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.

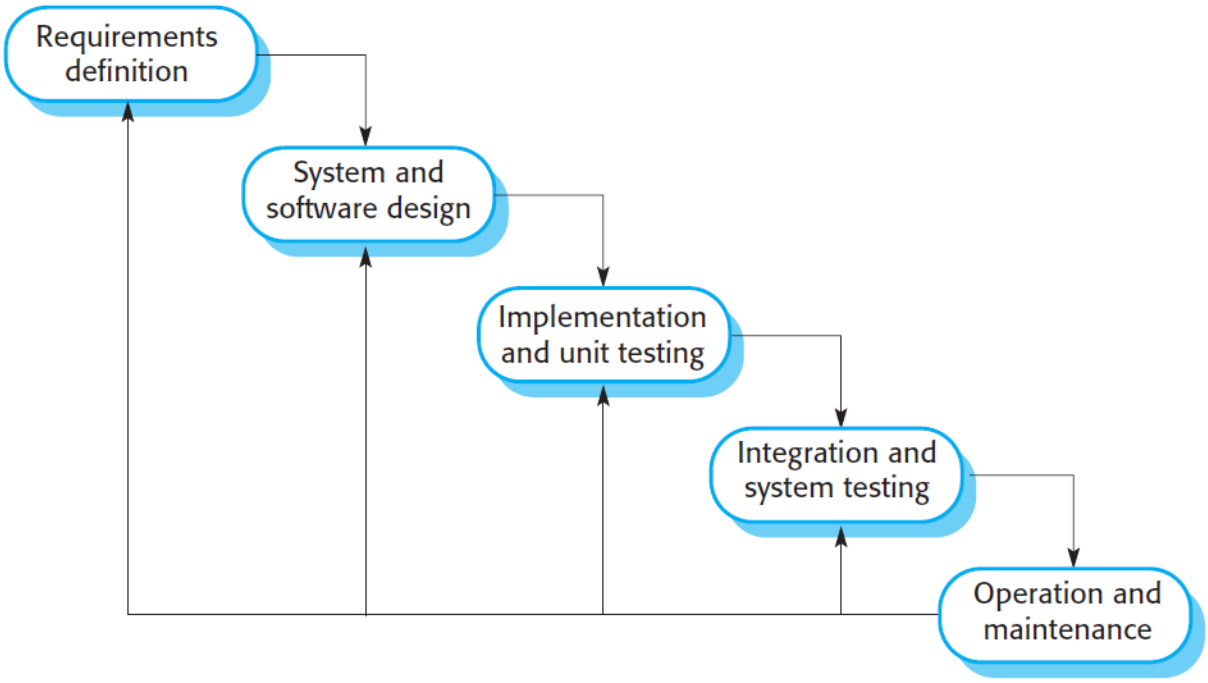


Figure 1: The Waterfall Model

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.

