The Complete Software Engineering Lifecycle

Methods, Patterns, and Implementation

Dr. Moody Amakobe

2025-12-08

Table of contents

The Complete Software Engineering Lifecycle

Methods, Patterns, and Implementation

Introduction

Welcome

Welcome to *The Complete Software Engineering Lifecycle*!
This open textbook is designed for graduate students, practitioners, and educators who want a modern, practical, and project-driven exploration of software engineering.

The book follows the **full lifecycle of software development**—from requirements gathering to deployment and long-term maintenance—integrating both **industry best practices** and **academic rigor**.
It is written to accompany a 16-week graduate course but can also be used independently by teams and self-learners.

Abstract

Software engineering is more than just writing code—it is a disciplined approach to **designing, building, testing, deploying, and evolving complex software systems**.
This book blends **technical foundations**, **architectural patterns**, and **hands-on exercises** that mirror the workflows used by professional engineering teams.

We explore:

- Requirements engineering and documentation
- UML and systems modeling
- Software architecture and design patterns
- Version control and collaborative development
- Testing methodologies & quality assurance
- DevOps, CI/CD, and cloud deployment strategies
- Security, maintainability, and long-term evolution

By the end, you will have both the **knowledge** and the **applied experience** to engineer robust, scalable, and maintainable software systems—supported by a semester-long project that builds from chapter to chapter.

Learning Objectives

By working through this book, you will be able to:

- Analyze user needs and translate them into actionable software requirements
- Model systems using UML and architectural design principles
- Apply software design patterns to build modular, extensible codebases
- Use Git and GitHub effectively for collaborative development
- Implement testing strategies across unit, integration, and acceptance levels
- Deploy applications using modern DevOps and cloud technologies
- Integrate security, maintainability, and quality assurance into every stage of development
- Deliver a complete, professional software project—from concept to deployment

License

This book is published by **Global Data Science Institute (GDSI)** as an
Open Educational Resource (OER).

It is licensed under the **Creative Commons Attribution 4.0 International (CC BY 4.0)** license.

You are free to **share**, **adapt**, and **build upon** this material for any purpose—even commercially—so long as proper attribution is provided.

Figure 1: CC BY 4.0

How to Use This Book

- The **HTML edition** is recommended for the best interactive reading experience.
- **PDF** and **EPUB** versions are available for offline reading.
- Code examples and templates are included in the `/assets/code/` directory.

- Each chapter includes a **project milestone**, allowing you to build a complete software system as you progress.
- This book pairs seamlessly with GitHub Classroom, GitHub Projects, and modern DevOps workflows.

Sample content for preface.qmd

Sample content for acknowledgments.qmd

Chapter 1: Introduction to Software Engineering

Learning Objectives

By the end of this chapter, you will be able to:

- Define software engineering and explain its significance in modern technology
- Describe the evolution of software engineering as a discipline
- Compare and contrast major software development lifecycle (SDLC) models
- Understand the fundamentals of version control using Git and GitHub
- Apply collaborative workflows for team-based software development
- Set up a project repository with proper structure and documentation

---

1.1 What Is Software Engineering?

Imagine you're building a house. You wouldn't just start stacking bricks randomly and hope for the best, would you? You'd need blueprints, a foundation plan, electrical and plumbing designs, a construction schedule, quality inspections, and a team of specialists working together. Building software is remarkably similar—except instead of bricks and mortar, we work with code, data, and digital infrastructure.

Software engineering is the systematic application of engineering principles to the design, development, testing, deployment, and maintenance of software systems. It's not just about writing code that works; it's about writing code that works *reliably*, *efficiently*, and *maintainably* over time.

The IEEE (Institute of Electrical and Electronics Engineers) defines software engineering as:

> "The application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software."

This definition highlights several key aspects:

- **Systematic**: Following organized methods and processes
- **Disciplined**: Adhering to standards and best practices
- **Quantifiable**: Measuring progress, quality, and outcomes
- **Comprehensive**: Covering the entire lifecycle, not just coding

1.1.1 Software Engineering vs. Programming

A common misconception among newcomers is that software engineering and programming are the same thing. While programming is certainly a core skill within software engineering, the discipline encompasses much more.

Aspect	Programming	Software Engineering
Focus	Writing code to solve specific problems	Designing and building complete systems
Scope	Individual tasks or features	Entire product lifecycle
Timeline	Short-term	Long-term (years of maintenance)
Team Size	Often individual	Usually collaborative
Documentation	Optional or minimal	Essential and comprehensive
Quality Assurance	Ad-hoc testing	Systematic testing strategies
Process	Flexible, informal	Structured methodologies

Think of it this way: a programmer might write an excellent function to sort a list of names. A software engineer asks questions like: How will this sorting function integrate with the rest of the system? What happens when the list contains millions of names? How will we test it? Who will maintain it? How do we deploy updates without breaking existing functionality?

1.1.2 A Brief History of Software Engineering

The term "software engineering" was first coined at the 1968 NATO Software Engineering Conference in Garmisch, Germany. This conference was convened in response to what was then called the **software crisis**—a period when software projects were consistently failing, running over budget, delivering late, and producing unreliable results.

In the early days of computing (1940s-1960s), software was often an afterthought. Hardware was expensive and precious; software was seen as a minor component. Programs were small, written by individuals, and often tied to specific machines. Documentation was rare, and the concept of "maintenance" barely existed—if a program didn't work, you wrote a new one.

As computers became more powerful and widespread, software grew in complexity. The 1960s saw ambitious projects like IBM's OS/360 operating system, which employed thousands of programmers and took years longer than planned. Frederick Brooks, who managed that project, later wrote "The Mythical Man-Month," a seminal book that observed:

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

This counterintuitive insight—that you can't just throw more programmers at a problem to solve it faster—underscored the need for better engineering practices.

The decades that followed brought waves of innovation in how we approach software development:

- **1970s**: Structured programming and the Waterfall model emerged

- **1980s**: Object-oriented programming and CASE (Computer-Aided Software Engineering) tools
- **1990s**: Component-based development, the rise of the internet, and early Agile ideas
- **2000s**: Agile Manifesto (2001), widespread adoption of iterative methods
- **2010s**: DevOps culture, continuous delivery, cloud computing
- **2020s**: AI-assisted development, platform engineering, and infrastructure as code

Today, software engineering continues to evolve rapidly. The principles you'll learn in this course represent decades of accumulated wisdom from millions of projects—both successful and failed.

1.2 The Role of Software Engineering in Modern Systems

Software has become the invisible infrastructure of modern civilization. Consider a typical morning: your smartphone alarm wakes you (software), you check the weather app (software connecting to distributed systems), your smart thermostat adjusts the temperature (embedded software), you drive to work with GPS navigation (software integrating satellite data), and you buy coffee with a tap of your phone (financial software processing transactions across multiple systems).

1.2.1 Software Is Everywhere

The scale of software's presence in our world is staggering:

Transportation: Modern vehicles contain 100+ million lines of code. The Boeing 787 Dreamliner runs on approximately 6.5 million lines of code. Self-driving cars process terabytes of sensor data through sophisticated software systems.

Healthcare: Electronic health records, diagnostic imaging systems, robotic surgery equipment, drug interaction databases, and pandemic tracking systems all depend on reliable software engineering.

Finance: High-frequency trading systems execute millions of transactions per second. Banking apps handle trillions of dollars in transfers. Cryptocurrencies run on complex distributed software systems.

Communication: Social media platforms serve billions of users simultaneously. Video conferencing software enables global collaboration. Messaging apps deliver hundreds of billions of messages daily.

Infrastructure: Power grids, water treatment plants, air traffic control systems, and emergency services all rely on software that must work correctly, all the time.

1.2.2 The Cost of Software Failures

When software fails, the consequences can range from minor inconveniences to catastrophic disasters. Understanding these failures helps us appreciate why rigorous software engineering practices matter.

The Therac-25 Accidents (1985-1987): A radiation therapy machine's software bugs caused massive overdoses, killing at least three patients and seriously injuring others. The failures resulted from poor software design, inadequate testing, and the removal of hardware safety interlocks that had been present in earlier models.

Ariane 5 Explosion (1996): The European Space Agency's rocket exploded 37 seconds after launch, resulting in a $370 million loss. The cause? A software error—specifically, an integer overflow when 64-bit floating-point data was converted to a 16-bit signed integer. Code reused from the Ariane 4 hadn't been tested for the new rocket's different flight parameters.

Knight Capital Glitch (2012): A software deployment error caused a trading firm to lose $440 million in just 45 minutes. Old, deprecated code was accidentally activated, executing millions of unintended trades. The company nearly went bankrupt overnight.

Healthcare.gov Launch (2013): The U.S. government's health insurance marketplace website failed spectacularly at launch, unable to handle user traffic and plagued with bugs. The problems stemmed from inadequate testing, poor project management, and insufficient integration between components built by different contractors.

These examples share common themes: inadequate testing, poor communication, rushed timelines, and insufficient attention to software engineering principles. They demonstrate that software engineering isn't just an academic exercise—it's a matter of safety, economics, and public trust.

1.2.3 The Value of Good Software Engineering

Conversely, excellent software engineering creates enormous value:

Reliability: Well-engineered systems work correctly, consistently, over time. Users trust them.

Scalability: Properly architected systems can grow to serve millions or billions of users without fundamental redesigns.

Maintainability: Good engineering practices make it possible to fix bugs, add features, and adapt to changing requirements efficiently.

Security: Systematic approaches to security protect users' data and privacy.

Cost Efficiency: While good engineering requires upfront investment, it dramatically reduces long-term costs by preventing bugs, reducing technical debt, and enabling faster development of new features.

1.3 The Software Development Life Cycle (SDLC)

The **Software Development Life Cycle** (SDLC) is a framework that describes the stages involved in building software, from initial concept through deployment and maintenance. Think of it as a roadmap for transforming an idea into a working system.

While different methodologies organize these stages differently, most include some version of:

1. **Requirements**: What should the system do?
2. **Design**: How will the system be structured?
3. **Implementation**: Writing the actual code
4. **Testing**: Verifying the system works correctly
5. **Deployment**: Releasing the system to users
6. **Maintenance**: Ongoing updates, fixes, and improvements

Different SDLC models arrange these stages in different ways, with different philosophies about planning, flexibility, and iteration. Let's explore the major models you'll encounter in professional practice.

1.3.1 The Waterfall Model

The **Waterfall model** is the oldest and most traditional approach to software development. Introduced by Winston Royce in 1970 (though he actually presented it as an example of a flawed approach!), it organizes development into sequential phases that flow downward, like a waterfall.

```
  Requirements

    Design

Implementation

   Testing

  Deployment

  Maintenance
```

Key Characteristics:

- Each phase must be completed before the next begins
- Extensive documentation at each stage
- Formal reviews and sign-offs between phases
- Changes are difficult and expensive once a phase is complete
- Testing occurs late in the process

When Waterfall Works Well:

- Requirements are well-understood and unlikely to change
- The technology is mature and well-known
- The project is relatively short
- Regulatory compliance requires extensive documentation
- The customer can articulate complete requirements upfront

When Waterfall Struggles:

- Requirements are unclear or likely to evolve
- The project is long-term (requirements will change)
- Rapid feedback is needed
- Innovation or experimentation is involved
- The customer wants to see working software early

Example Scenario: Developing software for a medical device that must meet FDA regulations might use Waterfall. The requirements are clear (based on medical standards), extensive documentation is mandatory, and changes after approval are extremely costly.

1.3.2 Agile Methodology

Agile is not a single methodology but a family of approaches that share common values and principles. The Agile Manifesto, published in 2001, articulates four core values:

> **Individuals and interactions** over processes and tools
> **Working software** over comprehensive documentation
> **Customer collaboration** over contract negotiation
> **Responding to change** over following a plan

This doesn't mean Agile ignores processes, documentation, contracts, or plans—but it prioritizes the items on the left when trade-offs must be made.

The Twelve Principles of Agile Software:

1. Satisfy the customer through early and continuous delivery of valuable software
2. Welcome changing requirements, even late in development
3. Deliver working software frequently (weeks rather than months)
4. Business people and developers must work together daily
5. Build projects around motivated individuals; give them support and trust
6. Face-to-face conversation is the most effective communication method
7. Working software is the primary measure of progress

8. Maintain a sustainable pace indefinitely
9. Continuous attention to technical excellence and good design
10. Simplicity—maximizing work not done—is essential
11. Self-organizing teams produce the best architectures and designs
12. Regular reflection on how to become more effective

Common Agile Frameworks:

Scrum is the most popular Agile framework. It organizes work into fixed-length iterations called *sprints* (typically 2-4 weeks). Key elements include:

- **Product Backlog**: Prioritized list of features and requirements
- **Sprint Planning**: Team commits to work for the upcoming sprint
- **Daily Standups**: Brief daily meetings to synchronize the team
- **Sprint Review**: Demonstration of completed work to stakeholders
- **Sprint Retrospective**: Team reflects on process improvements
- **Roles**: Product Owner, Scrum Master, Development Team

Kanban focuses on visualizing workflow and limiting work in progress. Work items move across a board through stages (e.g., To Do → In Progress → Review → Done). Unlike Scrum, Kanban doesn't use fixed-length iterations.

Extreme Programming (XP) emphasizes technical practices like pair programming, test-driven development, continuous integration, and frequent releases.

When Agile Works Well:

- Requirements are expected to change
- Customer feedback is available regularly
- The team is co-located or has good communication tools
- The organization supports iterative delivery
- Innovation and adaptation are valued

When Agile Struggles:

- Fixed-price contracts with rigid specifications
- Distributed teams with poor communication
- Regulatory environments requiring extensive upfront documentation
- Customers unwilling or unable to participate actively
- Very large-scale projects without proper scaling frameworks

1.3.3 The Spiral Model

The **Spiral model**, proposed by Barry Boehm in 1986, emphasizes **risk management**. Development proceeds through multiple iterations, each passing through four phases:

1. **Planning**: Determine objectives, alternatives, and constraints
2. **Risk Analysis**: Identify and evaluate risks; create prototypes
3. **Engineering**: Develop and verify the product
4. **Evaluation**: Review results and plan the next iteration

```
                    Planning




    Evaluation         Risk Analysis




                   Engineering

                    (repeat)
```

Each loop around the spiral represents a more complete version of the software. Early iterations might produce paper prototypes or proof-of-concept code; later iterations produce the actual system.

Key Characteristics:

- Explicit focus on identifying and mitigating risks
- Combines iterative development with systematic aspects of Waterfall
- Prototyping used to reduce uncertainty
- Flexibility to adapt the process to project needs

When Spiral Works Well:

- Large, complex projects
- High-risk systems where failure would be catastrophic
- Projects with uncertain or evolving requirements
- Situations requiring significant prototyping

1.3.4 DevOps

DevOps represents a cultural and technical movement that bridges the traditional gap between development (Dev) and operations (Ops) teams. Rather than a distinct SDLC model, DevOps is a set of practices that can be combined with other methodologies.

Traditionally, developers wrote code and "threw it over the wall" to operations teams, who were responsible for deploying and maintaining it in production. This separation created friction: developers optimized for features and speed; operations optimized for stability and reliability. The result was slow deployments, finger-pointing when problems occurred, and systems that worked in development but failed in production.

DevOps breaks down these silos through:

Cultural Practices:

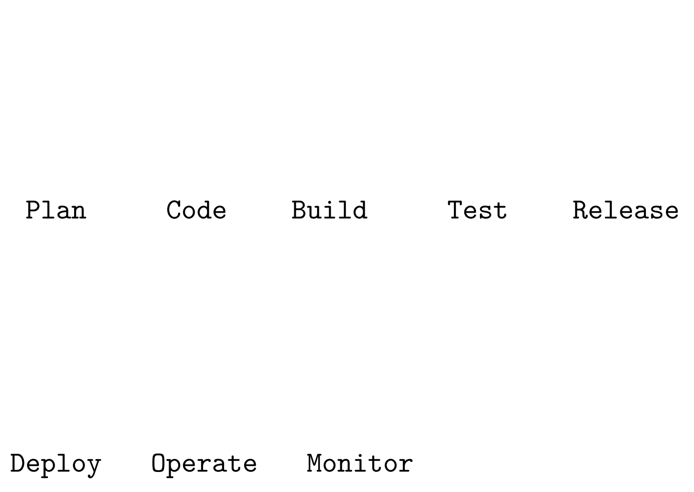
- Shared responsibility for the entire lifecycle
- Blameless post-mortems when things go wrong
- Continuous learning and improvement

- Collaboration between all roles

Technical Practices:

- **Continuous Integration (CI)**: Automatically building and testing code whenever changes are committed
- **Continuous Delivery (CD)**: Keeping software in a deployable state at all times
- **Continuous Deployment**: Automatically deploying every change that passes tests
- **Infrastructure as Code**: Managing servers and environments through version-controlled scripts
- **Monitoring and Logging**: Comprehensive visibility into system behavior
- **Automated Testing**: Extensive test suites that run automatically

The DevOps Lifecycle:



The cycle is continuous—monitoring in production feeds back into planning for the next iteration.

Key DevOps Metrics:

- **Deployment Frequency**: How often you release to production
- **Lead Time for Changes**: Time from commit to production
- **Mean Time to Recovery (MTTR)**: How quickly you recover from failures
- **Change Failure Rate**: Percentage of deployments causing problems

High-performing DevOps organizations deploy multiple times per day, with lead times measured in hours, recover from failures in minutes, and have change failure rates below 15%.

1.3.5 Choosing an SDLC Model

No single model is universally best. The right choice depends on your project's characteristics:

Factor	Waterfall	Agile	Spiral	DevOps
Requirement stability	High	Low	Variable	Variable
Project size	Any	Small-Medium	Large	Any

Factor	Waterfall	Agile	Spiral	DevOps
Risk level	Low	Low-Medium	High	Variable
Customer involvement	Low	High	Medium	Medium
Documentation needs	High	Low-Medium	High	Medium
Delivery frequency	End	Frequent	Iterative	Continuous
Team experience	Any	Experienced	Experienced	Experienced

In practice, many organizations use hybrid approaches. For example, a team might use Scrum for iteration planning while implementing DevOps practices for CI/CD, or use a Spiral approach at the program level while individual teams work in Agile sprints.

1.4 Version Control with Git and GitHub

Version control is one of the most fundamental tools in a software engineer's toolkit. It solves a problem you've probably encountered even outside of programming: how do you track changes to documents over time, collaborate with others, and recover from mistakes?

1.4.1 Why Version Control Matters

Without version control, teams resort to chaotic practices:

- Files named `project_final.doc`, `project_final_v2.doc`, `project_REALLY_final.doc`
- Emailing files back and forth
- Copying entire folders as "backups"
- Overwriting each other's changes
- No way to see what changed, when, or why

Version control systems solve these problems by:

- Tracking every change to every file
- Recording who made each change and why
- Enabling multiple people to work simultaneously
- Allowing you to revert to any previous state
- Supporting parallel lines of development (branches)
- Facilitating code review and collaboration

1.4.2 Understanding Git

Git is the dominant version control system in software development today. Created by Linus Torvalds in 2005 (yes, the same person who created Linux), Git is distributed, fast, and powerful.

