Unit 2: Software Processes (5 Hrs.)

Software Process

A software process is a set of related activities required to develop high-quality software systems. Since different types of software (e.g., embedded systems, web applications, AI-driven solutions) have unique requirements, there is no single universal process that fits all scenarios. Instead, organizations tailor their approaches based on factors such as:

- **Project complexity** (small app vs. enterprise system)
- **Customer needs** (fixed requirements vs. evolving demands)
- **Team expertise** (experienced engineers vs. distributed teams)
- **Regulatory constraints** (safety-critical systems vs. consumer apps)

Despite this variability, all effective software processes incorporate the following four core engineering activities:

1. **Software Specification (Requirements Engineering)**

   Defines what the software should do and its operational constraints.

   Task involved:

   - Gathering and analyzing stakeholder needs
   - Documenting functional & non-functional requirements
   - Prioritizing features based on business goals

   Outputs:

   - Software Requirements Specification (SRS)
   - Use case diagrams

2. **Software Development (Design & Implementation)**

   Transforms requirements into a working system.

   Task involved:

   - **Architectural design** (system structure, components)
   - **Detailed design** (algorithms, database schemas)
   - **Coding** (following best practices like clean code, SOLID principles)

   Outputs:

   - Design documents (UML diagrams, ER models)
   - Source code

- APIs and microservices (if applicable)

3. **Software Validation (Testing & Quality Assurance)**

   Purpose: Ensure the software meets specifications and is defect-free.

   Key Tasks:

   - Unit testing (individual components)
   - Integration testing (interactions between modules)
   - System & acceptance testing (end-to-end validation)

   Outputs:

   - Test cases and reports
   - Bug-fix logs
   - Performance benchmarks

4. **Software Evolution (Maintenance & Updates)**

   Purpose: Adapt the software to changing needs over time.

   Key Tasks:

   - Corrective maintenance (fixing defects)
   - Adaptive maintenance (updating for new environments)
   - Perfective maintenance (enhancing features)

   Outputs: Patches and updates

   - Versioned releases
   - Refactored code

Software Process Models

Software process models (sometimes called a Software Development Life Cycle or SDLC model) provide structured approaches to developing software systems. These models serve as **blueprints** that guide teams through the stages of specification, design, implementation, testing, and maintenance. While no single model fits all projects, understanding different paradigms helps teams choose the best approach for their needs.

1. The Waterfall Model

The **Waterfall Model** is a **linear, sequential** approach where each phase must be completed before moving to the next. It aligns closely with the fundamental software engineering activities:

1. Requirements Definition

2. System & Software Design
3. Implementation & Unit Testing
4. Integration & System Testing
5. Operation & Maintenance

2. Incremental Development
This model **interleaves** specification, development, and validation, delivering software in **small, functional increments**. Each iteration adds features to the previous version.

3. Integration & Configuration (Reuse-Oriented Model)
This approach focuses on **integrating existing components** (e.g., APIs, microservices, COTS software) rather than building from scratch.

Hybrid Approaches
Many modern processes (e.g., **Rational Unified Process - RUP**) combine elements from different models. They incorporate **structured planning** (like Waterfall) for core architecture, **iterative development** (like Incremental) for feature rollouts and **reusable components** (like Integration & Configuration) for efficiency.

Waterfall Model
The Waterfall Model is one of the earliest and most widely used structured software development methodologies. Originating from military and aerospace engineering practices, it follows a sequential, phase-by-phase approach where each stage must be completed before moving to the next. Since the phases fall from a higher level to lower level, like a waterfall, it's named as the waterfall model.

