Chapter 1 : Basics of Software Engineering

1.1 Software, software engineering as layered approach, characteristics of software, types of software

Software is a program or set of programs containing instructions that provide the desired functionality. Engineering is the process of designing and building something that serves a particular purpose and finds a cost-effective solution to problems.

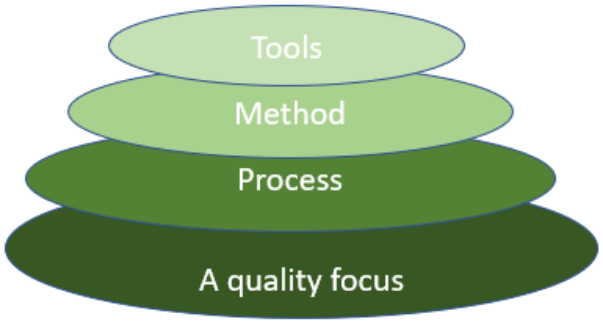
Software = Instructions + Data Structures + Documents

Engineering → A branch of science & technology focused on designing, building, and using engines, machines, and structures.
It applies science, tools, and methods to find cost-effective solutions to simple and complex problems.

Software Engineering → A systematic, disciplined, and measurable approach to develop, operate, and maintain software.

Layered Technology in Software Engineering

Software engineering is a fully layered technology.
To develop software, we must move step-by-step from one layer to the next.
All layers are connected, and each one depends on the successful completion of the previous layer.



Layers of Software Engineering

Layered technology is divided into four main parts:

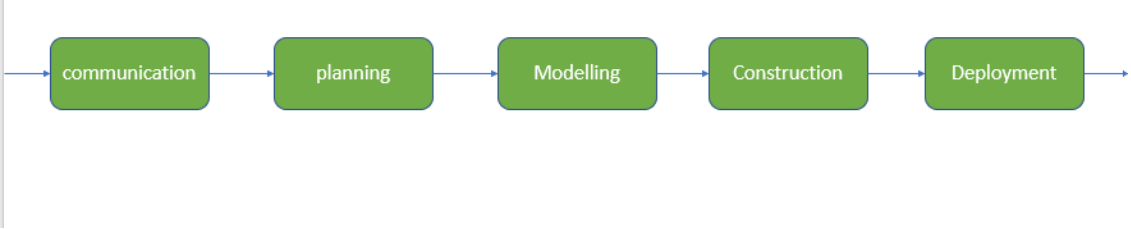
1. Quality Focus

This layer ensures continuous improvement in the software process.
It maintains integrity, meaning the software is secure — only authorized people can access data, and no outsider can misuse it.
It also focuses on:

- Maintainability – making sure the software can be easily updated or fixed.
- Usability – making the software easy for users to operate.

2. Process *(Foundation Layer)*

This is the base of software engineering.
It is the key that connects all layers together and helps in developing software on time.
The process defines a framework that must be followed for the effective delivery of software.
It includes all activities, actions, and tasks required in development.



Main Process Activities:

- Communication: The first and most important step. Developers talk with the client to understand the actual requirements.
- Planning: Creating a roadmap or plan to make development easier and avoid complications.
- Modeling: Making a model or design according to the client's needs for better understanding.
- Construction: Writing the code and testing it.
- Deployment: Delivering the final software to the client for evaluation and feedback.

3. Method

In the development process, methods answer all the "how to do" questions.
Methods provide guidelines and information for every task, such as:

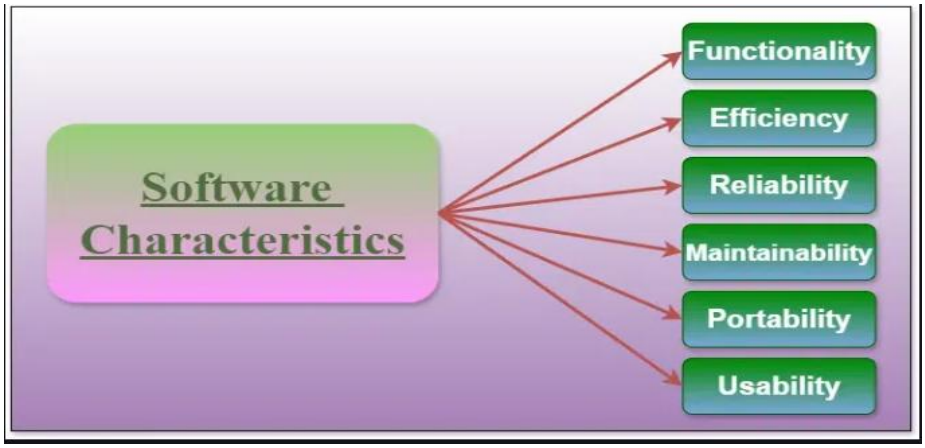
- Communication
- Requirement analysis
- Design modeling
- Program construction
- Testing
- Support

4. Tools

Software engineering tools give automatic support to processes and methods.
They are integrated, meaning the information created by one tool can be used directly by another tool.

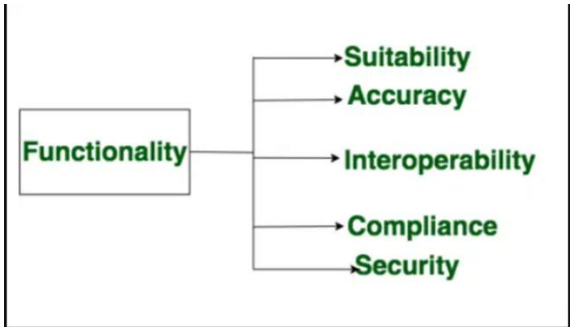
<u>Components of Software Characteristics</u>

There are 6 components of Software Characteristics.



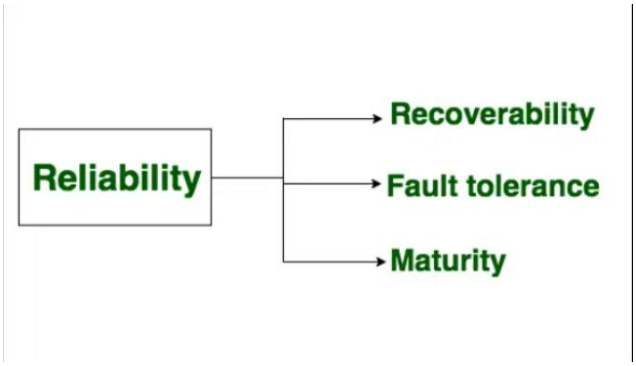
1. Functionality

- Measures how well software performs its intended purpose.
- Includes features like:
    - Data storage & retrieval
    - Data processing
    - User interface & navigation
    - Security & access control
    - Reporting & automation
- Note: More functionality = more powerful but can also be more complex. Must balance with ease of use & maintainability.



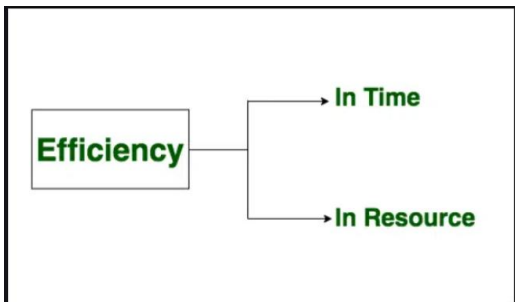
2. Reliability

- Ability of software to perform correctly and consistently over time.
- Affected by bugs, poor testing, bad design, lack of error handling, or incompatibility.
- Reliable software has a low failure rate and recovers quickly if it fails.



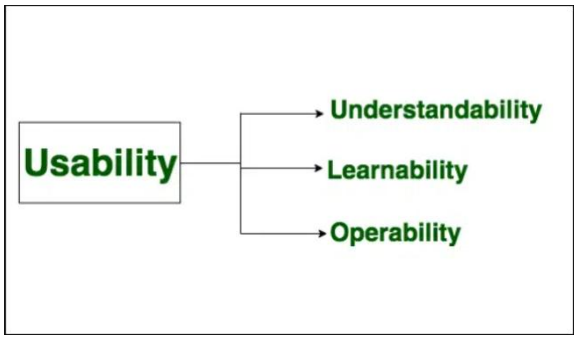
3. Efficiency

- How well software uses system resources (memory, CPU, bandwidth).
- High efficiency = fast performance + minimal resource use.
- Affected by poor algorithms, unoptimized code, unnecessary processing.
- Important for real-time, high-performance, and resource-limited systems.



