# Software Engineering And Project Management

Software Technology Engineering

Eduard Fekete

December 9, 2025

# Contents

# 1 Introduction

Software Engineering is the application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software. Unlike "programming" or "coding," which focuses on the act of writing instructions for a computer, software engineering focuses on the lifecycle of software systems, managing complexity, ensuring reliability, and scaling processes for teams.

Project Management in this context is the discipline of planning, organizing, and managing resources to bring about the successful completion of specific software project goals and objectives. It navigates the "Iron Triangle" of constraints: **Scope**, **Time**, and **Cost**, with **Quality** often sitting at the center or as a result of the balance.

The fundamental challenge of software engineering is **complexity**. As defined by Fred Brooks, software complexity is essential (inherent to the problem) or accidental (caused by the tools/methods). Engineering methodologies aim to minimize accidental complexity and manage essential complexity.

# 2 Software Process Models

A software process model is an abstraction of the software development process. It specifies the stages and order of a process.

## 2.1 The Waterfall Model

The classical linear approach. Activity flows downward through phases like a waterfall.

| Phase | Description |
|---|---|
| **Requirements** | Capturing all requirements upfront. |
| **Analysis** | detailed analysis of the system needs. |
| **Design** | Architecture and detailed design. |
| **Implementation** | Coding the system. |
| **Testing** | Verifying the system against requirements. |
| **Deployment** | Releasing to production. |
| **Maintenance** | Fixing bugs and adding features post-release. |

**Critique:**

- **Rigidity:** Difficult to go back to a previous phase.
- **Late Testing:** Integration happens at the end, often leading to "big bang" integration failures.
- **Risk:** High risk of building the wrong thing because the user sees the product only at the end.
- **Assumption:** Assumes requirements are static and fully knowable at the start (rarely true).

## 2.2 The V-Model

An extension of Waterfall that emphasizes the relationship between development phases and testing phases.

- **Verification:** Are we building the product right? (Left side of V)
- **Validation:** Are we building the right product? (Right side of V)

## 2.3 Iterative and Incremental

- **Iterative:** The system is developed through repeated cycles (iterative) and in smaller portions at a time (incremental).
- **Benefit:** Early feedback, risk management, and adaptability.

# 3 Agile Methodology

Agile is not a process but a philosophy or mindset defined by the **Agile Manifesto** (2001). It emerged as a reaction to heavy-weight, documentation-driven methods (like Waterfall).

## 3.1 The Four Values

1. **Individuals and interactions** over processes and tools.
2. **Working software** over comprehensive documentation.
3. **Customer collaboration** over contract negotiation.
4. **Responding to change** over following a plan.

## 3.2 The Twelve Principles (Summary)

1. Satisfy the customer through early and continuous delivery.
2. Welcome changing requirements, even late in development.
3. Deliver working software frequently.
4. Business people and developers must work together daily.
5. Build projects around motivated individuals.
6. Face-to-face conversation is the best form of communication.
7. Working software is the primary measure of progress.
8. Sustainable development (constant pace).
9. Technical excellence and good design enhance agility.
10. Simplicity (the art of maximizing work not done) is essential.
11. Self-organizing teams generate the best architectures.
12. Regular reflection and adjustment (Retrospectives).

# 4 Scrum

Scrum is an agile framework for developing, delivering, and sustaining complex products. It is an empirical process control theory based on **Transparency**, **Inspection**, and **Adaptation**.

## 4.1 The Scrum Team

| Role | Responsibilities |
|---|---|
| **Product Owner (PO)** | Maximizes value. Manages the Product Backlog. Represents the stakeholders/customers. Decides *what* to build. |
| **Scrum Master** | Servant-leader. Coaches the team. Removes impediments. Ensures Scrum is understood and enacted. |
| **Developers** | Cross-functional, self-organizing group. Creates the "Done" increment. Decides *how* to build. |

## 4.2 Scrum Artifacts

1. **Product Backlog:** Ordered list of everything that is known to be needed in the product. Dynamic and living.
2. **Sprint Backlog:** The set of Product Backlog items selected for the Sprint plus a plan for delivering them.
3. **Increment:** The sum of all Product Backlog items completed during a Sprint and the value of the increments of all previous Sprints. Must meet the **Definition of Done (DoD)**.

