Module-01

Introduction to Software Engineering Process Models

Chapter-01

Software and Software Engineering

1. **What is computer software?**
    - Computer software is the product created and supported by software professionals over the long term.
    - It includes programs running on computers of any size and architecture, along with associated content and descriptive information.

2. **What is software engineering?**
    - Software engineering is a process, a collection of methods, and an array of tools used by professionals to build high-quality computer software.

3. **Who are software engineers?**
    - Software engineers are individuals who build and support software, and almost everyone in the modern world uses software either directly or indirectly.

4. **Why is software important?**
    - Software is pervasive in commerce, culture, and everyday activities, affecting nearly every aspect of our lives.
    - Software engineering is important as it enables the timely development of complex systems with high quality.

5. **What are the steps in building software?**
    - Software is built using an agile, adaptable process that aims for a high-quality result meeting user needs.
    - A software engineering approach is applied to ensure the effectiveness of the process.

6. **What is the work product of software engineering?**
    - From a software engineer's perspective, the work product consists of programs, content, and other artifacts constituting computer software.
    - From the user's viewpoint, the work product is the resulting information that improves their world.

The Nature of Software

The Dual Role of Software

Software as a Product

- Delivers computing potential of hardware or a network of computers.
- Acts as an information transformer, managing and manipulating data from simple bits to complex multimedia.

Software as a Vehicle

- Controls computer operations (operating systems).
- Facilitates communication of information (networks).
- Enables creation and control of other programs (software tools and environments).

Importance of Software

- Information Transformation
    - Makes personal data more useful.
    - Manages business information to enhance competitiveness.
    - Provides access to global information networks like the Internet.
    - Enables the acquisition of information in various forms.

Evolution of Software

- Significant Changes Over 50 Years
- Hardware performance improvements.
- New computing architectures.
- Increased memory and storage capacities.
- Advanced input and output options.

Impact

- Led to more sophisticated and complex systems.
- While success can be dazzling, complexity poses challenges for developers.

Modern Software Industry

- **Economic Dominance**
    - A major factor in industrialized economies.
    - Teams of specialists focus on different technology aspects of complex applications.
- **Persistent Questions**
    - Why does software development take so long?
    - Why are development costs high?
    - Why can't all errors be found before release?
    - Why is maintaining software so time-consuming?
    - Why is progress measurement difficult?

Software Engineering Practice

- **Addressing Concerns**
    - The challenges in software development have led to the adoption of software engineering practices to improve efficiency, quality, and manageability of software projects.

Defining Software

- Instructions (programs) that provide desired features, function, and performance.
- Data structures that enable information manipulation.
- Descriptive information (hard copy and virtual) describing the operation and use of programs.

Characteristics of Software

Logical System Element

- Software is a logical, not a physical, system element.

Development vs. Manufacturing

- Software is developed or engineered, not manufactured.
- High quality in software is achieved through good design.
- Software projects cannot be managed like manufacturing projects.

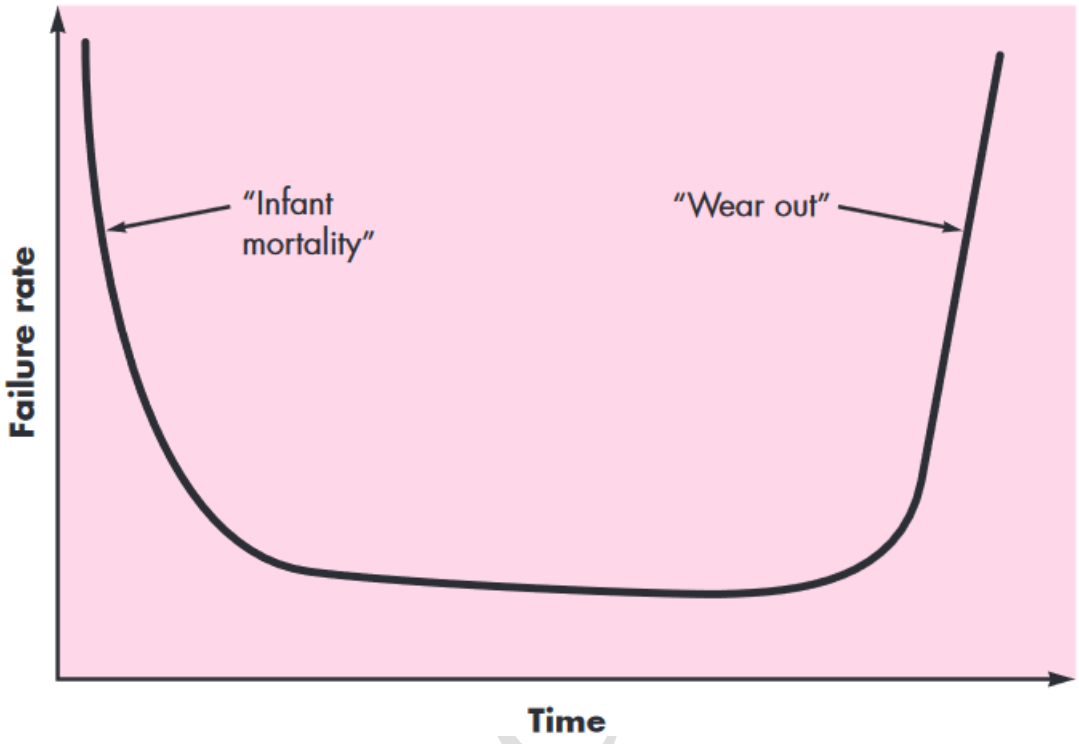
No Wear and Tear

- Software doesn't wear out like hardware.

Failure Rate Curve

- Hardware failure rate follows a "bathtub curve" with high initial failure rates, a low steady-state, and rising failure rates over time.

- Software failure rate ideally flattens after initial defects are fixed but can spike due to changes introducing new errors.