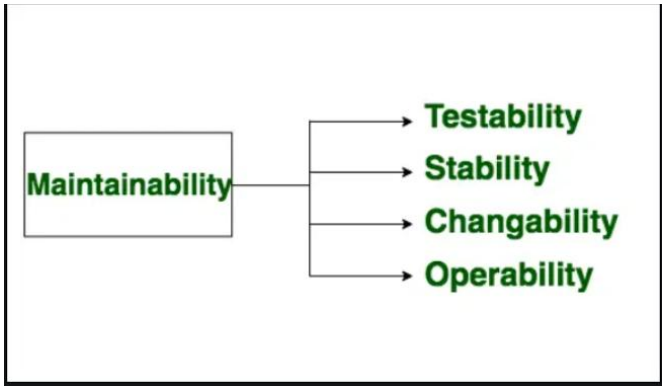
4. Usability

- How easy the software is to learn and use.
- Lower learning time → higher usability.



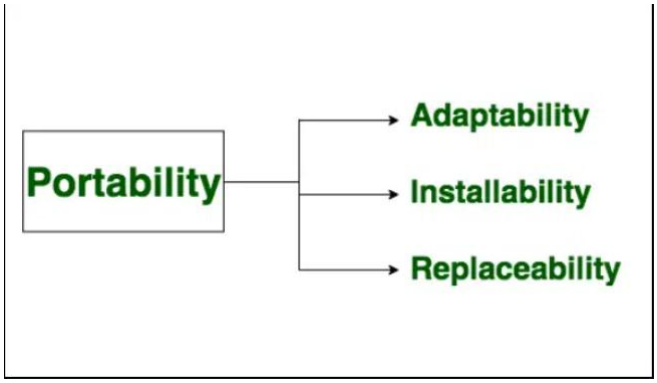
5. Maintainability

- Ease of making changes to fix errors, improve performance, or add features.
- Good maintainability reduces long-term cost and effort.



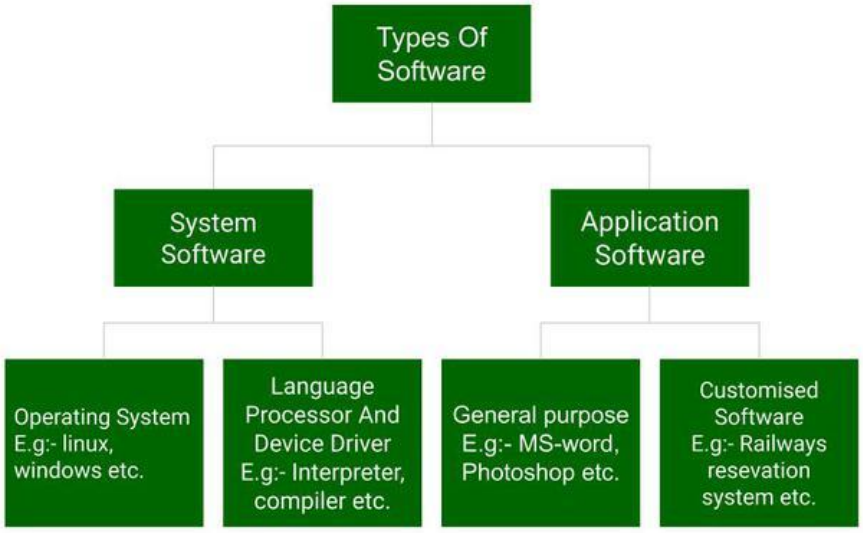
6. Portability

- Ability to transfer software from one environment to another with minimal changes.
- High portability = works on multiple platforms easily.



Types of Software

It is a collection of data that is given to the computer to complete a particular task. The chart below describes the types of software:



Above is the diagram of types of software. Now we will briefly describe each type and its subtypes:

1. System Software
    - Operating System
    - Language Processor
    - Device Driver
2. Application Software
    - General Purpose Software
    - Customize Software
    - Utility Software

System Software

System software is the software that directly operates the computer hardware and provides basic functions to both the user and other software so they can work smoothly.

In simple words, system software:

- Controls a computer’s internal functioning.
- Controls hardware devices like monitors, printers, and storage devices.
- Works as an interface between hardware and user applications.

Since hardware understands machine language (0s and 1s) and applications work in human-readable languages (English, Hindi, German, etc.), system software converts human language into machine language and vice versa.

Types of System Software

1. Operating System (OS)
    - Main program of a computer system.
    - First software to load when the system is switched ON.
    - Manages all resources (memory, CPU, printer, hard disk).
    - Provides an interface for user interaction.
    - Gives services to other software.
    - Examples: Linux, macOS, Windows.
2. Language Processor
    - Converts high-level programming languages (Java, C, C++, Python) into machine code.
    - High-level code → Source Code; Machine-readable code → Object Code.
3. Device Driver
    - Special program that controls hardware devices (printer, mouse, modem).
    - Needed for the operating system to communicate with the device.
    - Must be installed when a new device is connected.

Features of System Software

- Closer to the hardware.
- Usually written in low-level language.
- Difficult to design and understand.
- Works at high speed.
- Less interactive compared to application software.

Application Software

Application software performs specialized functions beyond the basic operations of a computer. It is designed for specific tasks to fulfill end-user needs.

Examples: Word processors, spreadsheets, database systems, payroll software.

Types of Application Software

1. General Purpose Software
    - Can perform many tasks, not limited to one purpose.
    - Examples: MS Word, MS Excel, PowerPoint.

2. Customized Software
    - Designed for a specific task or organization.
    - Examples: Railway reservation system, invoice management system.
3. Utility Software
    - Supports and maintains the computer system.
    - Functions: Analyze, configure, optimize, repair.
    - Examples: Antivirus, disk cleaner, memory tester, disk repair tools.

Features of Application Software

- Performs specialized tasks (word processing, email, spreadsheets).
- Usually large in size → requires more storage space.
- More interactive and user-friendly.
- Easier to design and understand.
- Generally written in high-level language.

Difference Between System Software and Application Software

System Software	Application Software
It is designed to manage the resources of the computer system, like memory and process management, etc.	It is designed to fulfill the requirements of the user for performing specific tasks.
Written in a low-level language.	Written in a high-level language.
Less interactive for the users.	More interactive for the users.
System software plays vital role for the effective functioning of a system.	Application software is not so important for the functioning of the system, as it is task specific.
It is independent of the application software to run.	It needs system software to run.

1.2 Software development framework: Software generic process framework activities and umbrella activities

Software Process Framework

A Software Process Framework is a structured way of developing software.
It defines the steps, tasks, and activities that guide the team during development.
The framework provides a foundation for software engineering, ensuring that the process is systematic, efficient, and consistent.

It helps in:

- Project planning (deciding what to do and when),
- Risk management (handling possible problems), and
- Quality assurance (maintaining product quality).

By following a process framework, teams can improve productivity, consistency, and quality of the final software.

What is a Software Process Framework?