## 4.3 Scrum Events

1. **Sprint:** A container for all other events. Fixed length (time-boxed), usually 2-4 weeks.
2. **Sprint Planning:** Team collaborates to select work for the Sprint and define a Sprint Goal.
3. **Daily Scrum:** 15-minute event for developers to synchronize activities and plan for the next 24 hours. (Stand-up).
4. **Sprint Review:** Inspect the outcome of the Sprint (the Increment) and determine future adaptations. Stakeholders present.
5. **Sprint Retrospective:** The team inspects itself to create a plan for improvements to be enacted during the next Sprint.

# 5 Extreme Programming (XP)

XP is an Agile software development framework that aims to improve software quality and responsiveness to changing customer requirements. It advocates frequent "releases" in short development cycles to improve productivity and introduce checkpoints.

## 5.1 Core Values

Communication, Simplicity, Feedback, Courage, Respect.

## 5.2 Key Practices

### 5.2.1 Fine-Scale Feedback

- **Pair Programming:** Two developers work on one workstation. One writes code (Driver), the other reviews it instantly (Navigator). Increases code quality and knowledge sharing.
- **Planning Game:** Determine scope of next release by combining business priorities and technical estimates.
- **Test-Driven Development (TDD):** Write unit tests *before* writing the code.
- **Whole Team:** Customer is part of the team and available daily.

### 5.2.2 Continuous Process

- **Continuous Integration (CI):** Integrate and build the system many times a day.
- **Refactoring:** Restructuring existing code without changing its external behavior.
- **Small Releases:** Put a simple system into production quickly, then release new versions on a very short cycle.

### 5.2.3 Shared Understanding

- **Coding Standards:** Programmers write code in accordance with rules emphasizing communication.
- **Collective Code Ownership:** Anyone can change any code anywhere in the system at any time.
- **Simple Design:** The system should be designed as simply as possible at any given moment. Extra complexity is removed as soon as it is discovered.
- **System Metaphor:** A shared story of how the system works.

# 6 Requirements Engineering

The process of defining, documenting, and maintaining requirements.

## 6.1 Types of Requirements

1. **Functional Requirements:** What the system should *do*. (e.g., "The system shall allow users to log in.")
2. **Non-Functional Requirements (NFRs / Quality Attributes):** How the system should *behave*. (e.g., Performance, Scalability, Security, Usability). Often called "-ilities".

## 6.2 User Stories

A standard format in Agile to capture functional requirements. Format: `As a <role>, I want <feature> so that <benefit>.`

### 6.2.1 INVEST Criteria

A good User Story must be:

- **I**ndependent
- **N**egotiable
- **V**aluable
- **E**stimable
- **S**mall
- **T**estable

## 6.3 Estimation Techniques

- **Planning Poker:** Consensus-based technique to estimate effort or relative size of user stories (using Fibonacci sequence: 1, 2, 3, 5, 8, 13...).
- **T-Shirt Sizing:** XS, S, M, L, XL for high-level estimation.

# 7 Unified Modeling Language (UML)

UML is a standardized modeling language consisting of an integrated set of diagrams, developed to help system and software developers for specifying, visualizing, constructing, and documenting the artifacts of software systems.

## 7.1 Structural Diagrams (Static)

### 7.1.1 Class Diagram

Describes the structure of a system by showing the system's classes, their attributes, operations (methods), and the relationships among objects.