Reusable Software Components

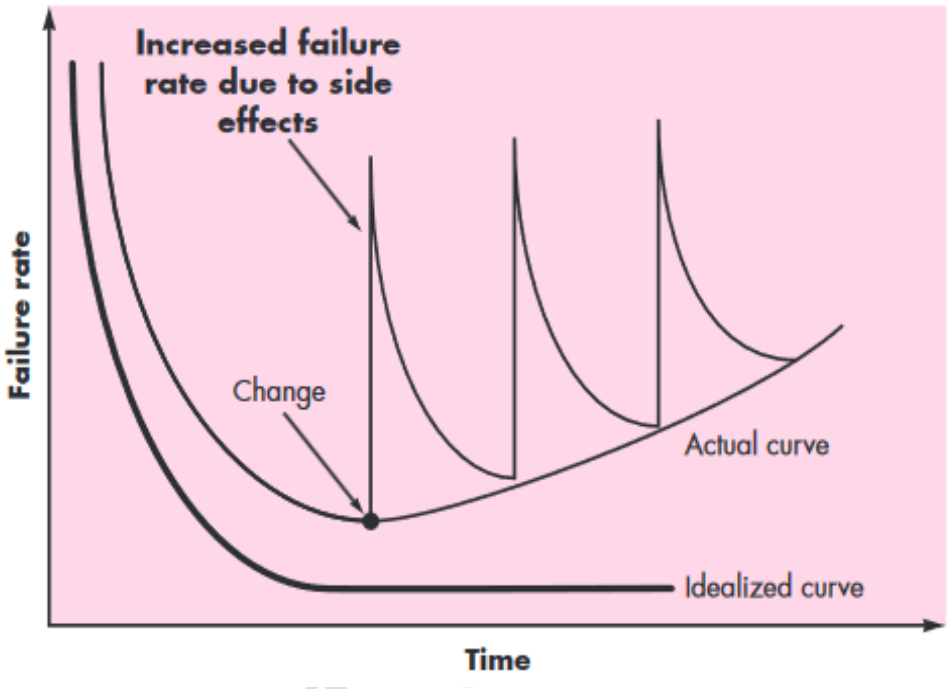
- Software components should be reusable in multiple programs.
- Encapsulate both data and processing for reuse.
- Reusable components are used to build interactive user interfaces (e.g., graphics windows, pull-down menus).
- Libraries of reusable components assist in constructing interfaces.

Software Engineering Complexity

- Software maintenance is more complex due to the lack of spare parts and the need to address design errors.

Economic Impact

- The software industry is a dominant factor in industrialized economies.
- Modern software development involves teams of specialists rather than lone programmers.



Persistent Challenges

- Software development takes a long time.
- Development costs are high.
- Difficulty in finding all errors before release.
- Significant time and effort spent on maintaining existing programs.
- Difficulty in measuring progress during development and maintenance.

Software Application Domains

1. **System Software**
    - Services other programs.
    - Includes compilers, editors, file management utilities, operating system components, drivers, networking software, and telecommunications processors.
    - Characterized by heavy interaction with hardware, multi-user environments, concurrent operation, complex data structures, and multiple external interfaces.

2. **Application Software**
    - Stand-alone programs solving specific business needs.
    - Processes business or technical data for operations or decision-making.
    - Includes real-time applications like point-of-sale transaction processing and real-time manufacturing process control.

3. **Engineering/Scientific Software**
    - Known for "number crunching" algorithms.
    - Applications range from astronomy to automated manufacturing.
    - Modern applications include computer-aided design and system simulation, integrating real-time and system software characteristics.

4. **Embedded Software**
    - Resides within products or systems.
    - Controls features and functions for end users and the system itself.
    - Examples: key pad control for a microwave oven, digital functions in automobiles like fuel control and braking systems.

5. **Product-Line Software**
    - Provides specific capabilities for many customers.
    - Targets both niche markets (e.g., inventory control) and mass markets (e.g., word processing, spreadsheets, multimedia).

6. **Web Applications (WebApps)**
    - Network-centric software spanning various applications.
    - Evolving from simple hypertext files to sophisticated computing environments.
    - Integrated with corporate databases and business applications.

7. **Artificial Intelligence Software**
    - Uses non-numerical algorithms to solve complex problems.
    - Applications include robotics, expert systems, pattern recognition, artificial neural networks, theorem proving, and game playing.

New Challenges in Software Engineering

- Growth of wireless networking leading to pervasive, distributed computing.
- Develop systems for communication across vast networks involving mobile devices, personal computers, and enterprise systems.
- The web becoming a computing engine and content provider.
- Architect applications that provide benefits to end-user markets worldwide.
- Distribution of source code for systems applications.
- Build self-descriptive source code and develop techniques to track changes made by various contributors.

Legacy Software

Definition

- Older computer programs still in use.
- Continuously modified to meet evolving business requirements and computing platforms.
- Support core business functions and are considered indispensable.

Characteristics

- Longevity and business criticality.
- Often have poor quality: inextensible designs, convoluted code, poor or nonexistent documentation, unarchived test cases and results, poorly managed change history.

Challenges

- Costly to maintain.
- Risky to evolve.

Necessity for Change

- Adapting to new computing environments or technologies.
- Enhancing to meet new business requirements.
- Extending for interoperability with modern systems or databases.
- Re-architecting for viability within network environments.

Strategy for Evolution

- Do nothing if the system runs reliably and meets user needs.
- Reengineer when significant changes are necessary.

Modern Software Engineering Goal

- Devise methodologies based on the notion of evolution.
- Ensure systems interoperate and cooperate with each other.

The Unique Nature of Webapps

- In the early 1990s, websites were simple hypertext files with text and limited graphics.
- The development of tools like XML and Java enabled more sophisticated Web-based systems and applications, referred to as WebApps.
- WebApps provide both stand-alone functionality and integration with corporate databases and business applications.

WebApps have unique characteristics that differentiate them from other software categories:

Network Intensiveness

- Reside on a network and serve a diverse community of clients.
- Can enable worldwide access (Internet) or limited access (corporate Intranet).

Concurrency

- Accessed by a large number of users simultaneously.
- Usage patterns among end users can vary greatly.

Unpredictable Load

- User numbers can vary drastically from day to day.
- Examples include 100 users on Monday versus 10,000 on Thursday.

Performance

- User retention depends on fast access, server-side processing, and client-side display.
- Slow performance can lead users to switch to other WebApps.

Availability

- Popular WebApps are expected to be accessible 24/7/365.
- Users in different time zones demand access at all hours.

