required to be implemented among the individual modules.

Objectives of Software Design

1. **Correctness:** A good design should be correct i.e., it should correctly implement all the functionalities of the system.
2. **Efficiency:** A good software design should address the resources, time, and cost optimization issues.
3. **Flexibility:** A good software design should have the ability to adapt and accommodate changes easily. It includes designing the software in a way, that allows for modifications, enhancements, and scalability without requiring significant rework or causing major disruptions to the existing functionality.
4. **Understandability:** A good design should be easily understandable, it should be modular, and all the modules are arranged in layers.
5. **Completeness:** The design should have all the components like data structures, modules, external interfaces, etc.
6. **Maintainability:** A good software design aims to create a system that is easy to understand, modify, and maintain over time. This involves using modular and well-structured design principles e.g.,(employing appropriate naming conventions and providing clear documentation). Maintainability in Software and design also enables developers to fix bugs, enhance features, and adapt the software to changing requirements without excessive effort or introducing new issues.

Software Design Concepts

Concepts are defined as a principal idea or invention that comes into our mind or in thought to understand something. The **software design concept** simply means the idea or principle behind the design. It describes how you plan to solve the problem of designing software, and the logic, or thinking behind how you will design software. It allows the software engineer to create the model of the system software or product that is to be developed or built. The software design concept provides a supporting and essential structure or model for developing the right software. There are many concepts of software design and some of them are given below:

Points to be Considered While Designing Software

1. **Abstraction** (**Hide Irrelevant data**)**:** Abstraction simply means to hide the details to reduce complexity and increase efficiency or quality. Different levels of Abstraction are necessary and must be applied at each stage of the design process so that any error that is present can be removed to increase the efficiency of the software solution and to refine the software solution. The solution should be described in broad ways that cover a wide range of different things at a higher level of abstraction and a more detailed description of a solution of software should be given at the lower level of abstraction.
2. **Modularity (subdivide the system):** Modularity simply means dividing the system or project into smaller parts to reduce the complexity of the system or project. In the same way, modularity in design means subdividing a system into smaller parts so that these parts can be created independently and then use these parts in different systems to perform different functions. It is necessary to divide the software into components known as modules because nowadays, there are different software available like Monolithic software that is hard to grasp for software engineers. So, modularity in design has now become a trend and is also important. If the system contains fewer components then it would mean the system is complex which requires a lot of effort (cost) but if we can divide the system into components then the cost would be small.
3. **Architecture (design a structure of something):** Architecture simply means a technique to design a structure of something. Architecture in designing software is a concept that focuses on various elements and the data of the structure. These components interact with each other and use the data of the structure in architecture.
4. **Refinement (removes impurities):** Refinement simply means to refine something to remove any impurities if present and increase the quality. The refinement concept of software design is a process of developing or presenting the software or system in a detailed manner which means elaborating a system or software. Refinement is very necessary to find out any error if present and then to reduce it.
5. **Pattern (a Repeated form):** A pattern simply means a repeated form or design in which the same shape is repeated several times to form a pattern. The pattern in the design process means the repetition of a solution to a common recurring problem within a certain context.
6. **Information Hiding (Hide the Information**)**:** Information hiding simply means to hide the information so that it cannot be accessed by an unwanted party. In software design, information hiding is achieved by designing the modules in a manner that the information gathered or contained in one module is hidden and can’t be accessed by any other modules.
7. **Refactoring (Reconstruct something):** Refactoring simply means reconstructing something in such a way that it does not affect the behavior of any other features. Refactoring in software design means reconstructing the design to reduce complexity and simplify it without impacting the behavior or its functions. Fowler has defined refactoring as

"the process of changing a software system in a way that it won't impact the behavior of the design and improves the internal structure".

3.2 DESIGN CONCEPTS

Design concepts provides the software designer with a foundation from which more sophisticated design methods can be applied and helps the software engineer to answer the following questions: • What criteria can be used to partition software into individual components? • How is function or data structure detail separated from a conceptual representation of the software? • What uniform criteria define the technical quality of a software design? Fundamental software design concepts provide the necessary framework for "getting it right."

Abstraction

Architecture

Patterns

Separation of Concerns

Modularity

Information Hiding

Functional Independence

Refinement

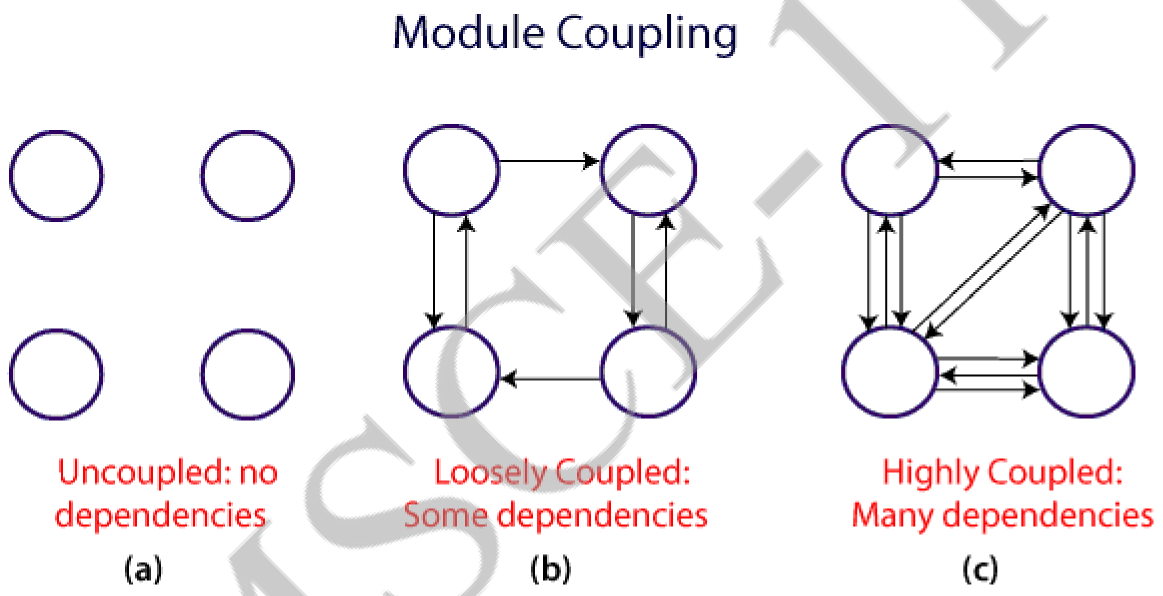
Aspects Refactoring Object-Oriented Design Concepts Design Classes

Coupling and Cohesion

Module Coupling

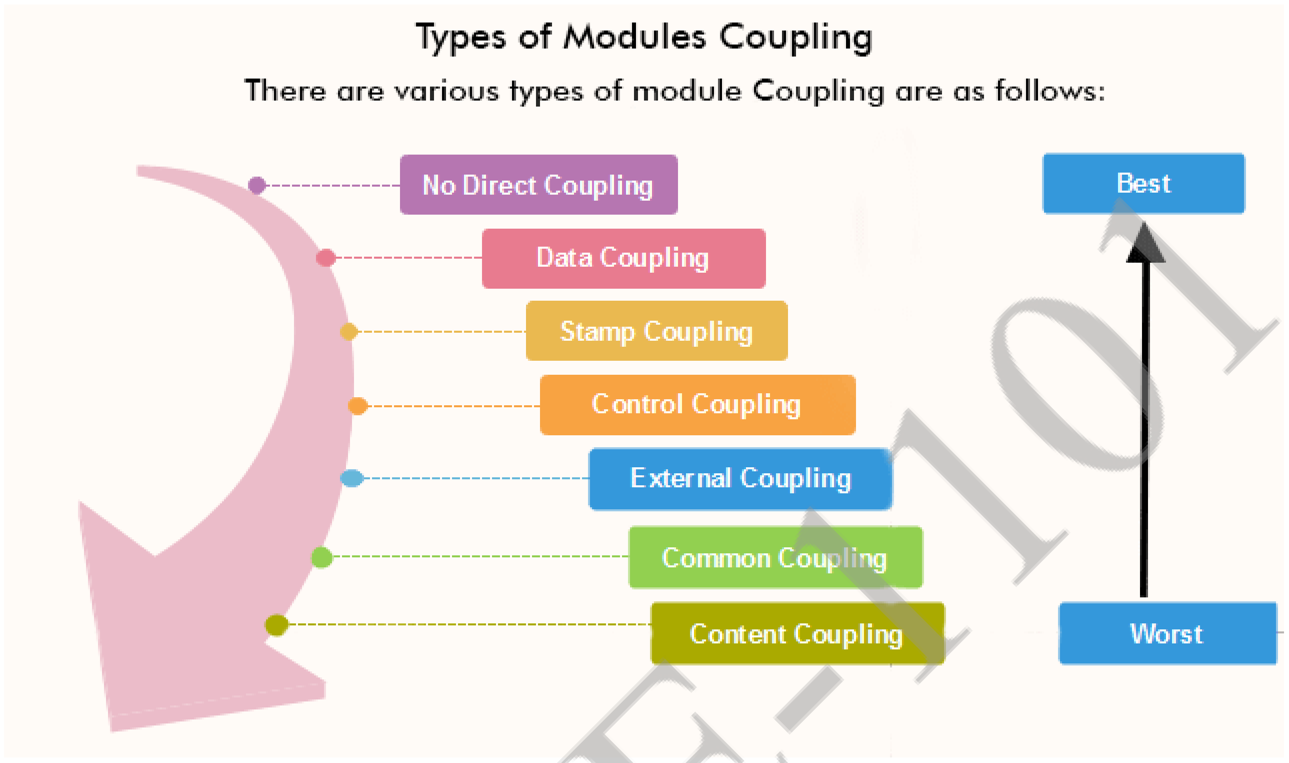
In software engineering, the coupling is the **degree of interdependence** between software modules. Two modules that are **tightly coupled** are strongly dependent on each other. However, two modules that are **loosely coupled** are not dependent on each other. **Uncoupled modules** have no interdependence at all within them.

The various types of coupling techniques are shown in fig:

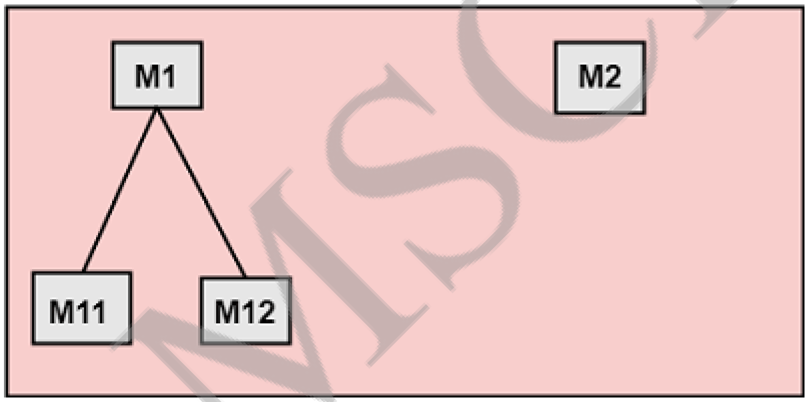

Module Coupling
Uncoupled: no dependencies
(a)
Loosely Coupled: Some dependencies
(b)
Highly Coupled: Many dependencies
(c)

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



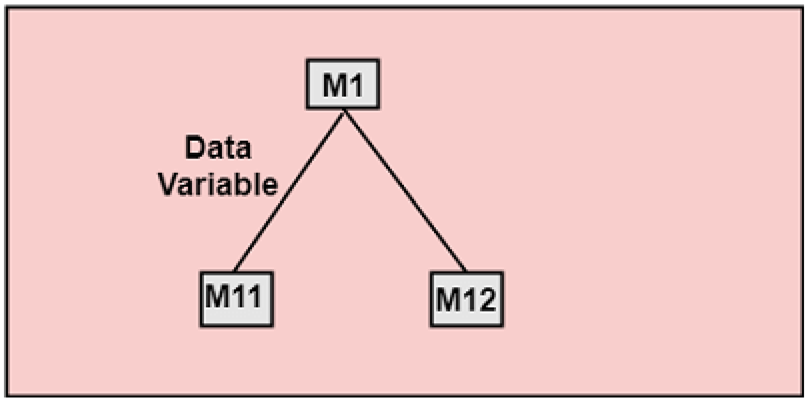
1. No Direct Coupling: There is no direct coupling between M1 and M2.



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.

.

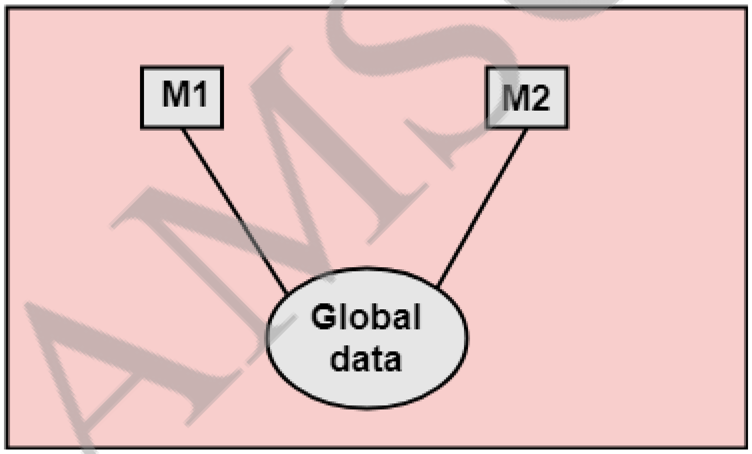


3. Stamp Coupling: Two modules are stamp coupled if they communicate using **composite data items such as structure, objects**, etc. When the module passes non-global data structure or entire structure to another module, they are said to be stamp coupled. For example, passing structure variable in C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

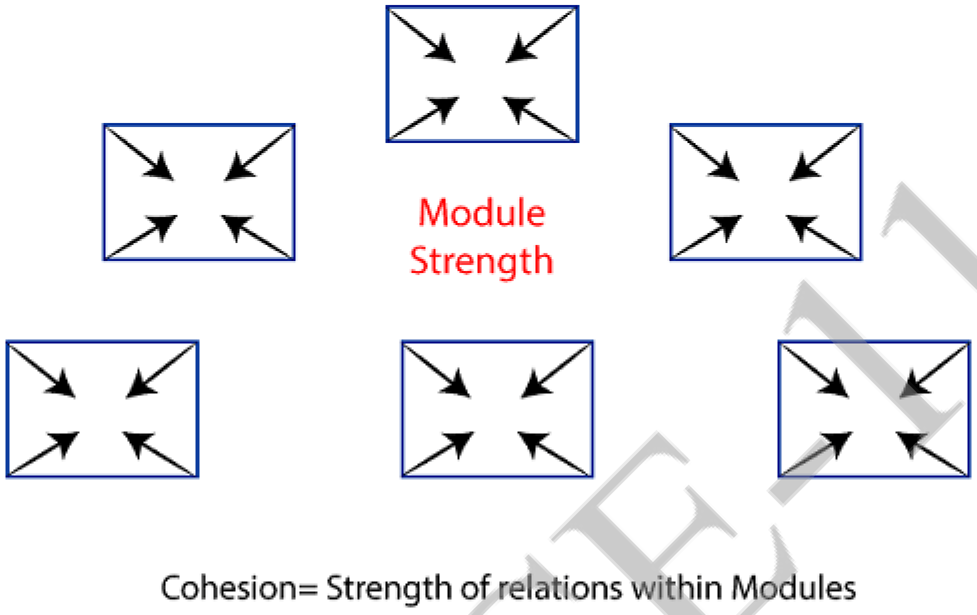


7. Content Coupling: Content Coupling exists among two modules if they **share code**, e.g., a branch from one module into another module.

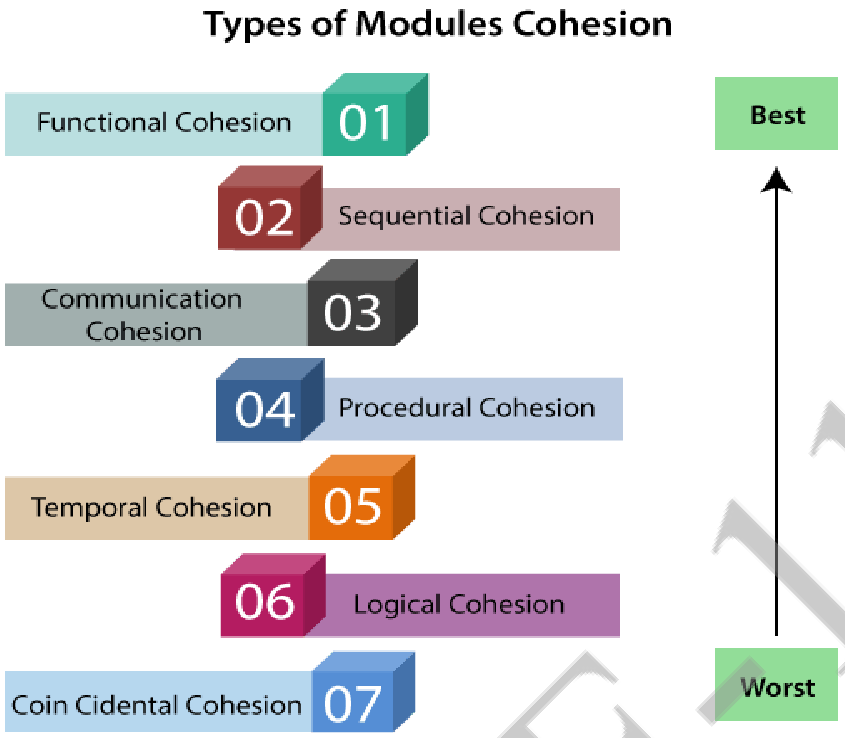
Module Cohesion

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." (Strength of relationship **within the module)**



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 cre**ating, 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**.

Software Design Patterns

Software design patterns **are communicating objects and classes** that are customized to solve a general design problem in a particular context. Software design patterns are general, r**eusable solutions to common problems** that arise during the design and development of software. They represent best practices for solving certain types of problems and provide a way for developers to communicate about effective design solutions. Design patterns capture expert knowledge and experience, making it easier for developers to create scalable, maintainable, and flexible software systems.

Software Design Patterns Tutorial

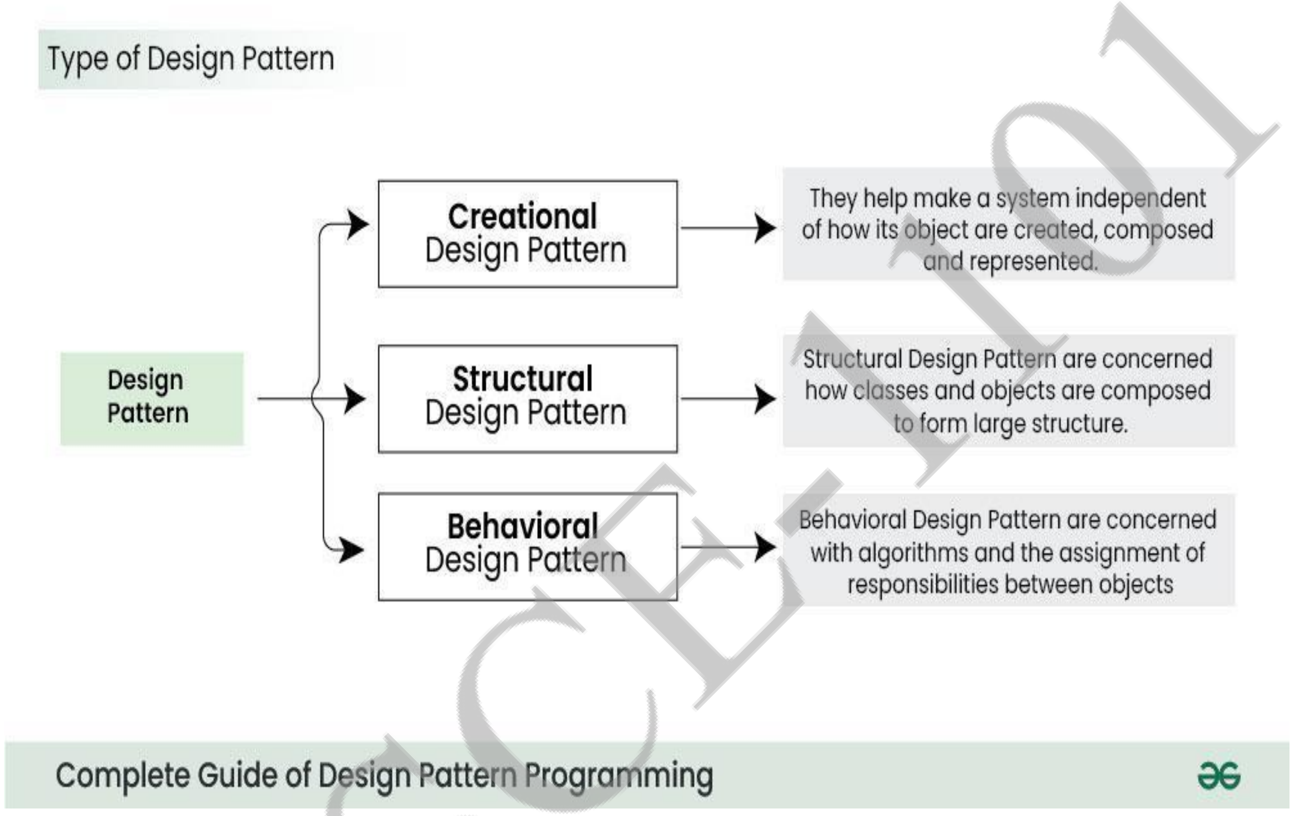
Types of Software Design Patterns

1. Creational Design Patterns
   - Factory Method Design Pattern
   - Abstract Factory Method Design Pattern
   - Singleton Method Design Pattern
   - Prototype Method Design Pattern
   - Builder Method Design Pattern
2. Structural Design Patterns
   - Adapter Method Design Patterns
   - Bridge Method Design Patterns
   - Composite Method Design Patterns
   - Decorator Method Design Patterns
   - Facade Method Design Patterns
   - Flyweight Method Design Patterns
   - Proxy Method Design Patterns
2. Behavioral Design Patterns
   - Chain Of Responsibility Method Design Pattern
   - Command Method Design Pattern
   - Interpreter Method Design Patterns
   - Mediator Method Design Pattern
   - Memento Method Design Patterns
   - Observer Method Design Pattern
   - State Method Design Pattern
   - Strategy Method Design Pattern
   - Template Method Design Pattern
   - Visitor Method Design Pattern

1. What are Design Patterns?

Design patterns are basically defined as r**eusable solutions to the common problems that arise during software design and development.** They are general templates or best practices that guide developers in creating well-structured, maintainable, and efficient code.