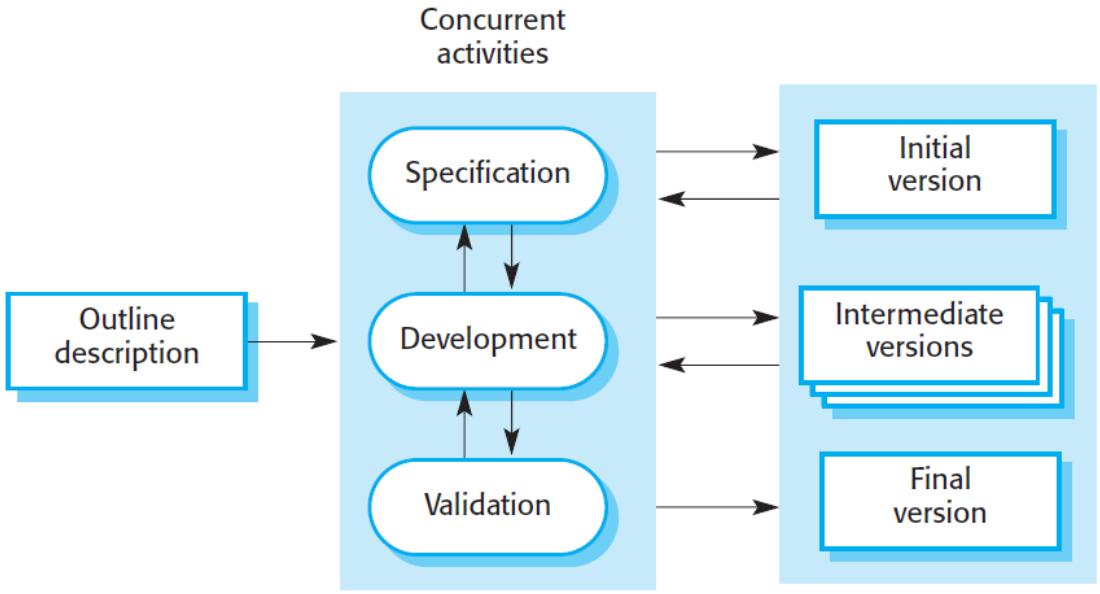


Figure 2: Incremental Development

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.

Integration and Configuration

Integration and Configuration is a **reuse-driven software development model** that focuses on assembling systems from existing components rather than building from scratch. This approach leverages **off-the-shelf software, libraries, APIs, and services** to accelerate development while reducing costs and risks.

Key Characteristics: **Reuse-Centric, Configurable and Integration-Focused**

Types of Reusable Components

- **Stand-Alone Application Systems**: **Example:** ERP systems, CMS platforms.
- **Object Collections & Frameworks**: **Example: .NET Core**
- **Web Services & APIs**: **Example:** Payment gateways (Stripe, PayPal), Cloud services (AWS S3, Google Maps API)

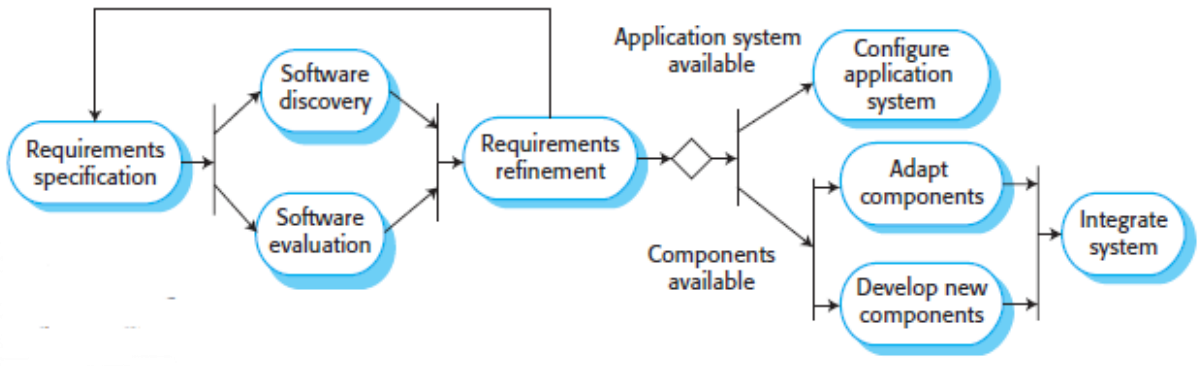


Figure 3: Reuse-oriented software engineering

Figure 3 shows a general process model for reuse-based development, based on integration and configuration. The stages in this process are:

1. Requirements Specification

- Define **high-level needs** (no deep detail required).

- Example: "The system must process payments and manage user profiles."

2. Software Discovery & Evaluation

- **Search** for reusable components (e.g., GitHub, vendor marketplaces).
- **Evaluate** based on:
    - Compatibility with requirements.
    - Licensing (open-source vs. proprietary).
    - Support and scalability.

3. Requirements Refinement

- Adjust requirements to **align with available components**.
- Example: If no pre-built fraud detection module exists, modify the system to use a third-party API.

4. Application System Configuration

- Customize **off-the-shelf (OTS) software** (e.g., Shopify for e-commerce).
- Example: Configuring WordPress with WooCommerce for an online store.

5. Component Adaptation & Integration

- **Modify components** (if needed) and integrate them.
- Example:
    - Extending an open-source dashboard library (e.g., Grafana) for custom analytics.
    - Combining Auth0 for login with a custom backend.