Data Driven

- Present hypermedia content like text, graphics, audio, and video.
- Often access external databases for information (e.g., e-commerce or financial apps).

Content Sensitive

- Quality and aesthetics of content are crucial for the overall quality of a WebApp.

Continuous Evolution

- WebApps evolve continuously, often with content updates on a minute-by-minute basis.
- Some content may be independently computed for each user request.

Immediacy

- WebApps have a fast time-to-market, often launched within days or weeks.

Security

- Network access makes it challenging to control end user population.
- Strong security measures are needed to protect content and ensure secure data transmission.

Aesthetics

- The look and feel of a WebApp play a significant role in its appeal.
- Especially important for marketing or selling products and ideas.

Software Engineering

Broad Stakeholder Involvement

- Software impacts virtually every aspect of life, increasing the number of stakeholders.
- Different stakeholders often have varying ideas of required features and functions.
- Understanding the problem thoroughly before developing a solution is crucial.

Increasing Complexity

- Information technology requirements are growing more complex.
- Large teams are now required to develop software previously built by individuals.
- Modern software systems and products are embedded in diverse devices.
- Careful attention to system element interactions is essential, making design a pivotal activity.

Critical Dependence on Software

- Individuals, businesses, and governments rely on software for decision-making and operations.
- Software failures can cause anything from minor inconveniences to catastrophic failures.
- Ensuring high software quality is vital.

Growth in User Base and Longevity

- The perceived value of software increases its user base and longevity.
- As software usage grows, demands for adaptation and enhancement also increase.
- Software should be maintainable to accommodate these demands.

Definitions of Software Engineering

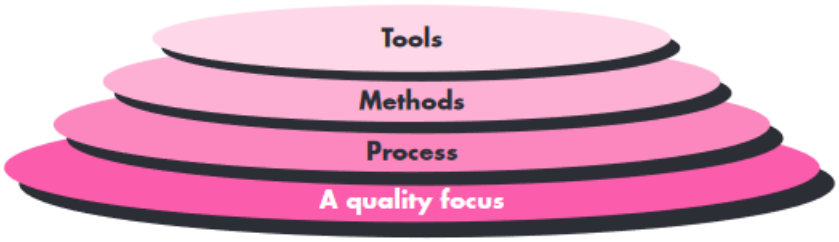
Fritz Bauer's Definition

- Software engineering is the use of sound engineering principles to develop software that is reliable and efficient on real machines.

IEEE's Definition

- Software engineering is a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software.
- It also includes the study of these approaches.

Software Engineering as a Layered Technology



Software engineering is layered to ensure effective software development:

1. **Quality Focus**
    - Quality management philosophies (e.g., Total Quality Management, Six Sigma) foster continuous process improvement.
    - Organizational commitment to quality is the foundation.

2. **Process Layer**
    - The software engineering process is the framework for effective software technology delivery.
    - It enables management control of projects and defines contexts for technical methods, work products, milestones, quality assurance, and change management.

3. **Methods Layer**
    - Provides technical how-to's for building software.
    - Covers tasks such as communication, requirements analysis, design modeling, program construction, testing, and support.
    - Relies on basic principles and modeling activities.

4. **Tools Layer**
    - Tools provide automated or semi-automated support for the process and methods.
    - Integrated tools create a computer-aided software engineering (CASE) system, facilitating the support of software development.

The Software Process

A software process is a collection of activities, actions, and tasks aimed at creating a work product. Here's a breakdown of its components:

1. **Activities:** Broad objectives applied across different domains, project sizes, and complexities. Example: Communication with stakeholders.
2. **Actions:** Sets of tasks producing major work products. Example: Architectural design.
3. **Tasks:** Focused on small, well-defined objectives with tangible outcomes. Example: Conducting a unit test.

Process Framework

- A process framework establishes the foundation for a comprehensive software engineering process.
- It includes a small number of framework activities applicable to all software projects and umbrella activities that span the entire process.

The five generic framework activities are:

1. **Communication:** Critical to understanding stakeholders' objectives and gathering requirements for software features and functions.
2. **Planning:** Creates a "map" or software project plan that guides the team. It defines technical tasks, risks, resources, work products, and schedules.
3. **Modeling:** Involves creating models to understand software requirements and design.
4. Construction: Combines code generation (manual or automated) and testing to uncover errors in the code.
5. **Deployment:** Delivers the software to the customer for evaluation and feedback. These activities apply iteratively through a number of project iterations, each producing a software increment that adds to the overall functionality and completeness.

Umbrella Activities

- Umbrella activities help manage and control progress, quality, change, and risk throughout the project. They include:

1. **Software Project Tracking and Control:** Allows the team to assess progress against the plan and take necessary actions to maintain the schedule.
2. **Risk Management:** Assesses risks that may affect the project outcome or product quality.
3. **Software Quality Assurance:** Defines and conducts activities to ensure software quality.
4. **Technical Reviews:** Evaluates work products to uncover and remove errors.

5. **Measurement:** Defines and collects process, project, and product measures to assist in delivering software that meets stakeholders' needs.
6. **Software Configuration Management:** Manages the effects of change throughout the process.
7. **Reusability Management:** Defines criteria for work product reuse and establishes mechanisms to achieve reusable components.
8. **Work Product Preparation and Production:** Encompasses activities required to create work products like models, documents, logs, forms, and lists.

Flexibility and Adaptability

- The software engineering process should be agile and adaptable to various factors like the problem, project, team, and organizational culture. Differences between processes adopted for different projects may include:

- Overall flow of activities, actions, and tasks
- Degree to which actions and tasks are defined
- Identification and requirements of work products
- Application of quality assurance activities
- Project tracking and control activities
- Detail and rigor in process description
- Involvement of customers and stakeholders
- Team autonomy and organization

Prescriptive vs. Agile Process Models

- **Prescriptive Process Models:** Emphasize detailed definition and application of activities and tasks to improve system quality, manage projects, predict delivery dates and costs, and guide teams. However, if applied dogmatically, they can increase bureaucracy and create difficulties.

- **Agile Process Models:** Focus on project agility, emphasizing maneuverability and adaptability. They follow principles leading to a more informal yet effective approach to software processes. Agile models are suitable for many projects, especially when engineering web applications.