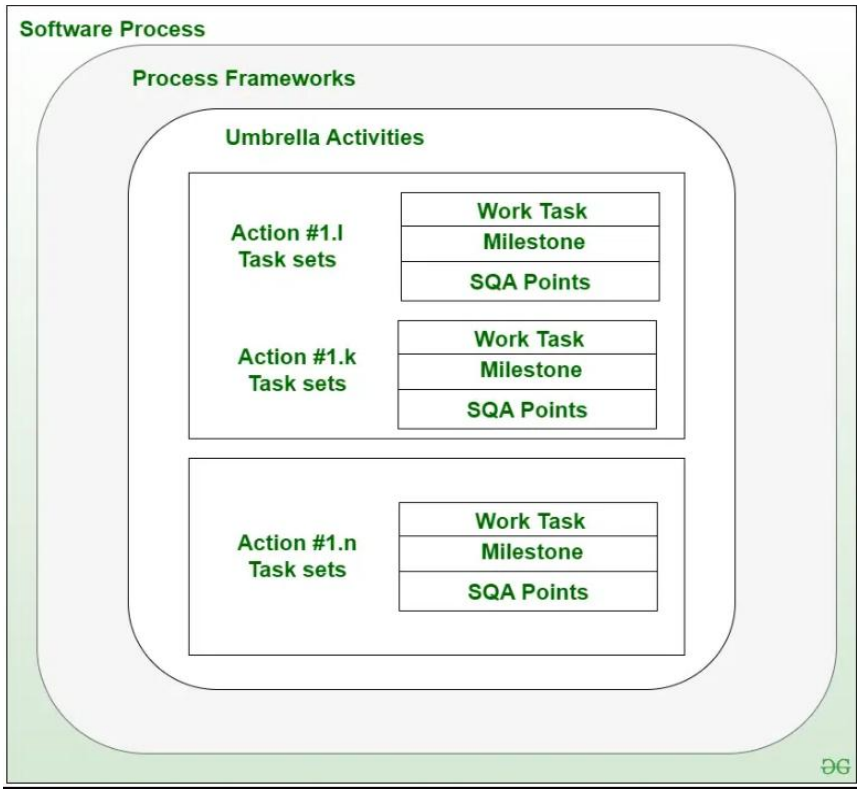
A software process framework gives the detailed order of steps in software development.
It acts as a base structure for most software projects.

It consists of:

- Task sets – small, focused goals.
- Umbrella activities – supporting activities that span the whole process (e.g., quality assurance, project management).
- Process framework activities – main development activities that form the backbone of the software lifecycle.

Elements of the Software Process

1. Tasks → Very small units of work, focusing on a single objective.
   Example: Writing a function, preparing test cases.
2. Action → A collection of related tasks that produce a major work product.
   Example: Coding module + reviewing + testing = completed software component.
3. Activities → A larger group of related actions and tasks, aimed at a big objective.
   Example: "Design activity" includes actions like architectural design, database design, and interface design.



What is a Software Development Framework?

A Software Development Framework is a ready-made structure that provides tools, libraries, best practices, and guidelines for building software applications.

It acts like a template or foundation that gives developers a starting point, so they don't need to build everything from scratch.

A framework usually includes:

- Pre-built components (functions, UI elements, libraries),
- Development tools, and
- Standard rules/guidelines to follow.

This helps developers build software in a more organized, faster, and efficient way. Developers can also customize the framework to fit their project's needs.

Key Points

1. Foundation → Provides a ready template/structure for software development.
2. Components & Tools → Includes built-in tools and reusable components.
3. Best Practices → Offers guidelines to ensure efficient and organized coding.
4. Customization → Can be adapted and extended for specific project needs.

Advantages of Software Development Framework

1. Increased Productivity → Developers focus on core logic since many components are already available.
2. Consistent Quality → Enforces coding standards and best practices.
3. Reduced Development Time → Ready-made libraries save time compared to starting from scratch.
4. Better Maintainability → Codebase remains well-structured and easier to update.
5. Enhanced Security → Most frameworks come with built-in security features.
6. Scalability → Designed to handle growth and increasing user demands.

Disadvantages of Software Development Framework

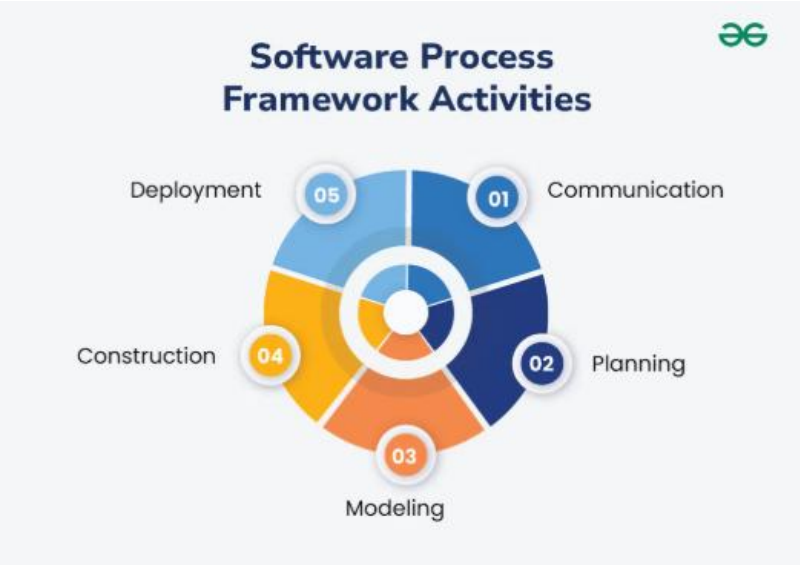
1. Learning Curve → Requires time and effort to understand framework rules and structure.
2. Restrictions → Some frameworks limit flexibility in design and implementation.
3. Complexity Overhead → Can be too heavy for small/simple projects.
4. Performance Overhead → Extra layers may reduce performance in resource-heavy apps.
5. Vendor Lock-in → Relying too much on one framework makes switching harder in the future.

Software Process Framework Activities

The **Software Process Framework** represents the **common set of activities** that are followed in almost all software projects.
In Software Engineering, there are **five main framework activities**:

Communication, Planning, Modeling, Construction, and Deployment.

Each activity produces specific **work outputs**, sets **milestones**, and includes **Software Quality Assurance (SQA) checks**.



1. Communication

- **Definition:** Communication means interacting with customers and stakeholders to understand the system's goals and requirements.
- **Activities:**
    - *Requirement Gathering* → Collecting needs through meetings, interviews, or surveys.
    - *Objective Setting* → Clearly defining what the software should achieve.
- **Explanation:** Good communication ensures that everyone has the same understanding of what is to be built. Without this, the project can fail due to unclear or wrong requirements.

2. Planning

- **Definition:** Planning focuses on preparing a **roadmap** for the project. It includes tasks, risks, resources, and schedules.
- **Activities:**
    - *Work Plan* → Listing tasks and activities for development.
    - *Risk Assessment* → Identifying possible risks and planning how to reduce them.
    - *Resource Allocation* → Assigning people, tools, and time.
    - *Schedule Definition* → Creating a timeline for each phase.
- **Explanation:** Planning helps organize the project, avoid surprises, and ensures that the team knows what to do and when.

3. Modeling

- **Definition:** Modeling means creating **designs and models** to represent how the system will work.
- **Activities:**
    - *Requirement Analysis* → Breaking down requirements into detailed specifications.
    - *Design* → Making architectural and detailed designs (like database design, UI design, etc.).
- **Explanation:** Modeling converts requirements into a **blueprint** for the developers. It helps in visualizing the system before actually building it.

4. Construction

- **Definition:** Construction is where the **real coding** and **testing** take place.
- **Activities:**
    - *Code Generation* → Writing code as per the design.
    - *Testing* → Checking the software, finding bugs, and fixing them.

- **Explanation:** This is the actual development phase. Testing is very important because it ensures that the software works correctly and meets the requirements.

5. Deployment

- **Definition:** Deployment is the process of **delivering the product** to customers and taking feedback.
- **Activities:**
    - *Product Release* → Handing over the software to users (final or partial version).
    - *Feedback Collection* → Asking users about their experience.
    - *Product Improvement* → Making necessary changes based on feedback.
- **Explanation:** Deployment makes the software available for real use. Feedback ensures continuous improvement.

Popular Software Development Frameworks

Some commonly used frameworks in industry are:

- **Frontend frameworks:** Angular, React, Vue.js
- **Backend frameworks:** Django, Flask, Express, Spring, Ruby on Rails, Laravel, ASP.NET Core

Umbrella Activities in Software Engineering

Definition

Umbrella activities are supporting activities that occur throughout the software development process.

They do not belong to one specific phase but span across all phases of the Software Development Life Cycle (SDLC).