When to Use This Model

- **Enterprise systems** (ERP, CRM)
- **Startups needing rapid MVPs**
- **Cloud-native applications**.

Not ideal for:

- Highly specialized systems (e.g., spacecraft firmware).
- Projects requiring full control over codebase.

Integration and Configuration is a **pragmatic approach** for modern software engineering, particularly suited for:

- **Business applications** (e.g., SaaS platforms).
- **Systems with reusable ecosystems** (e.g., web/mobile apps).

Software Process Activities

Software process activities are the core building blocks of software development, encompassing technical, collaborative, and managerial tasks to deliver high-quality systems. These activities are supported by modern tools and organized differently based on the chosen process model (e.g., Waterfall, Incremental).

The Four Fundamental Activities

1. Software Specification (Requirements Engineering)
2. Software Development (Design & Implementation)
3. Software Validation (Testing & QA)
4. Software Evolution (Maintenance & Updates)

We will study in details about these fundamental activities in the subsequent units.

Coping with Change

Change is an inherent part of software development due to:

- **Evolving business needs** (market shifts, competition, new regulations)
- **Emerging technologies** (new frameworks, platforms, tools)
- **User feedback** (discovered during development or after release)

These changes often require **rework**—modifying completed work to accommodate new requirements - which increases costs and delays delivery.

Strategies for Managing Change

1. Change Anticipation (Proactive Approach)

Goal: Predict and prepare for likely changes *before* they require major rework.

Methods:

✓ Prototyping

Prototyping is the process of creating an **early, simplified version** of a software system to:

- Demonstrate core concepts
- Explore design options

- Validate requirements
- Gather user feedback

It enables rapid, iterative development to **reduce risks** and **clarify system needs** before full-scale development.

Uses of Prototyping

1. Requirements Engineering

- Helps **elicit and validate** requirements by giving users a tangible system to interact with.
- Uncovers **hidden or misunderstood needs** early.

2. Design Exploration

- Tests **feasibility** of technical solutions (e.g., database structures, APIs).
- Refines **user interfaces (UI)**—critical for usability testing.

3. Stakeholder Communication

- Provides a **visual demo** for managers, clients, or investors to align expectations.

- Build quick, simplified versions of the system to validate requirements early.
- Example: A mockup of a UI to gather user feedback before full development.
- **Risk Analysis**
    - Identify volatile areas (e.g., payment processing in an e-commerce app) and design for flexibility.

The Prototyping Process

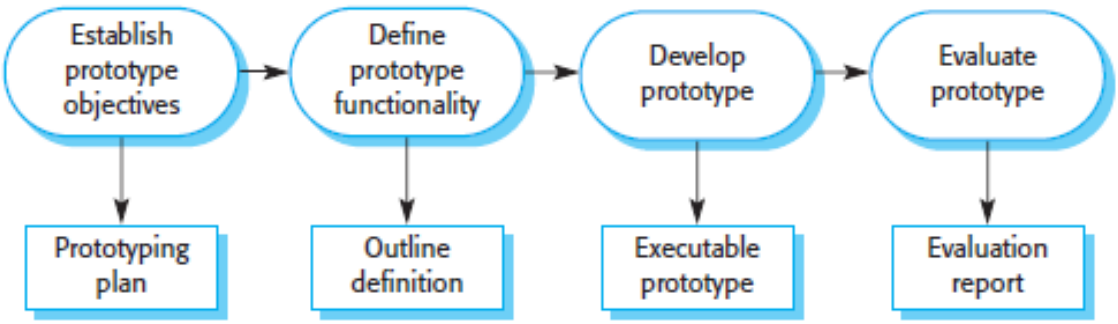


Figure 4: Prototyping development

1. **Define Objectives**
    - Clearly state goals (e.g., "Test checkout usability" vs. "Validate API performance").

2. **Select Features to Include/Exclude**
    - Prioritize **core functionalities** (e.g., ignore error handling for UI prototypes).
    - Relax **non-functional requirements** (e.g., speed, scalability).
3. **Develop & Iterate**
    - Use rapid tools (e.g., Figma for UI, Flask/Django for backend prototypes).
4. **Evaluate with Users**
    - Train users to avoid biased feedback.
    - Observe **real-world usage** (not just demos).