Key Concepts:

Repository (Repo): A repository is a directory containing your project files plus a hidden `.git` folder that stores the complete history of all changes. Every team member has a complete copy of the repository.

Commit: A commit is a snapshot of your project at a specific point in time. Each commit has a unique identifier (SHA hash), a message describing the change, and metadata about the author and timestamp.

```
commit 7f4e8d2 (HEAD -> main)
Author: Jane Developer <jane@example.com>
Date:   Mon Jan 15 10:30:00 2025 -0500

    Add user authentication module

    - Implement login/logout functionality
    - Add password hashing with bcrypt
    - Create session management
```

Branch: A branch is an independent line of development. You might create a branch to work on a new feature without affecting the main codebase. Once the feature is complete and tested, you merge it back.

```
         feature-auth



                 main
       ↑                 ↑
   branch point        merge
```

Staging Area (Index): Before committing, you add changes to the staging area. This lets you control exactly what goes into each commit—you might have modified five files but only want to commit three.

Remote: A remote is a copy of your repository hosted on a server (like GitHub). You push your local commits to the remote and pull others' commits from it.

1.4.3 Essential Git Commands

Let's walk through the fundamental Git operations you'll use daily.

Initial Setup:

```bash
# Configure your identity (do this once)
git config --global user.name "Your Name"
git config --global user.email "your.email@example.com"
```

Creating a Repository:

```bash
# Initialize a new repository
git init

# Or clone an existing one
git clone https://github.com/username/repository.git
```

Basic Workflow:

```bash
# Check status of your working directory
git status

# Add files to staging area
git add filename.py        # Add specific file
git add .                  # Add all changes

# Commit staged changes
git commit -m "Describe what this commit does"

# View commit history
git log
git log --oneline          # Compact view
```

Working with Remotes:

```bash
# Add a remote (usually done once)
git remote add origin https://github.com/username/repo.git

# Push commits to remote
git push origin main

# Pull commits from remote
git pull origin main

# Fetch without merging
git fetch origin
```

Branching:

```
# Create a new branch
git branch feature-name

# Switch to a branch
git checkout feature-name

# Create and switch in one command
git checkout -b feature-name

# List all branches
git branch -a

# Merge a branch into current branch
git merge feature-name

# Delete a branch
git branch -d feature-name
```

1.4.4 GitHub and Remote Collaboration

GitHub is a web-based platform that hosts Git repositories and adds collaboration features. While Git handles version control, GitHub provides:

- **Remote Hosting**: Store your repositories in the cloud
- **Pull Requests**: Propose changes for review before merging
- **Issues**: Track bugs, features, and tasks
- **Projects**: Kanban-style project management boards
- **Actions**: Automated workflows (CI/CD)
- **Wikis**: Project documentation
- **Social Features**: Stars, forks, followers

The GitHub Flow:

The most common collaborative workflow on GitHub follows these steps:

1. **Create a Branch**: Start from `main` with a descriptive branch name

```
git checkout -b feature/user-authentication
```

2. **Make Changes**: Write code, commit frequently with clear messages

```
git add .
git commit -m "Add login form component"
git commit -m "Implement authentication API endpoint"
git commit -m "Add input validation"
```

3. **Push to GitHub**: Upload your branch to the remote

```
git push origin feature/user-authentication
```

4. **Open a Pull Request**: On GitHub, create a PR to merge your branch into `main`. Describe what you've done and why.
5. **Code Review**: Team members review your changes, leave comments, and request modifications if needed.
6. **Address Feedback**: Make additional commits to address review comments.
7. **Merge**: Once approved, merge the PR into `main`. Delete the feature branch.
8. **Deploy**: The merge to `main` may trigger automated deployment.

1.4.5 Writing Good Commit Messages

Commit messages are documentation for your future self and your team. Good messages make it easy to understand the project history and find specific changes.

Structure of a Good Commit Message:

```
Short summary (50 chars or less)

More detailed explanation if necessary. Wrap at 72 characters.
Explain the what and why, not the how (the code shows how).

- Bullet points are okay
- Use the imperative mood: "Add feature" not "Added feature"

Fixes #123
```

Examples of Good Commit Messages:

```
Add password strength indicator to registration form

Users were creating weak passwords. This adds a visual indicator
showing password strength in real-time, using the zxcvbn library
for strength estimation.

Closes #456
```

```
Fix memory leak in image processing module

The image processor wasn't releasing buffer memory after use,
causing memory consumption to grow unbounded during batch processing.
Added explicit cleanup in the finally block.
```

Examples of Poor Commit Messages:

```
fix bug
```

```
Updates
```

```
WIP
```

```
asdfasdf
```

1.4.6 Repository Structure and Documentation

A well-organized repository helps team members navigate the codebase and understand the project. Here's a typical structure:

```
my-project/
  .github/
     workflows/          # CI/CD workflow definitions
     ISSUE_TEMPLATE.md   # Template for bug reports
  docs/                  # Documentation
  src/                   # Source code
     components/
     services/
     utils/
  tests/                 # Test files
  .gitignore             # Files Git should ignore
  LICENSE                # Software license
  README.md              # Project overview
  CONTRIBUTING.md        # Contribution guidelines
  package.json           # Dependencies (for Node.js projects)
```

The README File:

The README is often the first thing visitors see. A good README includes:

- **Project Title and Description**: What does this project do?
- **Installation Instructions**: How do I set this up?
- **Usage Examples**: How do I use it?
- **Configuration**: What can I customize?
- **Contributing**: How can I help?
- **License**: What are the terms of use?

The .gitignore File:

This file tells Git which files and directories to ignore. You typically ignore:

- Build outputs and compiled files

- Dependencies (which can be reinstalled)
- IDE configuration files
- Environment files with secrets
- Log files

Example `.gitignore`:

```
# Dependencies
node_modules/
venv/

# Build outputs
dist/
build/
*.pyc

# Environment files
.env
.env.local

# IDE files
.vscode/
.idea/

# Logs
*.log
```

1.5 Collaborative Workflows

Software development is inherently collaborative. Even if you're the only developer on a project, you're collaborating with your future self (who will have forgotten why you wrote that code) and potentially with future maintainers.

1.5.1 Branching Strategies

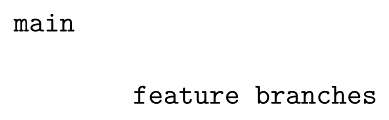
Teams adopt branching strategies to coordinate work and maintain code quality. Here are the most common approaches:

GitHub Flow:

The simplest strategy, ideal for continuous deployment:

- `main` is always deployable
- Create feature branches from `main`

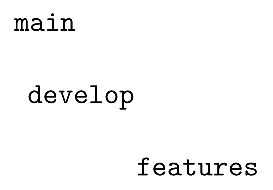
- Open pull requests for review
- Merge back to `main` after approval
- Deploy from `main`



Gitflow:

A more structured approach for projects with scheduled releases:

- `main`: Production-ready code
- `develop`: Integration branch for features
- `feature/*`: Individual features
- `release/*`: Preparation for release
- `hotfix/*`: Emergency production fixes



Trunk-Based Development:

Optimized for continuous integration:

- Everyone commits to `main` (trunk) frequently
- Feature flags hide incomplete work
- Short-lived branches (< 1 day) if any
- Requires strong CI/CD and testing

1.5.2 Code Reviews

Code review is the practice of having team members examine each other's code before it's merged. Benefits include:

- **Quality**: Catching bugs, design issues, and edge cases
- **Knowledge Sharing**: Team members learn from each other
- **Consistency**: Maintaining code style and architectural decisions
- **Mentorship**: Senior developers guide junior developers

Effective Code Reviews:

As a reviewer:

- Be constructive and kind—critique code, not people
- Explain *why* something should change, not just *what*

- Distinguish between requirements and suggestions
- Approve promptly when issues are addressed
- Look for logic errors, security issues, and maintainability

As an author:

- Keep pull requests small and focused
- Write clear descriptions explaining context
- Respond to feedback professionally
- Don't take criticism personally

1.5.3 Communication Tools

Modern software teams use various tools to collaborate:

- **Issue Trackers** (GitHub Issues, Jira): Track bugs and features
- **Documentation Platforms** (Confluence, Notion): Share knowledge
- **Chat** (Slack, Discord): Real-time communication
- **Video Conferencing** (Zoom, Meet): Face-to-face meetings
- **Design Tools** (Figma, Miro): Visual collaboration

1.6 Your Semester Project

This course is organized around a semester-long project where you'll apply everything you learn. By the end, you'll have built a complete software system from requirements through deployment.

1.6.1 Project Overview

You (or your team) will develop a software system of your choice. Examples include:

- An appointment scheduling system
- A small e-commerce platform
- An inventory management tool
- A classroom collaboration tool
- An API service for a specific domain
- A task management application
- A personal finance tracker

The specific application matters less than demonstrating mastery of software engineering practices. A simple, well-engineered system is better than an ambitious, poorly executed one.

1.6.2 Weekly Milestones

Each week, you'll complete a milestone that builds toward the final product:

Week	Milestone
1	Project proposal and repository setup
2	Software Requirements Specification
3	UML diagrams
4	Architecture and design document
5	UI/UX prototype
6	Agile sprint plan
7	Feature branch and pull request
8	Working prototype (midterm)
9	Test suite
10	CI pipeline and QA report
11	Database and API documentation
12	Deployed application
13	Security enhancements
14	Documentation package
15	Release candidate
16	Final presentation

1.6.3 This Week's Deliverables

For Week 1, you need to:

1. **Create a GitHub Repository**

   - Initialize with a README
   - Add a `.gitignore` appropriate for your technology stack
   - Set up initial folder structure

2. **Write a Project Proposal** including:

   - Problem statement: What problem does your system solve?
   - Target users: Who will use this system?
   - High-level features: What will the system do?
   - Technology choices: What languages/frameworks/tools will you use?
   - Success criteria: How will you know if the project succeeds?

1.7 Chapter Summary

Software engineering is the disciplined application of engineering principles to software development. Unlike ad-hoc programming, it encompasses the entire lifecycle of software systems, from initial conception through years of maintenance and evolution.

Key takeaways from this chapter:

- **Software engineering emerged** from the software crisis of the 1960s, when projects consistently failed due to lack of systematic approaches.
- **Modern systems depend on software** in virtually every domain. Failures can cost lives and billions of dollars; good engineering creates enormous value.
- **The SDLC provides a framework** for organizing development activities. Different models—Waterfall, Agile, Spiral, DevOps—suit different project characteristics.
- **Waterfall** works well for stable requirements and regulated environments but struggles with change.
- **Agile** embraces change and delivers working software frequently through iterative development.
- **Spiral** emphasizes risk management through prototyping and iteration.
- **DevOps** bridges development and operations, enabling continuous delivery and rapid feedback.
- **Git provides version control**, tracking every change to your codebase and enabling collaboration.
- **GitHub adds collaboration features** like pull requests, issues, and project management tools.
- **Effective collaboration** requires good branching strategies, code reviews, and communication.

1.8 Key Terms

Term	Definition
Software Engineering	Systematic application of engineering principles to software development
SDLC	Software Development Life Cycle; framework for development stages
Waterfall	Sequential SDLC model with distinct phases
Agile	Iterative approach emphasizing flexibility and customer collaboration
Scrum	Agile framework using sprints and defined roles

Term	Definition
DevOps	Cultural and technical practices bridging development and operations
CI/CD	Continuous Integration and Continuous Delivery/Deployment
Repository	A directory tracked by version control containing project files and history
Commit	A snapshot of changes in a version control system
Branch	An independent line of development
Pull Request	A proposal to merge changes, enabling code review
Merge	Combining changes from one branch into another

1.9 Review Questions

1. How does software engineering differ from programming? Give three specific examples of activities that are part of software engineering but not typically part of programming.
2. Describe the software crisis that led to the term “software engineering.” What characteristics of software projects during this period prompted the need for engineering discipline?
3. Compare and contrast the Waterfall and Agile approaches. For each, describe a project scenario where that approach would be most appropriate.
4. What are the four core values of the Agile Manifesto? In your own words, explain what each value means in practice.
5. Explain the relationship between DevOps culture and CI/CD practices. How do they reinforce each other?
6. What is the difference between `git add` and `git commit`? Why does Git have a staging area?
7. Describe the GitHub Flow workflow. What are the key steps, and why is each important?
8. What makes a good commit message? Write an example of a good commit message for adding a search feature to a web application.
9. Why is code review valuable? List at least three benefits for the team and three things to look for when reviewing someone else’s code.
10. Consider the software running an ATM machine. What SDLC model(s) might be appropriate for developing and maintaining this system? Justify your answer.

1.10 Hands-On Exercises

Exercise 1.1: Git Basics

Practice the fundamental Git commands:

```bash
# Create a new directory and initialize a repository
mkdir git-practice
cd git-practice
git init

# Create a file and make your first commit
echo "# Git Practice" > README.md
git add README.md
git commit -m "Initial commit: Add README"

# Make changes and commit again
echo "This is a practice repository." >> README.md
git add README.md
git commit -m "Add description to README"

# View your history
git log --oneline
```

Exercise 1.2: Branching Practice

Create and merge a feature branch:

```bash
# Create and switch to a new branch
git checkout -b feature/add-gitignore

# Create a .gitignore file
echo "*.log" > .gitignore
echo "node_modules/" >> .gitignore
git add .gitignore
git commit -m "Add .gitignore file"

# Switch back to main and merge
git checkout main
git merge feature/add-gitignore

# Delete the feature branch
git branch -d feature/add-gitignore
```

Exercise 1.3: Repository Setup

Set up your semester project repository:

1. Create a new repository on GitHub
2. Clone it to your local machine
3. Create an appropriate folder structure
4. Add a comprehensive README with project description
5. Create a `.gitignore` for your technology stack
6. Make your initial commit and push to GitHub

Exercise 1.4: Project Proposal

Write a one-page project proposal including:

- Project title
- Problem statement (2-3 paragraphs)
- Target users
- Key features (5-10 bullet points)
- Proposed technology stack
- Anticipated challenges
- Success criteria

1.11 Further Reading

Books:

- Brooks, F. P. (1995). *The Mythical Man-Month: Essays on Software Engineering* (Anniversary Edition). Addison-Wesley.
- Sommerville, I. (2015). *Software Engineering* (10th Edition). Pearson.
- Beck, K. et al. (2001). *Manifesto for Agile Software Development.* agilemanifesto.org

Online Resources:

- Pro Git Book (free online): https://git-scm.com/book
- GitHub Guides: https://guides.github.com
- Atlassian Git Tutorials: https://www.atlassian.com/git/tutorials
- The Twelve-Factor App: https://12factor.net

References

Beck, K., Beedle, M., van Bennekum, A., Cockburn, A., Cunningham, W., Fowler, M., … & Thomas, D. (2001). Manifesto for Agile Software Development. Retrieved from https://agilemanifesto.org/

Boehm, B. W. (1988). A spiral model of software development and enhancement. *Computer*, 21(5), 61-72.

Brooks, F. P. (1995). *The Mythical Man-Month: Essays on Software Engineering* (Anniversary Edition). Addison-Wesley.

IEEE. (1990). IEEE Standard Glossary of Software Engineering Terminology (IEEE Std 610.12-1990).

Kim, G., Humble, J., Debois, P., & Willis, J. (2016). *The DevOps Handbook*. IT Revolution Press.

Royce, W. W. (1970). Managing the development of large software systems. *Proceedings of IEEE WESCON*, 26(8), 1-9.

Schwaber, K., & Sutherland, J. (2020). *The Scrum Guide*. Scrum.org.

Chapter 2: Requirements Engineering

Learning Objectives

By the end of this chapter, you will be able to:

- Explain the importance of requirements engineering in software projects
- Distinguish between functional and non-functional requirements
- Apply various requirements elicitation techniques to gather stakeholder needs
- Write effective user stories with clear acceptance criteria
- Create a comprehensive Software Requirements Specification (SRS) document
- Develop and maintain a Requirements Traceability Matrix (RTM)
- Identify and manage common requirements engineering challenges

---

2.1 The Foundation of Software Projects

Picture this scenario: A client approaches your development team with an exciting idea. "I want an app," they say, "that helps people manage their tasks. You know, like a to-do list, but better." Your team gets to work immediately, spending three months building what you believe is an excellent task management application. You present the finished product, and the client's face falls. "This isn't what I meant at all. I needed something for teams to collaborate on projects, not a personal to-do list. And where's the integration with our existing calendar system?"

This scenario plays out in software projects far more often than anyone would like to admit. Studies consistently show that a significant percentage of software project failures can be traced back to poor requirements—requirements that were incomplete, ambiguous, misunderstood, or simply wrong.

Requirements engineering is the systematic process of discovering, documenting, validating, and managing the requirements for a software system. It answers the fundamental question: *What should this system do?*

2.1.1 Why Requirements Matter

Requirements engineering might seem like overhead—time spent not writing code. But consider the economics of software defects. The cost of fixing a bug increases dramatically depending on when it's discovered:

Phase Discovered	Relative Cost to Fix
Requirements	1x
Design	5x
Implementation	10x
Testing	20x
After Release	50-200x

A requirement error caught during the requirements phase might take an hour to fix—a conversation to clarify what the customer actually needs. That same error, if it survives into production code, might require redesigning components, rewriting thousands of lines of code, updating tests, redeploying, and dealing with unhappy users.

The Standish Group's research on software projects has consistently found that the top factors in project success include:

- Clear statement of requirements
- User involvement throughout the project
- Realistic expectations
- Clear vision and objectives

Notice that three of these four factors relate directly to requirements engineering.

2.1.2 The Requirements Engineering Process

Requirements engineering is not a one-time activity but an ongoing process throughout the project lifecycle. It typically involves four main activities:

1. Elicitation: Discovering requirements from stakeholders, documents, existing systems, and domain knowledge. This is often the most challenging phase because stakeholders may not know what they want, may disagree with each other, or may have difficulty articulating their needs.

2. Analysis: Examining requirements for conflicts, ambiguities, and incompleteness. This phase involves prioritization, negotiation between stakeholders, and feasibility assessment.

3. Specification: Documenting requirements in a clear, precise, and verifiable form. The output is typically a Software Requirements Specification (SRS) document.

4. Validation: Ensuring that the documented requirements actually reflect stakeholder needs and that they are achievable within project constraints.

```
 Elicitation      Analysis      Specification    Validation




                     (Iterative Process)
```

These activities are iterative and often overlap. As you document requirements (specification), you'll discover gaps that require more elicitation. Validation might reveal conflicts that require additional analysis. Requirements engineering continues throughout the project as understanding deepens and circumstances change.

2.2 Types of Requirements

Requirements come in different forms, each serving a different purpose. Understanding these categories helps ensure comprehensive coverage of what a system must do and how well it must do it.