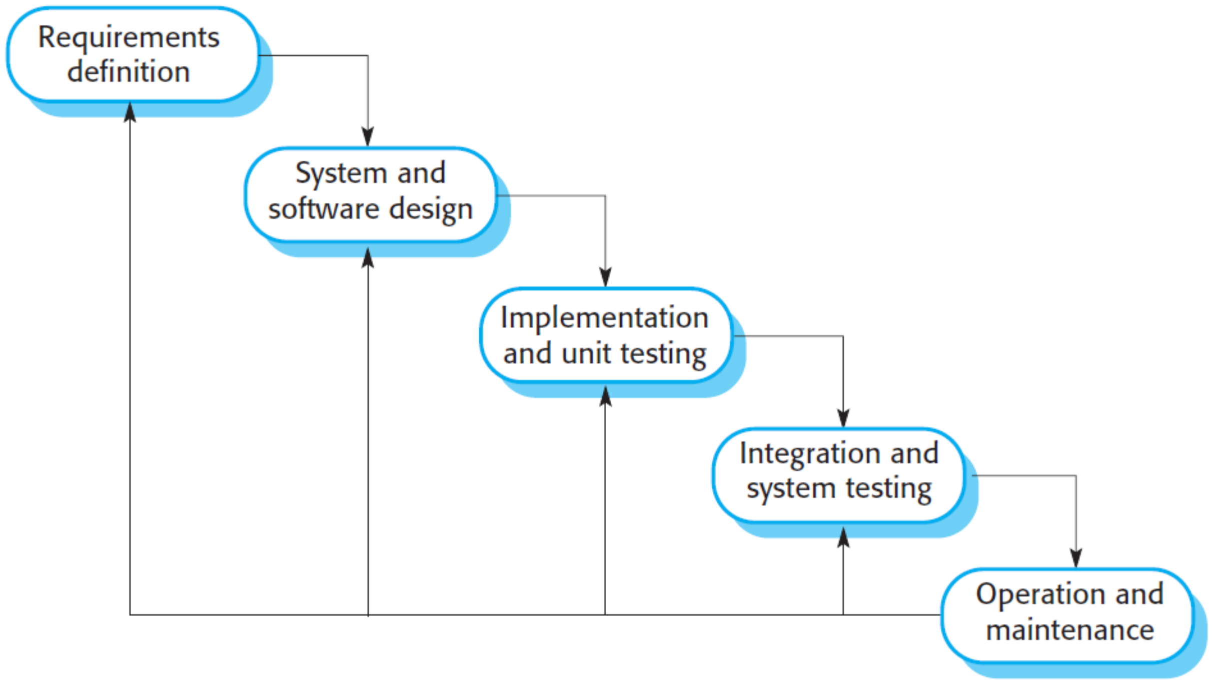
Requirements definition
System and software design
Implementation and unit testing
Integration and system testing
Operation and maintenance

Figure 1: The Waterfall Model

Stages of the Waterfall Model

- **Requirements analysis and definition:**

  This initial phase focuses on gathering and defining the project's requirements. The team works with stakeholders to understand their needs and expectations for the final product and document detailed software requirements specification (SRS).

- **System and software design:**

  Based on the solidified requirements, the team designs the system's architecture and software components. This includes creating system architecture diagrams (e.g., UML) and defining system interfaces, data structures, and algorithms.

- **Implementation and unit testing:**

  With the design finalized, developers begin coding and building the system. The team rigorously develop and test individual software components. Unit testing occurs here, where individual software units are tested for functionality.

- **Integration and system testing:**

  Once individual units are built, they are integrated into the complete system. This phase involves thorough testing to ensure all components work together seamlessly and meet the overall requirements. After the team perform system, performance, and acceptance testing, the tested and deployable software is obtained as output.

- **Operation and maintenance:**
  The main objectives of operation and maintenance phase is to deploy, monitor, and improve the system. This phase involves ongoing maintenance, fixing bugs, and addressing any user issues that may arise.

When to Use Waterfall Model?

- Projects with stable and well-defined requirements
- Low uncertainty (minimal expected changes during development).
- Regulated industries (where documentation is mandatory)
- In case of Safety-critical systems (e.g., medical devices, avionics) and Large-scale engineering projects (e.g., hardware-software integration).