The Software Engineering Practice

A generic software process model was introduced, consisting of activities that form a framework for software engineering practice. These include:

Framework Activities:

- **Communication:** Engaging with stakeholders to understand needs and requirements.
- **Planning:** Creating a roadmap with tasks, resources, and schedules.
- **Modeling:** Visualizing and understanding the system's structure and behavior.
- **Construction:** Writing code and conducting testing to build the software.
- **Deployment:** Delivering the software to users and gathering feedback.

Umbrella Activities:

- **Project Tracking and Control:** Monitoring progress and adjusting tasks and schedules.
- **Risk Management:** Identifying and mitigating potential risks.
- **Software Quality Assurance:** Ensuring compliance with quality standards.
- **Technical Reviews:** Performing peer reviews to catch defects early.
- **Measurement:** Collecting and analyzing process and product metrics.
- **Software Configuration Management:** Controlling changes to software artifacts.
- **Reusability Management:** Promoting the use of reusable components.
- **Work Product Preparation and Production:** Preparing high-standard deliverables and documentation.

The Essence of Practice

George Polya's approach to problem-solving, outlined in "How to Solve It," forms the foundation for software engineering practice, leading to a series of essential steps and questions:

1. **Understand the Problem (Communication and Analysis)**

   Questions:

   - Who are the stakeholders?
   - What are the unknowns (data, functions, features)?
   - Can the problem be compartmentalized?
   - Can the problem be represented graphically?

2. **Plan a Solution (Modeling and Software Design)**

Questions:

- Have you seen similar problems before (patterns, existing software)?
- Has a similar problem been solved (reusable elements)?
- Can subproblems be defined with apparent solutions?
- Can a design model be created for effective implementation?

3. **Carry Out the Plan (Code Generation)**

Questions:

- Does the solution conform to the plan (source code traceable to design)?
- Is each component part of the solution provably correct (reviews, correctness proofs)?

4. **Examine the Result for Accuracy (Testing and Quality Assurance)**

Questions:

- Is it possible to test each component part of the solution (reasonable testing strategy)?
- Does the solution meet the required data, functions, and features (validation against stakeholder requirements)?

General Principles

David Hooker proposed seven principles that underpin effective software engineering practice

1. **The Reason It All Exists**
    - Software systems exist to provide value to users.
    - All decisions should prioritize adding real value to the system.

2. **KISS (Keep It Simple, Stupid!)**
    - Design should aim for simplicity while maintaining elegance and functionality.
    - Simplicity facilitates understanding and maintenance of the system.

3. **Maintain the Vision**
    - A clear vision ensures consistency and coherence in the system design.
    - Compromising the architectural vision weakens the system's integrity.

4. **What You Produce, Others Will Consume**
    - Design and implement with the awareness that others will use, maintain, and extend the system.
    - Consider the usability and understandability of the system for its audience.

5. **Be Open to the Future**
    - Design systems to be flexible and adaptable to changing requirements and environments.
    - Avoid designing systems into a corner and anticipate future changes.

6. **Plan Ahead for Reuse**
    - Forethought and planning are essential to maximize the benefits of code and design reuse.
    - Reuse saves time and effort in system development.

7. **Think!**
    - Intense thought before action leads to better results and reduces errors.
    - Clear, complete thought results in systems with higher value and quality.

The Software Myths

Management Myths:

- **Myth:** Having a book of standards and procedures is sufficient for software development.
- **Reality**: Often, existing standards are underutilized, outdated, and not adaptable to modern practices.
- **Myth:** Adding more programmers can accelerate a project that's behind schedule.
- **Reality:** Adding people to a late project can actually delay it further due to the time needed for integration and coordination.
- **Myth:** Outsourcing a software project means the organization can relax and let the third party handle it.

- **Reality:** Without internal project management capabilities, outsourcing can lead to project difficulties.

Customer Myths:

- **Myth:** A general statement of objectives is enough to start programming, with details filled in later.
- **Reality:** Ambiguous objectives often lead to project failures; clear requirements are essential.
- **Myth:** Software is flexible enough to accommodate continuous changes in requirements.
- **Reality:** While software can be modified, late changes can significantly increase costs and complexity.

Practitioner's Myths:

- **Myth:** The majority of effort in software development is expended before the program is delivered.
- **Reality:** A significant portion of effort occurs after the initial delivery, often exceeding 60-80%.
- **Myth:** Software quality can only be assessed once the program is running.
- **Reality:** Technical reviews are effective quality assurance mechanisms from project inception.
- **Myth:** A working program is the only deliverable for a successful project.
- **Reality:** Various work products, including models and documents, are essential for successful software engineering.
- **Myth:** Software engineering leads to excessive documentation and slows down projects.
- **Reality:** Software engineering focuses on creating quality products, which ultimately speeds up delivery by reducing rework.

How It All Starts

- Every software project originates from a business need, whether it's fixing a defect, adapting a legacy system, extending existing features, or creating something entirely new.
- Initially, the business need is often expressed informally through conversations.
- The importance of software in meeting the business need may not be explicitly discussed in these conversations.
- However, the success of the engineering effort ultimately depends on the success of the software.
- The acceptance of the product in the market hinges on how well the embedded software meets the customer's needs, which may not be fully articulated at the outset.
- The narrative of SafeHome software engineering will be explored further in subsequent chapters.