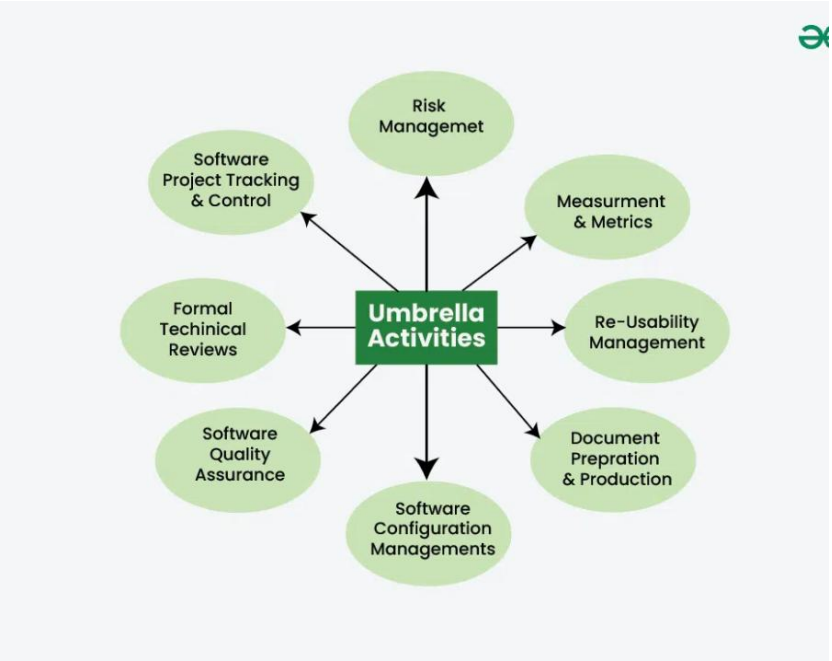
Their main purpose is to help in project management, quality control, risk handling, and tracking progress so that software development becomes more efficient and reliable.

These activities “cover” (like an umbrella) the entire process to ensure that the project stays on track and delivers quality results.

Umbrella activities are supportive activities that occur throughout the software development process. They do not belong to any one phase, but instead run across all framework activities to improve project management, quality, and tracking.

1. **Software Project Tracking and Control**
    - This activity monitors the project’s progress by comparing actual work with the project plan.
    - If any delay or deviation is found, corrective actions are taken to bring the project back on track.
    - It ensures that the project remains on time and within budget.

2. **Risk Management**
    - Risk management identifies possible risks that may negatively affect the project.
    - These risks are analyzed and strategies are prepared to reduce their impact.
    - It ensures better planning for uncertainties.
3. **Software Quality Assurance (SQA)**
    - Activities are performed to guarantee that the developed software meets defined quality standards.
    - This helps to maintain reliability, correctness, and performance of the product.
4. **Formal Technical Reviews (FTRs)**
    - Reviews are conducted at different stages to detect and correct errors early.
    - These reviews improve the quality of design, code, and documentation.
    - Errors are removed before they spread to later stages, saving time and cost.
5. **Software Configuration Management (SCM)**
    - This activity manages and controls changes in the software.
    - It ensures that updates, bug fixes, and enhancements are properly recorded and integrated.
6. **Work Product Preparation and Production**
    - In this activity, documents, models, reports, forms, and logs are created and maintained.
    - It ensures proper documentation and communication throughout the project.
7. **Reusability Management**
    - This defines guidelines for reusing existing software components or work products.
    - Reusable modules are stored and maintained for use in future projects, saving effort and cost.
8. **Measurement**
    - Data about the process, project, and product are collected and analyzed.
    - These measurements help in improving the process and assist the team in delivering high-quality software.

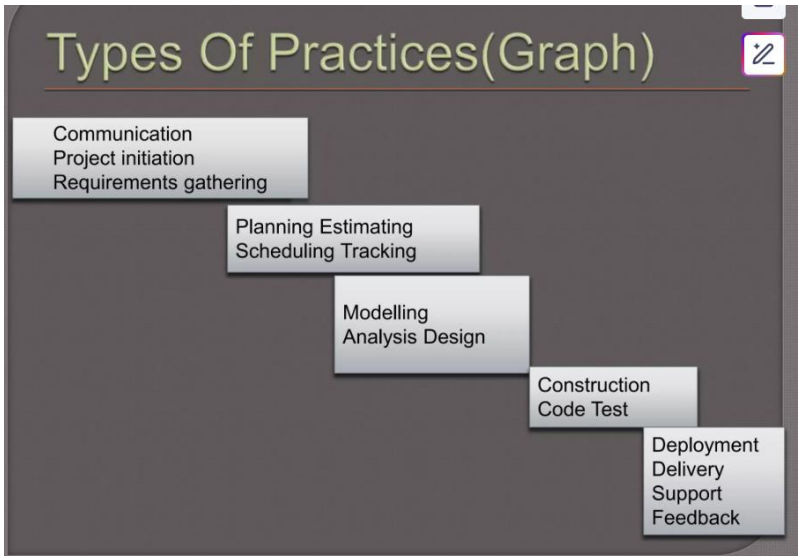


1.3 Software engineering core principles, communication practices, planning practices, modelling practices, construction practices, software deployment practices

Software engineering is guided by a set of **core principles** and **practices** that ensure the development of reliable, maintainable, and high-quality software. These principles act as a foundation, while the practices define how different phases of software development should be carried out.

Core Principles of Software Engineering

1. **Modularity** – Break software into small, independent modules. This makes it easier to test, maintain, and reuse.
2. **Abstraction** – Show only necessary details while hiding the internal implementation. This increases flexibility.
3. **Encapsulation** – Bind data and functions together, allowing controlled access. This protects against errors.
4. **DRY (Don't Repeat Yourself)** – Avoid duplication of code and data. Helps in reducing errors and maintenance effort.
5. **KISS (Keep It Simple, Stupid)** – Keep the design simple and clear for easy understanding and testing.
6. **YAGNI (You Aren't Gonna Need It)** – Avoid adding unnecessary features that are not required.
7. **SOLID Principles** – A set of five object-oriented design principles (Single Responsibility, Open/Closed, Liskov Substitution, Interface Segregation, Dependency Inversion) that improve reusability and maintainability.
8. **Test-Driven Development (TDD)** – Write tests before writing code and ensure code passes all tests.
9. **Reliability and Maintainability** – All principles aim to make the software reliable, extensible, and easier to maintain.



Types of Practices in Software Engineering

Software engineering applies specific practices across its major phases:

- **Communication Practices**
- **Planning Practices**
- **Modeling Practices (Analysis & Design)**
- **Construction Practices (Before & During Coding)**
- **Testing Practices**
- **Deployment Practices**

1. Communication Practices

Good communication ensures mutual understanding between developers, customers, and stakeholders.

- Listen actively and focus on the speaker.
- Prepare before meetings and use agendas.
- Prefer face-to-face communication, supported by documents.
- Take notes and record decisions.
- Encourage collaboration and consensus.
- Use diagrams/pictures when ideas are unclear.
- Negotiation should be win-win, not competitive.

2. Planning Practices

Planning provides a roadmap for the project and helps manage resources, risks, and schedules.

- Understand the scope of the project.
- Involve the customer in planning.
- Accept that planning is iterative and may change.
- Base estimates on known facts.
- Consider risks while preparing the plan.
- Be realistic about productivity.
- Define how quality will be ensured and how changes will be handled.
- Track progress regularly and update the plan when needed.

3. Modeling Practices

(a) Analysis Modeling

- Represent the information domain (input/output data).
- Define the required functions.
- Represent system behavior in response to external events.
- Partition models into layers for clarity.
- Move from essential information to implementation details gradually.

(b) Design Modeling

- Ensure design is traceable to analysis.
- Focus on overall architecture.
- Design both data and functions.
- Carefully design internal and external interfaces.
- User interface must be simple and user-friendly.
- Components should be highly cohesive and loosely coupled.
- Representations must be easy to understand.
- Develop design iteratively, aiming for simplicity at each step.

4. Construction Practices

Before Coding

- Understand the problem and basic design principles.
- Select a suitable programming language and tools.
- Prepare unit tests before starting.

During Coding

- Follow structured programming practices.
- Choose proper data structures.
- Maintain consistency with architecture.
- Keep logic simple and clear.
- Use meaningful names and coding standards.
- Write self-documenting code.
- Use proper indentation and formatting for readability.

5. Testing Practices

(You didn't provide this explicitly, but it's usually included. I'll keep it short for exams.)

- Conduct unit testing for individual modules.

- Perform integration testing to ensure modules work together.
- Carry out system and acceptance testing with users.
- Automate tests where possible.

6. Deployment Practices

Deployment delivers the software to customers and ensures its usability.