2.2.1 Functional Requirements

Functional requirements describe what the system should *do*—the specific behaviors, features, and functions it must provide. They define the system's capabilities and how it should respond to particular inputs or situations.

Functional requirements typically follow this pattern: *The system shall [perform some action] when [some condition occurs].*

Examples of Functional Requirements:

For an e-commerce system:

- The system shall allow users to search for products by name, category, or price range
- The system shall calculate shipping costs based on destination and package weight
- The system shall send an email confirmation when an order is placed
- The system shall allow users to save items to a wishlist
- The system shall process payments through credit cards, debit cards, and PayPal

For a library management system:

- The system shall allow librarians to add new books to the catalog
- The system shall track which member has borrowed each book
- The system shall calculate and display overdue fines
- The system shall send reminder notifications three days before a book is due
- The system shall allow members to reserve books that are currently checked out

Characteristics of Good Functional Requirements:

- **Specific**: Precisely defines behavior without ambiguity
- **Measurable**: Can be objectively verified through testing
- **Achievable**: Technically feasible within project constraints
- **Relevant**: Directly supports user or business needs
- **Traceable**: Can be linked to business objectives and test cases

2.2.2 Non-Functional Requirements

Non-functional requirements (NFRs) describe *how well* the system performs its functions—the quality attributes, constraints, and characteristics that define the system's operational qualities. They're sometimes called "quality requirements" or "-ilities" (because many end in "-ility": reliability, scalability, usability, etc.).

Non-functional requirements often have more impact on system architecture than functional requirements. You can add a search feature to an existing architecture, but retrofitting a system to handle millions of concurrent users requires fundamental architectural decisions.

Categories of Non-Functional Requirements:

Performance Requirements specify response times, throughput, and capacity:

- The system shall respond to search queries within 2 seconds
- The system shall support 10,000 concurrent users
- The system shall process at least 100 transactions per second
- Page load time shall not exceed 3 seconds on a 4G mobile connection

Reliability Requirements specify uptime, availability, and fault tolerance:

- The system shall maintain 99.9% uptime (less than 8.76 hours downtime per year)
- The system shall recover from failures within 5 minutes
- No data loss shall occur during system crashes
- The system shall maintain full functionality when one database server fails

Security Requirements specify protection against threats:

- All passwords shall be stored using bcrypt with a minimum cost factor of 12
- The system shall lock accounts after 5 failed login attempts
- All data transmission shall use TLS 1.3 or higher
- User sessions shall expire after 30 minutes of inactivity
- The system shall log all access to sensitive data

Usability Requirements specify ease of use and user experience:

- New users shall be able to complete a purchase within 5 minutes without training
- The system shall be accessible according to WCAG 2.1 Level AA guidelines
- Error messages shall clearly explain what went wrong and how to fix it
- The system shall work on screens from 320px to 4K resolution

Scalability Requirements specify growth capacity:

- The system architecture shall support horizontal scaling to 10x current load
- Database design shall accommodate 100 million records without performance degradation
- The system shall support adding new geographic regions within 2 weeks

Maintainability Requirements specify ease of modification:

- Code shall achieve a minimum of 80% test coverage
- All public APIs shall include documentation

- The system shall support zero-downtime deployments
- Configuration changes shall not require code redeployment

Compliance Requirements specify regulatory and legal constraints:

- The system shall comply with GDPR data protection requirements
- Payment processing shall comply with PCI DSS Level 1
- Medical records handling shall comply with HIPAA regulations
- The system shall maintain audit logs for 7 years

2.2.3 The Relationship Between Functional and Non-Functional Requirements

Functional and non-functional requirements are deeply intertwined. Consider a simple requirement: "The system shall allow users to search for products."

This functional requirement raises many non-functional questions:

- How fast should search results appear? (Performance)
- How many products should the search handle? (Scalability)
- What happens if the search service fails? (Reliability)
- How intuitive should the search interface be? (Usability)
- Should search queries be logged? For how long? (Compliance)

A complete specification addresses both what the system does and how well it does it.

2.2.4 Constraints and Assumptions

Beyond functional and non-functional requirements, specifications often include:

Constraints are restrictions on how the system can be built:

- The system must be developed using Java 17
- The database must be PostgreSQL (existing enterprise license)
- Development must be completed within 6 months
- The budget cannot exceed $500,000
- The system must integrate with the existing SAP installation

Assumptions are conditions believed to be true but not verified:

- Users will have modern web browsers (released within the last 2 years)
- Network connectivity between offices is reliable
- The client will provide access to subject matter experts during development
- Current server infrastructure has capacity for the new system

Documenting constraints and assumptions is crucial because they can significantly impact design decisions, and invalid assumptions are a common source of project problems.

2.3 Requirements Elicitation Techniques

Elicitation—discovering what stakeholders actually need—is often the most challenging aspect of requirements engineering. Stakeholders may not know what they want, may have conflicting needs, or may have difficulty expressing their requirements in terms developers can use.

Effective elicitation requires multiple techniques, as different approaches work better for different types of requirements and different stakeholders.

2.3.1 Stakeholder Interviews

Interviews are one-on-one or small group conversations with stakeholders to understand their needs, expectations, and concerns. They're particularly useful for understanding the context, goals, and priorities behind requirements.

Types of Interviews:

Structured interviews follow a predetermined set of questions asked in a specific order. They ensure consistency across multiple interviews and are useful when you need to compare responses from different stakeholders.

Unstructured interviews are open-ended conversations that follow wherever the discussion leads. They're useful early in the project when you're still discovering the problem domain.

Semi-structured interviews combine elements of both: a prepared set of questions with flexibility to explore interesting tangents.

Interview Best Practices:

Before the interview:

- Research the stakeholder's role and background
- Prepare questions but be ready to deviate
- Schedule appropriate time (typically 45-60 minutes)
- Clarify the interview's purpose with the stakeholder

During the interview:

- Start with open-ended questions ("Tell me about your current workflow…")
- Listen more than you talk (aim for 80/20)
- Ask follow-up questions to dig deeper
- Avoid leading questions that suggest answers
- Take notes, but maintain eye contact
- Use active listening techniques (paraphrasing, summarizing)

After the interview:

- Write up notes immediately while details are fresh
- Identify follow-up questions for future sessions
- Share notes with the stakeholder for validation
- Look for patterns across multiple interviews

Sample Interview Questions:

- What are your main responsibilities related to this system?
- Walk me through a typical day using the current system/process.
- What are the biggest challenges you face?
- If you could change one thing about the current system, what would it be?
- What would make your job easier?
- What absolutely must the new system do?
- What would be nice to have but isn't essential?
- What concerns do you have about the new system?
- Who else should I talk to about this?

2.3.2 Questionnaires and Surveys

Questionnaires allow you to gather information from many stakeholders efficiently. They're useful when you need quantitative data or when stakeholders are geographically distributed.

When to Use Questionnaires:

- Large number of stakeholders
- Need for statistical analysis
- Follow-up to validate interview findings
- Distributed or remote stakeholders
- Standardized information needed across groups

Questionnaire Design Tips:

- Keep it short (15-20 minutes maximum)
- Use clear, unambiguous language
- Mix question types (multiple choice, rating scales, open-ended)
- Order questions logically
- Pilot test with a small group first
- Provide context for why you're asking

Example Questions:

Rating scale: How satisfied are you with the current system's performance? [ ] Very Dissatisfied [ ] Dissatisfied [ ] Neutral [ ] Satisfied [ ] Very Satisfied

Multiple choice: How often do you use the reporting feature? [ ] Daily [ ] Weekly [ ] Monthly [ ] Rarely [ ] Never

Open-ended: What features would you most like to see in the new system?

2.3.3 Observation and Ethnography

Sometimes the best way to understand requirements is to watch users in their natural environment. **Observation** involves watching stakeholders perform their actual work to understand workflows, pain points, and unspoken needs.

Benefits of Observation:

- Reveals tacit knowledge users can't articulate
- Uncovers workarounds and unofficial processes
- Shows actual behavior vs. reported behavior
- Provides context for requirements
- Identifies environmental factors

Observation Techniques:

Passive observation: Watch without interfering, taking notes on what you see. Users may behave differently when watched (the Hawthorne effect), but this diminishes over time.

Active observation (contextual inquiry): Ask questions while observing. "I noticed you copied that data into a spreadsheet—can you tell me why?"

Apprenticing: Have the user teach you their job. This builds rapport and surfaces knowledge that might not emerge otherwise.

A Day in the Life: Shadow a user through an entire workday to understand the full context of their activities.

What to Look For:

- Steps in workflows that seem cumbersome
- Workarounds users have developed
- Frequent interruptions or context switches
- Information users need but don't have easy access to
- Paper notes, sticky notes, or personal tracking systems
- Frustration points
- Collaboration patterns

2.3.4 Workshops and Focus Groups

Workshops bring multiple stakeholders together to collaboratively explore requirements. They're particularly useful for building consensus, identifying conflicts, and generating ideas.

Types of Workshops:

Requirements workshops gather stakeholders to jointly define requirements. A facilitator guides the group through structured activities.

Joint Application Development (JAD) is a specific workshop methodology that brings together users, managers, and developers for intensive collaborative sessions.

Focus groups explore attitudes, opinions, and preferences with a group of representative users.

Workshop Best Practices:

- Limit group size (6-12 participants)
- Include diverse stakeholder perspectives
- Use a skilled facilitator (often external)
- Set clear objectives and agenda
- Use visual aids and collaborative tools
- Document outcomes in real-time
- Manage dominant personalities
- Allow for individual input before group discussion

Workshop Activities:

Brainstorming: Generate ideas without criticism, then consolidate and prioritize.

Affinity diagrams: Write ideas on sticky notes, then group related items to identify themes.

Dot voting: Give participants dots to vote on priorities; reveals group preferences quickly.

Use case walkthrough: Walk through scenarios step by step, identifying required functionality.

Card sorting: Have participants organize features or concepts into categories to understand mental models.

2.3.5 Document Analysis

Document analysis involves reviewing existing documentation to understand the current system, business rules, and context. It's particularly useful when working with established organizations or regulated industries.

Documents to Review:

- Current system documentation and user manuals
- Business process documentation
- Organizational charts
- Policy and procedure manuals
- Regulatory and compliance documents
- Previous project documentation
- Training materials
- Reports and forms currently in use
- Industry standards and benchmarks

What to Extract:

- Business rules and logic
- Data definitions and relationships
- Workflow steps
- Roles and responsibilities
- Compliance requirements
- Terminology and vocabulary

2.3.6 Prototyping

Prototyping involves building preliminary versions of the system to explore requirements. Users often find it easier to react to something concrete than to describe abstract needs.

Types of Prototypes:

Paper prototypes: Hand-drawn sketches of screens and interfaces. Quick to create, easy to modify, and effective for early exploration.

Wireframes: Low-fidelity digital mockups showing layout and navigation without visual design.

Clickable prototypes: Interactive mockups that simulate user flows without real functionality.

Proof of concept: Technical prototypes that test feasibility of specific features.

Evolutionary prototypes: Prototypes that evolve into the final system (requires disciplined development).

Throwaway prototypes: Built solely for learning, then discarded. Allows for quick, dirty experimentation.

When to Use Prototyping:

- Requirements are unclear or hard to articulate
- User interface is critical
- Stakeholders need to "see it to believe it"
- Technical feasibility is uncertain
- Novel or innovative features

Prototyping Risks:

- Users may expect the prototype to be the final product
- Pressure to ship the prototype as-is
- Time invested in throwaway prototypes
- Can focus too heavily on UI at expense of other requirements

2.3.7 Analyzing Existing Systems

If replacing or enhancing an existing system, that system is a valuable source of requirements. Understanding current functionality provides a baseline for the new system.

Analysis Approaches:

- Use the existing system yourself
- Review system documentation
- Study the database schema
- Examine reports and outputs
- Interview users about what works and what doesn't
- Analyze support tickets and bug reports
- Review change request history

Important Considerations:

Not everything in the current system needs to be in the new system. Some features may be unused, obsolete, or present only due to historical accidents. Ask users which features they actually use and value.

2.4 User Stories and Acceptance Criteria

User stories are a popular format for expressing requirements in Agile development. They capture requirements from the user's perspective, focusing on value delivered rather than technical implementation.

2.4.1 The User Story Format

The classic user story format is:

> **As a** [type of user], **I want** [some capability] **so that** [some benefit].

This format emphasizes three key elements:

- **Who** wants the capability (the persona or role)
- **What** they want to accomplish
- **Why** it matters to them (the value or benefit)

Examples:

> As a **customer**, I want to **save my shopping cart** so that I can **continue shopping later from a different device**.

> As a **librarian**, I want to **see overdue books for a specific member** so that I can **contact them about returns**.

> As a **sales manager**, I want to **view my team's performance dashboard** so that I can **identify who needs coaching**.

> As a **visually impaired user**, I want to **navigate the site using only my keyboard** so that I can **use the application without a mouse**.

2.4.2 Writing Effective User Stories

The INVEST Criteria:

Good user stories follow the INVEST principles:

I - Independent: Stories should be self-contained, without inherent dependencies on other stories. This allows them to be prioritized and scheduled flexibly.

N - Negotiable: Stories are not contracts. They're placeholders for conversations about requirements. Details emerge through discussion.

V - Valuable: Each story should deliver value to users or the business. Technical tasks that don't directly deliver value (like "refactor the database") aren't user stories.

E - Estimable: The team should be able to estimate the effort required. If a story is too vague to estimate, it needs clarification or splitting.

S - Small: Stories should be completable within a single sprint. Large stories (epics) should be broken down into smaller stories.

T - Testable: It must be possible to write tests that verify the story is complete. If you can't test it, you can't confirm it's done.

Common Mistakes:

Too vague:

> As a user, I want the system to be fast.

Better:

> As a customer, I want search results to appear within 2 seconds so that I can quickly find products.

Too technical:

> As a developer, I want to implement caching using Redis.

Better:

> As a customer, I want previously viewed products to load instantly so that I can quickly review items I've already seen.

Missing the "why":

> As an admin, I want to export data to CSV.

Better:

> As an admin, I want to export user data to CSV so that I can analyze trends in spreadsheet software I'm familiar with.

2.4.3 Acceptance Criteria

Acceptance criteria define the conditions that must be met for a user story to be considered complete. They provide clarity about scope and serve as the basis for testing.

Format Options:

Scenario format (Given-When-Then):

```
Given [precondition/context]
When [action occurs]
Then [expected outcome]
```

Example:

```
Story: As a customer, I want to reset my password so that I can
       regain access to my account if I forget it.

Acceptance Criteria:

Scenario 1: Requesting password reset
Given I am on the login page
When I click "Forgot Password" and enter my email address
Then I should receive a password reset email within 5 minutes

Scenario 2: Valid reset link
Given I have received a password reset email
When I click the reset link within 24 hours
Then I should see a form to enter a new password

Scenario 3: Expired reset link
Given I have received a password reset email
When I click the reset link after 24 hours
Then I should see a message that the link has expired
And I should see an option to request a new reset link

Scenario 4: Password requirements
Given I am on the password reset form
When I enter a new password
Then the password must be at least 8 characters
And contain at least one uppercase letter
And contain at least one number
And contain at least one special character
```

Checklist format:

```
Story: As a customer, I want to filter search results so that I can
       find products that match my specific needs.

Acceptance Criteria:
 Users can filter by price range (min and max)
 Users can filter by category
 Users can filter by customer rating (1-5 stars)
 Users can apply multiple filters simultaneously
 Filters update results without page reload
 Active filters are clearly displayed
 Users can remove individual filters or clear all
 Filter state is preserved when navigating back to results
```

2.4.4 Epics, Stories, and Tasks

User stories exist within a hierarchy:

```
                            EPIC
  Large body of work that can be broken into smaller pieces
  Example: "User Account Management"


      USER STORY 1        USER STORY 2        USER STORY 3
     User registration    Password reset     Profile editing


      Task  Task        Task  Task        Task  Task
```

Epics are large bodies of work that span multiple sprints. They represent major features or capabilities but are too big to complete in one iteration.

User Stories are the primary unit of work in Agile. Each story delivers a specific piece of value and can be completed within a sprint.

Tasks are the technical activities required to complete a story. Unlike stories, tasks describe implementation details.

Example Breakdown:

```
EPIC: Shopping Cart

User Story 1: Add items to cart
  Task: Create cart database schema
  Task: Implement add-to-cart API endpoint
  Task: Build cart UI component
  Task: Write unit tests for cart service
  Task: Write integration tests for cart API

User Story 2: Update cart quantities
  Task: Implement quantity update API
  Task: Add quantity controls to cart UI
  Task: Handle inventory validation
  Task: Write tests

User Story 3: Remove items from cart
  ...

User Story 4: Apply discount codes
  ...
```

2.4.5 Story Mapping

User story mapping is a technique for organizing user stories to understand the full picture of user experience. Created by Jeff Patton, it arranges stories in a two-dimensional map.

```
User Activities (left to right = user journey)


     Browse      Search      View         Add to      Checkout
    Products    Products    Product        Cart


    View       Search     View       Add        Enter
    catalog    by name    details    item      shipping      MVP
                                         Release 1
P
r
i    Filter     Search     View       Update     Choose
o    by         by         reviews   quantity    payment      Release 2
r    category   category
i
t
y
    View       Save       Zoom       Save       Apply
    featured   search     images     for        coupon        Release 3
    items                            later
```

The horizontal axis shows the user's journey through the system—the activities they perform from left to right. The vertical axis shows priority, with the most essential stories at the top.

Story mapping helps teams:

- See the big picture of user experience
- Identify gaps in functionality
- Plan releases by drawing horizontal lines
- Understand dependencies between stories
- Communicate the product vision

2.5 The Software Requirements Specification (SRS)

The **Software Requirements Specification** (SRS) is the primary document produced by requirements engineering. It serves as a contract between stakeholders about what the system will do and as a reference for designers, developers, and testers.

2.5.1 Purpose of the SRS

The SRS serves multiple audiences and purposes:

For customers and stakeholders:

- Confirms understanding of their needs
- Serves as basis for acceptance testing
- Documents agreed-upon scope

For project managers:

- Basis for estimating effort and cost
- Defines project scope
- Reference for change management

For designers and developers:

- Input for system design
- Reference during implementation
- Clarifies expected behavior

For testers:

- Basis for test planning
- Defines what to test
- Specifies expected results

2.5.2 SRS Structure (IEEE 830)

While formats vary, the IEEE 830 standard provides a widely-used template. Here's a typical structure:

```
1. Introduction
   1.1 Purpose
   1.2 Scope
   1.3 Definitions, Acronyms, and Abbreviations
   1.4 References
   1.5 Overview

2. Overall Description
   2.1 Product Perspective
   2.2 Product Functions
   2.3 User Classes and Characteristics
   2.4 Operating Environment
   2.5 Design and Implementation Constraints
   2.6 Assumptions and Dependencies

3. Specific Requirements
   3.1 Functional Requirements
   3.2 External Interface Requirements
       3.2.1 User Interfaces
       3.2.2 Hardware Interfaces
       3.2.3 Software Interfaces
       3.2.4 Communication Interfaces
   3.3 Non-Functional Requirements
       3.3.1 Performance Requirements
       3.3.2 Security Requirements
       3.3.3 Reliability Requirements
       3.3.4 Availability Requirements
   3.4 System Features

4. Appendices
   4.1 Glossary
   4.2 Analysis Models
   4.3 To Be Determined List
```

2.5.3 Writing an SRS: Section by Section

Let's walk through each section with guidance and examples.