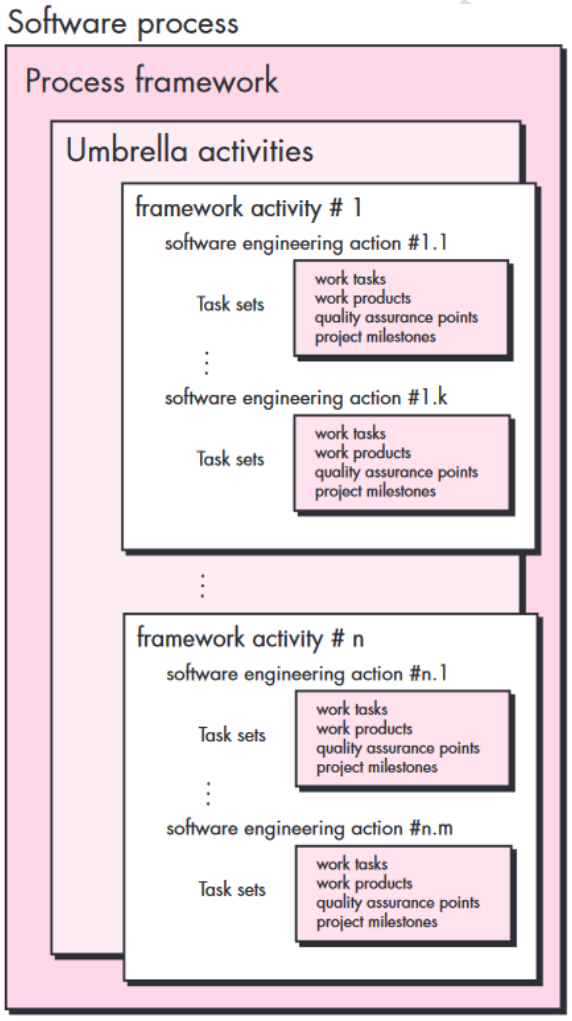
Process Models

Definition of Process:

- Collection of work activities, actions, and tasks.
- Reside within a framework or model defining their relationships.

Software Process Representation:

- Framework activity is populated by software engineering actions.
- Actions are defined by task sets (work tasks, work products, quality assurance points, and milestones).



Five Framework Activities:

1. Communication
2. Planning
3. Modeling
4. Construction
5. Deployment

Umbrella Activities (applied throughout the process):

1. Project tracking and control
2. Risk management
3. Quality assurance
4. Configuration management
5. Technical reviews
6. Others

Process Flow:

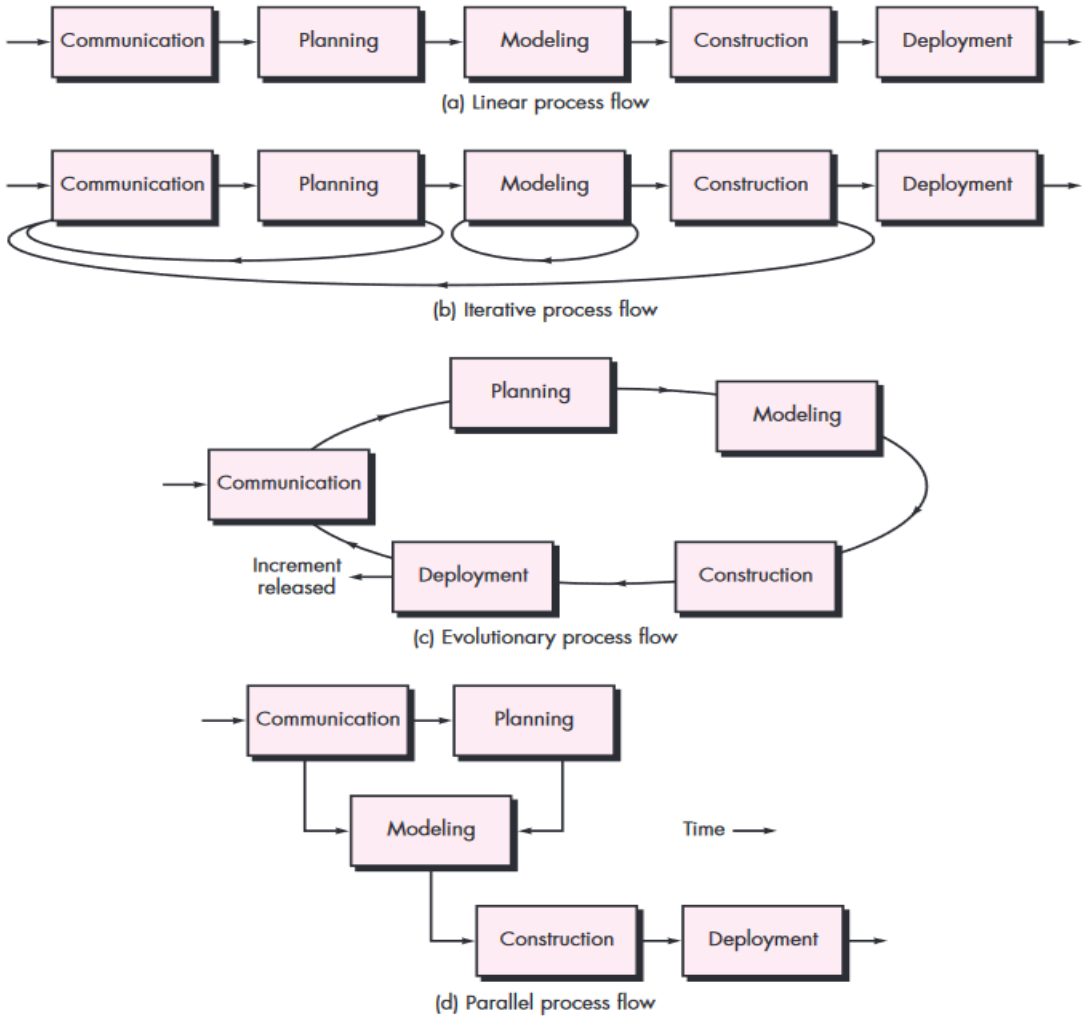
- Describes organization of framework activities, actions, and tasks with respect to sequence and time.

Types of Process Flows:

- Linear Process Flow: Executes each framework activity in sequence (Figure 2.2a).
- Iterative Process Flow: Repeats one or more activities before proceeding (Figure 2.2b).
- Evolutionary Process Flow: Activities executed in a circular manner, leading to a more complete version of the software (Figure 2.2c).
- Parallel Process Flow: Executes one or more activities in parallel with others (Figure 2.2d).

Defining a Framework Activity:

- Requires understanding the problem nature, characteristics of the team, and stakeholders.



Example:

- **Simple Project: Communication activity might involve:**
  1. Phone call with the stakeholder.
  2. Discussing requirements and taking notes.
  3. Organizing notes into a brief written statement of requirements.
  4. Emailing the statement to the stakeholder for review and approval.

- **Complex Project: Communication activity could include six actions:**
    - Inception
    - Elicitation
    - Elaboration
    - Negotiation
    - Specification
    - Validation

Each action has multiple tasks and work products.