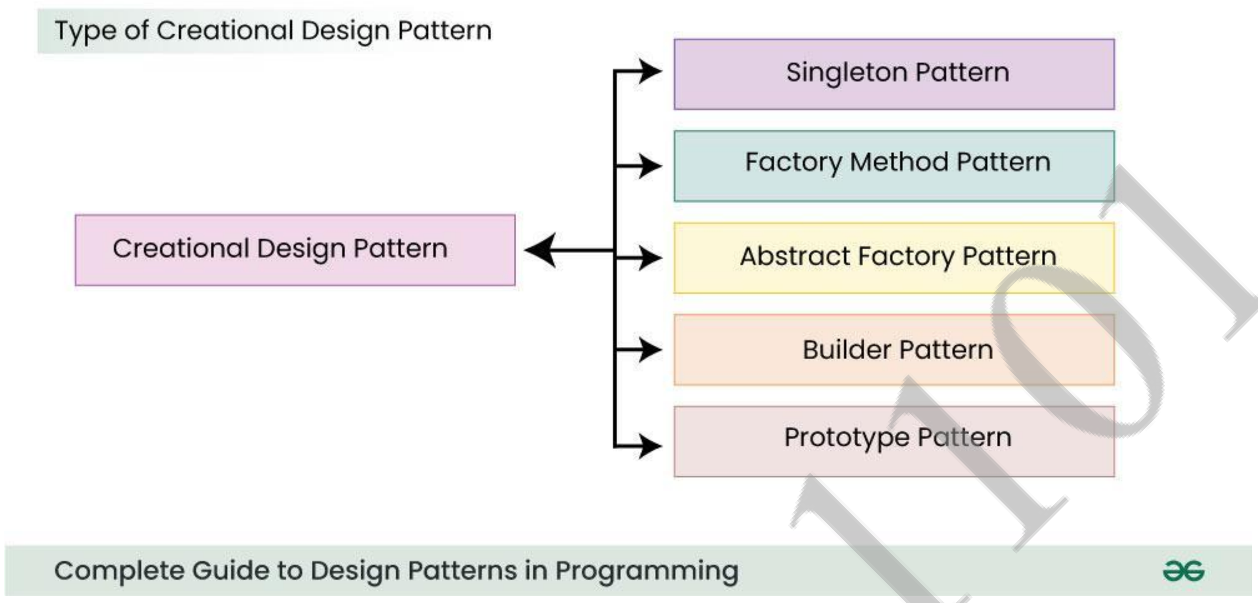
2. Types of Design Patterns


Type of Design Pattern
Design Pattern
Creational Design Pattern
They help make a system independent of how its object are created, composed and represented.
Structural Design Pattern
Structural Design Pattern are concerned how classes and objects are composed to form large structure.
Behavioral Design Pattern
Behavioral Design Pattern are concerned with algorithms and the assignment of responsibilities between objects
Complete Guide of Design Pattern Programming

Types of Design Patterns

Basically, there are **several types** of design patterns that are commonly used in software development. These patterns can be categorized into three main groups:

1. Creational Design Patterns

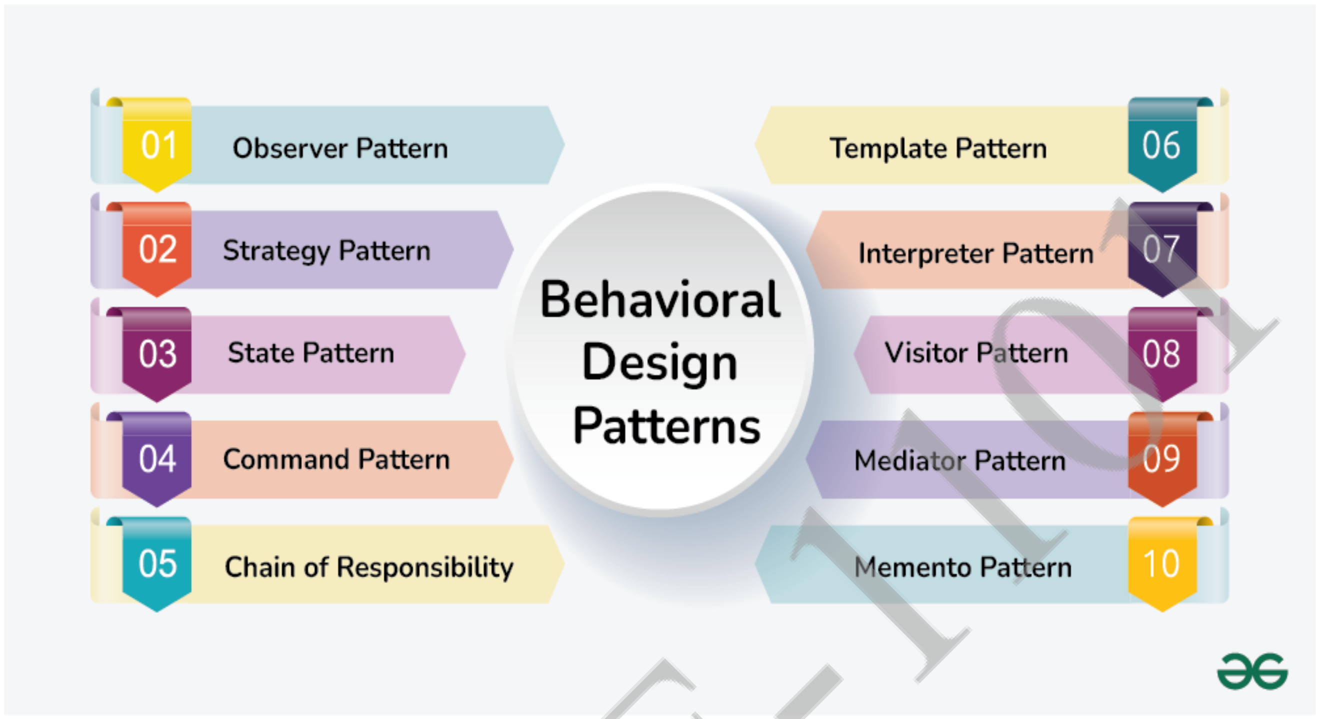

Type of Creational Design Pattern
Creational Design Pattern
Singleton Pattern
Factory Method Pattern
Abstract Factory Pattern
Builder Pattern
Prototype Pattern
Complete Guide to Design Patterns in Programming

- Singleton Pattern
    - The Singleton method or Singleton Design pattern is one of the simplest design patterns. It ensures a **class only has one instance,** and provides a global point of access to it.
- Factory Method Pattern
    - The Factory Method pattern is used to **create objects without specifying the exact class of object that will be created**. This pattern is useful when you need to decouple the creation of an object from its implementation.
- Abstract Factory Pattern
    - Abstract Factory pattern is almost s**imilar to Factory Pattern** and is considered as **another layer of abstraction over factory pattern**. Abstract Factory patterns work around a super-factory which creates other factories.
- Builder Pattern
    - Builder pattern aims to **"Separate the construction of a complex object from its representation** so that the same construction process can create different representations." It is used to construct a complex object step by step and the final step will return the object.
- Prototype Pattern
    - Prototype allows us to hide the complexity of making new instances from the client.
    - The concept is to copy an existing object rather than creating a new instance from scratch, something that may include **costly operations**. The existing object acts as a prototype and contains the state of the object.

2. Structural Design Patterns

- Adapter Pattern
    - The adapter pattern **converts the interface of a class into another interface clients expect.** Adapter lets classes work together that couldn't otherwise because of incompatible interfaces
- Bridge Pattern
    - The bridge pattern allows the **Abstraction and the Implementation to be developed independently and the client code** can access only the Abstraction part without being concerned about the Implementation part
- Composite Pattern
    - Composite pattern is a partitioning design pattern and describes a **group of objects that is treated** the same way as a single instance of the same type of object. The intent of a composite is to "compose" objects into tree structures to represent part-whole hierarchies.
- Decorator Pattern
    - It allows us to **dynamically add functionality and behavior to an object** without affecting the behavior of other existing objects within the same class.
    - We use inheritance to extend the behavior of the class. This takes place at compile-time, and all the instances of that class get the extended behavior.
- Facade Pattern
    - Facade Method Design Pattern provides a **unified interface** to a set of interfaces in a subsystem. Facade defines a high-level interface that makes the subsystem easier to use.
- Proxy Pattern
    - Proxy means 'in place of', representing' or 'in place of' or '**on behalf of**' are literal meanings of proxy and that directly explains Proxy Design Pattern.
    - Proxies are also called **surrogates, handles, and wrappers**. They are closely related in structure, but not purpose, to Adapters and Decorators.
- Flyweight Pattern
    - This pattern provides ways to **decrease object count** thus improving application required objects structure. Flyweight pattern is used when we need to create a large number of similar objects

3. **Behavioral Design Patterns**



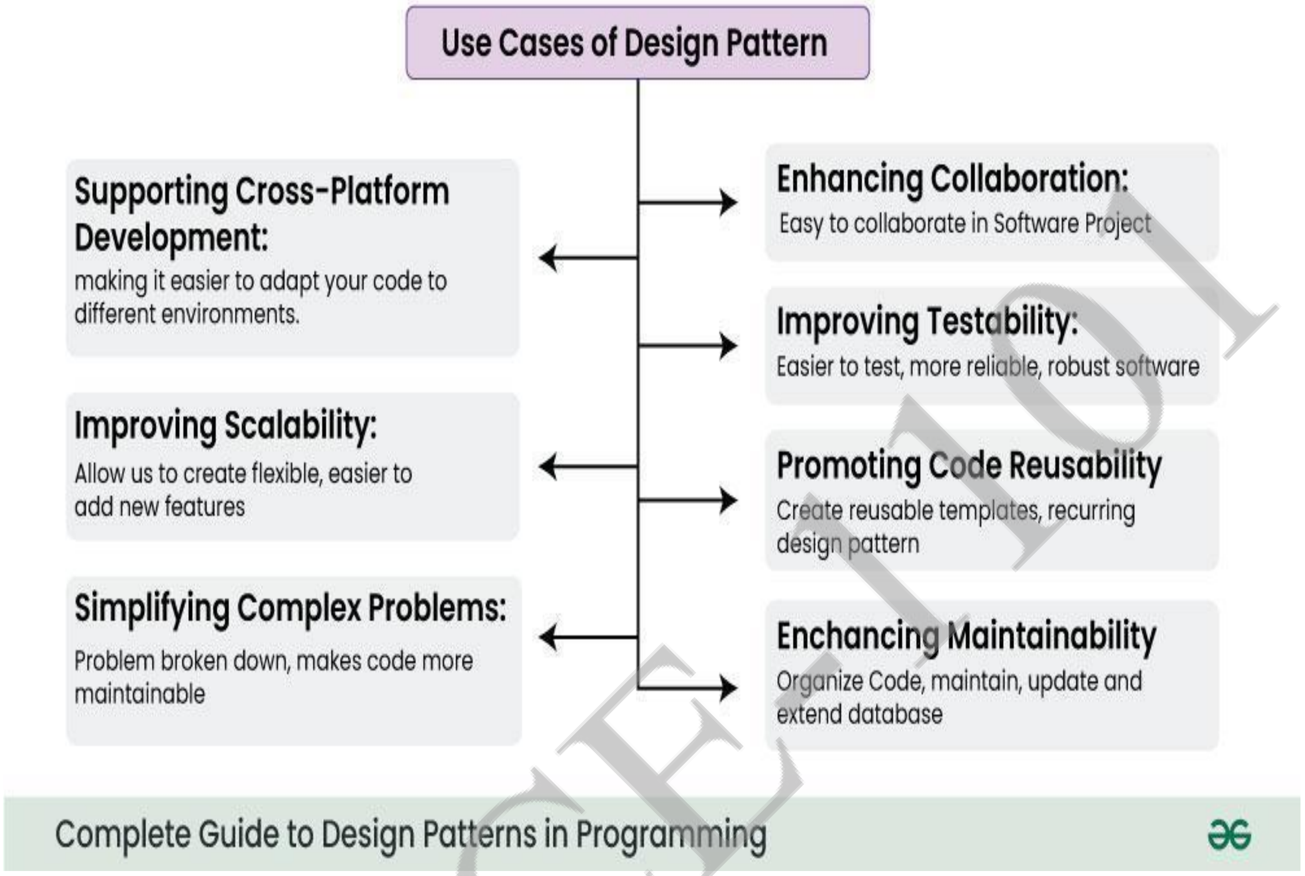
- Observer Pattern
    - It defines a **one-to-many dependency between objects**, so that when one object (the subject) changes its state, all its dependents (**observers**) are notified and updated automatically.

- Strategy Pattern
    - that allows the behavior of **an object to be selected at runtime**. It is one of the Gang of Four (GoF) design patterns, which are widely used in object-oriented programming.
    - The Strategy pattern is based on the **idea of encapsulating a family** of algorithms into separate classes that implement a common interface.
- Command Pattern
    - The Command Pattern is a behavioral design pattern that **turns a request into a stand-alone object**, containing all the information about the request. This object can be passed around, stored, and executed at a later time
- Chain of Responsibility Pattern
    - Chain of responsibility pattern is used t**o achieve loose coupling** in software design where a request from the client is passed to a chain of objects to process them.
    - Later, the object in the chain will decide themselves who will be processing the request and whether the request is required to be sent to the next object in the chain or not.
- State Pattern

- A state design pattern is used when an **Object changes its behavior based on its internal state**. If we have to change the behavior of an object based on its state, we can have a state variable in the Object and use the if-else condition block to perform different actions based on the state.

- Template Method Pattern
    - Template method design pattern is to define an and leave the details to be implemented by the child classe**algorithm as a skeleton of operations**s. The overall structure and sequence of the algorithm are preserved by the parent class.
- Visitor Pattern
    - It is used when we have to perform an **operation on a group of similar kind of Objects.** With the help of visitor pattern, we can move the operational logic from the objects to another class.
- Interpreter Pattern
    - Interpreter pattern is used to **defines a grammatical representation** for a language and provides an interpreter to deal with this grammar.
- Mediator Pattern
    - It enables decoupling of objects by introducing a layer in between so that the i**nteraction between objects** happen via the layer.
- Memento Pattern
    - It is used to r**estore the state of an object to a previous state**. As your application is progressing, you may want to save checkpoints in your application and restore back to those checkpoints later.
    - Intent of Memento Design pattern is without violating encapsulation, capture and externalize an object's internal state so that the object can be restored to this state later.

3. Use cases of Design Patterns


Use Cases of Design Pattern
Supporting Cross-Platform Development:
making it easier to adapt your code to different environments.
Improving Scalability:
Allow us to create flexible, easier to add new features
Simplifying Complex Problems:
Problem broken down, makes code more maintainable
Enhancing Collaboration:
Easy to collaborate in Software Project
Improving Testability:
Easier to test, more reliable, robust software
Promoting Code Reusability
Create reusable templates, recurring design pattern
Enchancing Maintainability
Organize Code, maintain, update and extend database
Complete Guide to Design Patterns in Programming

Use Cases of Design Pattern

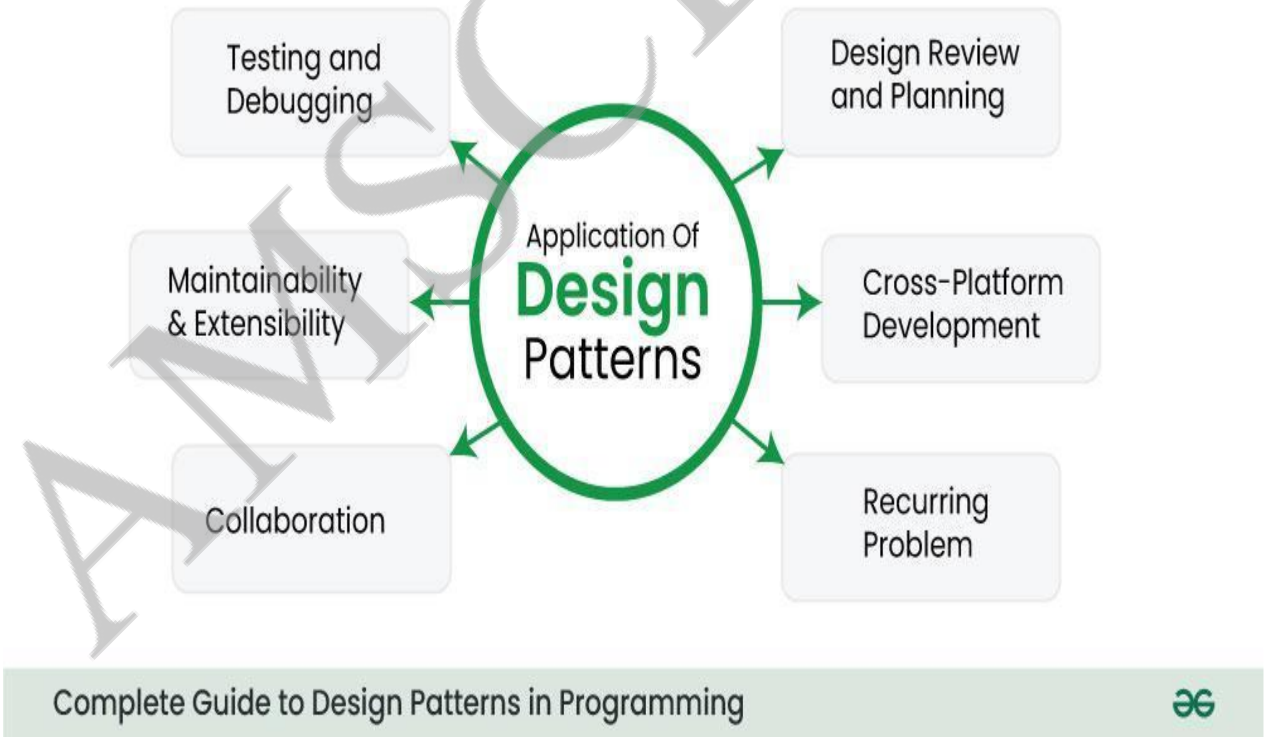
Design patterns are a valuable tool in software development, and they offer various benefits and uses, some of them are explained below :

- **Enhancing Maintainability**:
    - Design patterns help organize code in a structured and consistent way. This makes it easier to maintain, update, and extend the codebase. Developers familiar with the patterns can quickly understand and work on the code.
- **Promoting Code Reusability**:
    - Design patterns encapsulate solutions to recurring design problems. By using these patterns, we can create reusable templates for solving specific problems in different parts of your application.
- **Simplifying Complex Problems**:
    - Complex software problems can be broken down into smaller, more manageable components using design patterns. This simplifies development by addressing one problem at a time and, in turn, makes the code more maintainable.
- **Improving Scalability**:

    - Design patterns, particularly structural patterns, allow us to create a flexible and extensible architecture, making it easier to add new features or components.
- **Improving Testability**:
    - Code designed with patterns in mind is often more modular and easier to test. we can write unit tests for individual components or classes, leading to more reliable and robust software.
- **Supporting Cross-Platform Development**:
    - Design patterns are not tied to a specific programming language or platform. They are general guidelines that can be applied across different technologies, making it easier to adapt your code to different environments.
- **Enhancing Collaboration**:
    - Design patterns provide a common language and a shared understanding among team members. They enable developers to communicate effectively and collaborate on software projects by referring to well-known design solutions.

4. Applications of Design Patterns

Basically, design patterns should be used when they provide a clear and effective solution to a recurring problem in our software design. Here are some situations where we can use the design patterns.


Testing and Debugging
Design Review and Planning
Application Of Design Patterns
Maintainability & Extensibility
Cross-Platform Development
Collaboration
Recurring Problem
Complete Guide to Design Patterns in Programming

Application of Design Patterns

- **Collaboration:** Suppose if we are working in a team or on a project with multiple developers, so there design patterns can facilitate collaboration by providing a common language and shared understanding of how to address specific design challenges.
- **Recurring Problem:** Suppose if we encounter a design problem that we have seen in different forms in multiple projects or if it's a well-known and documented problem in software development, it's a good indicator that a design pattern might be useful.
- **Maintainability and Extensibility:** When we wants to create code that is easy to maintain, extend, and modify over time, design patterns can help by providing a structured approach to problem-solving.
- **Cross-Platform Development:** When we need to create code that works on different platforms or with various technologies, design patterns provide a platform-agnostic way to solve common problems.
- **Testing and Debugging:** Design patterns can make our code more modular and testable, leading to improved testing and debugging processes.
- **Design Review and Planning:** During the design and planning phase of a project, we can proactively consider the use of design patterns to solve anticipated design challenges.

It's important to note that design patterns are not a one-size-fits-all solution. They should be used judiciously, and not all problems require the application of a design pattern. Always consider the specific context and requirements of your project when deciding whether to use a design pattern.