1. Introduction

1.1 Purpose

Describe the purpose of this SRS document and its intended audience.

> This document specifies the software requirements for TaskFlow, a team task management application. It is intended for the development team, project stakeholders, and quality assurance personnel.

1.2 Scope

Describe the software being specified, its purpose, benefits, and objectives.

> TaskFlow is a web-based application that enables teams to create, assign, track, and collaborate on tasks and projects. The system will improve team productivity by centralizing task management, providing visibility into project progress, and facilitating collaboration through comments and notifications.
>
> The system will NOT include: time tracking functionality, billing/invoicing, or integration with version control systems. These features are planned for future releases.

1.3 Definitions, Acronyms, and Abbreviations

Define terms used throughout the document.

Term	Definition
Task	A single unit of work with a title, description, assignee, and due date
Project	A collection of related tasks
Sprint	A fixed time period (typically 2 weeks) for completing tasks
Board	A visual representation of tasks organized by status

1.4 References

List any documents referenced in the SRS.

1.5 Overview

Describe how the rest of the SRS is organized.

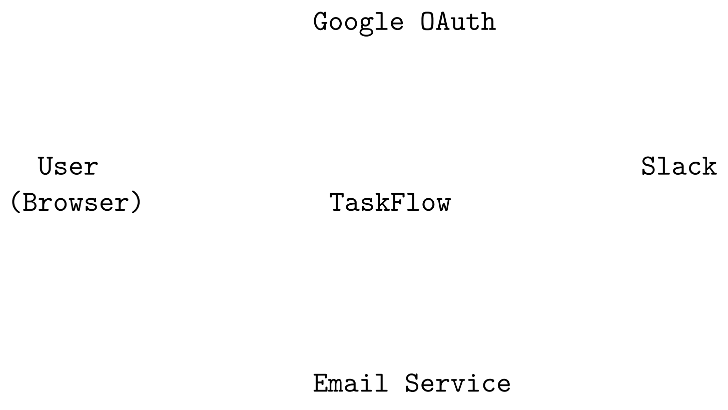
2. Overall Description

2.1 Product Perspective

Describe how the system fits into the broader environment. Is it standalone? Does it replace an existing system? What external systems does it interact with?

> TaskFlow is a new, standalone system that will replace the team's current use of spreadsheets and email for task tracking. The system will integrate with:
>
> - Google Workspace for user authentication
> - Slack for notifications
> - Email services for user communications

Include a context diagram showing the system and its external interfaces:



2.2 Product Functions

Provide a summary of major functions (detailed in Section 3).

Major functions include:

- User management: Registration, authentication, profile management
- Project management: Create, configure, and archive projects
- Task management: Create, assign, update, and complete tasks
- Collaboration: Comments, mentions, and activity feeds
- Notifications: Email and Slack notifications for relevant events
- Reporting: Project progress, team velocity, overdue tasks

2.3 User Classes and Characteristics

Describe the different types of users and their characteristics.

User Class	Description	Technical Expertise
Team Member	Creates and completes tasks	Basic
Project Manager	Creates projects, assigns tasks, monitors progress	Basic
Team Admin	Manages team membership and permissions	Intermediate
System Admin	Configures system settings, manages integrations	Advanced

2.4 Operating Environment

Describe the environment in which the software will operate.

- Server: Linux (Ubuntu 22.04 LTS), Docker containers
- Database: PostgreSQL 15
- Web Server: Nginx
- Client browsers: Chrome, Firefox, Safari, Edge (latest 2 versions)
- Mobile: Responsive design supporting iOS and Android devices

2.5 Design and Implementation Constraints

List any constraints that limit developer options.

- The system must be developed using React for the frontend and Node.js for the backend
- All data must be stored in the United States to comply with data residency requirements
- The system must use the existing corporate design system for UI components
- Development must be complete by [date] to coincide with team restructuring

2.6 Assumptions and Dependencies

Document assumptions that, if wrong, could affect requirements.

Assumptions:

- Users have reliable internet connectivity
- Users have accounts in Google Workspace for authentication
- Team sizes will not exceed 500 members

Dependencies:

- Google OAuth service availability
- Slack API stability
- Corporate design system components

3. Specific Requirements

This is the core of the SRS, containing detailed, testable requirements.

3.1 Functional Requirements

Organize by feature area or use case. Each requirement should have a unique identifier.

```
3.1.1 User Management

FR-UM-001: User Registration
The system shall allow new users to register using their Google Workspace account.

FR-UM-002: User Profile
The system shall allow users to view and edit their profile information, including:
- Display name
- Profile photo
- Notification preferences
```

```
FR-UM-003: Role Assignment
The system shall allow Team Admins to assign roles (Team Member, Project Manager,
Team Admin) to users.

3.1.2 Project Management

FR-PM-001: Project Creation
The system shall allow Project Managers to create new projects with the following
attributes:
- Project name (required, max 100 characters)
- Description (optional, max 500 characters)
- Start date (optional)
- Target completion date (optional)
- Team members (at least one required)

FR-PM-002: Project Status
The system shall allow projects to have one of the following statuses:
- Active (default)
- On Hold
- Completed
- Archived

FR-PM-003: Project Templates
The system shall allow Project Managers to create projects from templates that
pre-populate tasks and settings.
```

3.2 External Interface Requirements

3.2.1 User Interfaces

UI-001: The system shall provide a web-based interface accessible via modern browsers.

UI-002: The interface shall be responsive, supporting screen widths from 320px to 2560px.

UI-003: The system shall conform to WCAG 2.1 Level AA accessibility guidelines.

UI-004: The primary navigation shall include access to: Dashboard, Projects, My Tasks, Team, and Settings.

3.2.2 Software Interfaces

SI-001: The system shall authenticate users via Google OAuth 2.0.

SI-002: The system shall send notifications to Slack using the Slack Web API.

SI-003: The system shall expose a REST API for potential future integrations.

3.3 Non-Functional Requirements

3.3.1 Performance Requirements

NFR-PERF-001: Page load time shall not exceed 3 seconds on a 4G connection.

NFR-PERF-002: API responses shall return within 500ms for 95% of requests.

NFR-PERF-003: The system shall support 100 concurrent users without degradation.

3.3.2 Security Requirements

NFR-SEC-001: All data transmission shall use TLS 1.3.

NFR-SEC-002: User sessions shall expire after 8 hours of inactivity.

NFR-SEC-003: The system shall log all authentication events.

NFR-SEC-004: Passwords shall never be stored; only Google OAuth shall be used.

3.3.3 Reliability Requirements

NFR-REL-001: The system shall maintain 99.5% uptime, excluding scheduled maintenance.

NFR-REL-002: In the event of server failure, the system shall recover within 10 minutes.

NFR-REL-003: No user data shall be lost due to system failures.

2.5.4 Characteristics of Good Requirements

Individual requirements should be:

Clear: Unambiguous, meaning the same thing to all readers. Avoid vague terms like “user-friendly,” “fast,” or “intuitive” without specific definitions.

Complete: Contains all necessary information. A reader should be able to understand and implement the requirement without asking for clarification.

Consistent: Doesn’t contradict other requirements in the document.

Verifiable: Can be tested or measured. If you can’t write a test for a requirement, it’s not verifiable.

Traceable: Has a unique identifier and can be linked to its source and to downstream artifacts (design, code, tests).

Feasible: Technically achievable within project constraints.

Necessary: Supports a documented need. Requirements without clear justification should be questioned.

Prioritized: Stakeholders understand relative importance.

Bad Examples and Improvements:

Poor Requirement	Problem	Improved Requirement
The system shall be fast	Vague, not measurable	The system shall respond to user actions within 2 seconds

Poor Requirement	Problem	Improved Requirement
The system shall handle many users	“Many” is undefined	The system shall support 1,000 concurrent users
The system shall be easy to use	Subjective	New users shall complete the registration process in under 3 minutes without assistance
The system should have a login feature	Ambiguous (“should” vs “shall”)	The system shall require users to authenticate before accessing any features
The interface shall be attractive	Subjective, not testable	The interface shall conform to the corporate style guide (reference: design-system.company.com)

2.6 Requirements Traceability

Requirements traceability is the ability to follow a requirement from its origin through design, implementation, and testing. It ensures that every requirement is addressed and that all development work serves a documented need.

2.6.1 Why Traceability Matters

Traceability helps answer critical questions:

- **Completeness**: Is every requirement implemented and tested?
- **Impact Analysis**: If a requirement changes, what’s affected?
- **Coverage**: Are there any gaps in testing?
- **Justification**: Why does this code exist? What requirement does it satisfy?
- **Compliance**: Can we prove that regulatory requirements are met?

Without traceability, teams face significant risks:

- Requirements silently dropped during development
- Features implemented that nobody asked for
- Changes made without understanding full impact
- Testing gaps leading to defects
- Compliance audit failures

2.6.2 The Requirements Traceability Matrix (RTM)

A **Requirements Traceability Matrix** (RTM) is a document that maps requirements to other project artifacts. It creates explicit links between requirements and their downstream implementations.

Basic RTM Structure:

Req ID	Requirement Description	Design Reference	Code Module	Test Case ID	Status
FR-UM-001	User registration via Google OAuth	DES-AUTH-001	auth/google.js	TC-AUTH-001, TC-AUTH-002	Complete
FR-UM-002	User profile management	DES-USER-001	users/profile.js	TC-USER-001	In Progress
FR-PM-001	Project creation	DES-PROJ-001	projects/create.js	TC-PROJ-001	Complete
NFR-PERF-001	Page load < 3 seconds	DES-PERF-001	N/A	TC-PERF-001	Testing

Extended RTM with Additional Fields:

Req ID	Source	Priority	Risk	Stakeholder	Sprint	Notes
FR-UM-001	Interview-012	High	Low	Product Owner	Sprint 1	Core feature
FR-UM-002	Workshop-003	Medium	Low	Users	Sprint 2	May defer some fields
FR-PM-001	SRS v1.0	High	Medium	PM Team	Sprint 1	Complex validation
NFR-PERF-001	NFR Workshop	High	High	All Users	Sprint 3	Requires perf testing

2.6.3 Types of Traceability

Forward Traceability: From requirements to implementation

- Requirement → Design
- Requirement → Code
- Requirement → Test Cases

Forward traceability ensures every requirement is implemented and tested.

Backward Traceability: From implementation back to requirements

- Code → Requirement
- Test Case → Requirement

- Design → Requirement

Backward traceability ensures all development work serves a documented need—no “gold plating” or undocumented features.

Bi-directional Traceability: Both directions combined, providing complete coverage.

```
                Forward Traceability




Business       Software         Design         Test
  Needs         Req                     Cases




                Backward Traceability
```

2.6.4 Maintaining the RTM

The RTM is a living document that must be updated throughout the project. Best practices include:

Update triggers:

- New requirement added
- Requirement modified or deleted
- Design decision made
- Code module completed
- Test case created or executed
- Status changes

Review cadence:

- Weekly reviews during development
- Milestone reviews before releases
- Full audit before final delivery

Tooling options:

- Spreadsheets (simple projects)
- Requirements management tools (Jama, DOORS, Helix RM)
- Issue trackers with linking (Jira, GitHub Issues)
- Custom databases

2.6.5 Traceability in Agile Projects

In Agile environments, formal RTMs may seem heavyweight. However, traceability remains important. Agile approaches include:

Linking in issue trackers: User stories linked to epics (backward to business need), linked to tasks (forward to implementation), linked to test cases.

Definition of Done: Including "acceptance criteria verified" and "tests written" in the definition of done ensures traceability.

Living documentation: Tools like Cucumber connect executable specifications directly to tests, creating automatic traceability.

```
Epic: E-001 User Authentication
  Story: US-001 User Login
     Task: T-001 Implement login API
     Task: T-002 Build login form component
     Test: TC-001 Verify successful login
     Test: TC-002 Verify invalid credentials
  Story: US-002 Password Reset
     ...
```

2.7 Managing Requirements Challenges

Requirements engineering faces several common challenges. Recognizing and addressing these challenges is key to project success.

2.7.1 Scope Creep

Scope creep is the uncontrolled expansion of project scope—new requirements added without corresponding increases in time or budget. It's one of the most common causes of project overruns.

Causes:

- Unclear or incomplete initial requirements
- Stakeholders adding "just one more feature"
- Gold plating by developers
- Poor change management
- Lack of clear project boundaries

Prevention and Management:

Clear scope statements: Document what's in scope AND what's out of scope explicitly.

Change control process: All changes go through a formal review:

1. Document the change request
2. Assess impact on schedule, budget, and other requirements
3. Decide: approve, reject, or defer
4. Update documentation if approved

Baseline requirements: Freeze requirements at a specific point; changes after baseline require formal approval.

MoSCoW prioritization: Categorize requirements as:

- **Must have**: Essential, non-negotiable
- **Should have**: Important but not critical
- **Could have**: Nice to have if time permits
- **Won't have**: Explicitly out of scope (this time)

2.7.2 Ambiguous Requirements

Ambiguous requirements mean different things to different readers, leading to incorrect implementations and costly rework.

Common Sources of Ambiguity:

Vague adjectives: "fast," "user-friendly," "secure," "reliable"

Unbounded lists: "including but not limited to," "such as," "etc."

Ambiguous pronouns: "The system sends a notification to the user when they submit the form. It should be formatted as HTML." (What does "it" refer to?)

Missing conditions: "The system displays an error message." (When? Under what conditions?)

Unclear quantities: "The system supports multiple users." (How many? 10? 10,000?)

Strategies for Clarity:

Specific numbers: Replace "fast" with "within 2 seconds"

Complete lists: If the list is exhaustive, say so: "The system shall support exactly these payment methods: credit card, debit card, and PayPal"

Examples: Include concrete examples to illustrate requirements

Glossary: Define terms precisely in a glossary

Reviews: Multiple reviewers from different backgrounds catch different ambiguities

2.7.3 Conflicting Requirements

Different stakeholders often have different—sometimes contradictory—needs.

Examples:

- Marketing wants maximum features; development wants a sustainable pace
- Security wants strong authentication; UX wants minimal friction
- Sales wants customization for each client; architecture wants standardization

Resolution Strategies:

Identify conflicts early: Requirements analysis should explicitly look for conflicts.

Understand underlying needs: Often conflicts arise from different solutions to the same underlying need. Find the root cause.

Negotiate and prioritize: Bring stakeholders together to discuss trade-offs and agree on priorities.

Document decisions: Record what was decided and why, so the decision isn't relitigated later.

Escalate when necessary: Some conflicts require executive decision-making.

2.7.4 Changing Requirements

Requirements will change. Users learn what they actually need by seeing early versions. Market conditions shift. Technology evolves. Regulations change.

The question isn't whether requirements will change, but how you'll manage change.

Agile Approach: Embrace change. Short iterations deliver working software frequently. Requirements emerge and evolve based on feedback. The backlog is continuously refined.

Plan-Driven Approach: Manage change formally. Establish baselines. Evaluate change requests for impact. Maintain version control of requirements documents.

Hybrid Approach: Most real projects use a combination. Core requirements are stable (plan-driven), while details emerge iteratively (Agile).

Best Practices:

- Accept that change is inevitable
- Build processes to handle change efficiently
- Communicate the cost of late changes (not to prevent change, but to inform decisions)
- Keep requirements documentation up to date
- Maintain traceability so impacts are visible

2.8 Requirements in Practice: Tools and Techniques

2.8.1 Requirements Management Tools

Various tools support requirements engineering:

Document-based tools:

- Microsoft Word/Google Docs with templates
- Confluence
- Notion

Dedicated requirements tools:

- Jama Connect
- IBM DOORS
- Helix RM
- Modern Requirements

Agile tools with requirements support:

- Jira
- Azure DevOps
- GitHub Issues + Projects
- Linear
- Shortcut

Choosing a tool:

- Team size and distribution
- Project complexity
- Regulatory requirements
- Budget
- Integration with other tools
- Learning curve

For your course project, GitHub Issues and Projects provide adequate requirements management while learning fundamental concepts.

2.8.2 Using GitHub for Requirements

GitHub provides several features useful for requirements management:

Issues for user stories and requirements:

```
Title: As a customer, I want to reset my password

Description:
**User Story:**
As a customer, I want to reset my password so that I can
regain access if I forget it.

**Acceptance Criteria:**
- [ ] Reset link sent via email within 5 minutes
- [ ] Link expires after 24 hours
- [ ] New password must meet security requirements
- [ ] Confirmation shown after successful reset

**Priority:** High
**Sprint:** Sprint 2
```

Labels for categorization:

- `type: feature`
- `type: bug`
- `priority: high`
- `status: in-progress`
- `area: authentication`

Milestones for releases or sprints

Projects for Kanban boards and tracking

Linking issues to pull requests for traceability

2.9 Chapter Summary

Requirements engineering is the foundation of successful software projects. Investing time in understanding and documenting what the system should do—before writing code—dramatically reduces the risk of building the wrong thing.

Key takeaways from this chapter:

- **Requirements engineering** is the systematic process of discovering, documenting, validating, and managing requirements. It's iterative and continues throughout the project.
- **Functional requirements** describe what the system should do; **non-functional requirements** describe how well it should do it (performance, security, usability, etc.).
- **Multiple elicitation techniques** are needed: interviews, questionnaires, observation, workshops, document analysis, and prototyping each reveal different types of requirements.

- **User stories** capture requirements from the user's perspective ("As a… I want… so that…") and include acceptance criteria that define when the story is complete.
- **The SRS document** serves as a contract and reference for all project stakeholders. Good requirements are clear, complete, consistent, verifiable, traceable, feasible, and necessary.
- **Requirements traceability** links requirements to their sources and to downstream artifacts (design, code, tests), ensuring nothing falls through the cracks.
- **Common challenges** include scope creep, ambiguity, conflicts, and change. Each requires specific management strategies.