Benefits:

- Reduces late-stage surprises.
- Lowers costs by catching issues early.

2. Change Tolerance (Reactive Approach)

Goal: Make systems easy to modify *when* changes occur.

Methods:

✓ Incremental Development (or Incremental Delivery)

- Deliver software in small, functional chunks. Changes can be added to future increments.
- Example: Releasing a "basic" version of an app first, then adding social features later.

  Modular Design
    - Use loosely coupled components (e.g., microservices) to isolate changes.

  Refactoring
    - Continuously improve code structure to keep it adaptable.
    - Example: Breaking a monolithic codebase into reusable modules.

Benefits:

- Minimizes disruption from changes.
- Speeds up adaptation to new requirements.

Process Improvement

Process improvement is a systematic approach to **enhancing software development practices** to achieve **higher product quality**, **reduced costs** and **faster delivery times**

With increasing demands for **better, cheaper, and faster software**, organizations adopt structured methods to refine their workflows, eliminate inefficiencies, and align with industry best practices.

Two Key Approaches to Process Improvement

1. Process Maturity Approach

Focus: Enhancing process predictability and product quality through structured management practices

Characteristics:

- Emphasizes standardization and measurement (e.g., ISO 9001)
- Implements proven engineering practices
- Uses capability maturity models for assessment (CMMI)
- Best suited for large organizations and complex projects

Pros:

- Predictability and quality in outcomes
- Ideal for **large, regulated industries** (e.g., aerospace, healthcare).

Cons:

- High overhead (documentation, compliance).
- Less flexible for dynamic projects

2. Agile Approach

Focus: Maximizing responsiveness and minimizing overhead

Characteristics:

- Prioritizes working software over documentation
- Emphasizes iterative development
- Adapts quickly to changing requirements
- Particularly effective for dynamic environments

Pros:

- Adapts quickly to changing requirements.
- Lowers overhead (focus on working software).

Cons:

- Hard to scale for large, complex systems.

- Less emphasis on long-term process metrics.

The Process Improvement Cycle

The process improvement cycle represents a systematic and structured approach to enhancing software development practices. While different methodologies (plan-driven vs. agile) debate the optimal path forward, the maturity-based improvement cycle provides a structured framework for measurable progress.

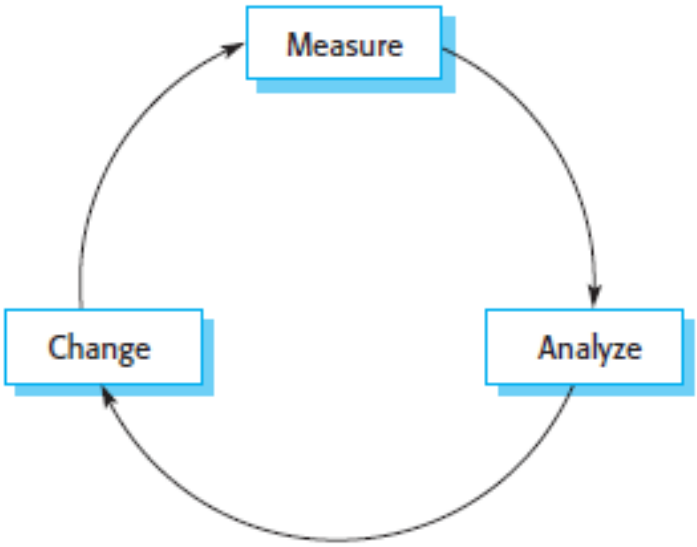


Figure 5: The process improvement cycle

The stages in this process are:

1. **Process Measurement**

   The improvement journey begins with quantitative assessment. Organizations must:

   - Identify key metrics that reflect process effectiveness (e.g., defect rates, cycle time, productivity)
   - Implement measurement systems to capture these metrics consistently and establish baseline values that represent current performance levels

2. **Process Analysis**

   With measurement data in hand, teams conduct thorough evaluations to:

   - Map existing workflows through process modeling techniques
   - Identify bottlenecks that slow development
   - Pinpoint quality issues and their root causes