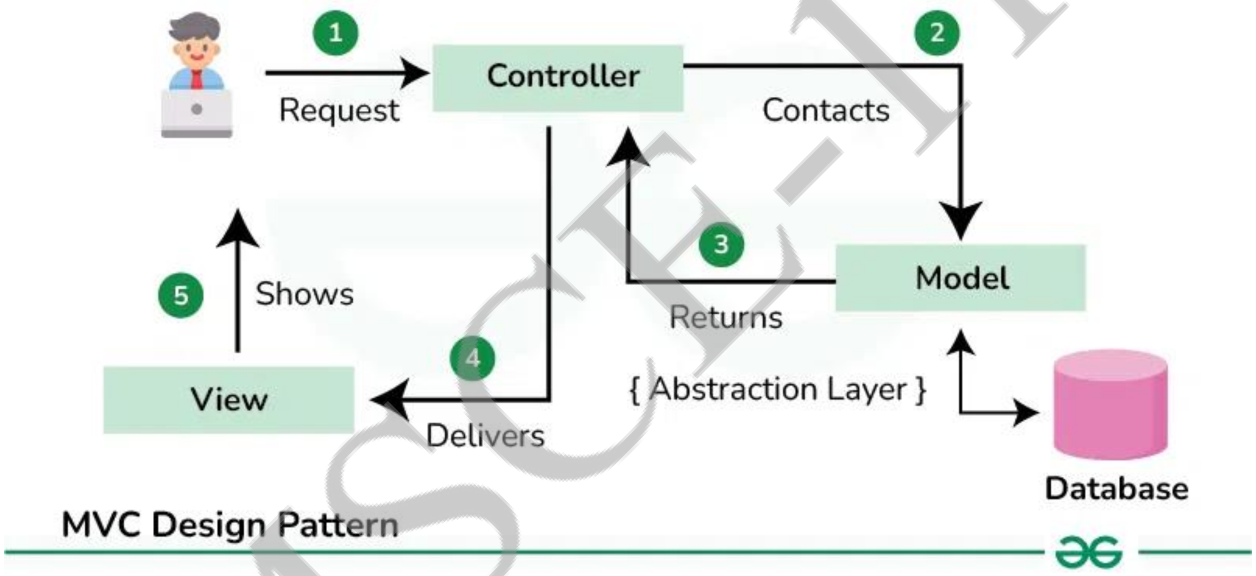
MVC Design Pattern

What is the MVC Design Pattern?

The **Model View Controller** (MVC) design pattern specifies that an application consists of a data model, presentation information, and control information. The pattern requires that each of these be separated into different objects.

- The MVC pattern separates the concerns of an application into three distinct components, each responsible for a specific aspect of the application's functionality.
- This separation of concerns makes the application easier to maintain and extend, as changes to one component do not require changes to the other components.

Components of the MVC Design Pattern



1. Model

The Model component in the MVC (Model-View-Controller) d**esign pattern represents the data and business logic of an application**. It is responsible for managing the application's data, processing business rules, and responding to requests for information from other components, such as the View and the Controller.

2. View

Displays the data from the Model to the user and sends user inputs to the Controller. It is passive and does not directly interact with the Model. Instead, it receives data from the Model and sends user inputs to the Controller for processing.

3. Controller

Controller acts as an **intermediary between the Model and the View**. It handles user input and updates the Model accordingly and updates the View to reflect changes in the Model. It contains application logic, such as input validation and data transformation.

Communication between the components

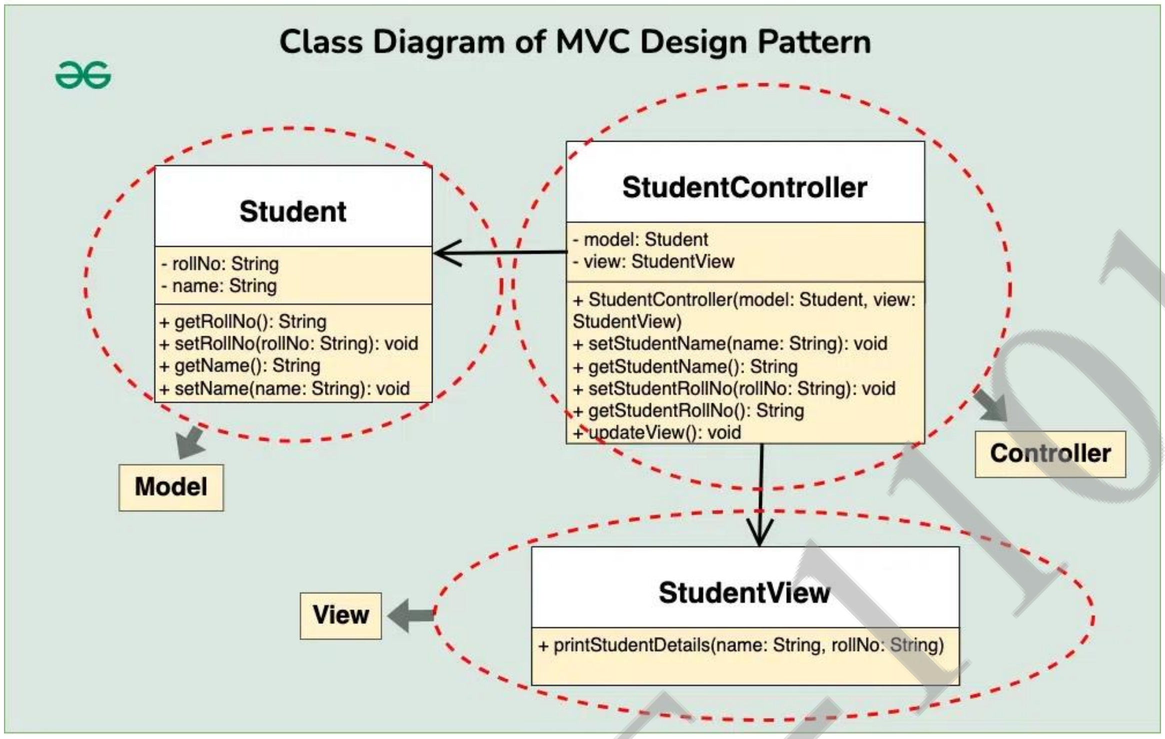
This below communication flow ensures that each component is responsible for a specific aspect of the application's functionality, leading to a more maintainable and scalable architecture

- **User Interaction with View:**
    - The user interacts with the View, such as clicking a button or entering text into a form.
- **View Receives User Input:**
    - The View receives the user input and forwards it to the Controller.
- **Controller Processes User Input:**
    - The Controller receives the user input from the View.
    - It interprets the input, performs any necessary operations (such as updating the Model), and decides how to respond.
- **Controller Updates Model:**
    - The Controller updates the Model based on the user input or application logic.
- **Model Notifies View of Changes:**
    - If the Model changes, it notifies the View.
- **View Requests Data from Model:**
    - The View requests data from the Model to update its display.
- **Controller Updates View:**
    - The Controller updates the View based on the changes in the Model or in response to user input.
- **View Renders Updated UI:**
    - The View renders the updated UI based on the changes made by the Controller.

Example of the MVC Design Pattern

Below is the code of above problem statement using MVC Design Pattern:
Let's break down into the component wise code:

1. Model (Student class)
Represents the data (student's name and roll number) and provides methods to access and modify this data.


Class Diagram of MVC Design Pattern
Student
- rollNo: String
- name: String
+ getRollNo(): String
+ setRollNo(rollNo: String): void
+ getName(): String
+ setName(name: String): void
StudentController
- model: Student
- view: StudentView
+ StudentController(model: Student, view: StudentView)
+ setStudentName(name: String): void
+ getStudentName(): String
+ setStudentRollNo(rollNo: String): void
+ getStudentRollNo(): String
+ updateView(): void
Model
Controller
View
StudentView
+ printStudentDetails(name: String, rollNo: String)

Advantages of the MVC Design Pattern

- **Separation of Concerns:** MVC separates the different aspects of an application **(data, UI, and logic)**, making the code easier to understand, maintain, and modify.
- **Modularity:** Each component (Model, View, Controller) can b**e developed and tested separately,** promoting code reusability and scalability.
- **Flexibility:** Since the **components are independent**, changes to one component do not affect the others, allowing for easier updates and modifications.
- **Parallel Development: Multiple developers** can work on different components simultaneously, speeding up the development process.
- **Code Reusability:** The components can be reused in other parts of the application or in different projects, reducing development time and effort.

Disadvantages of the MVC Design Pattern

- **Complexity:** Implementing the MVC pattern can add complexity to the code, **especially for simpler applications**, leading to overhead in development.
- **Learning Curve:** Developers **need to understand the concept** of MVC and how to implement it effectively, which may require additional time and resources.
- **Overhead:** The **communication** between components (Model, View, Controller) can lead to overhead, affecting the performance of the application, especially in resource-constrained environments.
- **Potential for Over-Engineering:** In some cases, **developers may over-engineer the application** by adding unnecessary abstractions and layers, leading to bloated and hard-to-maintain code.

- **Increased File Count:** MVC can result in a larger number of files and classes compared to simpler architectures, which may make the project structure more complex and harder to navigate.

ARCHITECTURAL STYLES

The architectural model or style is a pattern for **creating the system architecture for a given problem**. The software that is built for computer-based systems also exhibits many architectural styles.

Each style describes a system **category** that encompasses

(1) **A set of components**(e.g., a database, computational modules) that perform a function required by a system;

(2) **A set of connectors** that enable "communication, coordination and cooperation" among components;

(3) **Constraints** that define how components can be i**ntegrated to form the system;**

(4) **Semantic models** that enable a designer to understand the overall properties of a system by analyzing the known properties of its constituent parts. In the section that follows, we consider commonly used architectural patterns for software.

A Brief Taxonomy of Styles and Patterns

a)Data-centered architectures.

A data store (e.g., a file or database) resides at the center of this architecture and is accessed frequently by other components that update, add, delete, or otherwise modify data within the store. Figure (Data-centered architecture) illustrates typical data-centered style. Client software accesses a central repository.

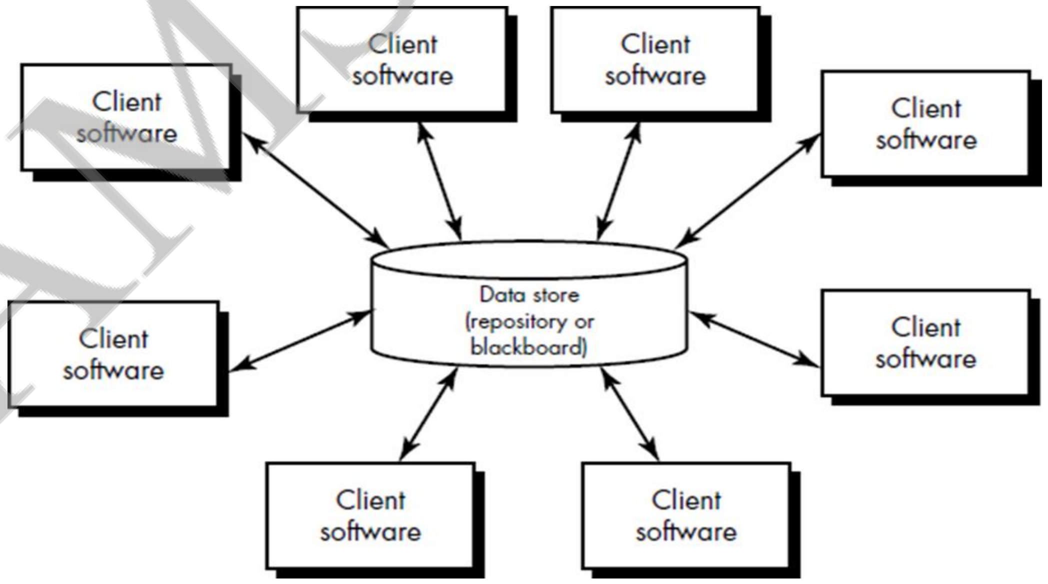


FIG Data-centered architecture

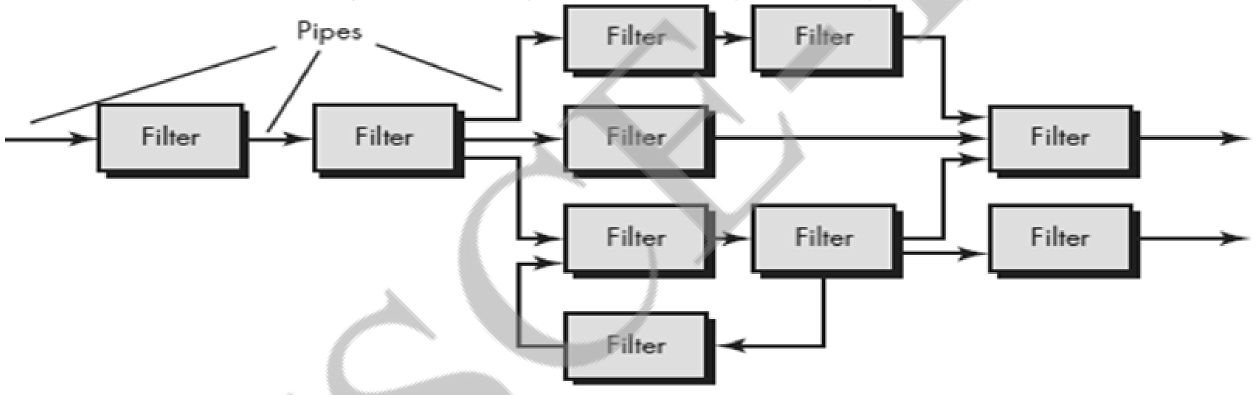
Data-centered architectures promote integrability. That is, **existing components can be changed and new client components can be added to the architecture without concern about other clients**

b) Data-flow architectures.

This architecture is applied when **input data is to be transformed through a series of computational or manipulative** components into output data.

A pipe and filter pattern has a set of components, called fi**lters, connected by pipes** that transmit data from one component to the next.

Each **filter works independently** of those components upstream and downstream, is designed to expect data input of a certain form, and produces data output (to the next filter) of a specified form. However, the filter does not require knowledge of the working of its neighboring filters.

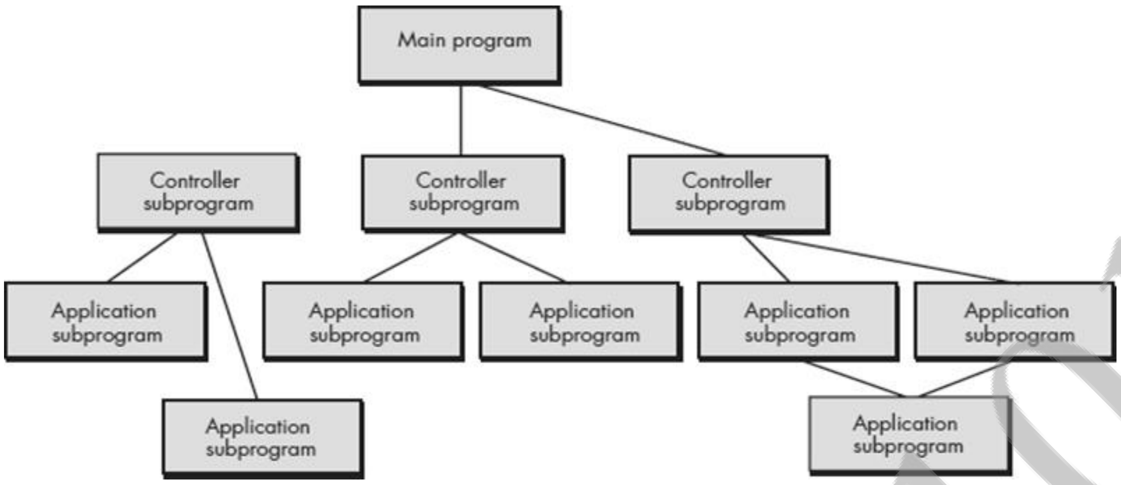


Pipes and filters

If the data flow degenerates into a single line of transforms, it is termed batch sequential. This pattern accepts a batch of data and then applies a series of sequential components (filters) to transform it

c)Call and return architectures.

This architectural style enables a software designer (system architect) t**o achieve a program structure that is relatively easy to modify and scale**. A number of sub styles exist within this category:



• **Main program/subprogram architectures.** This classic program structure decomposes function into a control hierarchy where a “main” program invokes a number of program components, which in turn may invoke still other components.

• **Remote procedure call architectures.** The components of main program/subprogram architecture are distributed across multiple computers on a network.

d)Object-oriented architectures.

The components of a system encapsulate data and the operations that must be applied to manipulate the data.
Communication and coordination between components is accomplished via message passing.

e) Layered architectures.

- The basic structure of a layered architecture is illustrated in Figure. A number of different layers are defined, each accomplishing operations that progressively become closer to the machine instruction set.
- At the outer layer, components service user interface operations.
- At the inner layer, components perform operating system interfacing.
- Intermediate layers provide utility services and application software functions.
- Once requirements engineering uncovers the characteristics and constraints of the system to be built, the architectural pattern (style) or combination of patterns (styles) that best fits those characteristics and constraints can be chosen.

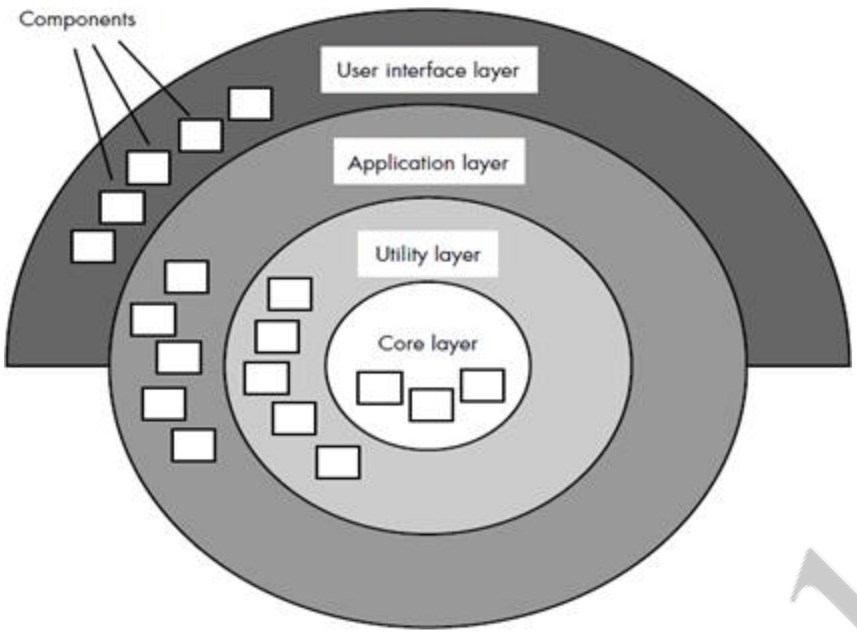


Figure Layered architecture

USER INTERFACE DESIGN

The Golden Rules

Mandel coins three golden rules:

1. Place the user in control.
2. Reduce the user's memory load.
3. Make the interface consistent.

These golden rules actually form the basis for a set of user interface design principles that guide this important aspect of software design.

1. Place the user in control.

Mandel defines a number of design principles that allow the user to maintain control:

Define interaction modes in a way that does not force a user into unnecessary or undesired actions. Provide for flexible interaction.

Allow user interaction to be interruptible and undoable.

Streamline interaction as skill levels advance and allow the interaction to be customized. Hide technical internals from the casual user.

Design for direct interaction with objects that appear on the screen.

2. Reduce the user's memory load.

Mandel defines design principles that enable an interface to reduce the user's memory load: Reduce demand on short-term memory.

Establish meaningful defaults.

Define shortcuts that are intuitive.

The visual layout of the interface should be based on a real-world metaphor. Disclose information in a progressive fashion.

3. Make the interface consistent.

The interface should present and acquire information in a consistent fashion. This implies that

(1) all visual information is organized according to design rules that are maintained throughout all screen displays,

(2) input mechanisms are constrained to a limited set that is used consistently throughout the application

(3) mechanisms for navigating from task to task are consistently defined and implemented. Mandel defines a set of design principles that help make the interface consistent:

Allow the user to put the current task into a meaningful context. Maintain consistency across a family of applications.

If past interactive models have created user expectations, do not make changes unless there is a compelling reason to do so.

UNIT IV SOFTWARE TESTING AND MAINTENANCE 9

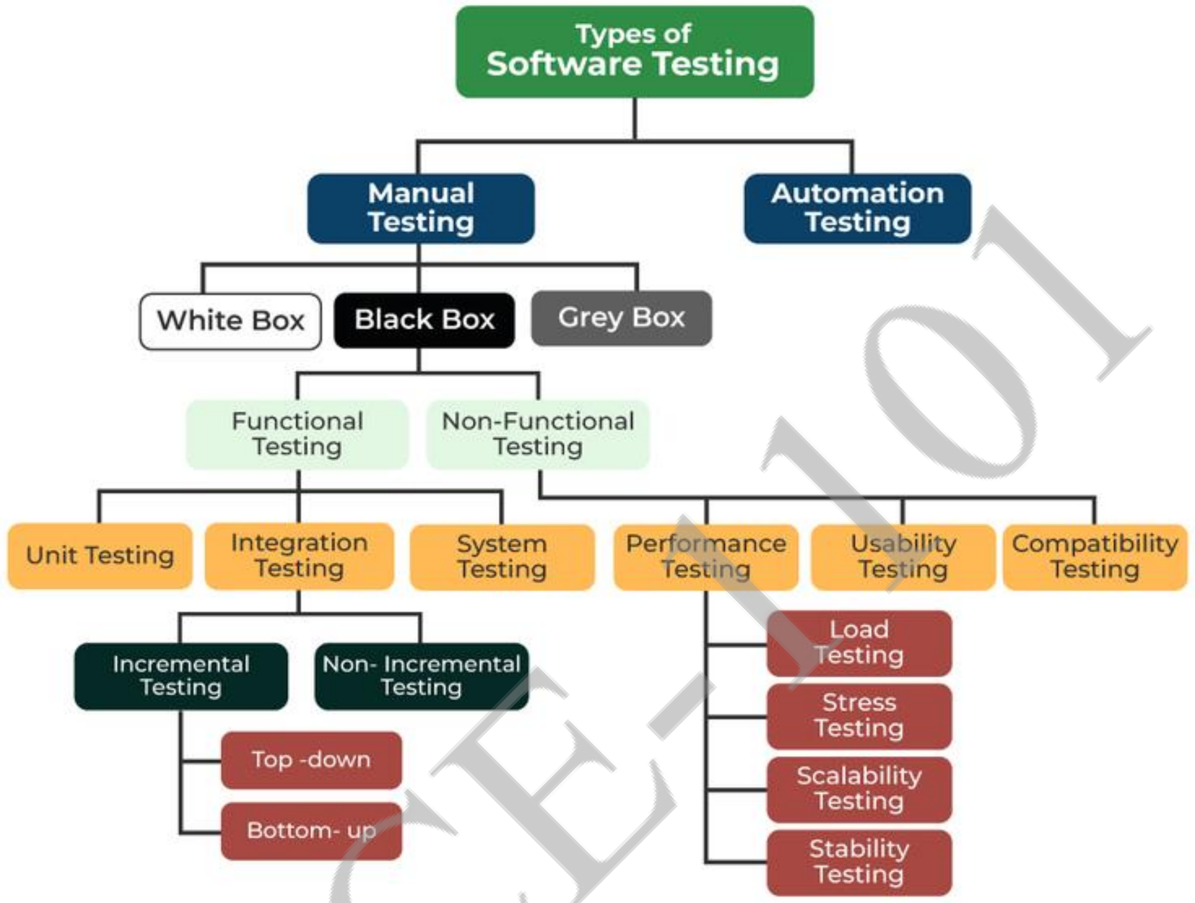
Testing – Unit testing – Black box testing– White box testing – Integration and System testing– Regression testing – Debugging - Program analysis – Symbolic execution – Model Checking-Case Study

Types of Software testing

Testing is the process of executing a program to find errors. To make our software perform well it should be error-free. If testing is done successfully it will remove all the errors from the software. In this article, we will discuss first the principles of testing and then we will discuss, the different types of testing.