2.10 Key Terms

Term	Definition
Requirements Engineering	The process of discovering, documenting, validating, and managing software requirements
Functional Requirement	A specification of what the system should do
Non-Functional Requirement	A specification of how well the system should perform (quality attributes)
Elicitation	The process of gathering requirements from stakeholders and other sources
User Story	A brief description of a feature from the perspective of a user
Acceptance Criteria	Conditions that must be met for a user story to be considered complete
Epic	A large body of work that can be broken down into smaller user stories
SRS	Software Requirements Specification; the primary requirements document
RTM	Requirements Traceability Matrix; a document linking requirements to other artifacts
Scope Creep	Uncontrolled expansion of project scope
MoSCoW	Prioritization method: Must have, Should have, Could have, Won't have
INVEST	Criteria for good user stories: Independent, Negotiable, Valuable, Estimable, Small, Testable

---

2.11 Review Questions

1. Explain the difference between functional and non-functional requirements. Why are both important? Give two examples of each for a mobile banking application.
2. Describe three different requirements elicitation techniques. For each, explain when it would be most appropriate and what types of requirements it's best suited to discover.
3. What makes a good user story according to the INVEST criteria? Write a user story for an online food ordering system and evaluate it against INVEST.
4. Why is the "so that" clause important in user stories? What happens when it's omitted?
5. Compare acceptance criteria written in Given-When-Then format versus checklist format. What are the advantages of each?
6. What are the key sections of an SRS document? Who are the different audiences for the SRS, and how does each use it?
7. Explain forward and backward traceability. Why is bi-directional traceability valuable?
8. What is scope creep? Describe three strategies for preventing or managing it.
9. You're reviewing a requirements document and find this requirement: "The system shall be secure." What's wrong with this requirement? How would you improve it?
10. A stakeholder says, "We don't have time for all this requirements documentation. Just start coding and we'll figure it out as we go." How would you respond?

2.12 Hands-On Exercises

Exercise 2.1: Elicitation Practice

Select a system you use regularly (a mobile app, website, or desktop application). Imagine you're replacing it with a new system.

1. Write 10 interview questions you would ask users of the current system.
2. Identify 5 things you would look for if you were observing users.
3. List 5 documents you would want to review.

Exercise 2.2: Writing User Stories

For your semester project, write 10 user stories following the "As a… I want… so that…" format. For each story:

1. Identify the user role
2. Write the story
3. Add 3-5 acceptance criteria
4. Evaluate against INVEST criteria

Exercise 2.3: Requirement Analysis

Review the following requirements and identify problems (ambiguity, incompleteness, conflicts, etc.). Rewrite each to improve it.

1. "The system should load quickly."
2. "Users can search for products."
3. "The system shall support all major browsers."
4. "The interface shall be intuitive."
5. "Reports should be generated daily, weekly, or on-demand."
6. "The system must be reliable."

Exercise 2.4: Software Requirements Specification

Create an SRS document for your semester project using the IEEE 830 structure as a guide. Include:

1. Introduction (purpose, scope, definitions)
2. Overall description (product perspective, user classes, constraints)
3. At least 15 functional requirements with unique IDs
4. At least 5 non-functional requirements covering different categories
5. Initial traceability to user stories

Exercise 2.5: Requirements Traceability Matrix

Create an RTM for your project that includes:

1. Requirement ID and description
2. Priority (MoSCoW)
3. Source (which elicitation activity or stakeholder)
4. Status (Not Started, In Progress, Complete)
5. Placeholder columns for Design, Code Module, and Test Case (to be filled in later)

Exercise 2.6: GitHub Project Setup

Set up requirements management for your project in GitHub:

1. Create issues for at least 10 user stories
2. Add appropriate labels (priority, type, area)
3. Create a milestone for your first release
4. Set up a project board with columns: Backlog, Ready, In Progress, Review, Done
5. Add acceptance criteria as checkboxes in each issue

2.13 Further Reading

Books:

- Wiegers, K. & Beatty, J. (2013). *Software Requirements* (3rd Edition). Microsoft Press.
- Robertson, S. & Robertson, J. (2012). *Mastering the Requirements Process* (3rd Edition). Addison-Wesley.
- Cohn, M. (2004). *User Stories Applied.* Addison-Wesley.
- Patton, J. (2014). *User Story Mapping.* O'Reilly Media.

Standards:

- IEEE 830-1998: Recommended Practice for Software Requirements Specifications
- ISO/IEC/IEEE 29148:2018: Systems and software engineering — Life cycle processes — Requirements engineering

Online Resources:

- Atlassian Agile Coach: User Stories (https://www.atlassian.com/agile/project-management/user-stories)
- Mountain Goat Software: User Stories (https://www.mountaingoatsoftware.com/agile/user-stories)
- Requirements Engineering Magazine (https://re-magazine.ireb.org/)

References

Cohn, M. (2004). *User Stories Applied: For Agile Software Development.* Addison-Wesley.

IEEE. (1998). IEEE Recommended Practice for Software Requirements Specifications (IEEE Std 830-1998).

Patton, J. (2014). *User Story Mapping: Discover the Whole Story, Build the Right Product.* O'Reilly Media.

Pohl, K. (2010). *Requirements Engineering: Fundamentals, Principles, and Techniques.* Springer.

Robertson, S., & Robertson, J. (2012). *Mastering the Requirements Process: Getting Requirements Right* (3rd Edition). Addison-Wesley.

Standish Group. (2020). *CHAOS Report 2020.* The Standish Group International.

Wake, B. (2003). INVEST in Good Stories, and SMART Tasks. Retrieved from https://xp123.com/articles/invest-in-good-stories-and-smart-tasks/

Wiegers, K., & Beatty, J. (2013). *Software Requirements* (3rd Edition). Microsoft Press.

Chapter 3: Systems Modeling and UML

Learning Objectives

By the end of this chapter, you will be able to:

- Explain the purpose and value of systems modeling in software engineering
- Read and create Use Case diagrams to capture system functionality
- Model workflows and processes using Activity diagrams
- Represent object interactions over time with Sequence diagrams
- Design system structure using Class diagrams and domain models
- Select appropriate diagram types for different modeling needs
- Apply UML notation correctly and consistently
- Use modeling tools to create professional diagrams

3.1 Why Model Software Systems?

Imagine trying to build a house by describing it only in words. "There's a living room connected to a kitchen, and upstairs there are three bedrooms…" You could write pages of description, but a single floor plan communicates the layout instantly and unambiguously. Architects don't just describe buildings—they draw them.

Software systems are far more complex than houses, yet we often try to describe them using only text: requirements documents, code comments, and verbal explanations. **Systems modeling** provides the visual blueprints that help us understand, communicate, and reason about software before and during its construction.

3.1.1 The Purpose of Models

A **model** is a simplified representation of reality that helps us understand complex systems. Models deliberately omit details to focus on what matters for a particular purpose.

Consider a map. A road map shows highways and cities but omits elevation, vegetation, and building footprints. A topographic map shows terrain but omits road numbers. A subway map distorts geography entirely to emphasize connections between stations. Each map serves a different purpose by including different information and making different simplifications.

Software models work the same way. Different diagrams serve different purposes:

- **Use Case diagrams** show what the system does from the user's perspective
- **Activity diagrams** show how processes flow through steps and decisions
- **Sequence diagrams** show how objects interact over time
- **Class diagrams** show the structure of the system's code

No single diagram captures everything. A complete understanding requires multiple views, each revealing different aspects of the system.

3.1.2 Benefits of Modeling

Communication: Models provide a common language between stakeholders. A business analyst, a developer, and a tester can all look at the same diagram and understand what the system should do. Visual representations often communicate more effectively than pages of text.

Understanding: The act of creating a model forces you to think through the system carefully. You can't draw a sequence diagram without understanding which objects interact and in what order. Modeling reveals gaps in your understanding early, when they're cheap to address.

Documentation: Models serve as documentation that remains useful throughout the project lifecycle. Unlike code comments that often become outdated, well-maintained models provide a high-level view that helps new team members understand the system.

Analysis: Models allow you to analyze designs before implementation. You can identify potential problems, evaluate alternatives, and make architectural decisions when changes are still inexpensive.

Abstraction: Models let you work at the right level of detail. When discussing system architecture with executives, you don't need to show individual methods and parameters. When designing a specific component, you don't need the entire system context.

3.1.3 Modeling in Different Contexts

The role of modeling varies across development methodologies:

Traditional/Waterfall approaches often emphasize extensive upfront modeling. Detailed models are created during the design phase before coding begins. Changes to models require formal reviews.

Agile approaches favor "just enough" modeling. Models are created as needed, often informally on whiteboards. The emphasis is on models as communication tools rather than formal documentation. "Working software over comprehensive documentation" doesn't mean no documentation—it means documentation that adds value.

The right balance depends on your context:

- Regulated industries may require formal models for compliance
- Distributed teams benefit from documented models for asynchronous communication
- Complex systems need more modeling than simple ones
- Novel designs require more exploration than familiar patterns

For most projects, a pragmatic middle ground works best: model enough to understand and communicate the design, but don't over-invest in documentation that won't be maintained.

3.2 Introduction to UML

The **Unified Modeling Language (UML)** is a standardized visual language for specifying, constructing, and documenting software systems. Developed in the 1990s by Grady Booch, James Rumbaugh, and Ivar Jacobson (the "Three Amigos"), UML unified several competing notations into a single standard, now maintained by the Object Management Group (OMG).

3.2.1 UML Diagram Types

UML 2.5 defines 14 diagram types, organized into two main categories:

Structural Diagrams show the static structure of the system—what exists and how it's organized:

Diagram	Purpose
Class Diagram	Classes, attributes, methods, and relationships
Object Diagram	Instances of classes at a specific moment
Component Diagram	High-level software components and dependencies
Deployment Diagram	Physical deployment of software to hardware
Package Diagram	Organization of model elements into packages
Composite Structure Diagram	Internal structure of a class
Profile Diagram	Extensions to UML itself

Behavioral Diagrams show the dynamic behavior of the system—what happens over time:

Diagram	Purpose
Use Case Diagram	System functionality from user perspective
Activity Diagram	Workflows and process flows
Sequence Diagram	Object interactions over time
Communication Diagram	Object interactions emphasizing structure
State Machine Diagram	States and transitions of an object
Timing Diagram	Timing constraints on behavior
Interaction Overview Diagram	High-level view of interaction flows

In practice, four diagrams cover most modeling needs:

- **Use Case diagrams** for requirements
- **Activity diagrams** for processes
- **Sequence diagrams** for interactions

- **Class diagrams** for structure

This chapter focuses on these four essential diagram types.

3.2.2 UML Notation Basics

Before diving into specific diagrams, let's understand some notation conventions that apply across UML:

Naming conventions:

- Class names: PascalCase (e.g., `ShoppingCart`, `UserAccount`)
- Attributes and operations: camelCase (e.g., `firstName`, `calculateTotal()`)
- Constants: UPPER_CASE (e.g., `MAX_ITEMS`)

Visibility markers:

- `+` Public: accessible from anywhere
- `-` Private: accessible only within the class
- `#` Protected: accessible within class and subclasses
- `~` Package: accessible within the same package

Multiplicity indicates how many instances participate in a relationship:

- `1` Exactly one
- `0..1` Zero or one (optional)
- `*` or `0..*` Zero or more
- `1..*` One or more
- `n..m` Between n and m

Stereotypes extend UML with additional meaning, shown in guillemets:

- `«interface»` An interface rather than a class
- `«abstract»` An abstract class
- `«enumeration»` An enumeration type
- `«actor»` A user or external system

3.3 Use Case Diagrams

Use Case diagrams capture the functional requirements of a system from the user's perspective. They show what the system does (use cases) and who interacts with it (actors), without detailing how the functionality is implemented.

3.3.1 Use Case Diagram Elements

Actors represent anyone or anything that interacts with the system from outside. Actors can be:

- Human users (Customer, Administrator, Manager)
- External systems (Payment Gateway, Email Service)
- Hardware devices (Barcode Scanner, Printer)
- Time-based triggers (Scheduled Task, Nightly Batch)

Actors are drawn as stick figures with their role name below:

```
    O
   /|\     Customer
   / \
```

Use Cases represent discrete pieces of functionality that provide value to an actor. They're drawn as ovals with the use case name inside:

```


       Place Order


```

System Boundary is a rectangle that defines what's inside the system versus outside. Actors are outside; use cases are inside.

```


           Online Store System


     Browse Catalog      Place Order



     Track Order       Manage Account




```

Associations connect actors to the use cases they participate in, shown as solid lines:

```
    O
   /|\
   / \                Place Order
 Customer
```

3.3.2 Use Case Relationships

Use cases can relate to each other in several ways:

Include Relationship («include»)

When one use case always includes the behavior of another. The included use case is mandatory. This is useful for extracting common behavior shared by multiple use cases.

```
   Place Order          Verify Payment
         «include»


   Renew Sub            Verify Payment
         «include»
```

Both "Place Order" and "Renew Subscription" always include payment verification.

Extend Relationship («extend»)

When one use case optionally adds behavior to another under certain conditions. The extension is not always executed.

```
  Apply Coupon            Place Order
         «extend»
```

"Apply Coupon" extends "Place Order" but only when the customer has a coupon.

Generalization (inheritance arrow)

When one actor or use case is a specialized version of another.

```
      O
     /|\
     / \
   Customer



    O     O
   /|\   /|\
   / \   / \
Guest   Registered
Customer Customer
```

3.3.3 Complete Use Case Diagram Example

Here's a use case diagram for a library management system:

```
                    Library Management System


     Search Catalog        View Book            Reserve Book
                      Details



    O
   /|\
   / \
 Member

            Borrow Book             Return Book




                     Update Account
               «include»



           Pay Fine



                   «extend»
                         Renew Book



    O
   /|\
   / \
Librarian

           Add Book                    Remove Book




         Manage Members               Generate Report
```

```
                              Email Service
                                «system»




  Payment
  Gateway
 «system»
```

3.3.4 Use Case Descriptions

While the diagram provides an overview, each use case needs detailed documentation. A **Use Case Description** (or Use Case Specification) expands on what happens within a use case.

Use Case Description Template:

```
USE CASE: Borrow Book

ID: UC-003
Actor(s): Member, Librarian
Preconditions:
  - Member is logged in
  - Member has no overdue books
  - Member has not exceeded borrowing limit

Main Success Scenario (Basic Flow):
  1. Member searches for a book
  2. System displays book details and availability
  3. Member selects "Borrow"
  4. System verifies member's borrowing eligibility
  5. System records the loan with due date (14 days)
  6. System updates book status to "On Loan"
  7. System sends confirmation email to member
  8. System displays loan confirmation with due date

Alternative Flows:
  3a. Book is not available:
      3a1. System displays "Book unavailable" message
      3a2. System offers reservation option
      3a3. Return to step 1 or end

  4a. Member has overdue books:
      4a1. System displays message about overdue books
      4a2. Use case ends

  4b. Member at borrowing limit:
      4b1. System displays borrowing limit message
      4b2. Use case ends

Postconditions:
  - Book is assigned to member
  - Due date is set
  - Book availability is updated
  - Transaction is logged

Business Rules:
  - Maximum 5 books per member
  - Loan period is 14 days
  - Members with overdue books cannot borrow

Frequency: ~200 times per day
```

3.3.5 Best Practices for Use Case Diagrams

Naming Use Cases:

- Use verb-noun format: “Place Order,” not “Order” or “Ordering”
- Focus on user goals, not system actions: “Register Account,” not “Store User Data”
- Keep names concise but descriptive

Choosing Actors:

- Name actors by their role, not their identity: “Customer,” not “John”
- If different user types have different access, make them separate actors
- Don’t forget non-human actors (external systems, scheduled jobs)

Scope:

- Keep diagrams focused; split into multiple diagrams if needed
- Show 5-15 use cases per diagram
- Each use case should deliver value to an actor

Relationships:

- Don’t overuse «include» and «extend»; simple is often better
- «include» for mandatory common behavior
- «extend» for optional behavior
- If unsure, just use simple associations

Common Mistakes:

- Drawing implementation details (login, database operations)
- Too many use cases (every button click is not a use case)
- Actors that don’t interact with any use case
- Use cases with no associated actor

- Confusing use cases with features or functions

3.4 Activity Diagrams

Activity diagrams model the flow of activities in a process. They're excellent for visualizing workflows, business processes, algorithms, and use case scenarios. Think of them as enhanced flowcharts with support for parallel activities.

3.4.1 Activity Diagram Elements

Initial Node (filled circle): Where the flow begins.

Final Node (circle with inner filled circle): Where the flow ends.

Action/Activity (rounded rectangle): A single step or task.

```
Verify Payment
```

Decision Node (diamond): A branch point where flow takes one of several paths based on a condition. Guards (conditions) are shown in brackets.

```
[yes]    [no]
```

Merge Node (diamond): Where multiple paths come back together.

Fork (thick horizontal bar): Splits flow into parallel paths.

Join (thick horizontal bar): Synchronizes parallel paths; waits for all to complete.

Swimlanes (vertical or horizontal partitions): Show who or what performs each activity.