- Manage customer expectations (do not overpromise).
- Deliver a complete, tested package.
- Provide user manuals and training materials.
- Establish support and maintenance plans.
- Fix bugs before release to avoid dissatisfaction.

1.4 Prescriptive process models: Waterfall model, incremental model, RAD model, prototyping model, spiral model

Software Development Models
These are different methods or processes used for project development, selected based on the project's goals and requirements. Many life cycle models exist to achieve important objectives. Each model defines the steps of development and the sequence in which they are carried out.

Software Modeling
It is the process of making abstract representations of a software system. These models act as blueprints that help developers, designers, and stakeholders understand the system's structure, behavior, and functions.

Software Process Model

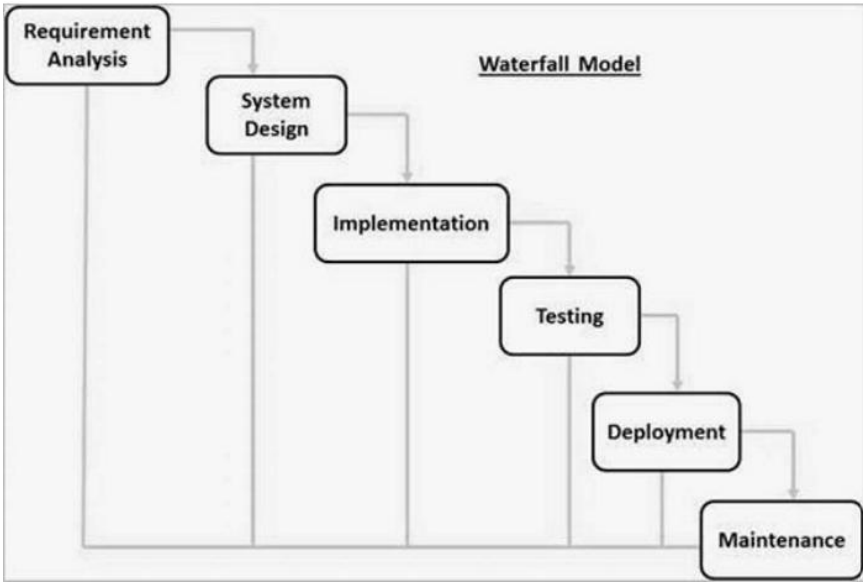
A software process model is a defined way of representing a software process from a specific viewpoint.
Since models are simplified versions, a software process model is an abstraction (simplification) of the real process.
It usually includes:

- Activities involved in software development,
- The software product, and
- Roles of the people working in software engineering.

1. The Waterfall Model

- Used when project requirements are clear from the start.
- Also called the **classic life cycle model**.
- Follows a **step-by-step, systematic, and sequential approach** to software development.

- Starts with customer requirements and moves through **planning, modeling, construction, and deployment**.



Phases in the Waterfall Model:

1. **Requirement Gathering & Analysis**
    - Collects all possible system requirements.
    - Documented in a **Requirement Specification Document**.
2. **System Design**
    - Requirements are studied and system design is prepared.
    - Helps define **hardware and software requirements**.
    - Provides the **overall system architecture**.
3. **Implementation**
    - Based on design, the system is developed in **small units (programs)**.
    - Each unit is tested for functionality (**Unit Testing**).
4. **Integration & Testing**
    - All tested units are integrated into a complete system.
    - The full system is tested for **errors and failures**.
5. **Deployment**
    - After functional and non-functional testing, the product is delivered to the **customer environment** or released in the market.
6. **Maintenance**
    - Issues faced by clients are fixed with **patches**.

- Improved versions with enhancements are released.
- Maintenance ensures smooth functioning in the customer environment.

👉 In this model, each phase **flows into the next like a waterfall**.
👉 A new phase begins **only after** the previous one is completed and approved.
👉 Phases do **not overlap**.

Problems in the Waterfall Model

- Real projects rarely follow a strict sequence; they are usually **iterative**.
- It demands **clear requirements at the beginning**, which is often hard to achieve.
- A **working system is available only at the later stages**, which delays feedback.

2. Increment Process Models

Iterative Enhancement Model

- In this model, development starts with **some specifications** and the **first version** of the software is built.
- If changes are needed after the first version, a **new version is created** in the next iteration.
- Each release is completed in a **fixed time period** called an **iteration**.
- Earlier phases can be revisited whenever changes are required.
- The **final output** is obtained at the end of the **SDLC process** after multiple iterations.

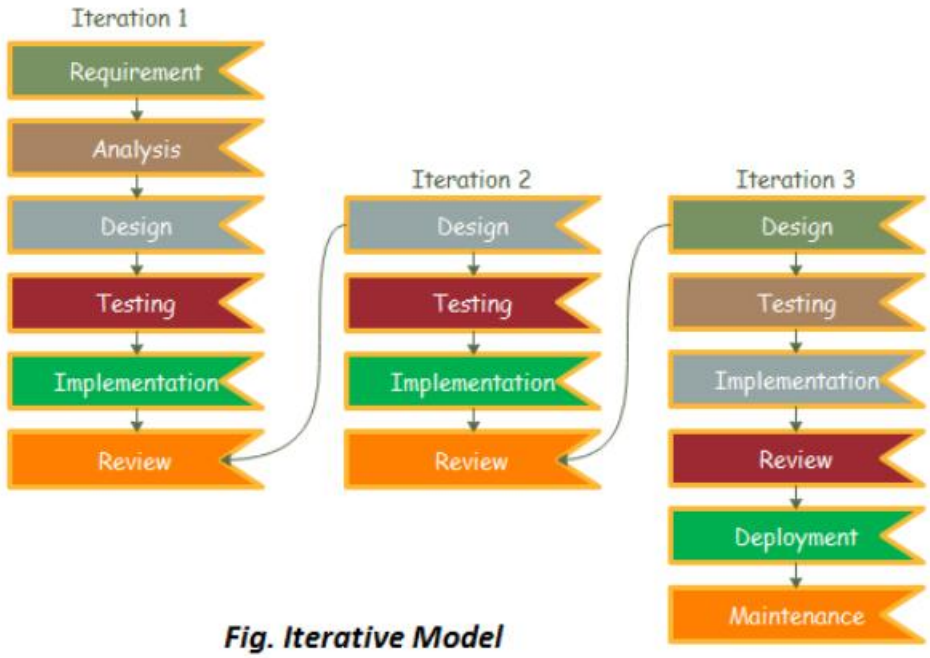


Fig. Iterative Model

Phases of Iterative Model

1. **Requirement Gathering & Analysis**

- Requirements are collected from customers.
- Analyst checks if requirements can be fulfilled within budget and scope.
- If acceptable, the team moves to the next phase.

2. **Design**
    - System design is prepared using diagrams like:
        - **Data Flow Diagram (DFD)**
        - **Activity Diagram**
        - **Class Diagram**
        - **State Transition Diagram**, etc.
3. **Implementation**
    - Requirements are converted into **code (programs)**.
    - This produces the software version.
4. **Testing**
    - After coding, the software is tested using methods such as:
        - **White Box Testing**
        - **Black Box Testing**
        - **Grey Box Testing**
5. **Deployment**
    - The tested software is deployed in the **working environment**.
6. **Review**
    - After deployment, the product is reviewed to check its **performance and validity**.
    - If errors are found, the process restarts from the **Requirement Phase**.
7. **Maintenance**
    - After deployment, bugs, errors, or new feature requests may arise.
    - Maintenance involves **debugging, updates, and enhancements**.

Advantages (Pros) of Iterative Model

1. Testing and debugging in **smaller iterations** is easier.
2. **Parallel development** can be planned.
3. Easily adapts to **changing project requirements**.
4. **Risks** are identified and resolved during iterations.

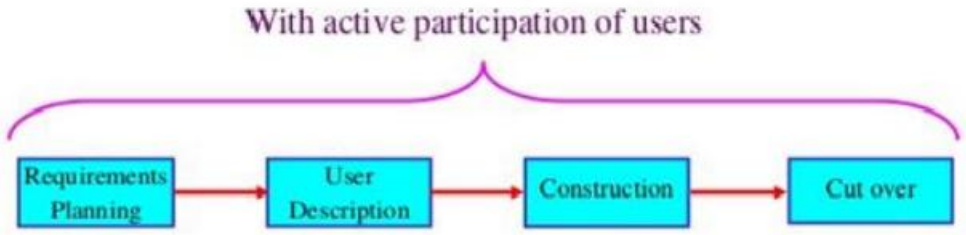
5. Less time on documentation, more time on **design and coding**.