Principles of Testing

- All the tests should meet the customer's requirements.
- To make our software testing should be performed by a third party.
- Exhaustive testing is not possible. As we need the optimal amount of testing based on the risk assessment of the application.
- All the tests to be conducted should be planned before implementing it
- It follows the Pareto rule(80/20 rule) which states that 80% of errors come from 20% of program components.
- Start testing with small parts and extend it to large parts.
- Types of Testing


Types of Software Testing
Manual Testing
Automation Testing
White Box
Black Box
Grey Box
Functional Testing
Non-Functional Testing
Unit Testing
Integration Testing
System Testing
Performance Testing
Usability Testing
Compatibility Testing
Incremental Testing
Non- Incremental Testing
Top -down
Bottom- up
Load Testing
Stress Testing
Scalability Testing
Stability Testing

Unit Testing – Software Testing

Unit testing is a type of software testing that focuses on individual units or components of a software system. The purpose of unit testing is to validate that each unit of the software works as intended and meets the requirements. Unit testing is typically performed by developers, and it is performed early in the development process before the code is integrated and tested as a whole system.

Unit tests are automated and are run each time the code is changed to ensure that new code does not break existing functionality. Unit tests are designed to validate the smallest possible unit of code, such as a function or a method, and test it in isolation from the rest of the system. This allows developers to quickly identify and fix any issues early in the development process, improving the overall quality of the software and reducing the time required for later testing.

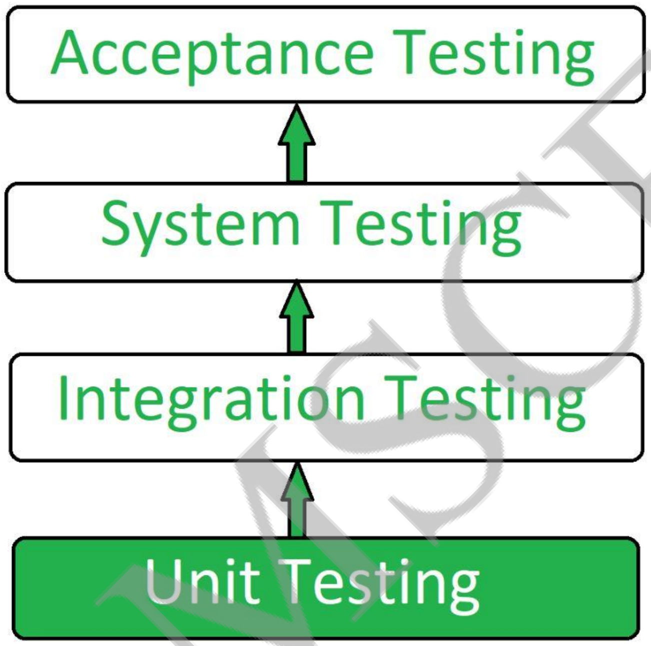
Prerequisite – Types of Software Testing

Unit Testing is a software testing technique using which individual units of software i.e. group of computer program modules, usage procedures, and operating procedures are tested to determine whether they are suitable for use or not. It is a testing method using which every independent module is tested to determine if there is an issue by the developer himself. It is correlated with the functional correctness of the independent modules. Unit Testing is defined as a type of software testing where individual components of a software are tested. Unit Testing of the software product is carried out during the development of an application. An individual component may be either an individual function or a procedure. Unit Testing is typically performed by the developer. In SDLC or V Model, Unit testing is the first level of testing done before integration testing. Unit testing is a type of testing technique that is usually performed by developers. Although due to the reluctance of developers to test, quality assurance engineers also do unit testing.

Objective of Unit Testing:

The objective of Unit Testing is:

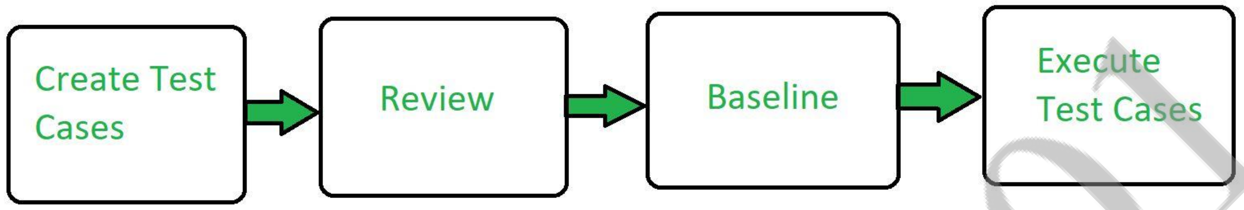
1. To isolate a section of code.
2. To verify the correctness of the code.
3. To test every function and procedure.
4. To fix bugs early in the development cycle and to save costs.
5. To help the developers understand the code base and enable them to make changes quickly.
6. To help with code reuse.



Types of Unit Testing:

There are 2 types of Unit Testing: Manual, and Automated.

Workflow of Unit Testing:



Unit Testing Techniques:

There are 3 types of Unit Testing Techniques. They are

1. Black Box Testing: This testing technique is used in covering the unit tests for input, user interface, and output parts.
2. White Box Testing: This technique is used in testing the functional behavior of the system by giving the input and checking the functionality output including the internal design structure and code of the modules.
3. Gray Box Testing: This technique is used in executing the relevant test cases, test methods, and test functions, and analyzing the code performance for the modules.

Unit Testing Tools:

Here are some commonly used Unit Testing tools:

1. Jtest
2. Junit
3. NUnit
4. EMMA
5. PHPUnit

6.

Advantages of Unit Testing:

1. Unit Testing allows developers to learn what functionality is provided by a unit and how to use it to gain a basic understanding of the unit API.
2. Unit testing allows the programmer to refine code and make sure the module works properly.
3. Unit testing enables testing parts of the project without waiting for others to be completed.
4. Early Detection of Issues: Unit testing allows developers to detect and fix issues early in the development process before they become larger and more difficult to fix.
5. Improved Code Quality: Unit testing helps to ensure that each unit of code works as intended and meets the requirements, improving the overall quality of the software.
6. Increased Confidence: Unit testing provides developers with confidence in their code, as they can validate that each unit of the software is functioning as expected.
7. Faster Development: Unit testing enables developers to work faster and more efficiently, as they can validate changes to the code without having to wait for the full system to be tested.
8. Better Documentation: Unit testing provides clear and concise documentation of the code and its behavior, making it easier for other developers to understand and maintain the software.
9. Facilitation of Refactoring: Unit testing enables developers to safely make changes to the code, as they can validate that their changes do not break existing functionality.
10. Reduced Time and Cost: Unit testing can reduce the time and cost required for later testing, as it helps to identify and fix issues early in the development process.

Disadvantages of Unit Testing:

1. The process is time-consuming for writing the unit test cases.
2. Unit Testing will not cover all the errors in the module because there is a chance of having errors in the modules while doing integration testing.

3. Unit Testing is not efficient for checking the errors in the UI(User Interface) part of the module.
4. It requires more time for maintenance when the source code is changed frequently.
5. It cannot cover the non-functional testing parameters such as scalability, the performance of the system, etc.
6. Time and Effort: Unit testing requires a significant investment of time and effort to create and maintain the test cases, especially for complex systems.
7. Dependence on Developers: The success of unit testing depends on the developers, who must write clear, concise, and comprehensive test cases to validate the code.
8. Difficulty in Testing Complex Units: Unit testing can be challenging when dealing with complex units, as it can be difficult to isolate and test individual units in isolation from the rest of the system.
9. Difficulty in Testing Interactions: Unit testing may not be sufficient for testing interactions between units, as it only focuses on individual units.
10. Difficulty in Testing User Interfaces: Unit testing may not be suitable for testing user interfaces, as it typically focuses on the functionality of individual units.
11. Over-reliance on Automation: Over-reliance on automated unit tests can lead to a false sense of security, as automated tests may not uncover all possible issues or bugs.
12. Maintenance Overhead: Unit testing requires ongoing maintenance and updates, as the code and test cases must be kept up-to-date with changes to the software.

Integration Testing

Integration testing is the process of testing the interface between two software units or modules. It focuses on determining the correctness of the interface. The purpose of integration testing is to expose faults in the interaction between integrated units. Once all the modules have been unit-tested, integration testing is performed.

Integration testing

It is a software testing technique that focuses on verifying the interactions and data exchange between different components or modules of a software application. The goal of integration testing is to identify any problems or bugs that arise when different components are combined and interact with each other. Integration testing is typically performed after unit testing and before system testing. It helps to identify and resolve integration issues early in the development cycle, reducing the risk of more severe and costly problems later on.

Integration testing can be done by picking module by module. This can be done so that there should be a proper sequence to be followed. And also if you don't want to miss out on any integration scenarios then you have to follow the proper sequence. Exposing the defects is the major focus of the integration testing and the time of interaction between the integrated units. Integration test approaches – There are four types of integration testing approaches. Those approaches are the following:

1. Big-Bang Integration Testing – It is the simplest integration testing approach, where all the modules are combined and the functionality is verified after the completion of individual module testing. In simple words, all the modules of the system are simply put together and tested. This approach is practicable only for very small systems. If an error is found during the integration testing, it is very difficult to localize the error as the error may potentially belong to any of the modules being integrated. So, debugging errors reported during Big Bang integration testing is very expensive to fix.

Big-bang integration testing is a software testing approach in which all components or modules of a software application are combined and tested at once. This approach is typically used when the software components have a low degree of interdependence or when there are constraints in the development environment that prevent testing individual components. The goal of big-bang integration testing is to verify the overall functionality of the system and to identify any integration problems that arise when the components are combined. While big-bang integration testing can be useful in some situations, it can also be a high-risk approach, as the complexity of the system and the number of interactions between components can make it difficult to identify and diagnose problems.

Advantages:

1. It is convenient for small systems.
2. Simple and straightforward approach.
3. Can be completed quickly.
4. Does not require a lot of planning or coordination.
5. May be suitable for small systems or projects with a low degree of interdependence between components.

Disadvantages:

1. There will be quite a lot of delay because you would have to wait for all the modules to be integrated.
2. High-risk critical modules are not isolated and tested on priority since all modules are tested at once.
3. Not Good for long projects.
4. High risk of integration problems that are difficult to identify and diagnose.
5. This can result in long and complex debugging and troubleshooting efforts.
6. This can lead to system downtime and increased development costs.
7. May not provide enough visibility into the interactions and data exchange between components.
8. This can result in a lack of confidence in the system's stability and reliability.

9. This can lead to decreased efficiency and productivity.
10. This may result in a lack of confidence in the development team.
11. This can lead to system failure and decreased user satisfaction.

2. Bottom-Up Integration Testing – In bottom-up testing, each module at lower levels are tested with higher modules until all modules are tested. The primary purpose of this integration testing is that each subsystem tests the interfaces among various modules making up the subsystem. This integration testing uses test drivers to drive and pass appropriate data to the lower-level modules.

Advantages:

- In bottom-up testing, no stubs are required.
- A principal advantage of this integration testing is that several disjoint subsystems can be tested simultaneously.
- It is easy to create the test conditions.
- Best for applications that uses bottom up design approach.
- It is Easy to observe the test results.

Disadvantages:

- Driver modules must be produced.
- In this testing, the complexity that occurs when the system is made up of a large number of small subsystems.
- As Far modules have been created, there is no working model can be represented.

3. Top-Down Integration Testing – Top-down integration testing technique is used in order to simulate the behaviour of the lower-level modules that are not yet integrated. In this integration testing, testing takes place from top to bottom. First, high-level modules are tested and then low-level modules and finally integrating the low-level modules to a high level to ensure the system is working as intended.

Advantages:

- Separately debugged module.
- Few or no drivers needed.
- It is more stable and accurate at the aggregate level.
- Easier isolation of interface errors.
- In this, design defects can be found in the early stages.

Disadvantages:

- Needs many Stubs.
- Modules at lower level are tested inadequately.
- It is difficult to observe the test output.
- It is difficult to stub design.

4. Mixed Integration Testing – A mixed integration testing is also called sandwiched integration testing. A mixed integration testing follows a combination of top down and bottom-up testing approaches. In top-down approach, testing can start only after the top-level module have been coded and unit tested. In bottom-up approach, testing can start only after the bottom level modules are ready. This sandwich or mixed approach overcomes this shortcoming of the top-down and bottom-up approaches. It is also called the hybrid integration testing. also, stubs and drivers are used in mixed integration testing.

Advantages:

- Mixed approach is useful for very large projects having several sub projects.
- This Sandwich approach overcomes this shortcoming of the top-down and bottom-up approaches.
- Parallel test can be performed in top and bottom layer tests.

Disadvantages:

- For mixed integration testing, it requires very high cost because one part has a Top-down approach while another part has a bottom-up approach.

- This integration testing cannot be used for smaller systems with huge interdependence between different modules.

Applications:

1. Identify the components: Identify the individual components of your application that need to be integrated. This could include the frontend, backend, database, and any third-party services.
2. Create a test plan: Develop a test plan that outlines the scenarios and test cases that need to be executed to validate the integration points between the different components. This could include testing data flow, communication protocols, and error handling.
3. Set up test environment: Set up a test environment that mirrors the production environment as closely as possible. This will help ensure that the results of your integration tests are accurate and reliable.
4. Execute the tests: Execute the tests outlined in your test plan, starting with the most critical and complex scenarios. Be sure to log any defects or issues that you encounter during testing.
5. Analyze the results: Analyze the results of your integration tests to identify any defects or issues that need to be addressed. This may involve working with developers to fix bugs or make changes to the application architecture.
6. Repeat testing: Once defects have been fixed, repeat the integration testing process to ensure that the changes have been successful and that the application still works as expected.

Integration Testing

Integration testing is the process of testing the interface between two software units or modules. It focuses on determining the correctness of the interface. The purpose of integration testing is to expose faults in the interaction between integrated units. Once all the modules have been unit-tested, integration testing is performed.

Integration testing is a software testing technique that focuses on verifying the interactions and data exchange between different components or modules of a software application. The goal of integration testing is to identify any problems or bugs that arise when different components are combined and interact with each other. Integration testing is typically performed after unit testing and before system testing. It helps to identify and resolve integration issues early in the development cycle, reducing the risk of more severe and costly problems later on.

Integration testing can be done by picking module by module. This can be done so that there should be a proper sequence to be followed. And also if you don't want to miss out on any integration scenarios then you have to follow the proper sequence. Exposing the defects is the major focus of the integration testing and the time of interaction between the integrated units. Integration test approaches – There are four types of integration testing approaches. Those approaches are the following:

Big-Bang Integration Testing – It is the simplest integration testing approach, where all the modules are combined and the functionality is verified after the completion of individual module testing. In simple words, all the modules of the system are simply put together and tested. This approach is practicable only for very small systems. If an error is found during the integration testing, it is very difficult to localize the error as the error may potentially belong to any of the modules being integrated. So, debugging errors reported during Big Bang integration testing is very expensive to fix.

Big-bang integration testing is a software testing approach in which all components or modules of a software application are combined and tested at once. This approach is typically used when the software components have a low degree of interdependence or when there are constraints in the development environment that prevent testing individual components. The goal of big-bang integration testing is to verify the overall functionality of the system and to identify any integration problems that arise when the components are combined. While big-bang integration testing can be useful in some situations, it can also be a high-risk approach, as the complexity of the system and the number of interactions between components can make it difficult to identify and diagnose problems.

Advantages:

1. It is convenient for small systems.

2. Simple and straightforward approach.
3. Can be completed quickly.
4. Does not require a lot of planning or coordination.
5. May be suitable for small systems or projects with a low degree of interdependence between components.

Disadvantages:

1. There will be quite a lot of delay because you would have to wait for all the modules to be integrated.
2. High-risk critical modules are not isolated and tested on priority since all modules are tested at once.
3. Not Good for long projects.
4. High risk of integration problems that are difficult to identify and diagnose.
5. This can result in long and complex debugging and troubleshooting efforts.
6. This can lead to system downtime and increased development costs.
7. May not provide enough visibility into the interactions and data exchange between components.
8. This can result in a lack of confidence in the system's stability and reliability.
9. This can lead to decreased efficiency and productivity.
10. This may result in a lack of confidence in the development team.
11. This can lead to system failure and decreased user satisfaction.

2. Bottom-Up Integration Testing – In bottom-up testing, each module at lower levels are tested with higher modules until all modules are tested. The primary purpose of this integration testing is that each subsystem tests the interfaces among various modules making up the subsystem. This integration testing uses test drivers to drive and pass appropriate data to the lower-level modules.

Advantages:

- In bottom-up testing, no stubs are required.
- A principal advantage of this integration testing is that several disjoint subsystems can be tested simultaneously.
- It is easy to create the test conditions.
- Best for applications that uses bottom up design approach.
- It is Easy to observe the test results.

Disadvantages:

- Driver modules must be produced.
- In this testing, the complexity that occurs when the system is made up of a large number of small subsystems.
- As Far modules have been created, there is no working model can be represented.

3. Top-Down Integration Testing – Top-down integration testing technique is used in order to simulate the behaviour of the lower-level modules that are not yet integrated. In this integration testing, testing takes place from top to bottom. First, high-level modules are tested and then low-level modules and finally integrating the low-level modules to a high level to ensure the system is working as intended.

Advantages:

- Separately debugged module.
- Few or no drivers needed.
- It is more stable and accurate at the aggregate level.
- Easier isolation of interface errors.
- In this, design defects can be found in the early stages.

Disadvantages:

- Needs many Stubs.
- Modules at lower level are tested inadequately.

- It is difficult to observe the test output.
- It is difficult to stub design.

4. Mixed Integration Testing – A mixed integration testing is also called sandwiched integration testing. A mixed integration testing follows a combination of top down and bottom-up testing approaches. In top-down approach, testing can start only after the top-level module have been coded and unit tested. In bottom-up approach, testing can start only after the bottom level modules are ready. This sandwich or mixed approach overcomes this shortcoming of the top-down and bottom-up approaches. It is also called the hybrid integration testing. also, stubs and drivers are used in mixed integration testing.

Advantages:

- Mixed approach is useful for very large projects having several sub projects.
- This Sandwich approach overcomes this shortcoming of the top-down and bottom-up approaches.
- Parallel test can be performed in top and bottom layer tests.

Disadvantages:

- For mixed integration testing, it requires very high cost because one part has a Top-down approach while another part has a bottom-up approach.
- This integration testing cannot be used for smaller systems with huge interdependence between different modules.

Applications:

1. Identify the components: Identify the individual components of your application that need to be integrated. This could include the frontend, backend, database, and any third-party services.
2. Create a test plan: Develop a test plan that outlines the scenarios and test cases that need to be executed to validate the integration points between the different

components. This could include testing data flow, communication protocols, and error handling.

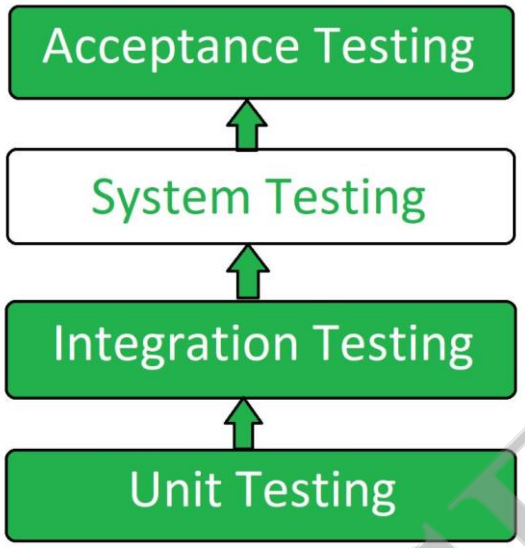
3. Set up test environment: Set up a test environment that mirrors the production environment as closely as possible. This will help ensure that the results of your integration tests are accurate and reliable.
4. Execute the tests: Execute the tests outlined in your test plan, starting with the most critical and complex scenarios. Be sure to log any defects or issues that you encounter during testing.
5. Analyze the results: Analyze the results of your integration tests to identify any defects or issues that need to be addressed. This may involve working with developers to fix bugs or make changes to the application architecture.
6. Repeat testing: Once defects have been fixed, repeat the integration testing process to ensure that the changes have been successful and that the application still works as expected.

System Testing

NTRODUCTION:

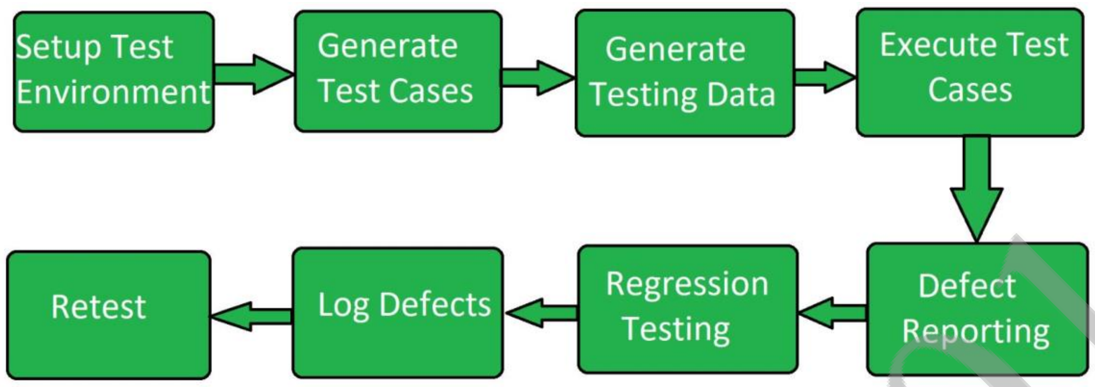
System testing is a type of software testing that evaluates the overall functionality and performance of a complete and fully integrated software solution. It tests if the system meets the specified requirements and if it is suitable for delivery to the end-users. This type of testing is performed after the integration testing and before the acceptance testing.