Identifying a Task Set

1. **Software Engineering Action:**
    - Each action (e.g., elicitation) can be represented by different task sets.
    - A task set is a collection of work tasks, related work products, quality assurance points, and project milestones.
    - Choose a task set based on project needs and team characteristics.
    - Adapt the action to the specific project needs and team characteristics.

Process Patterns

Definition:

- Process patterns describe process-related problems encountered during software engineering work.
- Identify the environment where the problem is encountered.
- Suggest proven solutions.

Purpose:

- Provide templates for describing problem solutions within the software process context.
- Help solve problems and construct processes that meet project needs.

Types of Patterns (as proposed by Ambler):

- Stage Pattern: Defines problems associated with a framework activity (e.g., Establishing Communication).
- Task Pattern: Defines problems associated with a software engineering action or work task (e.g., Requirements Gathering).
- Phase Pattern: Defines the sequence of framework activities within the process (e.g., Spiral Model, Prototyping).

Template for Describing a Process Pattern:

- **Pattern Name:** Meaningful name describing the pattern within the software process context.
- **Forces:** Environment and issues making the problem visible and affecting its solution.
- **Type:** Specifies the pattern type (stage, task, or phase).
- **Initial Context:** Conditions under which the pattern applies, including prior organizational or team activities, entry state, and existing information.
- **Problem:** Specific problem to be solved by the pattern.
- **Solution:** How to implement the pattern successfully, modifying the initial state and transforming information.
- **Resulting Context:** Conditions resulting from successful implementation, including exit state and developed information.
- **Related Patterns:** List of directly related patterns, possibly represented as a hierarchy or diagram.
- **Known Uses and Examples:** Specific instances where the pattern is applicable.

Usage:

- Process patterns enable hierarchical process descriptions.
- Start at a high abstraction level (phase pattern), refine into stage patterns (framework activities), and further refine into task patterns.
- Patterns can be reused to define process variants, allowing customization of process models using patterns as building blocks.

Process assessment and improvement

Approaches to Software Process Assessment and Improvement:

SCAMPI (Standard CMMI Assessment Method for Process Improvement):

- Five-step process assessment model: initiating, diagnosing, establishing, acting, and learning.
- Uses SEI CMMI as the basis for assessment.

CBA IPI (CMM-Based Appraisal for Internal Process Improvement):

- Diagnostic technique for assessing the maturity of a software organization.
- Based on SEI CMM.

SPICE (ISO/IEC 15504):

- Standard defining requirements for software process assessment.
- Helps organizations objectively evaluate the efficacy of any defined software process.

ISO 9001:2000 for Software:

- Generic standard for improving the quality of products, systems, or services.
- Applicable to software organizations and companies.

Prescriptive Process Models

Purpose:

- Proposed to bring order to the chaos of software development.
- Provide structure to software engineering work and serve as a road map for software teams.

Edge of Chaos Concept:

- Defined as a natural state between order and chaos, balancing structure and surprise.
- Unstable and partially structured state, constantly attracted to chaos or absolute order.
- Absolute order implies no variability, while too much chaos hinders coordination and coherence.
- Some structure allows organized change, while too much rigidity prevents it.

Philosophical Implications:

- Prescriptive process models strive for structure and order.
- Debate on whether they are appropriate for a software world that thrives on change.
- Rejection of traditional models may lead to loss of coordination and coherence.

Characteristics of Prescriptive Process Models:

Prescribe process elements:

- Framework activities
- Software engineering actions
- Tasks
- Work products
- Quality assurance
- Change control mechanisms

Define a process flow (or work flow):

- Describes how process elements are interrelated.

Application:

- All process models accommodate the generic framework activities.
- Each model emphasizes different activities and defines a unique process flow.

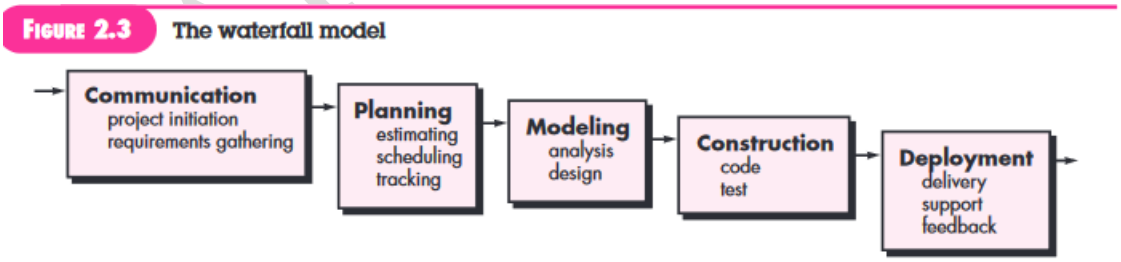
The Waterfall Model

Overview:

- Best suited for well-understood requirements where work flows linearly from communication to deployment.
- Often used for well-defined adaptations or enhancements to existing systems or in new development with stable requirements.

Description:

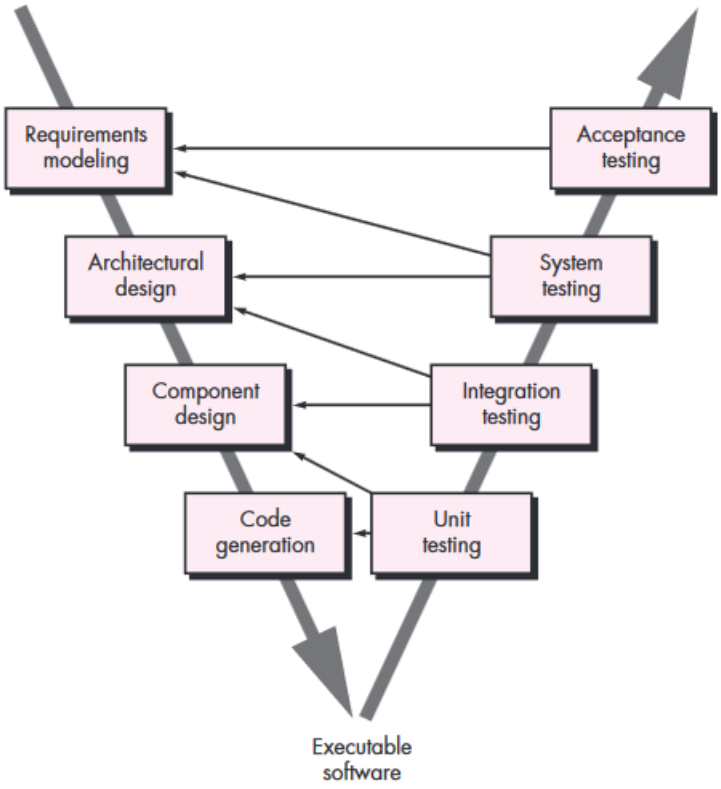
- Also known as the classic life cycle model.
- Suggests a systematic, sequential approach starting with customer specification and moving through planning, modeling, construction, deployment, and ongoing support.
- Visualized in Figure 2.3.