Disadvantages (Cons) of Iterative Model

1. Not suitable for **small projects**.
2. May require **more resources** (time, people, tools).
3. Frequent requirement changes → **design may change repeatedly**.
4. Changing requirements can cause **budget issues**.
5. Final project completion date is **uncertain** due to ongoing changes.

3. Rapid Application Development (RAD) Model

- In the **RAD model**, functional modules are developed **in parallel as prototypes**.
- These prototypes are later integrated to form the **complete product**, enabling **faster delivery**.
- Since there is **no detailed pre-planning**, changes can be easily made during development.
- RAD focuses on **iterative and incremental delivery** of working models to the customer.
- Customers remain **involved throughout development**, which reduces the risk of mismatch with actual requirements.



Advantages (Pros) of RAD Model

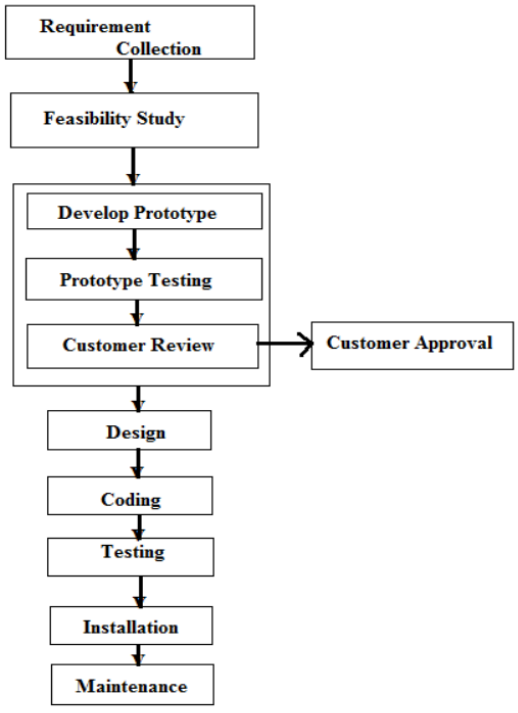
1. Changing requirements can be easily handled.
2. Project **progress is measurable** at each stage.
3. Iterations are shorter using **powerful RAD tools**.
4. Higher productivity with **fewer team members**.
5. **Faster development time** compared to other models.
6. Encourages **reuse of existing components**.
7. **Early reviews** of the product are possible.
8. Promotes **continuous customer feedback**.
9. **Integration from the beginning** reduces later integration issues.

Disadvantages (Cons) of RAD Model

1. Depends on **technically strong team members** to define business requirements.
2. Only systems that can be **modularized** can be developed using RAD.
3. Requires **highly skilled developers and designers**.
4. Heavy reliance on **modeling skills**.
5. Not suitable for **low-budget projects**, as modeling and automated code generation are expensive.
6. **Management complexity** is higher.
7. Best suited for **component-based and scalable systems**.
8. Needs **continuous user involvement** throughout the project.
9. More useful for projects needing **short development time**.

4.Prototyping Model

- A **prototype** is a **working model of software** with limited features.
- It may not contain the exact logic of the final software but is an **extra effort** considered in estimation.
- Purpose: Allows users to **evaluate developer proposals** and **test ideas** before full implementation.
- Helps in understanding **user-specific requirements** that developers might have missed during design.



Steps in Prototyping Model

1. **Basic Requirement Identification**
    - Focuses on understanding **basic product needs**, especially **user interface**.
    - Detailed aspects like performance and security are ignored at this stage.
2. **Developing the Initial Prototype**
    - A simple version of the system is developed.
    - Shows **basic requirements and UI design**.
    - Functions may not fully work internally, but provide a **look and feel** for the customer.
3. **Review of the Prototype**
    - The prototype is presented to **customers and stakeholders**.
    - Feedback is collected and analyzed for improvements.
4. **Revise and Enhance the Prototype**
    - Based on feedback, changes are discussed considering **time, cost, and feasibility**.
    - Accepted changes are included in the next version of the prototype.
    - The cycle repeats until customer expectations are met.

Types of Prototypes

- **Horizontal Prototype** → Focuses on the **user interface** and overall view of the system, without internal functions.
- **Vertical Prototype** → Gives a **detailed version of a specific function** or subsystem.

Advantages (Pros) of Prototyping Model

1. Users are **involved early** in development.
2. A working model helps users **understand the system better**.
3. Saves **time and cost** by detecting defects early.
4. **Quick feedback** leads to better solutions.
5. Missing functionalities can be identified.
6. Difficult or confusing functions can be detected early.

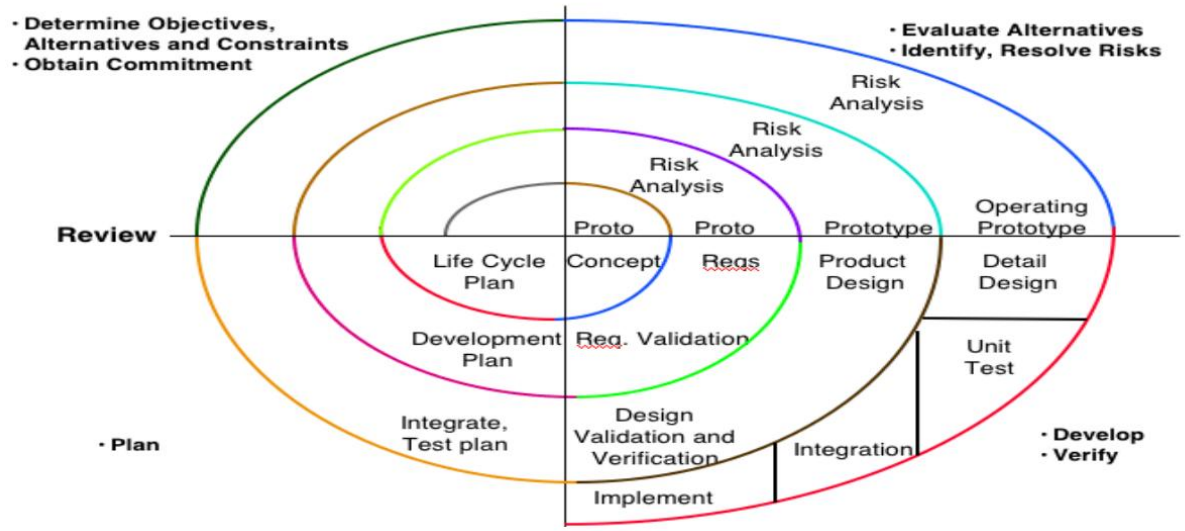
Disadvantages (Cons) of Prototyping Model

1. Risk of **poor requirement analysis** due to over-dependence on prototypes.
2. Users may get **confused between prototype and actual system**.

3. Can **increase system complexity** if scope keeps expanding.
4. Developers may wrongly **reuse prototypes** for final system, even if not feasible.
5. Building prototypes can require **high effort and cost** if not controlled.

5.Spiral Model

- The **Spiral Model** combines features of the **iterative model** and the **waterfall model**.
- Development progresses in **phases (iterations)** called **spirals**.
- After every spiral, customer feedback is taken and improvements are made in the next spiral.
- This cycle continues throughout the **life of the software**.



Phases of Spiral Model

1. **Identification**
    - Starts with gathering **business requirements** (baseline spiral).
    - In later spirals: system, subsystem, and unit requirements are identified.
    - Involves **continuous communication** between customer and system analyst.
    - At the end of spirals, the product is **deployed in the target market**.
2. **Design**
    - Starts with **conceptual design** in the baseline spiral.
    - Progresses to **architectural design**, **logical design**, **physical design**, and **final design** in later spirals.
3. **Construct / Build**
    - In baseline spiral → a **POC (Proof of Concept)** is developed for customer feedback.
    - In later spirals → **working builds (versions)** of the software are produced.

- Each build is sent to the customer for **review and suggestions**.