System Testing is a type of software testing that is performed on a complete integrated system to evaluate the compliance of the system with the corresponding requirements. In system testing, integration testing passed components are taken as input. The goal of integration testing is to detect any irregularity between the units that are integrated together. System testing detects defects within both the integrated units and the whole system. The result of system testing is the observed behavior of a component or a system when it is tested. System Testing is carried out on the whole system in the context of either system requirement specifications or functional requirement specifications or in the context of both. System testing tests the design and behavior of the system and also the expectations of the customer. It is performed to test the system beyond the bounds mentioned in the software requirements specification (SRS). System Testing is basically performed by a testing team that is independent of the development team that helps to test the quality of the system impartial. It has both functional and non-functional testing. System Testing is a black-box testing. System Testing is performed after the integration testing and before the acceptance testing.



System Testing Process: System Testing is performed in the following steps:

- Test Environment Setup: Create testing environment for the better quality testing.
- Create Test Case: Generate test case for the testing process.
- Create Test Data: Generate the data that is to be tested.
- Execute Test Case: After the generation of the test case and the test data, test cases are executed.
- Defect Reporting: Defects in the system are detected.
- Regression Testing: It is carried out to test the side effects of the testing process.
- Log Defects: Defects are fixed in this step.
- Retest: If the test is not successful then again test is performed.



Types of System Testing:

- Performance Testing: Performance Testing is a type of software testing that is carried out to test the speed, scalability, stability and reliability of the software product or application.
- Load Testing: Load Testing is a type of software Testing which is carried out to determine the behavior of a system or software product under extreme load.
- Stress Testing: Stress Testing is a type of software testing performed to check the robustness of the system under the varying loads.
- Scalability Testing: Scalability Testing is a type of software testing which is carried out to check the performance of a software application or system in terms of its capability to scale up or scale down the number of user request load.

Tools used for System Testing :

1. JMeter
2. Gallen Framework
3. Selenium

Here are a few common tools used for System Testing:

1. HP Quality Center/ALM
2. IBM Rational Quality Manager
3. Microsoft Test Manager

4. Selenium
5. Appium
6. LoadRunner
7. Gatling
8. JMeter
9. Apache JServ
10. SoapUI

    Note: The choice of tool depends on various factors like the technology used, the size of the project, the budget, and the testing requirements.

Advantages of System Testing :

- The testers do not require more knowledge of programming to carry out this testing.
- It will test the entire product or software so that we will easily detect the errors or defects which cannot be identified during the unit testing and integration testing.
- The testing environment is similar to that of the real time production or business environment.
- It checks the entire functionality of the system with different test scripts and also it covers the technical and business requirements of clients.
- After this testing, the product will almost cover all the possible bugs or errors and hence the development team will confidently go ahead with acceptance testing.

re are some advantages of System Testing:

- Verifies the overall functionality of the system.
- Detects and identifies system-level problems early in the development cycle.
- Helps to validate the requirements and ensure the system meets the user needs.
- Improves system reliability and quality.
- Facilitates collaboration and communication between development and testing teams.
- Enhances the overall performance of the system.

- Increases user confidence and reduces risks.
- Facilitates early detection and resolution of bugs and defects.
- Supports the identification of system-level dependencies and inter-module interactions.
- Improves the system’s maintainability and scalability.

Disadvantages of System Testing :

- This testing is time consuming process than another testing techniques since it checks the entire product or software.
- The cost for the testing will be high since it covers the testing of entire software.
- It needs good debugging tool otherwise the hidden errors will not be found.

Here are some disadvantages of System Testing:

- Can be time-consuming and expensive.
- Requires adequate resources and infrastructure.
- Can be complex and challenging, especially for large and complex systems.
- Dependent on the quality of requirements and design documents.
- Limited visibility into the internal workings of the system.
- Can be impacted by external factors like hardware and network configurations.
- Requires proper planning, coordination, and execution.
- Can be impacted by changes made during development.
- Requires specialized skills and expertise.
- May require multiple test cycles to achieve desired results.

Regression Testing

Regression Testing is the process of testing the modified parts of the code and the parts that might get affected due to the modifications to ensure that no new errors have been introduced in the software after the modifications have been made. Regression means the return of

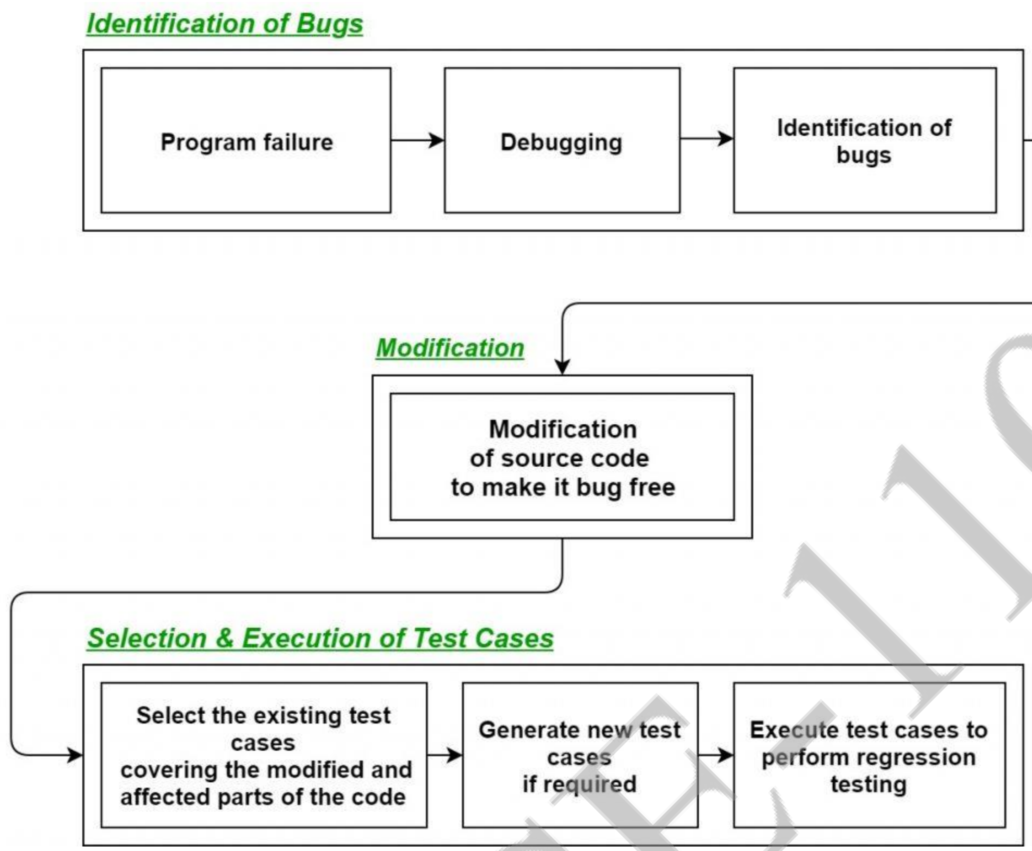
something and in the software field, it refers to the return of a bug.

When to do regression testing?

- When new functionality is added to the system and the code has been modified to absorb and integrate that functionality with the existing code.
- When some defect has been identified in the software and the code is debugged to fix it.
- When the code is modified to optimize its working.

Process of Regression testing

Firstly, whenever we make some changes to the source code for any reason like adding new functionality, optimization, etc. then our program when executed fails in the previously designed test suite for obvious reasons. After the failure, the source code is debugged to identify the bugs in the program. After identification of the bugs in the source code, appropriate modifications are made. Then appropriate test cases are selected from the already existing test suite which covers all the modified and affected parts of the source code. We can add new test cases if required. In the end, regression testing is performed using the selected test cases.

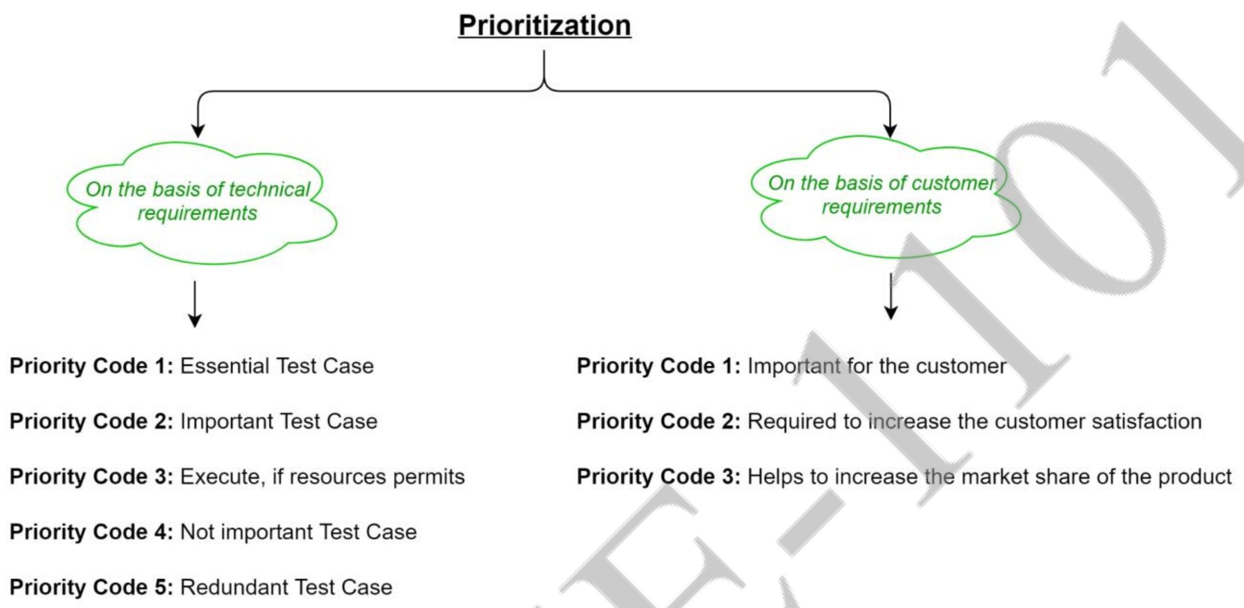

Identification of Bugs
Program failure
Debugging
Identification of bugs
Modification
Modification of source code to make it bug free
Selection & Execution of Test Cases
Select the existing test cases covering the modified and affected parts of the code
Generate new test cases if required
Execute test cases to perform regression testing

Techniques for the selection of Test cases for Regression Testing

- Select all test cases: In this technique, all the test cases are selected from the already existing test suite. It is the simplest and safest technique but not very efficient.
- Select test cases randomly: In this technique, test cases are selected randomly from the existing test suite, but it is only useful if all the test cases are equally good in their fault detection capability which is very rare. Hence, it is not used in most of the cases.
- Select modification traversing test cases: In this technique, only those test cases are selected that cover and test the modified portions of the source code and the parts that are affected by these modifications.
- Select higher priority test cases: In this technique, priority codes are assigned to each test case of the test suite based upon their bug detection capability, customer requirements, etc. After assigning the priority codes, test cases with the highest priorities are selected for the process of regression testing. The test case with the

highest priority has the highest rank. For example, a test case with priority code 2 is less important than a test case with priority code 1.

-


Prioritization
On the basis of technical requirements
On the basis of customer requirements
Priority Code 1: Essential Test Case
Priority Code 2: Important Test Case
Priority Code 3: Execute, if resources permits
Priority Code 4: Not important Test Case
Priority Code 5: Redundant Test Case
Priority Code 1: Important for the customer
Priority Code 2: Required to increase the customer satisfaction
Priority Code 3: Helps to increase the market share of the product

Tools for Regression testing
In regression testing, we generally select the test cases from the existing test suite itself and hence, we need not compute their expected output, and it can be easily automated due to this reason. Automating the process of regression testing will be very effective and time-saving. The most commonly used tools for regression testing are:

- Selenium
- WATIR (Web Application Testing In Ruby)
- QTP (Quick Test Professional)
- RFT (Rational Functional Tester)
- Winrunner
- Silktest

Advantages of Regression Testing
- It ensures that no new bugs have been introduced after adding new functionalities to the system.

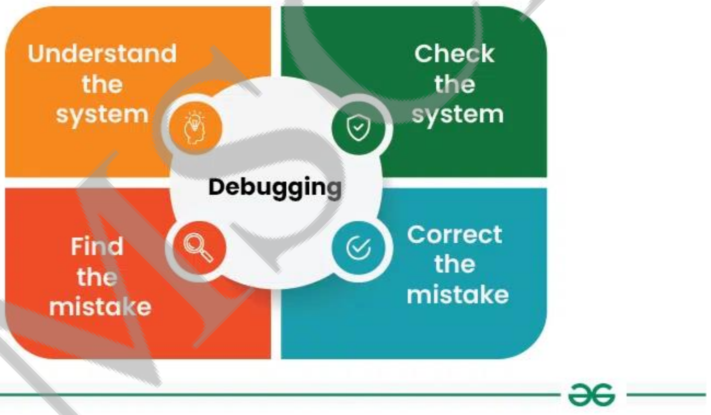
- As most of the test cases used in Regression Testing are selected from the existing test suite, and we already know their expected outputs. Hence, it can be easily automated by the automated tools.
- It helps to maintain the quality of the source code.

Disadvantages of Regression Testing

- It can be time and resource-consuming if automated tools are not used.
- It is required even after very small changes in the code.

What is Debugging in Software Engineering?

Debugging is the process of identifying and resolving errors, or bugs, in a software system. It is an important aspect of software engineering because bugs can cause a software system to malfunction, and can lead to poor performance or incorrect results. Debugging can be a time-consuming and complex task, but it is essential for ensuring that a software system is functioning correctly.



What is Debugging ?

In the context of software engineering, debugging is the process of fixing a bug in the software. When there’s a problem with software, programmers analyze the code to figure out why things aren’t working correctly. They use different debugging tools to carefully go through the code, step by step, find the issue, and make the necessary corrections.

Why is it called debugging?

The term “debugging” originated from an incident involving Grace Hopper in the 1940s when a moth caused a malfunction in the Mark II computer at Harvard University. The term stuck and is now commonly used to describe the process of finding and fixing errors in computer programs. In simpler terms, debugging got its name from removing a moth that caused a computer problem.

Methods and Techniques Used in Debugging
There are several common methods and techniques used in debugging, including:

1. Code Inspection: This involves manually reviewing the source code of a software system to identify potential bugs or errors.
2. Debugging Tools: There are various tools available for debugging such as debuggers, trace tools, and profilers that can be used to identify and resolve bugs.
3. Unit Testing: This involves testing individual units or components of a software system to identify bugs or errors.
4. Integration Testing: This involves testing the interactions between different components of a software system to identify bugs or errors.
5. System Testing: This involves testing the entire software system to identify bugs or errors.
6. Monitoring: This involves monitoring a software system for unusual behavior or performance issues that can indicate the presence of bugs or errors.
7. Logging: This involves recording events and messages related to the software system, which can be used to identify bugs or errors.

Process of Debugging

The steps involved in debugging are:

- Problem identification and report preparation.
- Assigning the report to the software engineer defect to verify that it is genuine.
- Defect Analysis using modeling, documentation, finding and testing candidate flaws, etc.
- Defect Resolution by making required changes to the system.

- Validation of corrections.

The debugging process will always have one of two outcomes :

1. The cause will be found and corrected.
2. The cause will not be found.

Why is debugging important?
Fixing mistakes in computer programming, known as bugs or errors, is necessary because programming deals with abstract ideas and concepts. Computers understand machine language, but we use programming languages to make it easier for people to talk to computers. Software has many layers of abstraction, meaning different parts must work together for an application to function properly. When errors happen, finding and fixing them can be tricky. That’s where debugging tools and strategies come in handy. They help solve problems faster, making developers more efficient. This not only improves the quality of the software but also makes the experience better for the people using it. In simple terms, debugging is important because it makes sure the software works well and people have a good time using it.

Debugging Approaches/Strategies

1. Brute Force: Study the system for a longer duration to understand the system. It helps the debugger to construct different representations of systems to be debugged depending on the need. A study of the system is also done actively to find recent changes made to the software.
2. Backtracking: Backward analysis of the problem which involves tracing the program backward from the location of the failure message to identify the region of faulty code. A detailed study of the region is conducted to find the cause of defects.
3. Forward analysis of the program involves tracing the program forwards using breakpoints or print statements at different points in the program and studying the results. The region where the wrong outputs are obtained is the region that needs to be focused on to find the defect.
4. Using A debugging experience with the software debug the software with similar problems in nature. The success of this approach depends on the expertise of the debugger.

5. Cause elimination: it introduces the concept of binary partitioning. Data related to the error occurrence are organized to isolate potential causes.
6. Static analysis: Analyzing the code without executing it to identify potential bugs or errors. This approach involves analyzing code syntax, data flow, and control flow.
7. Dynamic analysis: Executing the code and analyzing its behavior at runtime to identify errors or bugs. This approach involves techniques like runtime debugging and profiling.
8. Collaborative debugging: Involves multiple developers working together to debug a system. This approach is helpful in situations where multiple modules or components are involved, and the root cause of the error is not clear.
9. Logging and Tracing: Using logging and tracing tools to identify the sequence of events leading up to the error. This approach involves collecting and analyzing logs and traces generated by the system during its execution.
10. Automated Debugging: The use of automated tools and techniques to assist in the debugging process. These tools can include static and dynamic analysis tools, as well as tools that use machine learning and artificial intelligence to identify errors and suggest fixes.

Examples of error during debugging

Some common example of error during debugging are:

- Syntax error
- Logical error
- Runtime error
- Stack overflow
- Index Out of Bound Errors
- Infinite loops
- Concurrency Issues
- I/O errors
- Environment Dependencies
- Integration Errors

- Reference error
- Type error

Debugging Tools

A debugging tool is a computer program that is used to test and debug other programs. A lot of public domain software like gdb and dbx are available for debugging. They offer console-based command-line interfaces. Examples of automated debugging tools include code-based tracers, profilers, interpreters, etc. Some of the widely used debuggers are:

- Radare2
- WinDbg
- Valgrind

Difference Between Debugging and Testing

Debugging is different from testing. Testing focuses on finding bugs, errors, etc whereas debugging starts after a bug has been identified in the software. Testing is used to ensure that the program is correct and it was supposed to do with a certain minimum success rate. Testing can be manual or automated. There are several different types of testing unit testing, integration testing, alpha, and beta testing, etc.

Debugging requires a lot of knowledge, skills, and expertise. It can be supported by some automated tools available but is more of a manual process as every bug is different and requires a different technique, unlike a pre-defined testing mechanism.

Advantages of Debugging

Several advantages of debugging in software engineering:

1. Improved system quality: By identifying and resolving bugs, a software system can be made more reliable and efficient, resulting in improved overall quality.
2. Reduced system downtime: By identifying and resolving bugs, a software system can be made more stable and less likely to experience downtime, which can result in improved availability for users.

3. Increased user satisfaction: By identifying and resolving bugs, a software system can be made more user-friendly and better able to meet the needs of users, which can result in increased satisfaction.
4. Reduced development costs: Identifying and resolving bugs early in the development process, can save time and resources that would otherwise be spent on fixing bugs later in the development process or after the system has been deployed.
5. Increased security: By identifying and resolving bugs that could be exploited by attackers, a software system can be made more secure, reducing the risk of security breaches.
6. Facilitates change: With debugging, it becomes easy to make changes to the software as it becomes easy to identify and fix bugs that would have been caused by the changes.
7. Better understanding of the system: Debugging can help developers gain a better understanding of how a software system works, and how different components of the system interact with one another.
8. Facilitates testing: By identifying and resolving bugs, it makes it easier to test the software and ensure that it meets the requirements and specifications.

In summary, debugging is an important aspect of software engineering as it helps to improve system quality, reduce system downtime, increase user satisfaction, reduce development costs, increase security, facilitate change, a better understanding of the system, and facilitate testing.

Disadvantages of Debugging

While debugging is an important aspect of software engineering, there are also some disadvantages to consider:

1. Time-consuming: Debugging can be a time-consuming process, especially if the bug is difficult to find or reproduce. This can cause delays in the development process and add to the overall cost of the project.
2. Requires specialized skills: Debugging can be a complex task that requires specialized skills and knowledge. This can be a challenge for developers who are not familiar with the tools and techniques used in debugging.

3. Can be difficult to reproduce: Some bugs may be difficult to reproduce, which can make it challenging to identify and resolve them.
4. Can be difficult to diagnose: Some bugs may be caused by interactions between different components of a software system, which can make it challenging to identify the root cause of the problem.
5. Can be difficult to fix: Some bugs may be caused by fundamental design flaws or architecture issues, which can be difficult or impossible to fix without significant changes to the software system.
6. Limited insight: In some cases, debugging tools can only provide limited insight into the problem and may not provide enough information to identify the root cause of the problem.
7. Can be expensive: Debugging can be an expensive process, especially if it requires additional resources such as specialized debugging tools or additional development time

Program Analysis Tools in Software Engineering

The goal of developing software that is reliable, safe and effective is crucial in the dynamic and always changing field of software development. Programme Analysis Tools are a developer's greatest support on this trip, giving them invaluable knowledge about the inner workings of their code. In this article, we'll learn about it's importance and classification.