**Relationships:**

- **Association:** A generic link (solid line).
- **Inheritance (Generalization):** "Is-a" relationship (solid line with hollow triangle pointing to parent).
- **Realization/Implementation:** Implementing an interface (dashed line with hollow triangle).
- **Dependency:** Change in one affects the other (dashed line with open arrow).
- **Aggregation:** "Has-a" relationship, weak ownership (solid line with hollow diamond). Child can exist without parent.
- **Composition:** "Part-of" relationship, strong ownership (solid line with filled diamond). Child cannot exist without parent.

### 7.1.2 Component Diagram

Depicts how components are wired together to form larger components or software systems.

### 7.1.3 Deployment Diagram

Models the physical deployment of artifacts on nodes (hardware).

## 7.2 Behavioral Diagrams (Dynamic)

### 7.2.1 Use Case Diagram

Describes the functional behavior of the system as seen by external users (Actors).

- **Actor:** Stick figure representing a role.
- **Use Case:** Oval representing a function.
- **Relationships:** Include (mandatory), Extend (optional).

### 7.2.2 Sequence Diagram

Shows object interactions arranged in time sequence.

- **Lifeline:** Dashed vertical line representing the existence of an object over time.
- **Activation Bar:** Thin rectangle on lifeline showing when object is active.
- **Messages:** Horizontal arrows (Synchronous = solid arrowhead, Asynchronous = open arrowhead).

### 7.2.3 State Machine Diagram

Describes the states of an object and the transitions between those states triggered by events.

### 7.2.4 Activity Diagram

Graphical representations of workflows of stepwise activities and actions with support for choice, iteration, and concurrency. Similar to flowcharts but with support for parallel processing.

# 8 Design Patterns

Reusable solutions to common problems in software design. Originated by the "Gang of Four" (GoF).

## 8.1 Creational Patterns

Deal with object creation mechanisms.

- **Singleton:** Ensures a class has only one instance and provides a global point of access to it.
- **Factory Method:** Defines an interface for creating an object but lets subclasses decide which class to instantiate.
- **Builder:** Separates the construction of a complex object from its representation.

## 8.2 Structural Patterns

Deal with object composition.

- **Adapter:** Allows incompatible interfaces to work together (wrapper).
- **Facade:** Provides a simplified interface to a library, a framework, or any other complex set of classes.
- **Composite:** Composes objects into tree structures to represent part-whole hierarchies.

## 8.3 Behavioral Patterns

Deal with communication between objects.

- **Observer:** Defines a one-to-many dependency so that when one object changes state, all its dependents are notified (Publish-Subscribe).
- **Strategy:** Defines a family of algorithms, encapsulates each one, and makes them interchangeable.
- **Command:** Encapsulates a request as an object, thereby letting you parameterize clients with different requests.

# 9 Quality Assurance and Testing

## 9.1 The Testing Pyramid

A strategy that suggests having many low-level tests and fewer high-level tests.

1. **Unit Tests (Base):** Test individual components/functions in isolation. Fast, cheap, numerous.
2. **Integration Tests (Middle):** Test interactions between modules or services. Slower, more complex.
3. **End-to-End (E2E) / UI Tests (Top):** Test the full application flow from a user's perspective. Slow, brittle, expensive.

## 9.2 Testing Types

- **Black Box Testing:** Testing without knowledge of internal code structure. Focuses on inputs and outputs.
- **White Box Testing:** Testing with knowledge of internal logic, branches, and paths.
- **Regression Testing:** Verifying that recent changes have not broken existing functionality.
- **Acceptance Testing:** Determining if the requirements of a specification or contract are met.

## 9.3 Test Driven Development (TDD)

A development cycle where tests are written *before* the code.

### 9.3.1 The Red-Green-Refactor Cycle

1. **Red:** Write a failing test for a small piece of functionality.
2. **Green:** Write just enough code to make the test pass.
3. **Refactor:** Clean up the code (remove duplication, improve logic) while keeping the test green.

**Benefits:**

- High code coverage.
- Design quality (code must be testable).
- Living documentation.
- Confidence to refactor.