V-Model:

- A variation of the Waterfall Model (Figure 2.4).
- Depicts the relationship of quality assurance actions to communication, modeling, and early construction activities.
- Moves down the left side of the V to refine problem requirements and up the right side to validate each model through testing.
- No fundamental difference from the classic life cycle; provides visualization of verification and validation actions.

FIGURE 2.4
The V-model



Criticisms:

- Non-sequential Nature: Real projects rarely follow the proposed sequential flow. Iterations are accommodated indirectly, causing potential confusion during changes.
- Requirement Uncertainty: Difficult for customers to state all requirements explicitly at the beginning. The model struggles with the natural uncertainty of initial project phases.
- Delayed Working Version: Customers must wait until late in the project to see a working version. Undetected major errors can be disastrous.
- Blocking States: Linear nature can lead to blocking states where team members wait for others to complete dependent tasks. Waiting time can exceed productive work time, especially at the beginning and end of the process.

Incremental Process Models

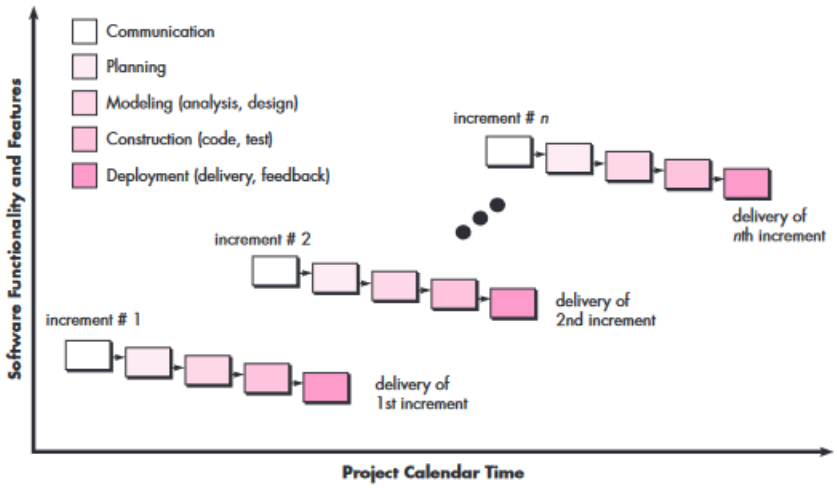
Purpose and Applicability:

- Suitable for situations where initial software requirements are well-defined, but the development scope precludes a purely linear process.
- Useful when there is a need to quickly provide limited software functionality and then expand it in later releases.

Description:

- Combines elements of linear and parallel process flows.
- Applies linear sequences in a staggered fashion over time.
- Each sequence produces deliverable "increments" of the software.
- Similar to evolutionary process flow but with a focus on producing operational software in stages.

FIGURE 2.5
The incremental model



Example:

- Word-processing software:
- First Increment: Basic file management, editing, and document production functions.
- Second Increment: More sophisticated editing and document production capabilities.
- Third Increment: Spelling and grammar checking.
- Fourth Increment: Advanced page layout capability.
- Each increment can incorporate the prototyping paradigm.

Process Flow:

- The first increment is often a core product addressing basic requirements.
- The core product undergoes use and evaluation by the customer.
- Feedback is used to develop a plan for the next increment.
- The process repeats until the complete product is produced.

Advantages:

- Delivers an operational product with each increment.
- Early increments are stripped-down versions but provide useful capabilities and a platform for user evaluation.
- Allows for the management of technical risks and staffing constraints.
- Incremental development enables partial functionality delivery, accommodating business deadlines and technical uncertainties.

Staffing and Technical Risk Management:

- Useful when complete implementation staffing is unavailable by the project deadline.
- Early increments can be implemented with fewer people.
- Additional staff can be added if the core product is well-received.
- Early increments can be planned to avoid dependencies on uncertain new hardware, allowing partial functionality delivery without delay.

Evolutionary Process Models

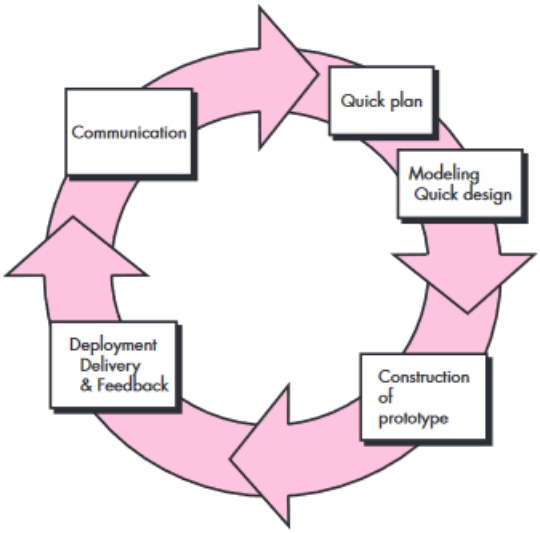
Purpose and Applicability:

Evolutionary process models are designed for scenarios where software requirements and market demands are dynamic. These models are iterative and focus on gradually developing increasingly complete versions of the software.

Characteristics:

- Iterative Development: The software evolves through repeated cycles, allowing for incremental improvements based on feedback and changing requirements.
- Adaptability: These models accommodate changes in requirements and deliver functional software at each iteration.
- Continuous Feedback: Stakeholders provide feedback at each iteration, refining requirements and guiding development.