- Assess process characteristics like speed, reliability, and adaptability

3. **Process Change**

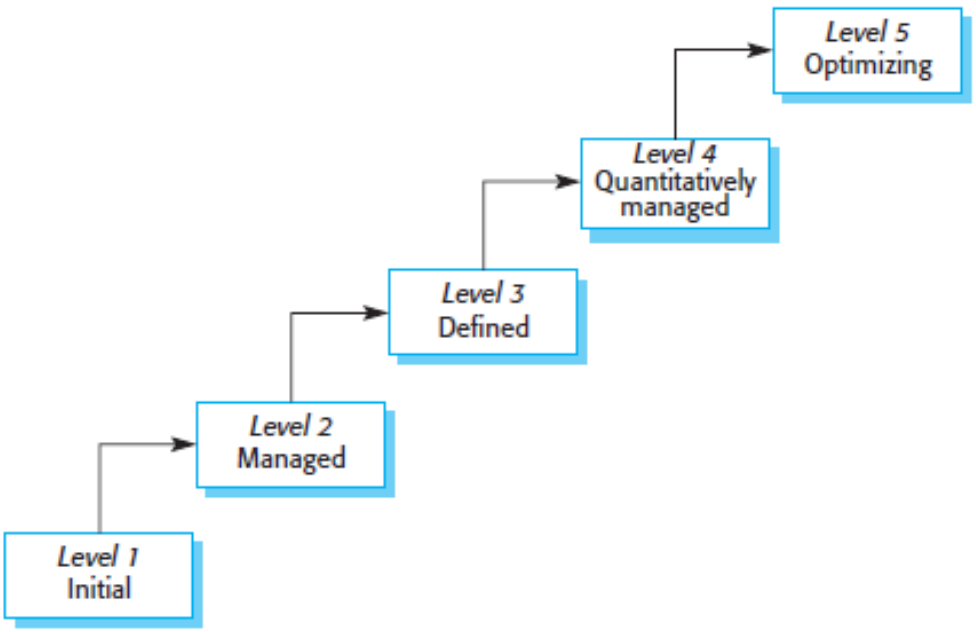
   The insights from analysis drive targeted modifications:

   - Redesigning inefficient workflows
   - Introducing new quality assurance practices
   - Removing redundant process steps
   - Adopting better tools or methodologies

Repeat the cycle to ensure continuous enhancement.

Process Maturity Levels

The **Process Maturity Model**, originally developed by the Software Engineering Institute (SEI) in the 1980s, provides a structured framework to assess and improve an organization's software development capabilities. Initially created for the U.S. Department of Defense to evaluate contractors, it has become a global standard for measuring process quality and efficiency.



1. Initial (Level 1)

- **Characteristics:**
  - Processes are **unpredictable and reactive**.
  - Success depends on individual heroics.
  - No formal documentation or consistency.

- **Example:** A startup coding without requirements or testing protocols.

2. Managed (Level 2)

- **Characteristics:**
  - Basic **project management** is established (schedules, budgets).
  - Processes are **repeatable** for similar projects.
  - Focuses on **meeting immediate goals**.
- **Example:** A mid-sized company using documented plans but lacking standardization.

3. Defined (Level 3)

- **Characteristics:**
  - Organization-wide **standardized processes** (e.g., coding standards, QA checklists).
  - Processes are **tailored** per project needs.
  - **Knowledge sharing** via process assets (templates, guidelines).
- **Example:** A firm with a centralized DevOps pipeline used by all teams.

4. Quantitatively Managed (Level 4)

- **Characteristics:**
  - **Data-driven decision-making** (metrics like defect rates, cycle time).
  - Statistical process control (SPC) to **predict performance**.
  - Subprocesses are **measured and optimized**.
- **Example:** An enterprise using AI to predict deployment failures based on historical data.

5. Optimizing (Level 5)

- **Characteristics:**
  - **Continuous improvement** via innovation and feedback loops.
  - **Proactive adaptation** to market/technology changes.
  - Focus on **preventing defects** (vs. fixing them).
- **Example:** Google's refined CI/CD pipeline with automated A/B testing.