The Waterfall Model remains relevant for high-certainty, regulated, or safety-critical projects where upfront planning is essential. However, its inflexibility makes it unsuitable for modern, fast-paced environments where requirements evolve.

Incremental Development Model

The **Incremental Development Model** is an iterative approach where software is built in small, functional increments. Incremental development is based on the idea of developing an initial implementation, getting feedback from users and others, and evolving the software through several versions until the required system has been developed. Each increment adds new features to the previous version, allowing for **continuous feedback, adaptation, and early delivery**. This model blends elements of **plan-driven and agile methodologies**, making it ideal for projects with evolving requirements.

Unlike the rigid **Waterfall Model**, incremental development **interleaves** specification, development, and validation in cycles:

1. **Initial Planning**
   - Define core requirements and prioritize features.
   - Plan increments
2. **Iterative Development**

   - **Develop:** Build a subset of features (Increment 1).
   - **Validate:** Test and gather user feedback.
   - **Refine:** Adjust requirements and improve the next increment.
3. **Final Deployment**
   - Combine all increments into a complete system.

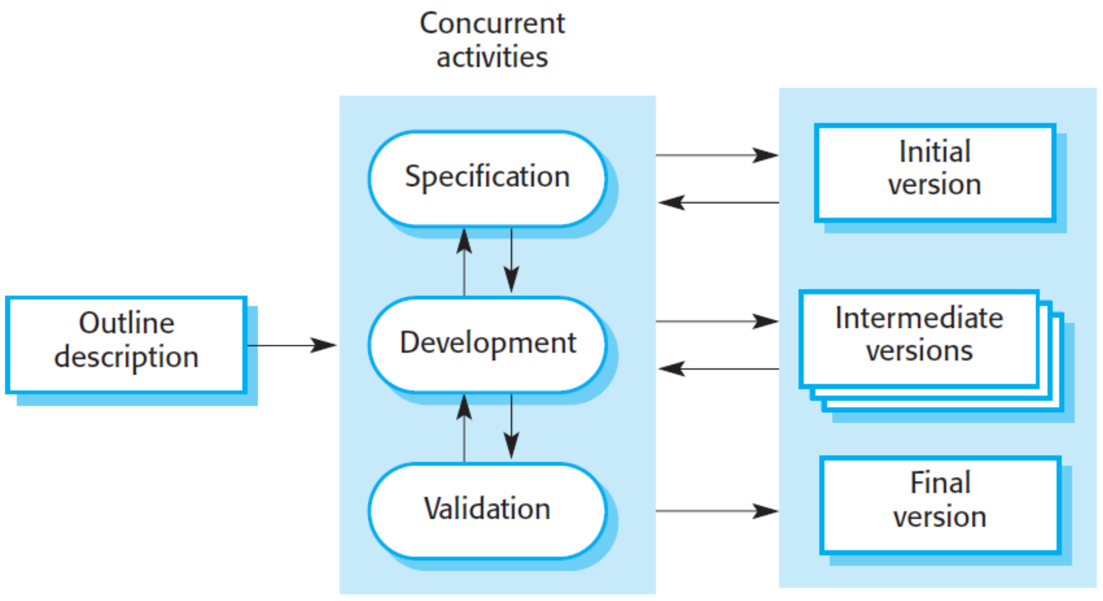

Figure 2: Incremental Development

Advantages Over Waterfall

Benefit	**Explanation**
Lower change costs	Only the current increment needs modification.
Early customer feedback	Users validate working features.
Faster ROI (Return of Inv)	Partial functionality can be deployed early.
Risk mitigation	Issues surface sooner, reducing late-stage failures.

The Incremental Development Model strikes a balance between **structure and flexibility**, making it a **go-to choice for modern software projects**. By delivering functional pieces early and often, teams can:

- **Reduce risk** through continuous validation.
- **Adapt to market changes** efficiently.
- **Maximize customer satisfaction** with tangible progress.