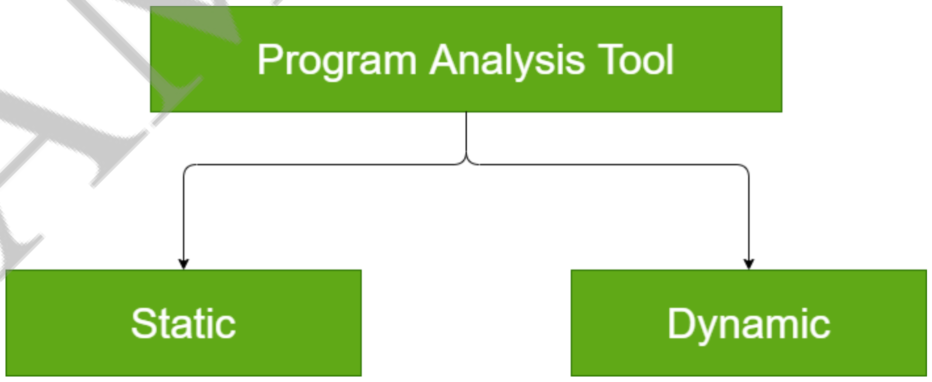
What is Program Analysis Tool?
Program Analysis Tool is an automated tool whose input is the source code or the executable code of a program and the output is the observation of characteristics of the program. It gives various characteristics of the program such as its size, complexity, adequacy of commenting, adherence to programming standards and many other characteristics. These tools are essential to software engineering because they help programmers comprehend, improve and maintain software systems over the course of the whole development life cycle.

Importance of Program Analysis Tools

1. Finding faults and Security Vulnerabilities in the Code: Automatic programme analysis tools can find and highlight possible faults, security flaws and bugs in the code. This lowers the possibility that bugs will get it into production by assisting developers in identifying problems early in the process.
2. Memory Leak Detection: Certain tools are designed specifically to find memory leaks and inefficiencies. By doing so, developers may make sure that their software doesn't gradually use up too much memory.
3. Vulnerability Detection: Potential vulnerabilities like buffer overflows, injection attacks or other security flaws can be found using programme analysis tools, particularly those that are security-focused. For the development of reliable and secure software, this is essential.
4. Dependency analysis: By examining the dependencies among various system components, tools can assist developers in comprehending and controlling the connections between modules. This is necessary in order to make well-informed decisions during refactoring.
5. Automated Testing Support: To automate testing procedures, CI/CD pipelines frequently combine programme analysis tools. Only well-tested, high-quality code is released into production thanks to this integration, helping in identifying problems early in the development cycle.

Classification of Program Analysis Tools

Program Analysis Tools are classified into two categories:



1. Static Program Analysis Tools

Static Program Analysis Tool is such a program analysis tool that evaluates and computes various characteristics of a software product without executing it. Normally, static program analysis tools analyze some structural representation of a program to reach a certain analytical conclusion. Basically some structural properties are analyzed using static program analysis tools. The structural properties that are usually analyzed are:

1. Whether the coding standards have been fulfilled or not.
2. Some programming errors such as uninitialized variables.
3. Mismatch between actual and formal parameters.
4. Variables that are declared but never used.

Code walkthroughs and code inspections are considered as static analysis methods but static program analysis tool is used to designate automated analysis tools. Hence, a compiler can be considered as a static program analysis tool.

2. Dynamic Program Analysis Tools

Dynamic Program Analysis Tool is such type of program analysis tool that require the program to be executed and its actual behavior to be observed. A dynamic program analyzer basically implements the code. It adds additional statements in the source code to collect the traces of program execution. When the code is executed, it allows us to observe the behavior of the software for different test cases. Once the software is tested and its behavior is observed, the dynamic program analysis tool performs a post execution analysis and produces reports which describe the structural coverage that has been achieved by the complete testing process for the program.

For example, the post execution dynamic analysis report may provide data on extent statement, branch and path coverage achieved. The results of dynamic program analysis tools are in the form of a histogram or a pie chart. It describes the structural coverage obtained for different modules of the program. The output of a dynamic program analysis tool can be stored and printed easily and provides evidence that complete testing has been done. The result of dynamic analysis is the extent of testing performed as white box testing. If the testing result is not satisfactory then more test cases are designed and added to the test scenario. Also dynamic analysis helps in elimination of redundant test cases.

Master Software Testing and Automation in an efficient and time-bound manner by mentors with real-time industry experience. Join our Software Automation Course and embark on an exciting journey, mastering the skill set with ease!

Symbolic Execution

What is Symbolic Execution?

Symbolic execution is a software testing technique that is useful to aid the generation of test data and in proving the program quality.

Steps to use Symbolic Execution:

The execution requires a selection of paths that are exercised by a set of data values. A program, which is executed using actual data, results in the output of a series of values.

In symbolic execution, the data is replaced by symbolic values with set of expressions, one expression per output variable.

The common approach for symbolic execution is to perform an analysis of the program, resulting in the creation of a flow graph.

The flowgraph identifies the decision points and the assignments associated with each flow. By traversing the flow graph from an entry point, a list of assignment statements and branch predicates is produced.

Issues with Symbolic Execution:

Symbolic execution cannot proceed if the number of iterations in the loop is known.

The second issue is the invocation of any out-of-line code or module calls.

Symbolic execution cannot be used with arrays.

The symbolic execution cannot identify of infeasible paths.

Symbolic Execution Application:

Path domain checking

Test Data generation

Partition analysis

Symbolic debugging

Symbolic Model Checking

What is Symbolic Model Checking?

Symbolic model checking is a conventional technique utilized in the field of software engineering and the hypothesis of calculation to confirm the rightness of equipment and programming situations. The methodology includes addressing the framework under examination as a numerical model, normally as a limited state machine or progress framework, and afterward utilizing robotized calculations to dissect the model and check for blunders or properties.

The procedure is designated “emblematic symbolic” in light of the fact that it utilizes representative control strategies, for example, Paired Choice Outlines (BDDs) and Satisfiability Modulo Speculations (SMT), to proficiently deal with enormous and complex state spaces. The emblematic model really takes a look at works by addressing the framework’s state space as a bunch of sensible recipes, and afterward utilizing computerized hypotheses demonstrating devices to reason about these equations.

Attributes of Symbolic Model Checking

Symbolic model checking is a strong procedure for confirming the rightness of equipment and programming situation, and it has a few qualities that make it a viable device for formal confirmation:

- Emblematic control: Symbolic model checking utilizes emblematic control methods, for example, Twofold Choice Charts (BDDs) and Satisfiability Modulo Speculations (SMT), to address and control enormous and complex state spaces. These methods can fundamentally lessen the computational intricacy of confirmation issues, making it conceivable to actually take a look at frameworks with billions of potential states.
- Robotization: symbolic model checking is a completely computerized method that requires practically zero human intercession. When the framework under examination and its determination are encoded in a reasonable conventional language, the confirmation cycle can be performed naturally utilizing a model really looking at the device.

- Culmination: symbolic model checking can give total confirmation, implying that it can really take a look at all potential conditions of a framework and check that the framework fulfills its determination.
- Adaptability: Emblematic model checking scales well with framework size, making it appropriate for confirming enormous and complex frameworks. It can deal with frameworks with many cooperating parts, complex information designs, and simultaneousness.

Support for different symbolic models really looks at upholds various formalisms for framework detail, including transient rationales, automata, and other conventional dialects. This adaptability makes it conceivable to determine and confirm a great many frameworks.

- Broadly utilized: symbolic model checking is a deeply grounded procedure that has been generally utilized in scholarly world and industry to confirm security basic frameworks, including airplane control frameworks, rail route frameworks, and clinical gadgets.

By and large, the symbolic model really looking at offers a strong and successful way to deal with formal check, giving a serious level of computerization and versatility, while supporting different formalisms for framework detail.

Methods Associated with Symbolic Model Checking

Symbolic model checking includes a few methods that are utilized to emblematically address and control the framework under investigation. Here are a portion of the key strategies utilized in emblematic model checking:

- Emblematic portrayal: The framework under investigation is addressed emblematically utilizing numerical designs, for example, Boolean equations, Paired Choice Outlines (BDDs), and propositional rationale. This permits the framework to be dissected all the more effectively by staying away from unequivocal list of states.
- Model development: A model of the framework under investigation is built utilizing formal strategies, for example, automata, Petri nets, or change frameworks. The

    model is then changed into an emblematic portrayal utilizing strategies, for example, BDDs or other choice graphs.

    - State-space investigation: symbolic model checking investigates the state space of the framework under examination to confirm in the event that it fulfills a given detail. State-space investigation methods include utilizing calculations that cross the state space of the framework and check assuming the determination is fulfilled.
    - Choice strategies: symbolic model checking frequently involves choice techniques for satisfiability modulo speculations (SMT) or satisfiability (SAT) issues to check regardless of whether a bunch of consistent recipes is satisfiable. This assists with distinguishing counterexamples and check the accuracy of the framework.

- Property detail: The determination that the framework needs to fulfill is regularly indicated by utilizing a transient rationale, like Straight Worldly Rationale (LTL) or Calculation Tree Rationale (CTL). These rationales take into consideration the declaration of transient connections between framework ways of behaving.
- Deliberation: Symbolic model checking might utilize reflection methods to lessen the intricacy of the framework under examination, for example, by eliminating unimportant subtleties or amassing comparative states. This can assist with making the confirmation cycle more proficient.

UNIT V PROJECT MANAGEMENT 9

Software Project Management- Software Configuration Management - Project Scheduling- DevOps: Motivation-Cloud as a platform-Operations- Deployment Pipeline:Overall Architecture Building and Testing-Deployment- Tools- Case Study

Software Project Management (SPM) – Software Engineering

Software Project Management (SPM) is a proper way of planning and leading software projects. It is a part of project management in which software projects are planned, implemented, monitored, and controlled. This article focuses on discussing Software Project Management (SPM).

Need for Software Project Management

Software is a non-physical product. Software development is a new stream in business and there is very little experience in building software products. Most of the software products are made to fit clients' requirements. The most important is that basic technology changes and advances so frequently and rapidly that the experience of one product may not be applied to the other one. Such types of business and environmental constraints increase risk in software development hence it is essential to manage software projects efficiently. It is necessary for an organization to deliver quality products, keep the cost within the client's budget constraint, and deliver the project as per schedule. Hence, in order, software project management is necessary to incorporate user requirements along with budget and time constraints.

Types of Management in SPM

1. Conflict Management

Conflict management is the process to restrict the negative features of conflict while increasing the positive features of conflict. The goal of conflict management is to improve learning and group results including efficacy or performance in an organizational setting. Properly managed conflict can enhance group results.

2. Risk Management

Risk management is the analysis and identification of risks that is followed by synchronized and economical implementation of resources to minimize, operate and control the possibility or effect of unfortunate events or to maximize the realization of opportunities.

3. Requirement Management

It is the process of analyzing, prioritizing, tracking, and documenting requirements and then supervising change and communicating to pertinent stakeholders. It is a continuous process during a project.

4. Change Management

Change management is a systematic approach to dealing with the transition or transformation of an organization's goals, processes, or technologies. The purpose of change management is to execute strategies for effecting change, controlling change, and helping people to adapt to change.

5. Software Configuration Management

Software configuration management is the process of controlling and tracking changes in the software, part of the larger cross-disciplinary field of configuration management. Software configuration management includes revision control and the inauguration of baselines.

6. Release Management

Release Management is the task of planning, controlling, and scheduling the built-in deploying releases. Release management ensures that the organization delivers new and enhanced services required by the customer while protecting the integrity of existing services.

Aspects of Software Project Management
The list of focus areas it can tackle and the broad upsides of Software Project Management is:

1. Planning

The software project manager lays out the complete project's blueprint. The project plan will outline the scope, resources, timelines, techniques, strategy, communication, testing, and maintenance steps. SPM can aid greatly here.

2. Leading

A software project manager brings together and leads a team of engineers, strategists, programmers, designers, and data scientists. Leading a team necessitates exceptional communication, interpersonal, and leadership abilities. One can only hope to do this effectively if one sticks with the core SPM principles.

3. Execution

SPM comes to the rescue here also as the person in charge of software projects (if well versed with SPM/Agile methodologies) will ensure that each stage of the project is completed successfully. measuring progress, monitoring to check how teams function, and generating status reports are all part of this process.

4. Time Management

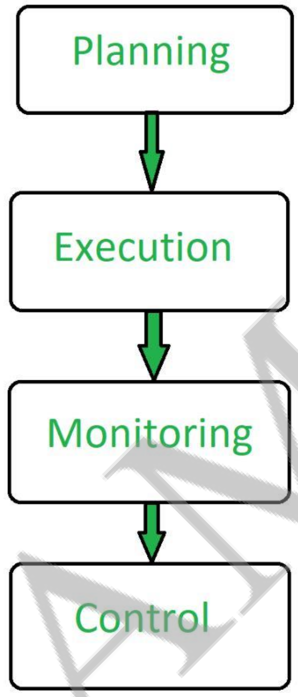
Abiding by a timeline is crucial to completing deliverables successfully. This is especially difficult when managing software projects because changes to the original project charter are unavoidable over time. To assure progress in the face of blockages or changes, software project managers ought to be specialists in managing risk and emergency preparedness. This Risk Mitigation and management is one of the core tenets of the philosophy of SPM.

5. Budget

Software project managers, like conventional project managers, are responsible for generating a project budget and adhering to it as closely as feasible, regulating spending, and reassigning funds as needed. SPM teaches us how to effectively manage the monetary aspect of projects to avoid running into a financial crunch later on in the project.

6. Maintenance

Software project management emphasizes continuous product testing to find and repair defects early, tailor the end product to the needs of the client, and keep the project on track. The software project manager ensures that the product is thoroughly tested, analyzed, and adjusted as needed. Another point in favor of SPM.



Downsides of Software Project Management
Numerous issues can develop if a Software project manager lacks the necessary expertise or knowledge. Software Project management has several drawbacks, including resource loss, scheduling difficulty, data protection concerns, and interpersonal conflicts between Developers/Engineers/Stakeholders. Furthermore, outsourcing work or recruiting additional personnel to complete the project may result in hefty costs for one's company.

1. Costs are High

Consider spending money on various kinds of project management tools, software, & services if ones engage in Software Project Management strategies. These initiatives can be expensive and time-consuming to put in place. Because your team will be using them as well, they may require training. One may need to recruit subject-matter experts or specialists to assist with a project, depending on the circumstances. Stakeholders will frequently press for the inclusion of features that were not originally envisioned. All of these factors can quickly drive up a project's cost.

2. Complexity will be increased

Software Project management is a multi-stage, complex process. Unfortunately, some specialists might have a propensity to overcomplicate everything, which can lead to confusion among teams and lead to delays in project completion. Their expressions are very strong and specific in their ideas, resulting in a difficult work atmosphere. Projects having a larger scope are typically more arduous to complete, especially if there isn't a dedicated team committed completely to the project. Members of cross-functional teams may lag far behind their daily tasks, adding to the overall complexity of the project being worked on.

3. Overhead in Communication

Recruits enter your organization when we hire software project management personnel. This provides a steady flow of communication that may or may not match a company's culture. As a result, it is advised that you maintain your crew as small as feasible. The communication overhead tends to skyrocket when a team becomes large enough. When a large team is needed for a project, it's critical to identify software project managers who can conduct effective communication with a variety of people.

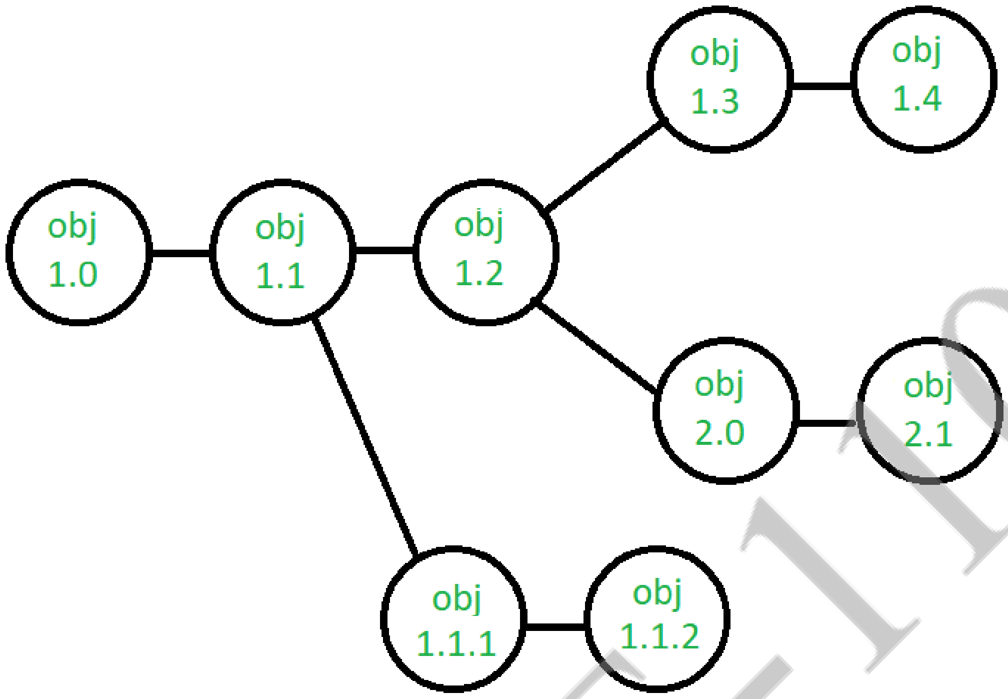
4. Lack of Originality

Software Project managers can sometimes provide little or no space for creativity. Team leaders either place an excessive amount of emphasis on management processes or impose hard deadlines on their employees, requiring them to develop and operate code within stringent guidelines. This can stifle innovative thought and innovation that could be beneficial to the project. When it comes to Software project management, knowing when to encourage creativity and when to stick to the project plan is crucial. Without Software project management personnel, an organization can perhaps build and ship code more quickly. However, employing a trained specialist to handle these areas, on the other hand, can open up new doors and help the organization achieve its objectives more quickly and more thoroughly.

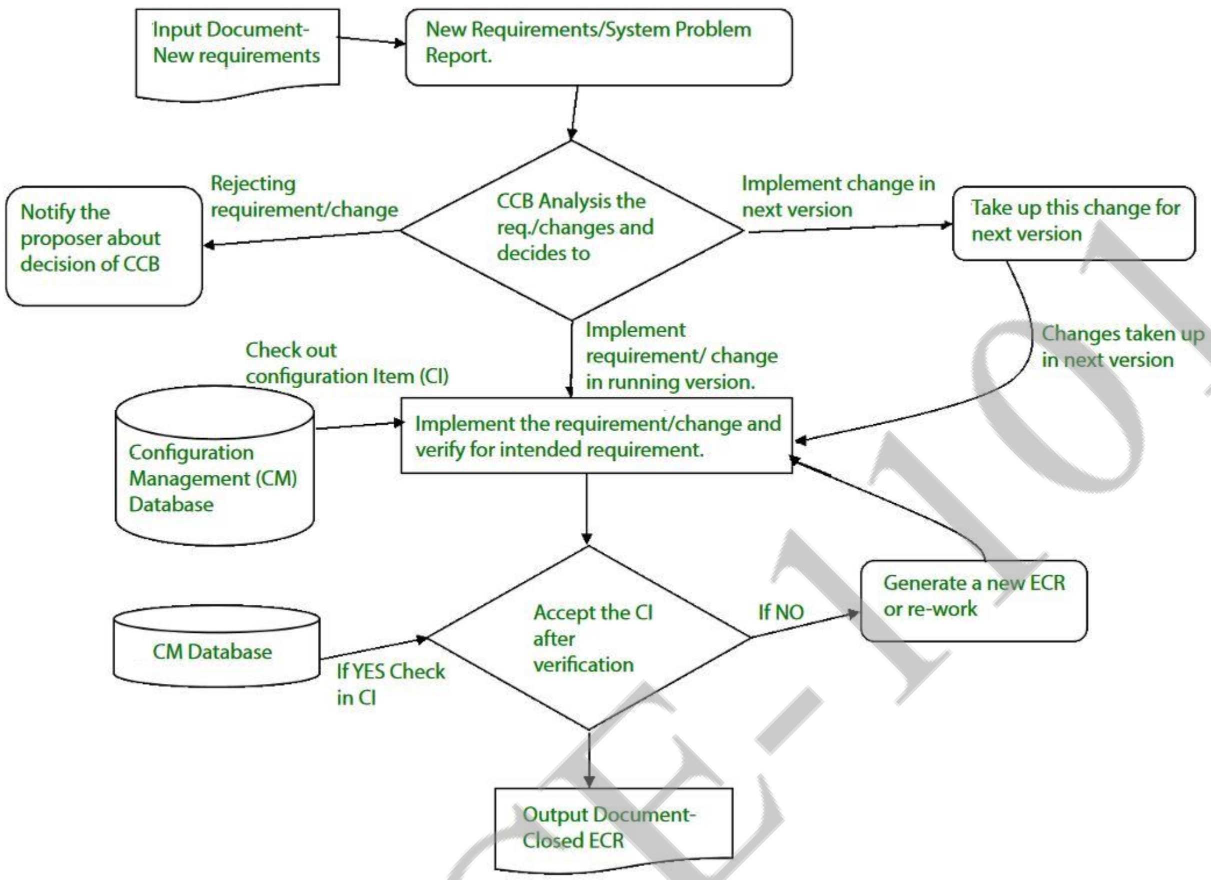
System configuration management – Software Engineering
Whenever software is built, there is always scope for improvement and those improvements bring picture changes. Changes may be required to modify or update any existing solution or to create a new solution for a problem. Requirements keep on changing daily so we need to keep on upgrading our systems based on the current requirements and needs to meet desired outputs. Changes should be analyzed before they are made to the existing system, recorded before they are implemented, reported to have details of before and after, and controlled in a manner that will improve quality and reduce error. This is where the need for System Configuration Management comes. **System Configuration Management (SCM)** is an arrangement of exercises that controls change by recognizing the items for change, setting up connections between those things, making/characterizing instruments for overseeing diverse variants, controlling the changes being executed in the current framework, inspecting and revealing/reporting on the changes made. It is essential to control the changes because if the changes are not checked legitimately then they may wind up undermining a well-run programming. In this way, SCM is a fundamental piece of all project management activities.