FIGURE 2.6
The prototyping paradigm



Common Evolutionary Models:

Prototyping:

- Purpose: To clarify requirements and validate functionality when initial requirements are unclear or when experimenting with new technologies.

Process:

- Communication: Meet with stakeholders to define overall objectives, identify known requirements, and outline areas needing further definition.
- Quick Design: Create a simplified representation of the software, focusing on aspects visible to end users.
- Construction of Prototype: Build a working model based on the quick design.
- Evaluation: Deploy the prototype for stakeholder evaluation and gather feedback.
- Iteration: Refine the prototype based on feedback and repeat the process until requirements are well-understood.

Challenges:

- Misinterpretation: Stakeholders may mistake the prototype for the final product, unaware of its temporary nature and potential lack of quality.
- Compromises: Engineers might make suboptimal choices (e.g., using an inappropriate operating system or inefficient algorithms) to quickly develop the prototype, which may become hard to change later.

Guidelines for Effective Use:

- Clearly define the purpose of the prototype.
- Agree that the prototype is for requirements definition and will be discarded or significantly reworked for the final product.
- Focus on quality in the final engineering of the software.

Concurrent Models

Purpose and Applicability:

The concurrent development model, also known as concurrent engineering, is designed to handle iterative and parallel aspects of software development. This model is suitable for complex projects where various activities need to be performed concurrently and where dynamic changes in project states are frequent.

Characteristics:

- Parallelism: Multiple software engineering activities (e.g., communication, modeling, construction) occur simultaneously rather than sequentially.
- Dynamic State Management: Activities transition between different states based on specific events, allowing for flexible and adaptive development processes.
- Iterative Refinement: Iterative cycles are integrated, enabling continuous refinement and evolution of the software.

Key Components:

- Activities and States: Software engineering activities (such as modeling) can be in various states at any given time. Typical states include:
- Inactive: The activity is not currently being worked on.
- Under Development: The activity is actively being worked on.
- Awaiting Changes: The activity is paused, waiting for changes or further input.
- Done: The activity is completed but can be revisited if needed.
- Event-Driven Transitions: Specific events trigger transitions between states. For example, an inconsistency in the requirements model might trigger a transition from "done" to "awaiting changes."

Example Process Flow:

Initial Phase:

- Communication: Initially in the "under development" state as requirements are gathered and analyzed.
- Modeling: In the "inactive" state until initial communication is completed.

Transition Phase:

- Once initial requirements are gathered, the communication activity transitions to "awaiting changes."
- The modeling activity transitions to "under development" to start creating design models.

Dynamic Adjustments:

- If new requirements or changes are identified, the modeling activity might transition to "awaiting changes" while communication might re-enter the "under development" state to refine requirements.
- Construction and other activities can proceed in parallel, transitioning between states as necessary.

Advantages:

- Flexibility: Allows for adaptive and flexible project management, accommodating changes without significant disruption.
- Real-Time Progress Monitoring: Provides a real-time view of the project's status, helping in better tracking and management.
- Improved Coordination: Enables better coordination among team members as multiple activities are managed concurrently.

Challenges:

- Complexity: Managing concurrent activities can be complex and requires robust project management practices.
- Communication: Requires effective communication and collaboration among team members to ensure synchronization and avoid conflicts.

Applicability:

- Complex Projects: Suitable for projects with complex requirements and high levels of uncertainty.
- Large Teams: Beneficial for large teams where different sub-teams can work on different activities simultaneously.
- Dynamic Environments: Ideal for environments where requirements and technologies are rapidly evolving.

Specialized Process Models

Specialized process models take on many characteristics of traditional process models but are tailored for specific software engineering approaches. They are applied in situations requiring a unique or narrowly defined methodology.

Component-Based Development

- Component-based development (CBD) utilizes commercial off-the-shelf (COTS) software components to construct applications.
- This model is evolutionary, incorporating iterative development and focusing on assembling prepackaged software components with well-defined interfaces.

Key Steps:

- Research and Evaluation: Identify and assess available components for the application domain.
- Integration Considerations: Address issues related to the integration of selected components.
- Architectural Design: Develop a software architecture that accommodates the components.
- Component Integration: Integrate the components into the architecture.
- Comprehensive Testing: Ensure proper functionality through rigorous testing.

Advantages:

- Promotes software reuse.
- Reduces development time and costs.
- Leverages pre-built, tested components, potentially increasing reliability.

Challenges:

- Integration complexity.
- Dependency on third-party components.
- Potential issues with component compatibility and maintainability.

The Formal Methods Model

- The formal methods model employs rigorous mathematical notation to specify, develop, and verify software systems.
- This approach aims for high reliability and defect-free software through formal specification and mathematical analysis.

Key Characteristics:

- Mathematical Specification: Use of formal mathematical methods to specify system requirements.
- Program Verification: Employ mathematical techniques to verify software correctness.
- Error Detection: Discover and correct ambiguities, incompleteness, and inconsistencies through formal analysis.

Advantages:

- Potential for high reliability and defect-free software.
- Effective for safety-critical systems (e.g., medical devices, aircraft avionics).

Challenges:

- Time-consuming and expensive development process.
- Requires specialized training and expertise.
- Difficult to communicate formal models to non-technical stakeholders.

Aspect-Oriented Software Development (AOSD)

- AOSD, or aspect-oriented programming (AOP), addresses crosscutting concerns that affect multiple parts of a software system.
- These concerns can include systemic properties like security, fault tolerance, and transaction processing.

Key Concepts:

- Crosscutting Concerns: Issues that span multiple components or system functions.
- Aspects: Mechanisms for localizing the expression of crosscutting concerns beyond subroutines and inheritance.
- Aspectual Requirements: Define the impact of crosscutting concerns across the software architecture.

Development Approach:

- Horizontal Slices: Aspects are horizontal slices through vertically decomposed software components.
- Parallel Engineering: Aspects are engineered independently but have a direct impact on software components.
- Evolutionary and Concurrent: Combines evolutionary development of aspects with concurrent development of localized components.

Challenges:

- Immature process model.
- Requires asynchronous communication between aspect engineering and component development.
- Complexity in managing and integrating aspects with core components.