# 10 Refactoring

Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure.

## 10.1 Code Smells

Indicators of deeper problems in the code.

- **Duplicated Code:** Copy-paste logic.
- **Long Method:** A method doing too much.
- **Large Class:** A class trying to do too much (God Class).
- **Feature Envy:** A method that seems more interested in a class other than the one it actually is in.
- **Shotgun Surgery:** A single change requires making many little changes to many different classes.

## 10.2 Refactoring Techniques

- **Extract Method:** Turn a code fragment into a method whose name explains the purpose of the method.
- **Rename Method:** Change the name of a method to reveal its purpose.
- **Move Method:** Move a method to the class where it is used most.
- **Replace Conditional with Polymorphism:** Move each leg of the conditional to an overriding method in a subclass.

## 10.3 Technical Debt

A metaphor reflecting the implied cost of additional rework caused by choosing an easy (limited) solution now instead of using a better approach that would take longer.

- **Reckless/Inadvertent:** "What's layering?"
- **Prudent/Deliberate:** "We must ship now and deal with consequences later."

# 11 Group Dynamics and Formation

Software engineering is a human activity. Understanding team dynamics is crucial for project success.

## 11.1 Tuckman's Stages of Group Development

1. **Forming:** The team meets and learns about the opportunities and challenges, and then agrees on goals and begins to tackle the tasks. Team members tend to behave quite independently.
2. **Storming:** Different ideas compete for consideration. The team addresses issues such as what problems they are supposed to solve, how they will function independently and together and what leadership model they will accept. Conflict is common here.
3. **Norming:** The team manages to have one goal and come to a mutual plan for the team at this stage. Some team members may have to give up their own ideas and agree with others to make the team function.
4. **Performing:** The team members are now competent, autonomous and able to handle the decision-making process without supervision. High performance.
5. **Adjourning:** Breaking up the team after the task is completed.

## 11.2 Brooks's Law

> "Adding manpower to a late software project makes it later."

**Reasoning:**

- **Ramp up time:** New people need time to become productive (learning the codebase).
- **Communication overhead:** The number of communication channels increases quadratically with the number of people. Formula for channels: $\frac{n(n-1)}{2}$.

## 11.3 The Bus Factor

The minimum number of team members that have to disappear from a project (hit by a bus) before the project stalls due to lack of knowledge. High bus factor is good; low bus factor (1) is a risk.

# 12 DevOps and Modern Practices

## 12.1 Continuous Integration (CI)

The practice of merging all developers' working copies to a shared mainline continuously (usually at least several times a day).

- **Automated Builds:** The system is compiled automatically.
- **Automated Testing:** Unit and integration tests run on every commit.

## 12.2 Continuous Delivery (CD)

An extension of CI where the software is always in a deployable state. Deployment to production can be triggered manually or automatically.

## 12.3 Version Control (Git)

Distributed version control system.

- **Commit:** A snapshot of changes.
- **Branch:** A parallel version of the code.
- **Merge:** Combining branches.
- **Pull Request (PR):** A mechanism for code review before merging changes.

## 12.4 Documentation

- **Code as Documentation:** Clean code with meaningful naming is the best primary documentation.
- **Internal Documentation:** Comments (only 'why', not 'what'), Readme, Architecture Decision Records (ADR).
- **External Documentation:** User manuals, API references (Swagger/OpenAPI).

# 13 Ethics in Software Engineering (ACM/IEEE Code)

Software engineers shall commit themselves to making the analysis, specification, design, development, testing and maintenance of software a beneficial and respected profession.

1. **Public:** Act consistently with the public interest.
2. **Client and Employer:** Act in a manner that is in the best interests of their client and employer consistent with the public interest.
3. **Product:** Ensure that their products and related modifications meet the highest professional standards possible.
4. **Judgment:** Maintain integrity and independence in their professional judgment.