4. **Evaluation and Risk Analysis**
    - Focuses on **risk identification, estimation, and monitoring** (technical, cost, schedule risks).
    - After testing each build, customer provides **feedback**.
    - Helps in deciding the next steps and risk handling.

Advantages (Pros) of Spiral Model

1. Can easily accommodate **changing requirements**.
2. Allows **extensive use of prototypes**.
3. Requirements are captured more **accurately**.
4. Users see the system **early in development**.
5. Supports **risk management** by developing risky parts earlier.
6. Development is broken into **smaller manageable parts**.

Disadvantages (Cons) of Spiral Model

1. Project **management becomes complex**.
2. Project end date may not be known early.
3. Not suitable for **small or low-risk projects** (too costly for small projects).
4. The process itself is **complex to implement**.
5. Spiral may continue **indefinitely**.
6. Requires **extensive documentation** at many intermediate stages.

1.5 Agile software development: Agile process, and its importance, extreme programming, scrum

What is Agile?

In software development, the term 'agile' is adapted to mean 'the ability to respond to changes − changes from Requirements, Technology and People.'

Agile Software Development is a way of making software that focuses on **flexibility, teamwork, and customer happiness**.

It follows the **Agile Manifesto**, which gives principles that prefer:

- people and communication over strict processes,
- working software over long documents,
- customer involvement over formal contracts,

- and adapting to changes over sticking to a fixed plan.

Agile is done in **small, repeated steps (iterative and incremental)**. The goal is to deliver a **working product quickly and often**. Developers and customers work closely together so the final product truly fits the customer's needs.

Agile Software Development Process

Agile, often just called Agile, is about being **practical and flexible** while making software. Instead of finishing and releasing everything at once, Agile delivers **small useful updates** to users step by step.

This way, the team can **make changes and improvements along the way**, and each update is checked to ensure it brings **real value**.

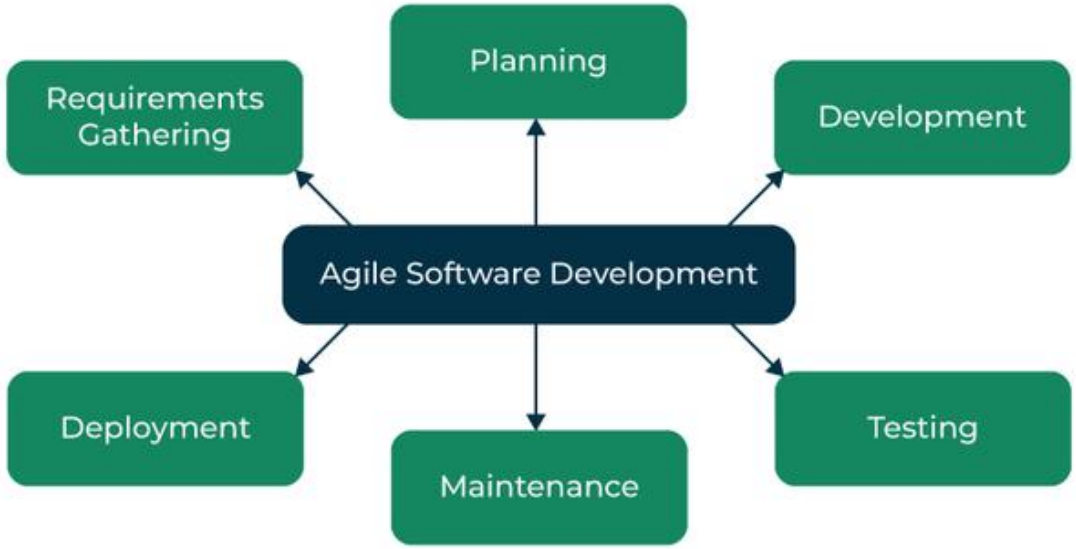
Agile is about **progress in small steps** and **quickly responding to change**.

Agile Software Development Process

Agile software development, often called **Agile**, focuses on being **flexible and practical** while making software.
Instead of delivering everything at once, Agile provides **small, useful updates** step by step.

This helps teams **improve and adjust** the product as they go, making sure each update gives **real value** to users.

Agile is about **small steps, steady progress, and quick response to change**.



Agile Software Development

1. Requirements Gathering

- First step: team works with the customer to know what they really need.
- Collect requirements, **sort and prioritize** them.

- Focus on most important features **first**.

2. Planning

- Team prepares a **clear plan** for development.
- Decide which features to build in each **iteration (cycle)**.
- It's like making a **roadmap**, so everyone knows what will be done and when.

3. Development

- Team starts building the software in **short cycles**.
- Each cycle creates a **small working part** of the product.
- Builds step by step, with quick feedback to improve.

4. Testing

- As features are developed, they are **tested immediately**.
- Makes sure the product works correctly and meets customer needs.
- Errors are caught **early**, ensuring high quality.

5. Deployment

- After testing, the software is **released to users**.
- Customers can now use the product.
- This is when the **work becomes real**.

6. Maintenance

- Work continues even after release.
- The team fixes issues, updates features, and adapts to new customer needs.
- Keeps the software **relevant and useful**.

Agile Software Development is widely used because it is **flexible and adaptable**.

It suits projects where **requirements change frequently** and speed matters.

Agile is a **time-bound, iterative process** that delivers software **step by step**, instead of all at once.

Advantages of Agile Software Development

1. **Flexibility & Adaptability** – Can quickly respond to changing requirements or market needs.
2. **Better Collaboration** – High teamwork, communication, and involvement of stakeholders/customers.
3. **High Quality** – Continuous testing, feedback, and improvement reduce defects.
4. **Customer Satisfaction** – Regular feedback ensures product matches customer needs.

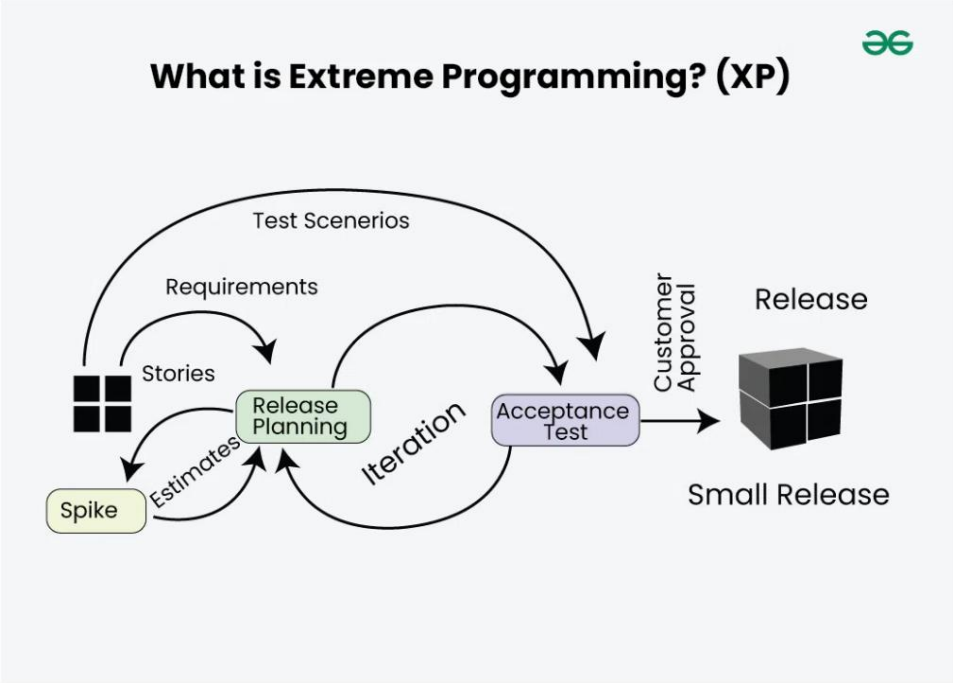
5. **Faster Delivery** – Software is released in small increments, gaining customer trust.
6. **Improved Team Morale** – Supportive environment increases motivation & productivity.