3.4.2 Control Flow vs. Object Flow

Control flow (solid arrows) shows the sequence of activities:

```
  Activity A       Activity B
```

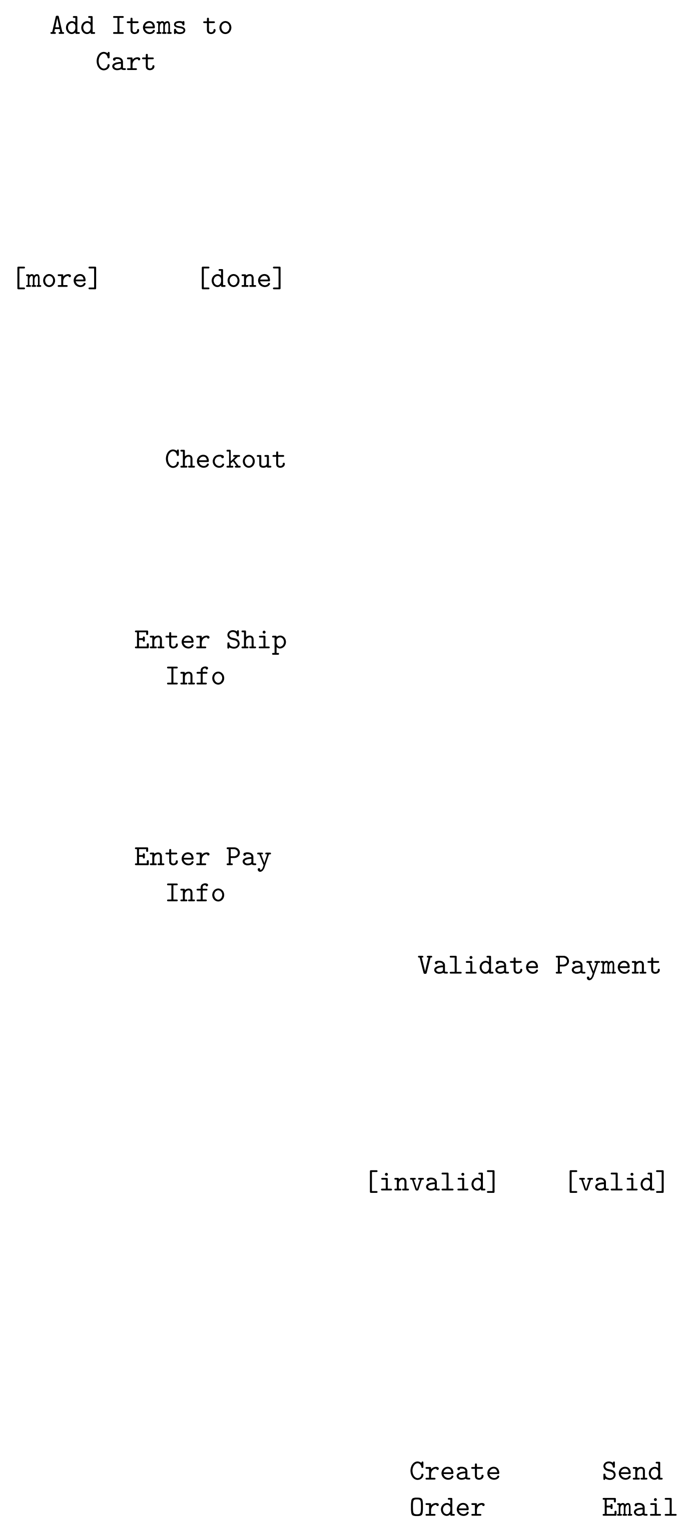
Object flow (solid arrows with object nodes) shows data passing between activities:

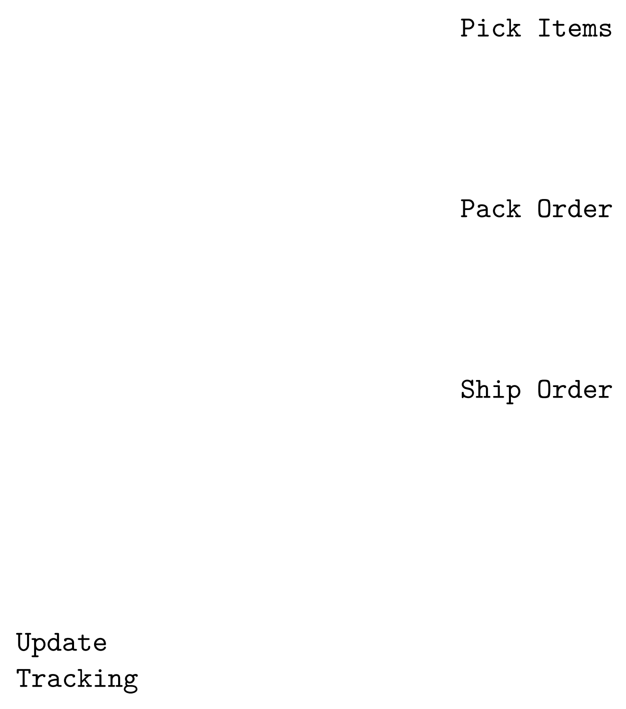
```
Create Order      [Order]     Process Order
```

3.4.3 Complete Activity Diagram Example

Here’s an activity diagram for an online order process with swimlanes:

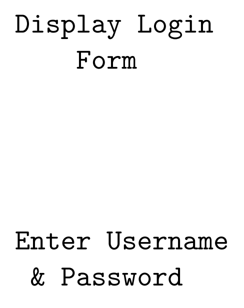
```
        Customer                  System                 Warehouse




    Browse Catalog
```


Add Items to Cart
[more]
[done]
Checkout
Enter Ship Info
Enter Pay Info
Validate Payment
[invalid]
[valid]
Create Order
Send Email



3.4.4 Activity Diagram for Algorithm Logic

Activity diagrams can also model algorithms. Here's a diagram for a simple login process:



Validate
Credentials

[valid] [invalid]

Increment
Failed Attempts

[< 3] [3]

Lock Account
(30 minutes)

Display Lock
Message

Display Error
Message

(loop back to form)

Create Session

```
Redirect to
 Dashboard
```

3.4.5 Best Practices for Activity Diagrams

Structure:

- Start with a single initial node
- End with one or more final nodes (or flow nodes for ongoing processes)
- Every path from the initial node should eventually reach a final node or loop

Decisions and Merges:

- Every decision needs at least two outgoing flows
- Guard conditions should be mutually exclusive and complete
- Use merges to rejoin split paths (optional but clarifies the diagram)

Parallelism:

- Use forks when activities can happen simultaneously
- Use joins to synchronize parallel paths
- All forked paths must eventually join (or reach a final node)

Swimlanes:

- Use when multiple actors or systems are involved
- Helps clarify responsibility for each activity
- Swimlanes can be vertical or horizontal

Level of Detail:

- Match detail level to the diagram's purpose
- High-level process diagrams: fewer, larger activities
- Detailed workflow diagrams: more granular steps
- Avoid mixing abstraction levels in one diagram

Common Mistakes:

- Missing guard conditions on decision branches
- Unbalanced forks and joins
- No path to final node
- Activities that are too vague ("Process stuff") or too detailed ("Set variable x to 5")

3.5 Sequence Diagrams

Sequence diagrams show how objects interact with each other over time. They're particularly useful for modeling the behavior of use cases, showing the messages exchanged between objects to accomplish a task.

3.5.1 Sequence Diagram Elements

Lifelines represent participants in the interaction. Each lifeline has a name and optionally a type, with a dashed line extending downward representing the participant's existence over time.

```
   :Customer


         (lifeline)
```

Messages are communications between lifelines, shown as arrows:

Synchronous message (solid arrow, filled head): Sender waits for response

Asynchronous message (solid arrow, open head): Sender continues without waiting

Return message (dashed arrow): Response to a synchronous call

```
- - - - - - - - - -
```

Self-message (arrow back to same lifeline): Object calls itself

Activation bars (rectangles on lifelines) show when an object is active (executing):

```
 :Client                          :Server


            request()


                                         (processing)

            response
```

3.5.2 Combined Fragments

Combined fragments represent control structures like loops, conditions, and alternatives:

alt (alternatives): Conditional logic (if-else)

```
    alt  [condition]

             (messages if true)

             [else]
             (messages if false)
```

opt (optional): Conditional execution (if without else)

```
    opt  [condition]

             (messages if condition true)
```

loop: Repeated execution

```
    loop [condition or count]

             (repeated messages)
```

par (parallel): Concurrent execution

```
par
        (parallel region 1)

        (parallel region 2)
```

3.5.3 Complete Sequence Diagram Example

Here's a sequence diagram for a user login process:

```
   :User         :LoginController     :AuthService          :Database


        enterCredentials(username, password)


                          authenticate(username, password)


                                               findUser(username)


                                                 user




           alt  [user exists]

                                               verifyPassword(
                                                 password,
                                                 user.hashedPassword)




               alt  [password valid]

                                              createSession(user)
```

sessionToken

loginSuccess(token)

[else]

AuthException

loginFailed("Invalid password")

[else]

AuthException

loginFailed("User not found")

displayResult()

3.5.4 Object Creation and Destruction

Objects can be created during the interaction:

:Factory

create()

:Product

Objects can be destroyed (shown with an X):

```
               destroy()
                             X
```

3.5.5 Best Practices for Sequence Diagrams

Focus:

- One diagram per scenario or use case
- Show the main success path; use separate diagrams for alternatives
- Include enough detail to understand the interaction, but not implementation minutiae

Naming:

- Name lifelines with role:Type format (e.g., `:Customer`, `cart:ShoppingCart`)
- Use descriptive message names that indicate what happens
- Include parameters when they add clarity

Layout:

- Arrange lifelines left-to-right in order of first involvement
- Place the initiating actor on the left
- Keep crossing message lines to a minimum

Level of Detail:

- High-level diagrams: show major components and their interactions
- Detailed diagrams: show individual method calls and returns
- Match detail level to your audience and purpose

Common Mistakes:

- Too many lifelines (hard to read; consider splitting the diagram)
- Missing return messages for synchronous calls
- Unclear message sequencing
- Mixing abstraction levels (business actions and technical implementation)

3.6 Class Diagrams

Class diagrams show the static structure of a system: the classes, their attributes and methods, and the relationships between them. They're the most commonly used UML diagram for designing object-oriented systems.

3.6.1 Class Notation

A class is shown as a rectangle divided into three compartments:

```
          ClassName                ← Name compartment

 - privateAttribute: Type          ← Attributes compartment
 # protectedAttribute: Type
 + publicAttribute: Type

 + publicMethod(): ReturnType      ← Operations compartment
 - privateMethod(param: Type)
 # protectedMethod(): void
```

Visibility markers:

- + Public
- - Private
- # Protected
- ~ Package

Attribute syntax:

```
visibility name: type [multiplicity] = defaultValue
```

Examples:

```
- id: int
+ name: String
- items: Product [0..*]
+ status: OrderStatus = PENDING
```

Operation syntax:

```
visibility name(parameters): returnType
```

Examples:

```
+ calculateTotal(): Decimal
- validateInput(data: String): Boolean
+ addItem(product: Product, quantity: int): void
```

3.6.2 Relationships Between Classes

Association: A general relationship between classes. Objects of one class know about objects of the other.

```
   Student                  Course
```

With role names and multiplicity:

```
            enrolledIn
   Student                 Course
         1..*        0..*
```

A student is enrolled in zero or more courses; a course has one or more students.

Navigability: Arrows indicate which class knows about which:

```
    Order                 Customer
```

Order knows about Customer, but Customer doesn't have a direct reference to Order.

Aggregation (hollow diamond): "Has-a" relationship where parts can exist independently of the whole.

```
  Department              Employee
```

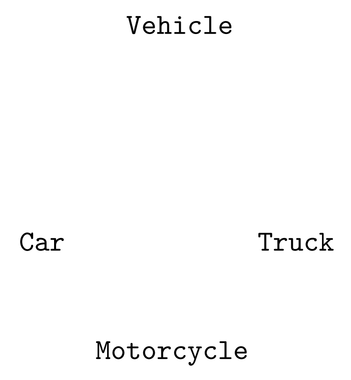
A department has employees, but employees can exist without the department.

Composition (filled diamond): "Has-a" relationship where parts cannot exist without the whole.

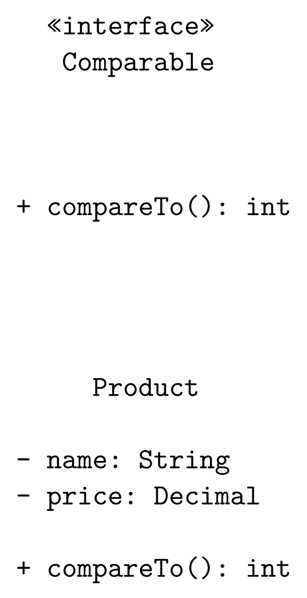
```
    Order                 OrderLine
```

An order contains order lines; order lines cannot exist without an order.

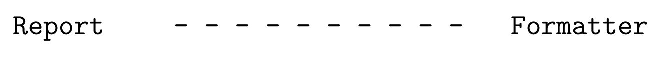
Inheritance/Generalization (hollow triangle): "Is-a" relationship; one class is a specialized version of another.



Realization/Implementation (dashed line, hollow triangle): A class implements an interface.



Dependency (dashed arrow): A weaker relationship; one class uses another temporarily.



Report depends on Formatter (perhaps uses it as a parameter or local variable) but doesn't hold a long-term reference.

3.6.3 Association Classes

Sometimes a relationship itself has attributes. An **association class** captures this:

```
  Student                  Course



                   Enrollment

                   - grade
                   - date
```

The Enrollment class captures attributes of the student-course relationship (grade, enrollment date).

3.6.4 Complete Class Diagram Example

Here's a class diagram for a simplified e-commerce system:

```
     «interface»                         «abstract»
      Payable                              User

                                     - id: int
                           - email: String
  + pay(): Boolean                   - password: String
  + getAmount(): Dec                 - createdAt: Date

                                      + login(): Boolean
                                      + logout(): void




      Order             Customer         Admin         Vendor

  - id: int           - firstName    - department   - companyName
  - orderDate: Date   - lastName     - role         - rating
  - status: Status    - address
  - total: Decimal           +manageUsers   +addProduct
```

```
       +placeOrder()  +viewReports   +viewSales
+ pay(): Boolean       +getOrders()
+ getAmount(): Dec
+ cancel(): void                places
+ ship(): void                  1


     contains
     1..*

  OrderLine                                      0..*

- quantity: int
- unitPrice: Decimal                    Product

+ getSubtotal(): Dec      - id: int
        - name: String
                          - description: String
     1                    - price: Decimal
                          - stock: int

                          + updateStock(qty: int): void
                          + isAvailable(): Boolean


                                     belongsTo
                                     *

                                    Category

                                 - id: int
                                 - name: String
                                 - parent: Category

                                 + getProducts(): []
```

3.6.5 Domain Models vs. Design Class Diagrams

Class diagrams serve different purposes at different project stages:

Domain Model (conceptual class diagram):

- Created during requirements/analysis
- Shows concepts in the problem domain
- Focuses on what exists, not implementation

- Uses business terminology
- Minimal or no methods
- No implementation-specific types

Design Class Diagram:

- Created during design
- Shows software classes
- Includes implementation details
- Uses programming terminology
- Complete methods with signatures
- Specific types (String, int, List)

Example: Domain Model

```
               places
  Customer                    Order
        1                0..*
 name                                date
 address                             total
                               status
```

Example: Design Class Diagram

```
         Customer                                Order

 - id: Long                             - id: Long
 - firstName: String           1    *   - orderDate: LocalDate
 - lastName: String                 - total: BigDecimal
 - email: String                        - status: OrderStatus
 - addresses: List<Address>             - customerId: Long

 + getFullName(): String                + calculateTotal(): void
 + addAddress(a: Address)               + cancel(): Boolean
 + getOrders(): List<Order>             + ship(): Boolean
```

3.6.6 Best Practices for Class Diagrams

Organization:

- Group related classes together
- Use packages for larger diagrams
- Consider multiple diagrams for different views or subsystems

Detail Level:

- Domain models: concepts and relationships only
- Design diagrams: full detail for implementation
- Overview diagrams: key classes and relationships, minimal detail

Relationships:

- Choose the right relationship type (association vs. dependency)
- Include multiplicities for associations
- Add role names when they add clarity
- Use navigability arrows to show direction of knowledge

Naming:

- Classes: noun phrases (Customer, ShoppingCart)
- Attributes: noun phrases (firstName, orderTotal)
- Methods: verb phrases (calculateTotal, validateInput)
- Use consistent naming conventions

Common Mistakes:

- Too much detail (every attribute and method)
- Too little detail (just boxes with names)
- Incorrect relationship types
- Missing multiplicities
- Confusing domain concepts with implementation classes

3.7 Choosing the Right Diagram

With multiple diagram types available, how do you decide which to use?

3.7.1 Matching Diagrams to Questions

Question You're Answering	Diagram Type
What can the system do? Who uses it?	Use Case Diagram
How does a process flow? What are the steps?	Activity Diagram
How do objects interact to accomplish a task?	Sequence Diagram
What classes exist? How are they related?	Class Diagram
What states can an object be in?	State Machine Diagram
How is the system deployed?	Deployment Diagram
How is code organized into packages?	Package Diagram

3.7.2 Diagrams Through the Development Lifecycle

Requirements Phase:

- Use Case diagrams to capture functionality
- Activity diagrams for business processes
- Domain models for key concepts

Design Phase:

- Sequence diagrams for use case realizations
- Class diagrams for detailed design
- State diagrams for complex object behavior
- Component and deployment diagrams for architecture

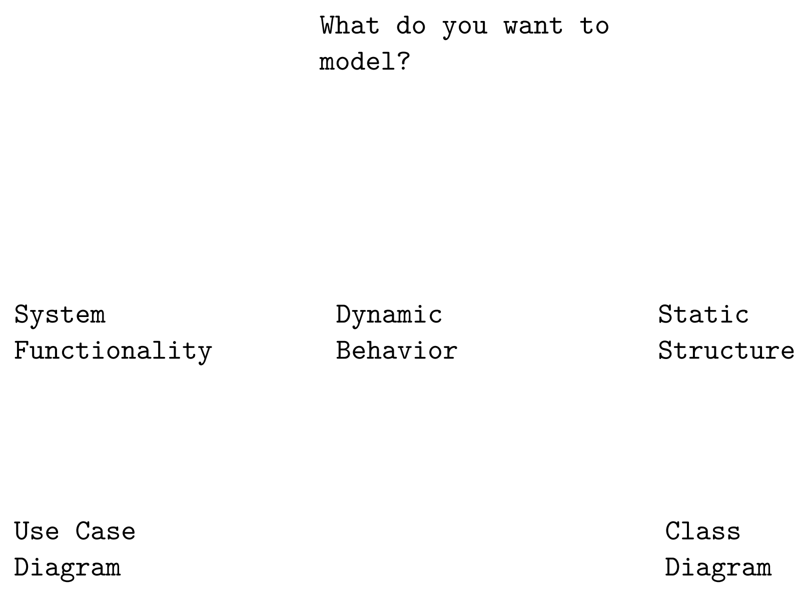
Implementation Phase:

- Class diagrams as code reference
- Sequence diagrams for complex interactions
- Activity diagrams for algorithms

Testing Phase:

- Use cases and activity diagrams for test scenarios
- Sequence diagrams for integration test design

3.7.3 Diagram Selection Guide



```
   Process/        Object          Object
   Workflow        Interactions    Lifecycle




 Activity          Sequence          State Machine
 Diagram           Diagram           Diagram
```

3.8 Modeling Tools

While you can sketch UML diagrams on paper or whiteboards, tools provide benefits like professional appearance, easy modification, and collaboration features.

3.8.1 Categories of Tools

Full-Featured UML Tools:

- Enterprise Architect (commercial)
- Visual Paradigm (commercial/free community edition)
- StarUML (commercial/free)
- Modelio (open source)

These tools offer complete UML support, code generation, reverse engineering, and team collaboration.

Diagramming Tools with UML Support:

- Lucidchart (web-based, collaborative)
- Draw.io/diagrams.net (free, web and desktop)
- Microsoft Visio (commercial)
- Miro (web-based, collaborative)

These tools support UML shapes but aren't specialized UML tools.

Text-Based Tools:

- PlantUML (text to diagram)
- Mermaid (text to diagram, integrates with Markdown)
- Nomnoml (text to diagram)

These tools let you write diagrams in a text format that's version-control friendly.