Processes involved in SCM – Configuration management provides a disciplined environment for smooth control of work products. It involves the following activities:

1. **Identification and Establishment –** Identifying the configuration items from products that compose baselines at given points in time (a baseline is a set of mutually consistent Configuration Items, which has been formally reviewed and agreed upon, and serves as the basis of further development). Establishing relationships among items, creating a mechanism to manage multiple levels of control and procedure for the change management system.
2. **Version control –** Creating versions/specifications of the existing product to build new products with the help of the SCM system. A description of the version is given below:



1. Suppose after some changes, the version of the configuration object changes from 1.0 to 1.1. Minor corrections and changes result in versions 1.1.1 and 1.1.2, which is followed by a major update that is object 1.2. The development of object 1.0 continues through 1.3 and 1.4, but finally, a noteworthy change to the object results in a new evolutionary path, version 2.0. Both versions are currently supported.
2. **Change control –** Controlling changes to Configuration items (CI). The change control process is explained in Figure below:


Input Document-
New requirements
New Requirements/System Problem Report.
Rejecting requirement/change
CCB Analysis the req./changes and decides to
Implement change in next version
Notify the proposer about decision of CCB
Take up this change for next version
Check out configuration Item (CI)
Implement requirement/ change in running version.
Changes taken up in next version
Configuration Management (CM) Database
Implement the requirement/change and verify for intended requirement.
Generate a new ECR or re-work
CM Database
Accept the CI after verification
If NO
If YES Check in CI
Output Document-
Closed ECR

A change request (CR) is submitted and evaluated to assess technical merit, potential side effects, the overall impact on other configuration objects and system functions, and the projected cost of the change. The results of the evaluation are presented as a change report, which is used by a change control board (CCB) —a person or group who makes a final decision on the status and priority of the change. An engineering change Request (ECR) is generated for each approved change. Also, CCB notifies the developer in case the change is rejected with proper reason. The ECR describes the change to be made, the constraints that must be respected, and the criteria for review and audit. The object to be changed is “checked out” of the project database, the change is made, and then the object is tested again. The object is then “checked in” to the database and appropriate version control mechanisms are used to create the next version of the software.

1. **Configuration auditing –** A software configuration audit complements the formal technical review of the process and product. It focuses on the technical correctness of the configuration object that has been modified. The audit confirms the completeness, correctness, and consistency of items in the SCM system and tracks action items from the audit to closure.
2. **Reporting –** Providing accurate status and current configuration data to developers, testers, end users, customers, and stakeholders through admin guides, user guides, FAQs, Release notes, Memos, Installation Guide, Configuration guides, etc.

Importance of Software Configuration Management

1. Effective Bug Tracking: Linking code modifications to issues that have been reported, makes bug tracking more effective.

2. Continuous Deployment and Integration: SCM combines with continuous processes to automate deployment and testing, resulting in more dependable and timely software delivery.
3. Risk management: SCM lowers the chance of introducing critical flaws by assisting in the early detection and correction of problems.
4. Support for Big Projects: Source Code Control (SCM) offers an orderly method to handle code modifications for big projects, fostering a well-organized development process.
5. Reproducibility: By recording precise versions of code, libraries, and dependencies, source code versioning (SCM) makes builds repeatable.
6. Parallel Development: SCM facilitates parallel development by enabling several developers to collaborate on various branches at once.

Why need for System configuration management?

1. **Replicability: Software version control (SCM) makes ensures that a software system can be replicated at any stage of its development. This is necessary for testing, debugging, and upholding consistent environments in production, testing, and development.**
2. **Identification of Configuration: Source code, documentation, and executable files are examples of configuration elements that SCM helps in locating and labeling. The management of a system's constituent parts and their interactions depend on this identification.**
3. **Effective Process of Development: By automating monotonous processes like managing dependencies, merging changes, and resolving disputes, SCM simplifies the development process. Error risk is decreased and efficiency is increased because of this automation.**

Key objectives of SCM

1. **Control the evolution of software systems: SCM helps to ensure that changes to a software system are properly planned, tested, and integrated into the final product.**
2. **Enable collaboration and coordination: SCM helps teams to collaborate and coordinate their work, ensuring that changes are properly integrated and that everyone is working from the same version of the software system.**
3. **Provide version control: SCM provides version control for software systems, enabling teams to manage and track different versions of the system and to revert to earlier versions if necessary.**
4. **Facilitate replication and distribution: SCM helps to ensure that software systems can be easily replicated and distributed to other environments, such as test, production, and customer sites.**
5. **SCM is a critical component of software development, and effective SCM practices can help to improve the quality and reliability of software systems, as well as increase efficiency and reduce the risk of errors.**

The main advantages of SCM

1. **Improved productivity and efficiency by reducing the time and effort required to manage software changes.**
2. **Reduced risk of errors and defects by ensuring that all changes were properly tested and validated.**

3. **Increased collaboration and communication among team members by providing a central repository for software artifacts.**
4. **Improved quality and stability of software systems by ensuring that all changes are properly controlled and managed.**

The main disadvantages of SCM

1. **Increased complexity and overhead, particularly in large software systems.**
2. **Difficulty in managing dependencies and ensuring that all changes are properly integrated.**
3. **Potential for conflicts and delays, particularly in large development teams with multiple contributors.**

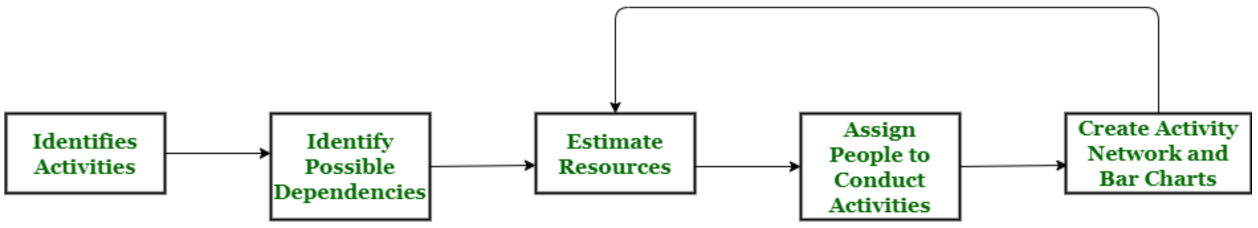
System configuration management – Software Engineering

[redacted]

Whenever software is built, there is always scope for improvement and those improvements bring picture changes. Changes may be required to modify or update any existing solution or to create a new solution for a problem. Requirements keep on changing daily so we need to keep on upgrading our systems based on the current requirements and needs to meet desired outputs. Changes should be analyzed before they are made to the existing system, recorded before they are implemented, reported to have details of before and after, and controlled in a manner that will improve quality and reduce error. This is where the need for System Configuration Management comes. System Configuration Management (SCM) is an arrangement of exercises that controls change by recognizing the items for change, setting up connections between those things, making/characterizing instruments for overseeing diverse variants, controlling the changes being executed in the current framework, inspecting and revealing/reporting on the changes made. It is essential to control the changes because if the changes are not checked legitimately then they may wind up undermining a well-run programming. In this way, SCM is a fundamental piece of all project management activities.
Processes involved in SCM – Configuration management provides a disciplined environment for smooth control of work products.

Short note on Project Scheduling

A schedule in your project's time table actually consists of sequenced activities and milestones that are needed to be delivered under a given period of time.
Project schedule simply means a mechanism that is used to communicate and know about that tasks are needed and has to be done or performed and which organizational resources will be given or allocated to these tasks and in what time duration or time frame work is needed to be performed. Effective project scheduling leads to success of project, reduced cost, and increased customer satisfaction. Scheduling in project management means to list out activities, deliverables, and milestones within a project that are delivered. It contains more notes than your average weekly planner notes. The most common and important form of project schedule is Gantt chart.


Identifies Activities
Identify Possible Dependencies
Estimate Resources
Assign People to Conduct Activities
Create Activity Network and Bar Charts

Project Scheduling Process

Process :
The manager needs to estimate time and resources of project while scheduling project. All activities in project must be arranged in a coherent sequence that means activities should be arranged in a logical and well-organized manner for easy to understand. Initial estimates of project can be made optimistically which means estimates can be made when all favorable things will happen and no threats or problems take place.
The total work is separated or divided into various small activities or tasks during project schedule. Then, Project manager will decide time required for each activity or task to get completed. Even some activities are conducted and performed in parallel for efficient performance. The project manager should be aware of fact that each stage of project is not problem-free.
Problems arise during Project Development Stage :

- People may leave or remain absent during particular stage of development.
- Hardware may get failed while performing.
- Software resource that is required may not be available at present, etc.

The project schedule is represented as set of chart in which work-breakdown structure and dependencies within various activities are represented. To accomplish and complete project within a given schedule, required resources must be available when they are needed. Therefore, resource estimation should be done before starting development.
Resources required for Development of Project :

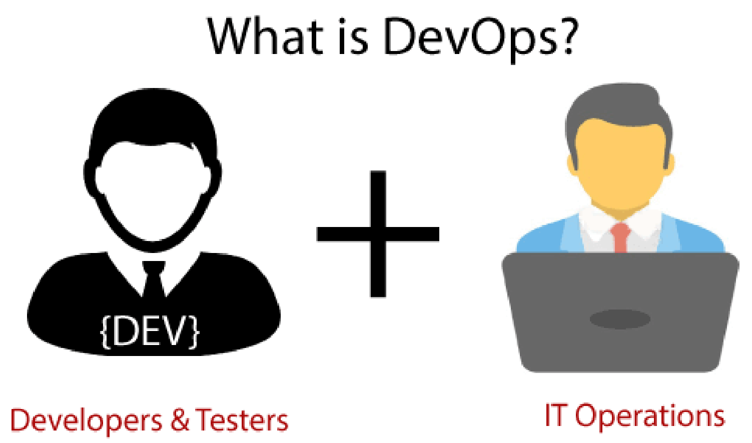
- Human effort
- Sufficient disk space on server
- Specialized hardware
- Software technology
- Travel allowance required by project staff, etc.

Advantages of Project Scheduling :
There are several advantages provided by project schedule in our project management:

- It simply ensures that everyone remains on same page as far as tasks get completed, dependencies, and deadlines.
- It helps in identifying issues early and concerns such as lack or unavailability of resources.
- It also helps to identify relationships and to monitor process.
- It provides effective budget management and risk mitigation.

What is DevOps?

The DevOps is a combination of two words, one is software Development, and second is Operations. This allows a single team to handle the entire application lifecycle, from development to **testing, deployment**, and **operations**. DevOps helps you to reduce the disconnection between software developers, quality assurance (QA) engineers, and system administrators.



DevOps promotes collaboration between Development and Operations team to deploy code to production faster in an automated & repeatable way.
DevOps helps to increase organization speed to deliver applications and services. It also allows organizations to serve their customers better and compete more strongly in the market.
DevOps can also be defined as a sequence of development and IT operations with better communication and collaboration.
DevOps has become one of the most valuable business disciplines for enterprises or organizations. With the help of DevOps, **quality**, and **speed** of the application delivery has improved to a great extent.
DevOps is nothing but a practice or methodology of making "**Developers**" and "**Operations**" folks work together. DevOps represents a change in the IT culture with a complete focus on rapid IT service delivery through the adoption of agile practices in the context of a system-oriented approach.
DevOps is all about the integration of the operations and development process. Organizations that have adopted DevOps noticed a 22% improvement in software quality and a 17% improvement in application deployment frequency and achieve a 22% hike in customer satisfaction. 19% of revenue hikes as a result of the successful DevOps implementation.

Why DevOps?

Before going further, we need to understand why we need the DevOps over the other methods.

- The operation and development team worked in complete isolation.
- After the design-build, the testing and deployment are performed respectively. That's why they consumed more time than actual build cycles.
- Without the use of DevOps, the team members are spending a large amount of time on designing, testing, and deploying instead of building the project.
- Manual code deployment leads to human errors in production.

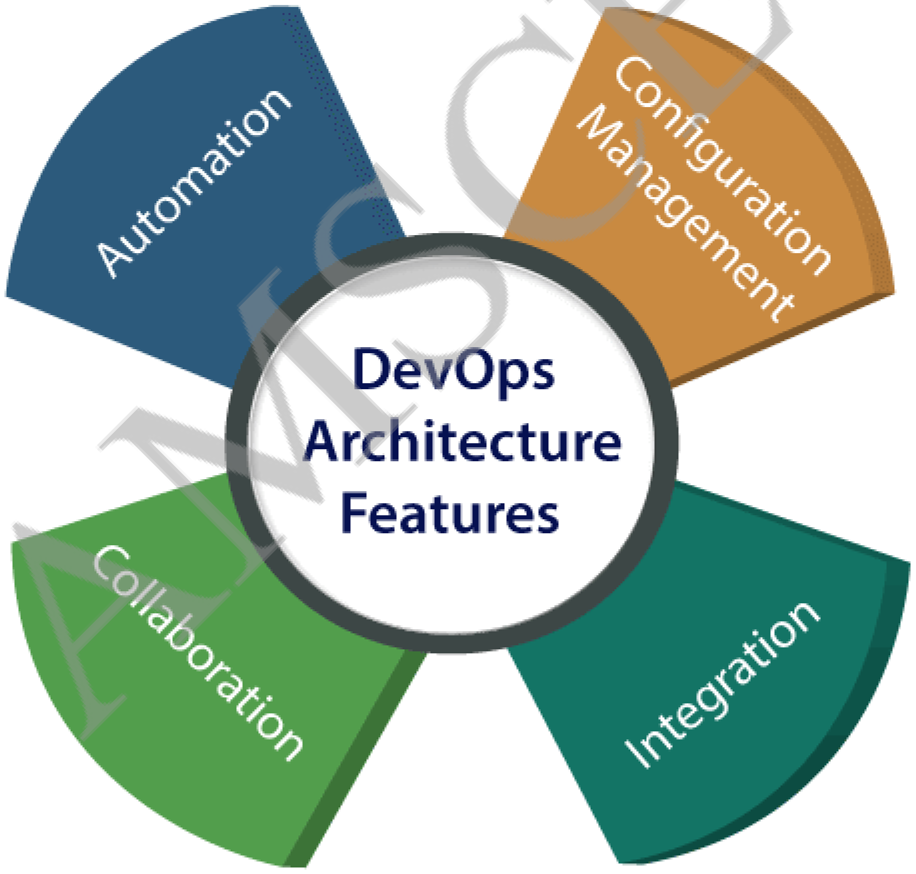
- Coding and operation teams have their separate timelines and are not in synch, causing further delays.

DevOps History

- In 2009, the first conference named **DevOpsdays** was held in Ghent Belgium. Belgian consultant and Patrick Debois founded the conference.
- In 2012, the state of DevOps report was launched and conceived by Alanna Brown at Puppet.
- In 2014, the annual State of DevOps report was published by Nicole Forsgren, Jez Humble, Gene Kim, and others. They found DevOps adoption was accelerating in 2014 also.
- In 2015, Nicole Forsgren, Gene Kim, and Jez Humble founded DORA (DevOps Research and Assignment).
- In 2017, Nicole Forsgren, Gene Kim, and Jez Humble published "Accelerate: Building and Scaling High Performing Technology Organizations".

DevOps Architecture Features

Here are some key features of DevOps architecture, such as:



1) Automation

Automation can reduce time consumption, especially during the testing and deployment phase. The productivity increases, and releases are made quicker by automation. This will lead in catching bugs quickly so that it can be fixed easily. For contiguous delivery, each code is defined through automated tests, cloud-based services, and builds. This promotes production using automated deploys.

2) Collaboration

The Development and Operations team collaborates as a DevOps team, which improves the cultural model as the teams become more productive with their productivity, which strengthens accountability and ownership. The teams share their responsibilities and work closely in sync, which in turn makes the deployment to production faster.

3) Integration

Applications need to be integrated with other components in the environment. The integration phase is where the existing code is combined with new functionality and then tested. Continuous integration and testing enable continuous development. The frequency in the releases and micro-services leads to significant operational challenges. To overcome such problems, continuous integration and delivery are implemented to deliver in a **quicker, safer**, and **reliable manner**.

4) Configuration management

It ensures the application to interact with only those resources that are concerned with the environment in which it runs. The configuration files are not created where the external configuration to the application is separated from the source code. The configuration file can be written during deployment, or they can be loaded at the run time, depending on the environment in which it is running.

DevOps Advantages and Disadvantages

Here are some advantages and disadvantages that DevOps can have for business, such as:

Advantages

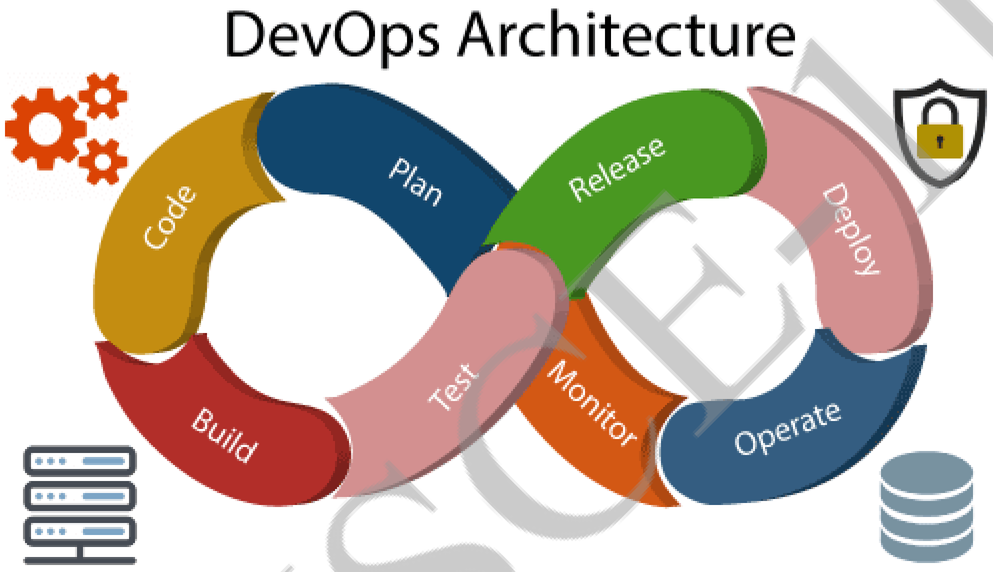
- DevOps is an excellent approach for quick development and deployment of applications.
- It responds faster to the market changes to improve business growth.
- DevOps escalate business profit by decreasing software delivery time and transportation costs.
- DevOps clears the descriptive process, which gives clarity on product development and delivery.
- It improves customer experience and satisfaction.
- DevOps simplifies collaboration and places all tools in the cloud for customers to access.

- DevOps means collective responsibility, which leads to better team engagement and productivity.

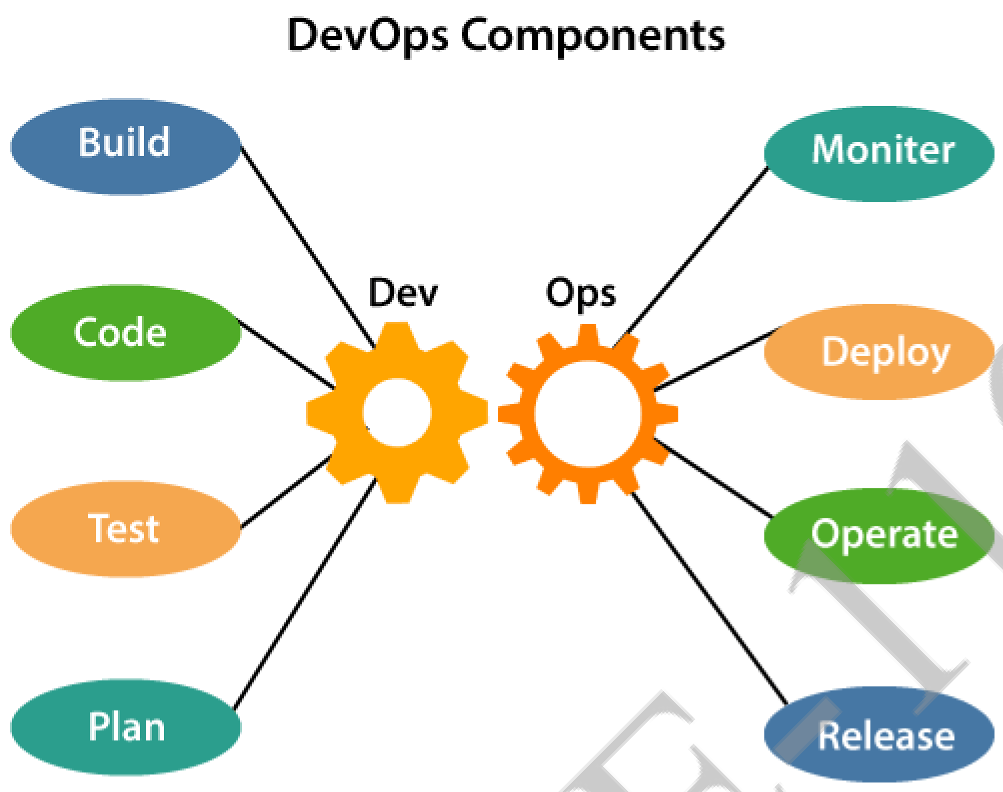
Disadvantages

- DevOps professional or expert's developers are less available.
- Developing with DevOps is so expensive.
- Adopting new DevOps technology into the industries is hard to manage in short time.
- Lack of DevOps knowledge can be a problem in the continuous integration of automation projects

DevOps Architecture



Development and operations both play essential roles in order to deliver applications. The deployment comprises analyzing the **requirements, designing, developing**, and **testing** of the software components or frameworks.
The operation consists of the administrative processes, services, and support for the software. When both the development and operations are combined with collaborating, then the DevOps architecture is the solution to fix the gap between deployment and operation terms; therefore, delivery can be faster.
DevOps architecture is used for the applications hosted on the cloud platform and large distributed applications. Agile Development is used in the DevOps architecture so that integration and delivery can be contiguous. When the development and operations team works separately from each other, then it is time-consuming to **design, test**, and **deploy**. And if the terms are not in sync with each other, then it may cause a delay in the delivery. So DevOps enables the teams to change their shortcomings and increases productivity.
Below are the various components that are used in the DevOps architecture:



1) Build

Without DevOps, the cost of the consumption of the resources was evaluated based on the pre-defined individual usage with fixed hardware allocation. And with DevOps, the usage of cloud, sharing of resources comes into the picture, and the build is dependent upon the user's need, which is a mechanism to control the usage of resources or capacity.