Disadvantages of Agile Software Development

1. **Unpredictable** – Hard to estimate timelines, budgets, and final outcomes.
2. **Scope Creep** – Frequent changes can cause lack of control over scope.
3. **Risk of Burnout** – Continuous sprints put pressure on the team.
4. **Less Documentation** – Focus on code over formal documentation.
5. **Customer Dependence** – If customers lack clarity, project may go off-track.
6. **Hard for Large Projects** – Difficult to scale in big organizations; face-to-face communication and governance become challenging.

What is Extreme Programming (XP)?

Extreme Programming (XP) is an **Agile software development methodology** that aims to deliver **high-quality software** through **continuous feedback, teamwork, and adaptation**. It stresses **close collaboration** between the development team, customers, and stakeholders, with a focus on **rapid, iterative development and deployment**.



Extreme Programming (XP)

Agile methods emerged in the 1990s as a response to **heavy documentation and rigid processes** like the waterfall model.

Agile approaches follow some **key principles**, which include:

1. **Working software** is the main measure of progress.

2. Software should be **developed and delivered quickly in small increments**.
3. **Changes in requirements** should be accepted, even if they come late.
4. **Face-to-face communication** is better than written documentation.
5. **Customer involvement and continuous feedback** are necessary for good-quality software.
6. **Simple design** that improves over time is preferred over making a **complex design up front**.
7. **Delivery dates** are decided by **empowered and skilled teams**.

Extreme Programming is one of the **most popular and widely used Agile methods**. An XP project starts with **User Stories**:

- These are short descriptions of the scenarios and features that customers want.
- Each story is written on a **separate card**, so they can be easily arranged and grouped.

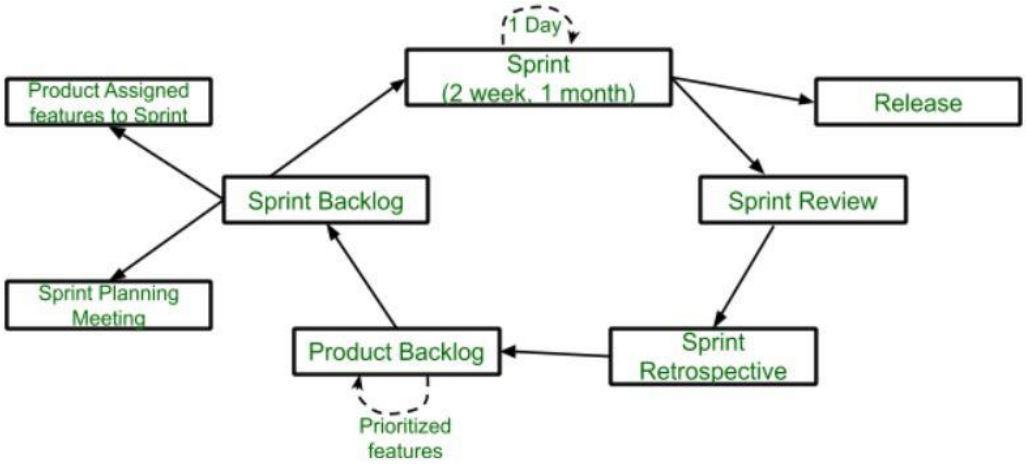
What is Scrum in Software Development?

Scrum is a **framework for Agile software development**.

- It works in **iterative cycles called Sprints**.
- Involves **daily stand-up meetings**.
- Uses a **Product Backlog**, which is **prioritized by the customer**.

Scrum is a **management framework** where teams **self-organize** their tasks and work together towards a **common goal**.

It provides a structure to solve **complex, adaptive problems** while keeping **creativity and productivity high**, and delivering products of the **highest value**.



Key Points:

- Scrum helps to **develop high-value products** while maintaining **creativity and productivity**.
- It uses an **iterative and incremental approach**, allowing teams to **adapt to changing requirements**.

Salient Features of Scrum

- Scrum is a **lightweight framework**.
- It emphasizes **self-organization**.
- It is **simple to understand**.
- It helps the team **work together effectively**.

Lifecycle of Scrum

- **Sprint**: A Sprint is a **time-box of one month or less**. A new Sprint begins immediately after the previous one ends.
- **Release**: Once the product is complete, it moves to the **Release stage**.
- **Sprint Review**: If some features are still incomplete, they are checked here before moving to **Sprint Retrospective**.
- **Sprint Retrospective**: Here the **quality and status** of the product are reviewed.
- **Product Backlog**: A list of product features, arranged according to **priority**.
- **Sprint Backlog**: Divided into two parts:
    - Features assigned to the Sprint.
    - Sprint Planning Meeting outcomes.

Advantages of Scrum Framework

- **Fast-moving and cost-efficient**.
- Works by dividing a **large product into small sub-products** (divide and conquer).
- Ensures **high customer satisfaction**.
- Adaptive in nature because of **short Sprints**.
- Relies on **constant feedback**, improving product quality in **less time**.

Disadvantages of Scrum Framework

- Scrum does **not allow changes** once a Sprint has started.
- Scrum is **not a fully defined model** → teams need to **fill gaps** with methods like **XP, Kanban, DSDM**.
- Planning and organizing is difficult if the **project lacks clear definition**.
- **Daily meetings and reviews** require a lot of **time and resources**.

1.6 Selection criteria for software process model

Introduction

Software development is **not the same for all projects**.

Each project requires a **suitable process model** depending on its nature, size, and requirements.

- A **software process model** is a framework used to **plan, structure, and control** the development of software systems.
- Common models: **Waterfall, Agile, Spiral, RAD, Incremental, Prototyping**.
- **Wrong selection** → wasted resources, delays, poor quality, failure.
- **Right selection** → better efficiency, higher quality, customer satisfaction.

✅ **Factors / Criteria for Selecting a Software Process Model**

1. Project Size

- **Small Projects (mini tools, web forms): Waterfall and RAD are suitable because they are simple and have less overhead.**
- **Large Projects (ERP, defense software): Spiral and Incremental are better as they can manage complexity and provide flexibility.**

2. Clarity of Requirements

- **Clear & Stable Requirements: Waterfall is suitable since everything can be planned upfront.**
- **Unclear / Changing Requirements: Agile, Spiral, and Prototyping are preferred because they support continuous changes.**

3. Customer Involvement

- **High Involvement: Agile, Scrum, and Prototyping are effective as they involve continuous feedback from the customer.**
- **Low Involvement: Waterfall is useful since customer input is required only at the start and end.**

4. Risk Level

- **High Risk (new technology, uncertain outcomes): Spiral is chosen because it explicitly includes risk analysis at every stage.**
- **Low Risk: Waterfall and Incremental can be used as they are sufficient when risk is low.**

5. Project Complexity

- **Simple Projects: Waterfall and RAD are suitable because they are straightforward.**
- **Complex Projects: Spiral and Agile are better as they can handle complexity effectively.**

6. Time-to-Market Requirement

- **Tight Deadlines: Agile and RAD are useful since they provide quick iterations and early versions.**
- **No Urgency: Waterfall can be used because it delivers the final product at the end.**

7. Team Experience & Skill

- **Inexperienced Teams: Waterfall is preferred as it provides clear steps and proper documentation.**
- **Experienced Teams: Agile and XP are suitable because they require flexibility and strong technical skills.**

8. Budget Constraints

- **Limited Budget: RAD and Incremental are cost-effective and avoid long development cycles.**
- **Flexible Budget: Spiral is suitable since it allows gradual investment and is good for large systems.**

9. Documentation Requirement

- **High Documentation: Waterfall and V-Model are preferred for government and safety-critical systems as they demand detailed documents.**
- **Low Documentation: Agile and XP are better since they focus more on working software than paperwork.**

10. Flexibility & Change Management

- **Frequent Changes Expected: Agile, Spiral, and Prototyping are suitable as they can easily handle changes.**

- **Fixed Plan: Waterfall and V-Model are better as they follow a rigid and predefined sequence.**

11. Stakeholder Expectations

- **Quick Progress Visibility: Agile and Incremental are good as they provide regular updates and partial deliveries.**
- **Full Product at Once: Waterfall and V-Model are suitable as they deliver the entire system at the end.**

📝 **Conclusion**

The **selection of a software process model** depends on multiple factors such as **project size, risk, requirements, time, cost, team skills, documentation, and stakeholder expectations**. Choosing the right model ensures **better quality, timely delivery, and customer satisfaction**.