3.8.2 PlantUML Example

PlantUML uses a simple text syntax to generate diagrams:

Use Case Diagram:

```
@startuml
left to right direction
actor Customer
actor Admin

rectangle "Online Store" {
    Customer --> (Browse Products)
    Customer --> (Place Order)
    Customer --> (Track Order)
    Admin --> (Manage Products)
    Admin --> (Process Orders)
    (Place Order) .> (Process Payment) : include
}
@enduml
```

Sequence Diagram:

```
@startuml
actor User
participant "Login Controller" as LC
participant "Auth Service" as AS
database "User DB" as DB

User -> LC: enterCredentials(user, pass)
LC -> AS: authenticate(user, pass)
AS -> DB: findUser(user)
DB --> AS: user
AS -> AS: verifyPassword()
AS --> LC: token
LC --> User: loginSuccess(token)
@enduml
```

Class Diagram:

```
@startuml
class Customer {
    -id: Long
    -name: String
    -email: String
    +placeOrder(): Order
}

```

```
class Order {
    -id: Long
    -date: Date
    -status: OrderStatus
    +calculateTotal(): Decimal
    +cancel(): void
}

class OrderLine {
    -quantity: int
    -unitPrice: Decimal
    +getSubtotal(): Decimal
}

Customer "1" -- "0..*" Order : places
Order "1" *-- "1..*" OrderLine : contains
@enduml
```

3.8.3 Mermaid Example

Mermaid integrates well with Markdown and is supported by GitHub, GitLab, and many documentation platforms:

Sequence Diagram:

```
sequenceDiagram
    participant U as User
    participant L as LoginController
    participant A as AuthService
    participant D as Database

    U->>L: enterCredentials(user, pass)
    L->>A: authenticate(user, pass)
    A->>D: findUser(user)
    D-->>A: user
    A->>A: verifyPassword()
    A-->>L: token
    L-->>U: loginSuccess(token)
```

Class Diagram:

```
classDiagram
    class Customer {
        -Long id
        -String name
        -String email
        +placeOrder() Order
    }

    class Order {
        -Long id
        -Date date
        -OrderStatus status
        +calculateTotal() Decimal
        +cancel() void
    }

    Customer "1" --> "0..*" Order : places
```

3.8.4 Tool Selection Considerations

When choosing a modeling tool, consider:

- **Learning curve**: How quickly can you become productive?
- **Collaboration**: Does your team need to work together on diagrams?
- **Integration**: Does it integrate with your other tools (IDE, documentation)?
- **Cost**: Is it within budget?
- **Version control**: Can diagrams be tracked in Git?
- **Export options**: What formats can you export to?

For your course project, Draw.io (free, easy) or PlantUML (text-based, version-control friendly) are excellent choices.

3.9 Chapter Summary

Systems modeling provides visual blueprints that help us understand, communicate, and design software. UML offers a standardized notation for these models, with different diagram types serving different purposes.

Key takeaways from this chapter:

- **Models** are simplified representations that help us understand complex systems. Different diagrams reveal different aspects of the system.

- **Use Case diagrams** capture system functionality from the user's perspective. They show actors (who uses the system) and use cases (what they can do).

- **Activity diagrams** model workflows and processes. They're excellent for showing the steps in a process, decision points, parallel activities, and swimlanes for responsibility.

- **Sequence diagrams** show how objects interact over time. They're particularly useful for modeling use case scenarios and understanding the flow of messages between components.
- **Class diagrams** show the static structure of the system: classes, their attributes and methods, and relationships between them. They range from conceptual domain models to detailed design specifications.
- **Choosing the right diagram** depends on what you're trying to communicate. Use case diagrams for requirements, activity diagrams for processes, sequence diagrams for interactions, and class diagrams for structure.
- **Tools** range from simple drawing applications to sophisticated modeling environments. Text-based tools like PlantUML offer version-control-friendly alternatives.

3.10 Key Terms

Term	Definition
UML	Unified Modeling Language; a standardized visual notation for software systems
Actor	An external entity (person, system, device) that interacts with the system
Use Case	A discrete piece of functionality that provides value to an actor
Activity Diagram	A diagram showing the flow of activities in a process
Swimlane	A partition in an activity diagram showing who performs each activity
Sequence Diagram	A diagram showing object interactions over time
Lifeline	The representation of a participant in a sequence diagram
Class Diagram	A diagram showing classes, their attributes/methods, and relationships
Association	A relationship between classes indicating objects of one class know about objects of another
Aggregation	A "has-a" relationship where parts can exist independently
Composition	A "has-a" relationship where parts cannot exist without the whole
Generalization	An "is-a" (inheritance) relationship between classes
Domain Model	A conceptual class diagram showing concepts in the problem domain

Term	Definition
Multiplicity	The number of instances that participate in a relationship

3.11 Review Questions

1. Explain the purpose of systems modeling in software engineering. What are three benefits of creating models before writing code?
2. What is the difference between structural and behavioral UML diagrams? Give two examples of each.
3. In a use case diagram, what is the difference between the «include» and «extend» relationships? When would you use each?
4. Create a use case diagram for an ATM system. Include at least three actors and eight use cases with appropriate relationships.
5. Explain the purpose of swimlanes in activity diagrams. When are they most useful?
6. What is the difference between a fork and a decision in an activity diagram? How do they differ visually and semantically?
7. In a sequence diagram, what is the difference between synchronous and asynchronous messages? When would you use each?
8. Explain the difference between aggregation and composition in class diagrams. Provide an example of each.
9. What is the difference between a domain model and a design class diagram? At what project phase would you create each?
10. You're designing a ride-sharing application. Which UML diagrams would you create, and what would each show?

3.12 Hands-On Exercises

Exercise 3.1: Use Case Diagram

Create a use case diagram for a hotel reservation system. Include:

1. At least 3 actors (consider guests, staff, external systems)
2. At least 10 use cases
3. At least 2 «include» relationships

4. At least 1 «extend» relationship
5. A system boundary

Write detailed use case descriptions for 2 of your use cases.

Exercise 3.2: Activity Diagram

Create an activity diagram for one of the following processes:

Option A: Online food ordering (from browsing menu to delivery) Option B: Library book borrowing (including reservation if unavailable) Option C: Job application process (from submission to hire/reject)

Include:

1. Initial and final nodes
2. At least 2 decision points with guards
3. At least 1 fork/join for parallel activities
4. Swimlanes showing different participants

Exercise 3.3: Sequence Diagram

Create a sequence diagram for one of the following scenarios:

Option A: User purchasing an item online (including payment processing) Option B: User posting a message to a social media platform Option C: ATM withdrawal transaction

Include:

1. At least 4 lifelines
2. At least one combined fragment (alt, opt, or loop)
3. Synchronous and return messages
4. Activation bars

Exercise 3.4: Class Diagram

Create a class diagram for a course registration system. Include:

1. At least 8 classes
2. Appropriate attributes and methods for each class
3. At least one inheritance relationship
4. At least one composition relationship
5. At least one association with multiplicity
6. An interface

Exercise 3.5: Project UML Package

For your semester project, create a UML package containing:

1. A use case diagram showing the main functionality
2. An activity diagram for a key process or workflow
3. A sequence diagram for a primary use case
4. A domain model (conceptual class diagram)

Upload all diagrams to your GitHub repository in a `docs/diagrams` folder.

Exercise 3.6: Tool Exploration

Choose one of these modeling approaches and create the class diagram from Exercise 3.4:

1. Draw.io: Create the diagram using the web-based tool
2. PlantUML: Write the diagram in text format
3. Mermaid: Write the diagram in Mermaid syntax in a Markdown file

Compare the experience. Which do you prefer and why?

3.13 Further Reading

Books:

- Fowler, M. (2003). *UML Distilled: A Brief Guide to the Standard Object Modeling Language* (3rd Edition). Addison-Wesley.
- Larman, C. (2004). *Applying UML and Patterns* (3rd Edition). Prentice Hall.
- Rumbaugh, J., Jacobson, I., & Booch, G. (2004). *The Unified Modeling Language Reference Manual* (2nd Edition). Addison-Wesley.

Online Resources:

- UML Specification (OMG): https://www.omg.org/spec/UML/
- PlantUML Documentation: https://plantuml.com/
- Mermaid Documentation: https://mermaid.js.org/
- Draw.io: https://app.diagrams.net/
- Visual Paradigm UML Guides: https://www.visual-paradigm.com/guide/uml-unified-modeling-language/

Tutorials:

- UML Diagrams (Lucidchart): https://www.lucidchart.com/pages/uml
- UML Tutorial (Tutorialspoint): https://www.tutorialspoint.com/uml/

References

Booch, G., Rumbaugh, J., & Jacobson, I. (2005). *The Unified Modeling Language User Guide* (2nd Edition). Addison-Wesley.

Fowler, M. (2003). *UML Distilled: A Brief Guide to the Standard Object Modeling Language* (3rd Edition). Addison-Wesley.

Larman, C. (2004). *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development* (3rd Edition). Prentice Hall.

Object Management Group. (2017). *OMG Unified Modeling Language (OMG UML) Version 2.5.1*. Retrieved from https://www.omg.org/spec/UML/2.5.1/

Rumbaugh, J., Jacobson, I., & Booch, G. (2004). *The Unified Modeling Language Reference Manual* (2nd Edition). Addison-Wesley.

Chapter 4: Software Architecture and Design Patterns

Learning Objectives

By the end of this chapter, you will be able to:

- Define software architecture and explain its importance in system development
- Compare and contrast major architectural styles and patterns
- Apply the SOLID principles to create maintainable, flexible designs
- Recognize and implement common creational, structural, and behavioral design patterns
- Make informed architectural decisions based on system requirements
- Create a Software Architecture Document (SAD) for a project
- Evaluate trade-offs between different architectural approaches

---

4.1 What Is Software Architecture?

When you look at a building, you don't see a random pile of bricks, steel, and glass. You see structure: floors stacked upon floors, walls dividing spaces, a roof keeping out the rain. That structure isn't accidental—an architect designed it to serve the building's purpose while meeting constraints of physics, budget, and building codes.

Software systems have architecture too. While you can't see or touch it, the architecture profoundly affects how the system behaves, how it can be modified, and whether it will succeed or fail over its lifetime.

Software architecture refers to the fundamental structures of a software system, the discipline of creating such structures, and the documentation of these structures. It encompasses the high-level decisions about how components are organized, how they communicate, and how the system achieves its quality requirements.

4.1.1 Why Architecture Matters

Architecture decisions are among the most consequential choices in software development. They're also among the hardest to change later.

Early decisions with lasting impact: Architectural choices made in the first weeks of a project constrain what's possible for years afterward. Choosing a monolithic architecture means you can't easily scale individual components. Choosing microservices means you need to handle distributed system complexity. Neither choice is inherently right or wrong, but both have long-lasting consequences.

Quality attributes depend on architecture: How fast is your system? How reliable? How secure? How easy to modify? These quality attributes aren't primarily determined by code quality—they emerge from architectural decisions. A well-designed system can be fast and reliable even with some sloppy code. A poorly architected system will struggle no matter how carefully each line is written.

Communication framework: Architecture provides a vocabulary for discussing the system. When you say "the payment service calls the order service," everyone understands what you mean. Without architecture, conversations about the system devolve into discussions of individual files and functions.

Risk management: Architectural decisions address the biggest risks in a project. If scalability is critical, the architecture must support it from the start. If security is paramount, architectural controls must be in place. Trying to add these qualities later is expensive at best, impossible at worst.

4.1.2 Architecture vs. Design

People often confuse software architecture with software design, and the boundary between them is genuinely fuzzy. Here's a useful distinction:

Architecture concerns the decisions that are:

- Hard to change later
- Affect multiple components or the entire system
- Related to quality attributes (performance, security, maintainability)
- About structure at a high level of abstraction

Design concerns the decisions that are:

- Relatively easy to change
- Affect individual components or modules
- Related to implementing specific functionality
- About structure at a lower level of abstraction

Consider a house analogy: Architecture is deciding to have three stories, placing load-bearing walls, and running plumbing through certain walls. Design is choosing cabinet hardware, paint colors, and light fixtures. You can change the paint without affecting the building's structure; you can't easily move a load-bearing wall.

In software, architecture might decide that the system uses a microservices approach with an API gateway, message queues for asynchronous communication, and a separate database per service. Design might decide that a particular service uses the Repository pattern for data access, or that a specific class uses the Strategy pattern for algorithm selection.

The line between architecture and design shifts based on context. In a small application, the decision to use a particular database might be "just design." In a large enterprise system, that same decision might be architectural because it affects so many components.

4.1.3 The Role of the Software Architect

In some organizations, "software architect" is a formal title held by senior technical staff. In others, architecture is a responsibility shared among the team. Either way, architectural thinking involves:

Understanding requirements: Both functional requirements and quality attributes (often called non-functional requirements). A system that needs to handle 100 users has different architectural needs than one serving 10 million.

Making trade-offs: Every architectural decision involves trade-offs. Microservices offer scalability but add complexity. Caching improves performance but risks stale data. The architect's job is to make these trade-offs explicitly and wisely.

Communicating decisions: Architecture must be documented and communicated. If the team doesn't understand the architecture, they'll inadvertently undermine it with every coding decision.

Evolving the architecture: Requirements change. Technology evolves. Architectures must adapt. The best architectures anticipate change and make evolution possible.

4.2 Architectural Styles and Patterns

An **architectural style** is a named collection of architectural decisions that are commonly applied in a given context, along with the constraints that produce certain desirable qualities. Think of architectural styles as templates or patterns that have proven effective for certain types of systems.

Let's explore the major architectural styles you'll encounter in modern software development.

4.2.1 Layered Architecture

The **layered architecture** (also called n-tier architecture) organizes the system into horizontal layers, each providing services to the layer above it and consuming services from the layer below.

```
          Presentation Layer
   (UI, Views, Controllers, API endpoints)
```

```
             Business Logic Layer
        (Services, Domain logic, Rules)




             Data Access Layer
      (Repositories, DAOs, ORM mappings)




               Database Layer
        (Database, File system, External APIs)
```

Key Principles:

- Each layer has a specific responsibility
- Layers only communicate with adjacent layers (typically downward)
- Higher layers depend on lower layers, not vice versa
- Each layer can be developed and tested somewhat independently

Common Layer Configurations:

Three-tier architecture:

1. Presentation (UI)
2. Business Logic
3. Data

Four-tier architecture:

1. Presentation
2. Application/API
3. Business Logic
4. Data

Advantages:

- **Separation of concerns**: Each layer focuses on one aspect of the system
- **Testability**: Layers can be tested independently with mocks for adjacent layers
- **Maintainability**: Changes to one layer typically don't affect others
- **Team organization**: Different teams can work on different layers
- **Familiar pattern**: Well understood by most developers

Disadvantages:

- **Performance overhead**: Requests must pass through all layers
- **Monolithic deployment**: Usually deployed as a single unit

- **Rigidity**: Strict layering can feel constraining
- **God classes risk**: Business logic layer can become bloated

When to Use:

- Traditional enterprise applications
- Applications with clear separation between presentation, logic, and data
- Teams familiar with this pattern
- Systems where simplicity is valued over flexibility

Example Structure (Web Application):

```
src/
   presentation/
       controllers/
           UserController.java
           OrderController.java
       views/
            ...
   business/
       services/
           UserService.java
           OrderService.java
       domain/
            User.java
            Order.java
   data/
       repositories/
           UserRepository.java
           OrderRepository.java
       entities/
            ...
   config/
        ...
```

4.2.2 Model-View-Controller (MVC)

MVC is an architectural pattern that separates an application into three interconnected components, originally developed for desktop GUIs but now ubiquitous in web applications.

```
                                 User


                          interacts with
                          VIEW
                  (Displays data to user)




    updates                                   user actions




                  manipulates
      MODEL                    CONTROLLER
   (Data and                          (Handles input,
    Logic)                     coordinates)
                   notifies
```

Components:

Model: Manages the data, logic, and rules of the application. It's independent of the user interface. When data changes, the model notifies observers (often the view).

View: Presents data to the user. It receives data from the model and renders it. Multiple views can display the same model data differently.

Controller: Accepts input from the user (via the view), converts it to commands for the model or view. It's the intermediary between user interaction and system response.

MVC Variants:

Traditional MVC (as above): Model notifies View directly of changes.

MVP (Model-View-Presenter): The Presenter mediates all communication between Model and View. The View is passive.

```
  View         Presenter       Model
```

MVVM (Model-View-ViewModel): Common in modern frontend frameworks. ViewModel exposes data streams that the View binds to.

```
        data binding
  View                ViewModel       Model
```

MVC in Web Frameworks:

Most web frameworks implement a variation of MVC:

- **Ruby on Rails**: Traditional MVC with ActiveRecord models
- **Django**: Often called MTV (Model-Template-View)
- **Spring MVC**: Java-based MVC framework
- **ASP.NET MVC**: Microsoft's MVC implementation
- **Express.js**: Flexible, but commonly structured as MVC

Example Flow (Web Application):

```
1. User submits login form

2. Controller receives POST /login

3. Controller extracts credentials, calls UserService.authenticate()

4. Model (UserService) validates credentials against database

5. Model returns result to Controller

6. Controller selects appropriate View (dashboard or error page)

7. View renders response and returns to user
```

Advantages:

- Clear separation of concerns
- Multiple views for same data
- Easier testing (test model independently)
- Parallel development (UI team and backend team)
- Well-supported by many frameworks

Disadvantages:

- Can be complex for simple applications
- Controllers can become bloated ("fat controllers")
- Tight coupling between View and Controller
- Learning curve for proper implementation

4.2.3 Microservices Architecture

Microservices architecture structures an application as a collection of small, autonomous services that communicate over a network. Each service is independently deployable, scalable, and can be written in different programming languages.

```
                              API Gateway







   User             Order            Product           Payment
  Service          Service           Service           Service

  [Users DB]       [Orders DB]      [Products DB]     [Payments DB]






                           Message Queue
                           (Async Comm)
```

Key Characteristics:

- **Single responsibility**: Each service does one thing well
- **Autonomy**: Services are independently deployable
- **Decentralized data**: Each service manages its own database
- **Smart endpoints, dumb pipes**: Services contain the logic; communication infrastructure is simple
- **Design for failure**: Services expect other services to fail
- **Evolutionary design**: Easy to replace or rewrite individual services

Service Communication:

Synchronous (Request-Response):

- REST APIs over HTTP
- gRPC for high-performance communication
- GraphQL for flexible querying

Asynchronous (Event-Based):

- Message queues (RabbitMQ, Amazon SQS)
- Event streaming (Apache Kafka)
- Pub/sub patterns

Common Microservices Patterns:

API Gateway: Single entry point that routes requests to appropriate services, handles cross-cutting concerns (authentication, rate limiting).

Service Discovery: Services register themselves and discover other services dynamically (Consul, Eureka, Kubernetes).

Circuit Breaker: Prevents cascade failures by stopping calls to failing services temporarily.

Saga Pattern: Manages distributed transactions across multiple services.

Advantages:

- **Independent deployment**: Update one service without deploying the entire system
- **Technology flexibility**: Use different languages/frameworks for different services
- **Scalability**: Scale individual services based on demand
- **Resilience**: Failure in one service doesn't bring down the whole system
- **Team autonomy**: Teams own their services end-to-end
- **Easier to understand**: Each service is small and focused

Disadvantages:

- **Distributed system complexity**: Network failures, latency, data consistency
- **Operational overhead**: Many services to deploy, monitor, and manage
- **Testing challenges**: Integration testing is complex
- **Data consistency**: No ACID transactions across services
- **Initial development speed**: More infrastructure to set up
- **Debugging difficulty**: Requests span multiple services

When to Use:

- Large, complex applications
- Systems requiring high scalability
- Organizations with multiple autonomous teams
- Systems with varying scalability needs across components
- When technology diversity is beneficial

When to Avoid:

- Small applications or startups (start with a modular monolith)
- Teams without DevOps expertise
- Applications where strong consistency is critical
- When operational maturity is low

4.2.4 Event-Driven Architecture

Event-driven architecture (EDA) is built around the production, detection, consumption, and reaction to events. An event represents a significant change in state.

```
             event                          event
   Event                  Event Bus /                Event
  Producer                       Message Queue                    Consumer
```

```
                                event

                            Event
                          Consumer
```

Key Concepts:

Event: A record of something that happened. Events are immutable facts. “OrderPlaced,” “User-Registered,” “PaymentReceived.”