2) Code

Many good practices such as Git enables the code to be used, which ensures writing the code for business, helps to track changes, getting notified about the reason behind the difference in the actual and the expected output, and if necessary reverting to the original code developed. The code can be appropriately arranged in **files, folders**, etc. And they can be reused.

3) Test

The application will be ready for production after testing. In the case of manual testing, it consumes more time in testing and moving the code to the output. The testing can be automated, which decreases the time for testing so that the time to deploy the code to production can be reduced as automating the running of the scripts will remove many manual steps.

4) Plan

DevOps use Agile methodology to plan the development. With the operations and development team in sync, it helps in organizing the work to plan accordingly to increase productivity.

5) Monitor

Continuous monitoring is used to identify any risk of failure. Also, it helps in tracking the system accurately so that the health of the application can be checked. The monitoring becomes more comfortable with services where the log data may get monitored through many third-party tools such as **Splunk**.

6) Deploy

Many systems can support the scheduler for automated deployment. The cloud management platform enables users to capture accurate insights and view the optimization scenario, analytics on trends by the deployment of dashboards.

7) Operate

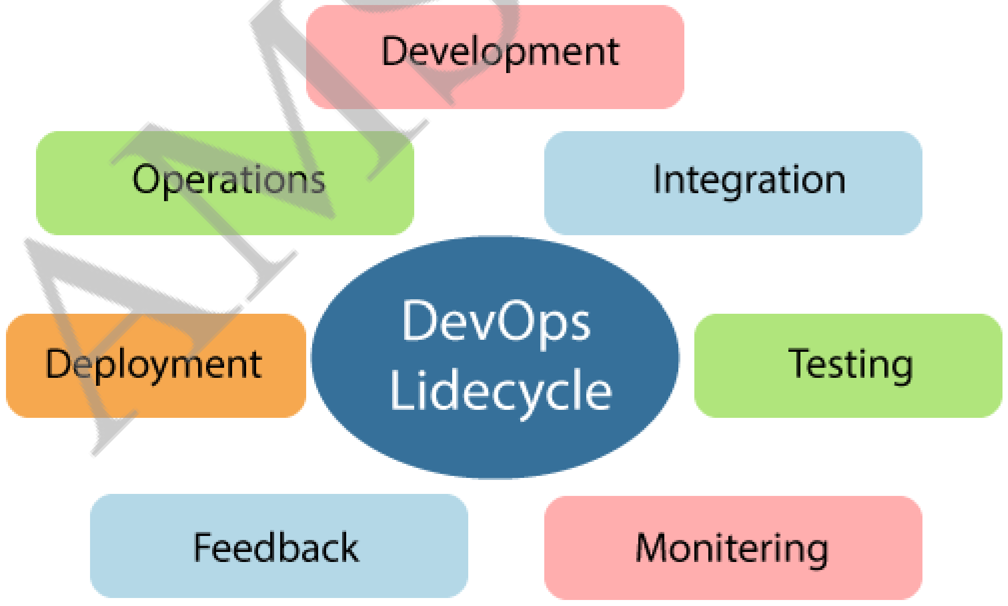
DevOps changes the way traditional approach of developing and testing separately. The teams operate in a collaborative way where both the teams actively participate throughout the service lifecycle. The operation team interacts with developers, and they come up with a monitoring plan which serves the IT and business requirements.

8) Release

Deployment to an environment can be done by automation. But when the deployment is made to the production environment, it is done by manual triggering. Many processes involved in release management commonly used to do the deployment in the production environment manually to lessen the impact on the customers.

DevOps Lifecycle

DevOps defines an agile relationship between operations and Development. It is a process that is practiced by the development team and operational engineers together from beginning to the final stage of the product.



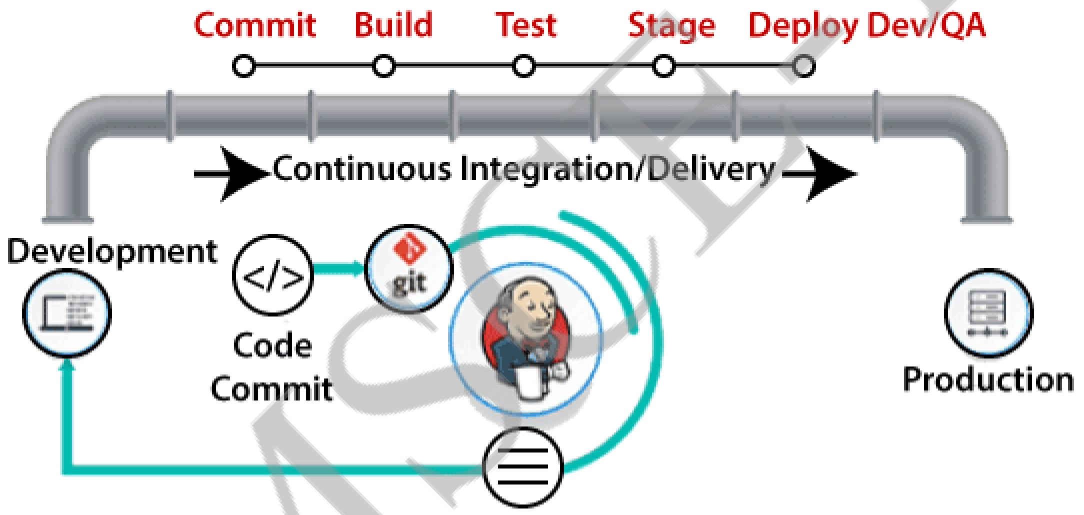
Learning DevOps is not complete without understanding the DevOps lifecycle phases. The DevOps lifecycle includes seven phases as given below:

1) Continuous Development

This phase involves the planning and coding of the software. The vision of the project is decided during the planning phase. And the developers begin developing the code for the application. There are no DevOps tools that are required for planning, but there are several tools for maintaining the code.

2) Continuous Integration

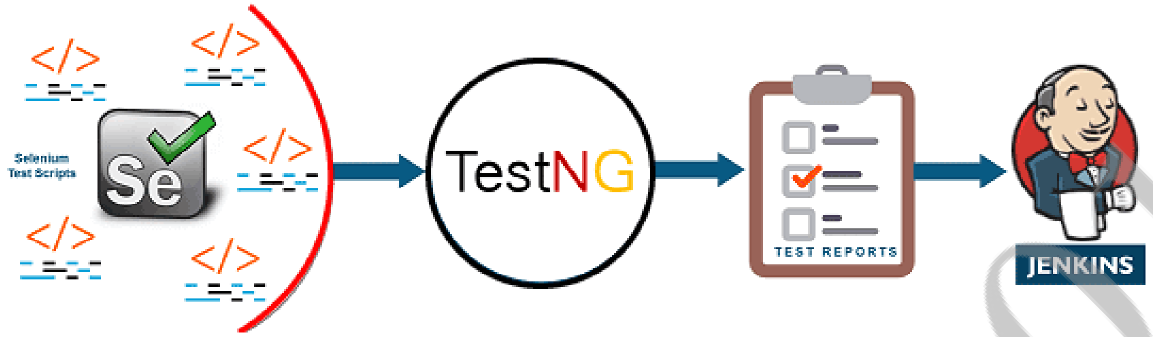
This stage is the heart of the entire DevOps lifecycle. It is a software development practice in which the developers require to commit changes to the source code more frequently. This may be on a daily or weekly basis. Then every commit is built, and this allows early detection of problems if they are present. Building code is not only involved compilation, but it also includes **unit testing, integration testing, code review**, and **packaging**.
The code supporting new functionality is continuously integrated with the existing code. Therefore, there is continuous development of software. The updated code needs to be integrated continuously and smoothly with the systems to reflect changes to the end-users.



Jenkins is a popular tool used in this phase. Whenever there is a change in the Git repository, then Jenkins fetches the updated code and prepares a build of that code, which is an executable file in the form of war or jar. Then this build is forwarded to the test server or the production server.

3) Continuous Testing

This phase, where the developed software is continuously testing for bugs. For constant testing, automation testing tools such as **TestNG, JUnit, Selenium**, etc are used. These tools allow QAs to test multiple code-bases thoroughly in parallel to ensure that there is no flaw in the functionality. In this phase, **Docker** Containers can be used for simulating the test environment.



Selenium does the automation testing, and TestNG generates the reports. This entire testing phase can automate with the help of a Continuous Integration tool called **Jenkins**.
Automation testing saves a lot of time and effort for executing the tests instead of doing this manually. Apart from that, report generation is a big plus. The task of evaluating the test cases that failed in a test suite gets simpler. Also, we can schedule the execution of the test cases at predefined times. After testing, the code is continuously integrated with the existing code.

4) Continuous Monitoring

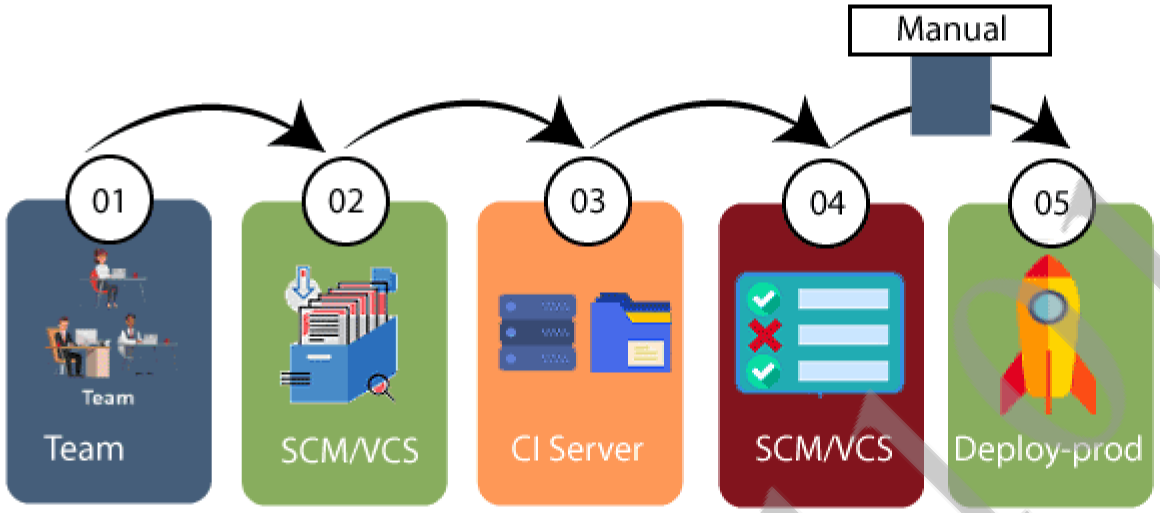
Monitoring is a phase that involves all the operational factors of the entire DevOps process, where important information about the use of the software is recorded and carefully processed to find out trends and identify problem areas. Usually, the monitoring is integrated within the operational capabilities of the software application.
It may occur in the form of documentation files or maybe produce large-scale data about the application parameters when it is in a continuous use position. The system errors such as server not reachable, low memory, etc are resolved in this phase. It maintains the security and availability of the service.

5) Continuous Feedback

The application development is consistently improved by analyzing the results from the operations of the software. This is carried out by placing the critical phase of constant feedback between the operations and the development of the next version of the current software application.
The continuity is the essential factor in the DevOps as it removes the unnecessary steps which are required to take a software application from development, using it to find out its issues and then producing a better version. It kills the efficiency that may be possible with the app and reduce the number of interested customers.

6) Continuous Deployment

In this phase, the code is deployed to the production servers. Also, it is essential to ensure that the code is correctly used on all the servers.



The new code is deployed continuously, and configuration management tools play an essential role in executing tasks frequently and quickly. Here are some popular tools which are used in this phase, such as **Chef, Puppet, Ansible**, and **SaltStack**.
Containerization tools are also playing an essential role in the deployment phase. **Vagrant** and **Docker** are popular tools that are used for this purpose. These tools help to produce consistency across development, staging, testing, and production environment. They also help in scaling up and scaling down instances softly.
Containerization tools help to maintain consistency across the environments where the application is tested, developed, and deployed. There is no chance of errors or failure in the production environment as they package and replicate the same dependencies and packages used in the testing, development, and staging environment. It makes the application easy to run on different computers.

7) Continuous Operations

All DevOps operations are based on the continuity with complete automation of the release process and allow the organization to accelerate the overall time to market continuingly.
It is clear from the discussion that continuity is the critical factor in the DevOps in removing steps that often distract the development, take it longer to detect issues and produce a better version of the product after several months. With DevOps, we can make any software product more efficient and increase the overall count of interested customers in your product.

DevOps Pipeline

A pipeline in software engineering team is a set of automated processes which allows DevOps professionals and developer to reliably and efficiently compile, build, and deploy their code to their production compute platforms.
The most common components of a pipeline in DevOps are build automation or continuous integration, test automation, and deployment automation.
A pipeline consists of a set of tools which are classified into the following categories such as:

- Source control
- Build tools
- Containerization
- Configuration management
- Monitoring

Continuous Integration Pipeline

Continuous integration (CI) is a practice in which developers can check their code into a version-controlled repository several times per day. Automated build pipelines are triggered by these checks which allows fast and easy to locate error detection.
Some significant benefits of CI are:

- Small changes are easy to integrate into large codebases.
- More comfortable for other team members to see what you have been working.
- Fewer integration issues allowing rapid code delivery.
- Bugs are identified early, making them easier to fix, resulting in less debugging work.

AD

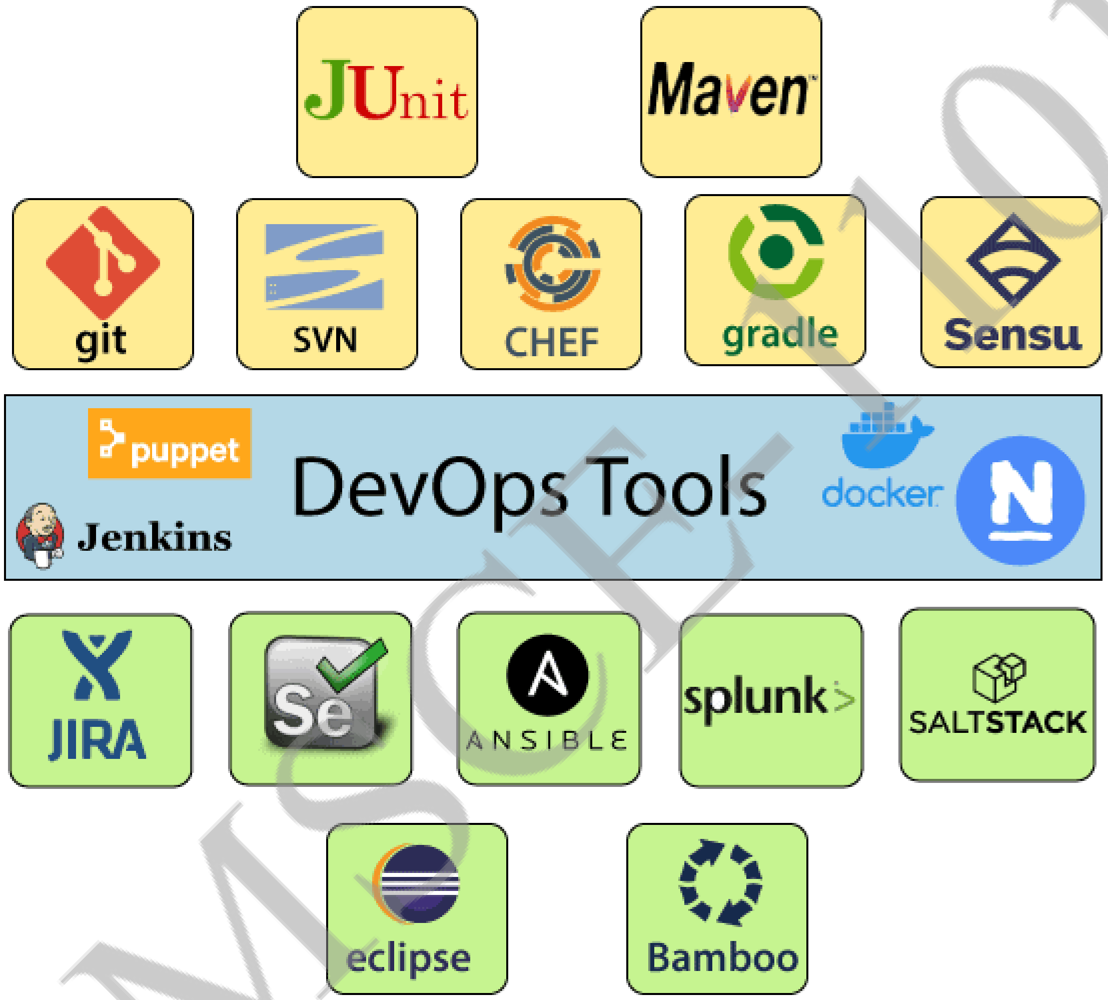
Continuous Delivery Pipeline

Continuous delivery (CD) is the process that allows operation engineers and developers to deliver bug fixes, features, and configuration change into production reliably, quickly, and sustainably. Continuous delivery offers the benefits of code delivery pipelines, which are carried out that can be performed on demand.
Some significant benefits of the CD are:

- Faster bug fixes and features delivery.
- CD allows the team to work on features and bug fixes in small batches, which means user feedback received much quicker. It reduces the overall time and cost of the project.

DevOps Tools

Here are some most popular DevOps tools with brief explanation shown in the below image, such as:



1) Puppet

Puppet is the most widely used DevOps tool. It allows the delivery and release of the technology changes quickly and frequently. It has features of versioning, automated testing, and continuous delivery. It enables to manage entire infrastructure as code without expanding the size of the team.

Features

- Real-time context-aware reporting.
- Model and manage the entire environment.
- Defined and continually enforce infrastructure.
- Desired state conflict detection and remediation.

- It inspects and reports on packages running across the infrastructure.
- It eliminates manual work for the software delivery process.
- It helps the developer to deliver great software quickly.

2) Ansible

Ansible is a leading DevOps tool. Ansible is an open-source IT engine that automates application deployment, cloud provisioning, intra service orchestration, and other IT tools. It makes it easier for DevOps teams to scale automation and speed up productivity.
Ansible is easy to deploy because it does not use any **agents** or **custom security** infrastructure on the client-side, and by pushing modules to the clients. These modules are executed locally on the client-side, and the output is pushed back to the Ansible server.
Features

- It is easy to use to open source deploy applications.
- It helps in avoiding complexity in the software development process.
- It eliminates repetitive tasks.
- It manages complex deployments and speeds up the development process.

3) Docker

Docker is a high-end DevOps tool that allows building, ship, and run distributed applications on multiple systems. It also helps to assemble the apps quickly from the components, and it is typically suitable for container management.
Features

- It configures the system more comfortable and faster.
- It increases productivity.
- It provides containers that are used to run the application in an isolated environment.
- It routes the incoming request for published ports on available nodes to an active container. This feature enables the connection even if there is no task running on the node.
- It allows saving secrets into the swarm itself.

4) Nagios

Nagios is one of the more useful tools for DevOps. It can determine the errors and rectify them with the help of network, infrastructure, server, and log monitoring systems.
Features

- It provides complete monitoring of desktop and server operating systems.
- The network analyzer helps to identify bottlenecks and optimize bandwidth utilization.
- It helps to monitor components such as services, application, OS, and network protocol.
- It also provides complete monitoring of Java Management Extensions.

5) CHEF

A chef is a useful tool for achieving scale, speed, and consistency. The chef is a cloud-based system and open source technology. This technology uses Ruby encoding to develop essential building blocks such as recipes and cookbooks. The chef is used in infrastructure automation and helps in reducing manual and repetitive tasks for infrastructure management.
Chef has got its convention for different building blocks, which are required to manage and automate infrastructure.

Features

- It maintains high availability.
- It can manage multiple cloud environments.
- It uses popular Ruby language to create a domain-specific language.
- The chef does not make any assumptions about the current status of the node. It uses its mechanism to get the current state of the machine.

6) Jenkins

Jenkins is a DevOps tool for monitoring the execution of repeated tasks. Jenkins is a software that allows continuous integration. Jenkins will be installed on a server where the central build will take place. It helps to integrate project changes more efficiently by finding the issues quickly.

Features

- Jenkins increases the scale of automation.
- It can easily set up and configure via a web interface.
- It can distribute the tasks across multiple machines, thereby increasing concurrency.
- It supports continuous integration and continuous delivery.
- It offers 400 plugins to support the building and testing any project virtually.
- It requires little maintenance and has a built-in GUI tool for easy updates.

7) Git

Git is an open-source distributed version control system that is freely available for everyone. It is designed to handle minor to major projects with speed and efficiency. It is developed to co-ordinate the work among programmers. The version control allows you to track and work together with your team members at the same workspace. It is used as a critical distributed version-control for the DevOps tool.

- It is a free open source tool.
- It allows distributed development.
- It supports the pull request.
- It enables a faster release cycle.
- Git is very scalable.
- It is very secure and completes the tasks very fast.

8) SALTSTACK

Stackify is a lightweight DevOps tool. It shows real-time error queries, logs, and more directly into the workstation. SALTSTACK is an ideal solution for intelligent orchestration for the software-defined data center.

Features

- It eliminates messy configuration or data changes.
- It can trace detail of all the types of the web request.
- It allows us to find and fix the bugs before production.
- It provides secure access and configures image caches.
- It secures multi-tenancy with granular role-based access control.
- Flexible image management with a private registry to store and manage images.

9) Splunk

Splunk is a tool to make machine data usable, accessible, and valuable to everyone. It delivers operational intelligence to DevOps teams. It helps companies to be more secure, productive, and competitive.

Features

- It has the next-generation monitoring and analytics solution.
- It delivers a single, unified view of different IT services.
- Extend the Splunk platform with purpose-built solutions for security.
- Data drive analytics with actionable insight.

10) Selenium

Selenium is a portable software testing framework for web applications. It provides an easy interface for developing automated tests.

Features

- It is a free open source tool.
- It supports multiplatform for testing, such as Android and ios.
- It is easy to build a keyword-driven framework for a WebDriver.
- It creates robust browser-based regression automation suites and tests.
-