Event Producer: A component that detects or creates events and publishes them.

Event Consumer: A component that listens for events and reacts to them.

Event Channel: The mechanism that transports events from producers to consumers (message queue, event stream).

Event-Driven Patterns:

Simple Event Notification: Producer publishes an event; consumers react. The event contains minimal data—just that something happened.

```
Event: { type: "OrderPlaced", orderId: "12345", timestamp: "..." }
```

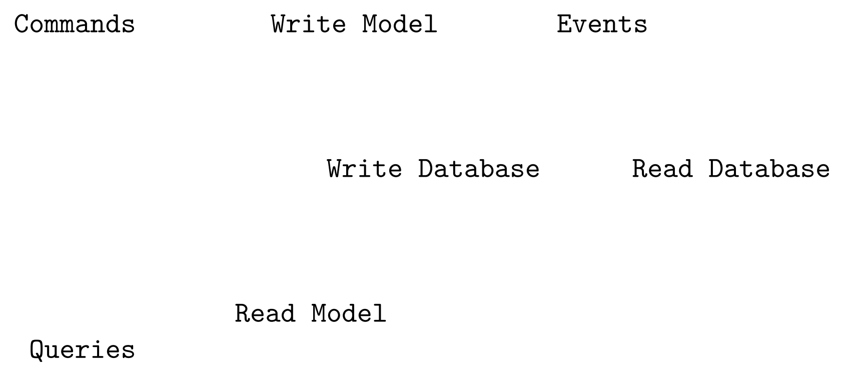
Event-Carried State Transfer: Events contain all data needed by consumers, reducing the need for callbacks.

```
Event: {
  type: "OrderPlaced",
  orderId: "12345",
  customer: { id: "789", name: "Alice", email: "..." },
  items: [...],
  total: 150.00
}
```

Event Sourcing: Instead of storing current state, store the sequence of events that led to current state. The current state is derived by replaying events.

```
Events for Account #123:
1. AccountOpened { amount: 0 }
2. Deposited { amount: 100 }
3. Withdrawn { amount: 30 }
4. Deposited { amount: 50 }

Current balance: 0 + 100 - 30 + 50 = 120
```

CQRS (Command Query Responsibility Segregation): Separate models for reading and writing data. Often combined with event sourcing.



Advantages:

- **Loose coupling**: Producers don't know about consumers
- **Scalability**: Consumers can be scaled independently
- **Flexibility**: Easy to add new consumers without changing producers
- **Responsiveness**: Asynchronous processing improves perceived performance
- **Audit trail**: Events provide natural logging
- **Temporal decoupling**: Producers and consumers don't need to be available simultaneously

Disadvantages:

- **Complexity**: Harder to trace the flow of operations
- **Eventual consistency**: Data may be inconsistent temporarily
- **Debugging difficulty**: Asynchronous flows are hard to debug
- **Event ordering**: Ensuring correct order across distributed systems is challenging
- **Event schema evolution**: Changing event formats requires careful migration

When to Use:

- Systems with many independent components
- High-throughput systems with varying load
- Systems requiring real-time reactions
- Audit and compliance requirements
- Complex workflows spanning multiple services

4.2.5 Monolithic Architecture

Before moving on, let's acknowledge the **monolithic architecture**—often presented as the opposite of microservices, but still a valid choice for many systems.

A monolith is a single deployable unit containing all application functionality.

```
                     Monolithic Application


        User             Order           Product
       Module           Module           Module




       Payment         Inventory        Reporting
       Module           Module           Module


                    Shared Database
```

Advantages:

- Simple to develop, test, deploy, and scale (initially)
- No distributed system complexity
- Easy debugging and tracing
- ACID transactions across the whole application
- Lower operational overhead

Disadvantages:

- Harder to scale specific components
- Technology stack is uniform
- Large codebase becomes unwieldy
- Deployment requires full redeployment
- Team coordination becomes challenging as system grows

The Modular Monolith:

A middle ground between monolith and microservices. The application is deployed as one unit but internally organized into well-defined, loosely-coupled modules.

```
                     Modular Monolith


        User Module               Order Module

      Public   Private          Public   Private
        API     Impl               API     Impl

         [User DB]                [Order DB]
```

Each module:

- Has a well-defined public API
- Keeps implementation details private
- Could have its own database schema
- Communicates with other modules only through APIs

This approach provides many benefits of microservices (modularity, team ownership, clear boundaries) while avoiding distributed system complexity. It's often a good starting point, with the option to extract modules into microservices later if needed.

4.2.6 Comparing Architectural Styles

Aspect	Layered	MVC	Microservices	Event-Driven
Complexity	Low	Low-Medium	High	High
Scalability	Limited	Limited	Excellent	Excellent
Deployment	Monolithic	Monolithic	Independent	Varies
Team Structure	Horizontal	By function	By service	By domain
Technology Flexibility	Low	Low	High	High
Data Consistency	Strong	Strong	Eventual	Eventual
Best For	Traditional apps	Web apps	Large systems	Reactive systems

---

4.3 The SOLID Principles

The **SOLID principles** are five design principles that help developers create software that is easy to maintain, understand, and extend. Introduced by Robert C. Martin (Uncle Bob), these principles apply at the class and module level but inform architectural decisions as well.

4.3.1 Single Responsibility Principle (SRP)

> **A class should have one, and only one, reason to change.**

The Single Responsibility Principle states that a class should have only one job. “Reason to change” refers to the actors or stakeholders who might request changes.

Violation Example:

```java
public class Employee {
    private String name;
    private double salary;

    // Business logic - reason to change: business rules
    public double calculatePay() {
        // Calculate salary, overtime, bonuses
        return salary * 1.0;
    }

    // Persistence - reason to change: database schema
    public void save() {
        // Save to database
        Database.execute("INSERT INTO employees...");
    }

    // Reporting - reason to change: report format requirements
    public String generateReport() {
        // Create performance report
        return "Employee Report: " + name + "...";
    }
}
```

This class has three reasons to change: business rules, database schema changes, and reporting requirements.

Refactored:

```java
// Handles employee data and business rules
public class Employee {
    private String name;
    private double salary;

    public double calculatePay() {
        return salary * 1.0;
    }

    // Getters and setters
}

// Handles persistence
public class EmployeeRepository {
    public void save(Employee employee) {
        Database.execute("INSERT INTO employees...");
    }

    public Employee findById(Long id) {
        // Load from database
    }
}

// Handles reporting
public class EmployeeReportGenerator {
    public String generateReport(Employee employee) {
        return "Employee Report: " + employee.getName() + "...";
    }
}
```

Now each class has one reason to change.

Benefits:

- Classes are smaller and more focused
- Changes are isolated to specific classes
- Testing is simplified
- Code is easier to understand

4.3.2 Open/Closed Principle (OCP)

Software entities should be open for extension but closed for modification.

You should be able to add new functionality without changing existing code. This is achieved through abstraction and polymorphism.

Violation Example:

```
public class AreaCalculator {
    public double calculateArea(Object shape) {
        if (shape instanceof Rectangle) {
            Rectangle r = (Rectangle) shape;
            return r.width * r.height;
        } else if (shape instanceof Circle) {
            Circle c = (Circle) shape;
            return Math.PI * c.radius * c.radius;
        }
        // Adding a new shape requires modifying this method!
        return 0;
    }
}
```

Every time we add a new shape, we must modify `AreaCalculator`.

Refactored:

```
public interface Shape {
    double calculateArea();
}

public class Rectangle implements Shape {
    private double width;
    private double height;

    @Override
    public double calculateArea() {
        return width * height;
    }
}

public class Circle implements Shape {
    private double radius;

    @Override
    public double calculateArea() {
        return Math.PI * radius * radius;
    }
}

// New shapes can be added without modifying this class
public class AreaCalculator {
    public double calculateArea(Shape shape) {
        return shape.calculateArea();
    }
}

// Adding a new shape - no modification to existing code
public class Triangle implements Shape {
    private double base;
    private double height;

    @Override
    public double calculateArea() {
        return 0.5 * base * height;
    }
}
```

Now we can add new shapes by creating new classes, without modifying existing code.

Benefits:

- Reduced risk of breaking existing functionality
- New features can be added safely
- Promotes use of abstractions

- Easier to test new functionality in isolation

4.3.3 Liskov Substitution Principle (LSP)

> **Objects of a superclass should be replaceable with objects of its subclasses without affecting the correctness of the program.**

Subtypes must be substitutable for their base types. If class B is a subtype of class A, you should be able to use B anywhere you use A without unexpected behavior.

Violation Example:

```java
public class Rectangle {
    protected int width;
    protected int height;

    public void setWidth(int width) {
        this.width = width;
    }

    public void setHeight(int height) {
        this.height = height;
    }

    public int getArea() {
        return width * height;
    }
}

public class Square extends Rectangle {
    @Override
    public void setWidth(int width) {
        this.width = width;
        this.height = width;  // Maintain square invariant
    }

    @Override
    public void setHeight(int height) {
        this.width = height;  // Maintain square invariant
        this.height = height;
    }
}
```

This seems logical—a square is a rectangle—but it violates LSP:

```
public void testRectangle(Rectangle r) {
    r.setWidth(5);
    r.setHeight(4);
    assert r.getArea() == 20;  // Fails for Square! Area would be 16.
}
```

Code written for `Rectangle` breaks when given a `Square`.

Refactored:

```
public interface Shape {
    int getArea();
}

public class Rectangle implements Shape {
    private int width;
    private int height;

    public Rectangle(int width, int height) {
        this.width = width;
        this.height = height;
    }

    @Override
    public int getArea() {
        return width * height;
    }
}

public class Square implements Shape {
    private int side;

    public Square(int side) {
        this.side = side;
    }

    @Override
    public int getArea() {
        return side * side;
    }
}
```

Now Square and Rectangle don't have an inheritance relationship that creates behavioral conflicts.

Signs of LSP Violations:

- Subclasses that throw UnsupportedOperationException

- Subclasses that override methods to do nothing
- Type checking with instanceof before calling methods
- Unexpected behavior when substituting subtypes

4.3.4 Interface Segregation Principle (ISP)

> **Clients should not be forced to depend on interfaces they do not use.**

Large interfaces should be split into smaller, more specific ones so that clients only need to know about methods relevant to them.

Violation Example:

```java
public interface Worker {
    void work();
    void eat();
    void sleep();
}

public class HumanWorker implements Worker {
    @Override
    public void work() { /* ... */ }

    @Override
    public void eat() { /* ... */ }

    @Override
    public void sleep() { /* ... */ }
}

public class RobotWorker implements Worker {
    @Override
    public void work() { /* ... */ }

    @Override
    public void eat() {
        throw new UnsupportedOperationException("Robots don't eat");
    }

    @Override
    public void sleep() {
        throw new UnsupportedOperationException("Robots don't sleep");
    }
}
```

`RobotWorker` is forced to implement methods it doesn't use.

Refactored:

```java
public interface Workable {
    void work();
}

public interface Eatable {
    void eat();
}

public interface Sleepable {
    void sleep();
}

public class HumanWorker implements Workable, Eatable, Sleepable {
    @Override
    public void work() { /* ... */ }

    @Override
    public void eat() { /* ... */ }

    @Override
    public void sleep() { /* ... */ }
}

public class RobotWorker implements Workable {
    @Override
    public void work() { /* ... */ }
}
```

Now each class implements only the interfaces it needs.

Benefits:

- Classes aren't forced to implement unused methods
- Interfaces are more cohesive
- Changes to one interface don't affect unrelated clients
- Easier to understand what a class does

4.3.5 Dependency Inversion Principle (DIP)

> **High-level modules should not depend on low-level modules. Both should depend on abstractions. Abstractions should not depend on details. Details should depend on abstractions.**

This principle is about decoupling. High-level business logic should not directly depend on low-level implementation details like databases or file systems.

Violation Example:

```java
public class MySQLDatabase {
    public void save(String data) {
        // Save to MySQL
    }
}

public class UserService {
    private MySQLDatabase database;  // Direct dependency on implementation

    public UserService() {
        this.database = new MySQLDatabase();
    }

    public void createUser(String userData) {
        // Business logic
        database.save(userData);
    }
}
```

`UserService` (high-level) directly depends on `MySQLDatabase` (low-level). Changing databases requires modifying `UserService`.

Refactored:

```java
// Abstraction
public interface Database {
    void save(String data);
}

// Low-level implementation depends on abstraction
public class MySQLDatabase implements Database {
    @Override
    public void save(String data) {
        // Save to MySQL
    }
}

public class MongoDatabase implements Database {
    @Override
    public void save(String data) {
        // Save to MongoDB
    }
}

// High-level module depends on abstraction
public class UserService {
    private Database database;  // Depends on interface, not implementation
    public UserService(Database database) {  // Dependency injection
        this.database = database;
    }

    public void createUser(String userData) {
        // Business logic
        database.save(userData);
    }
}
```

Now both high-level (`UserService`) and low-level (`MySQLDatabase`) depend on the abstraction (`Database`).

Dependency Injection:

DIP is often implemented through **dependency injection**, where dependencies are provided to a class rather than created by it:

```
// Constructor injection
UserService service = new UserService(new MySQLDatabase());

// Or for testing
UserService testService = new UserService(new MockDatabase());
```

Benefits:

- Loose coupling between components
- Easier testing (inject mocks)
- Flexibility to change implementations
- High-level modules are insulated from low-level changes

4.3.6 SOLID Summary

Principle	Focus	Key Benefit
Single Responsibility	One reason to change	Maintainability
Open/Closed	Open for extension, closed for modification	Extensibility
Liskov Substitution	Subtypes are substitutable	Correctness
Interface Segregation	Small, specific interfaces	Flexibility
Dependency Inversion	Depend on abstractions	Loose coupling

4.4 Design Patterns

Design patterns are reusable solutions to common problems in software design. They're not code you can copy directly but templates for solving problems that can be adapted to many situations.

The seminal book "Design Patterns: Elements of Reusable Object-Oriented Software" by the Gang of Four (GoF)—Gamma, Helm, Johnson, and Vlissides—cataloged 23 patterns in three categories: Creational, Structural, and Behavioral.

4.4.1 Creational Patterns

Creational patterns deal with object creation mechanisms, trying to create objects in a manner suitable to the situation.

Singleton Pattern

Intent: Ensure a class has only one instance and provide a global point of access to it.

When to Use:

- Exactly one instance is needed (database connection pool, configuration manager)
- Controlled access to a shared resource
- Global state that needs to be consistent

Structure:

```
          Singleton

 - instance: Singleton

 - Singleton()
 + getInstance(): Singleton
 + operation(): void
```

Implementation:

```java
public class DatabaseConnection {
    private static DatabaseConnection instance;
    private Connection connection;

    // Private constructor prevents direct instantiation
    private DatabaseConnection() {
        this.connection = createConnection();
    }
    // Thread-safe lazy initialization
    public static synchronized DatabaseConnection getInstance() {
        if (instance == null) {
            instance = new DatabaseConnection();
        }
        return instance;
    }

    public void query(String sql) {
        // Execute query using connection
    }

    private Connection createConnection() {
        // Create database connection
        return null;
    }
}

// Usage
DatabaseConnection db = DatabaseConnection.getInstance();
db.query("SELECT * FROM users");
```

Cautions:

- Singletons introduce global state, which can make testing difficult
- They can hide dependencies (classes use the singleton without it being in their interface)
- Consider dependency injection instead in many cases

Factory Method Pattern

Intent: Define an interface for creating an object, but let subclasses decide which class to instantiate.

When to Use:

- A class can't anticipate the class of objects it must create
- A class wants its subclasses to specify the objects it creates
- You want to localize the logic of which class to instantiate

Structure:

```
        Creator (abstract)

  + factoryMethod(): Product
  + operation(): void
```

```
ConcreteCreatorA      ConcreteCreatorB

+ factoryMethod()     + factoryMethod()
  : ProductA            : ProductB
```

Implementation:

```java
// Product interface
public interface Notification {
    void send(String message);
}

// Concrete products
public class EmailNotification implements Notification {
    @Override
    public void send(String message) {
        System.out.println("Sending email: " + message);
    }
}

public class SMSNotification implements Notification {
    @Override
    public void send(String message) {
        System.out.println("Sending SMS: " + message);
    }
}

public class PushNotification implements Notification {
    @Override
    public void send(String message) {
        System.out.println("Sending push notification: " + message);
    }
}

// Creator with factory method
public class NotificationFactory {
    public Notification createNotification(String type) {
        switch (type.toLowerCase()) {
            case "email":
                return new EmailNotification();
            case "sms":
                return new SMSNotification();
            case "push":
                return new PushNotification();
            default:
                throw new IllegalArgumentException("Unknown type: " + type);
        }
    }
}

// Usage
NotificationFactory factory = new NotificationFactory();
Notification notification = factory.createNotification("email");
notification.send("Hello, World!");
```

Builder Pattern

Intent: Separate the construction of a complex object from its representation, allowing the same construction process to create different representations.

When to Use:

- Object construction requires many parameters
- Some parameters are optional
- Object creation involves multiple steps
- Constructors with many parameters are confusing

Structure:

```
            Builder

 + setPartA(): Builder
 + setPartB(): Builder
 + setPartC(): Builder
 + build(): Product
```

Implementation:

```java
public class User {
    private final String firstName;     // Required
    private final String lastName;      // Required
    private final String email;         // Required
    private final String phone;         // Optional
    private final String address;       // Optional
    private final int age;              // Optional

    private User(UserBuilder builder) {
        this.firstName = builder.firstName;
        this.lastName = builder.lastName;
        this.email = builder.email;
        this.phone = builder.phone;
        this.address = builder.address;
        this.age = builder.age;
    }

    // Getters...

    public static class UserBuilder {
        private final String firstName;
        private final String lastName;
        private final String email;
        private String phone;
        private String address;
        private int age;

        public UserBuilder(String firstName, String lastName, String email) {
            this.firstName = firstName;
            this.lastName = lastName;
            this.email = email;
        }

        public UserBuilder phone(String phone) {
            this.phone = phone;
            return this;
        }

        public UserBuilder address(String address) {
            this.address = address;
            return this;
        }

        public UserBuilder age(int age) {
            this.age = age;
            return this;
        }

        public User build() {
            return new User(this);
        }
    }
}

// Usage - fluent interface
User user = new User.UserBuilder("John", "Doe", "john@example.com")
```

```
    .phone("555-1234")
    .age(30)
    .build();
```

4.4.2 Structural Patterns

Structural patterns deal with object composition—how classes and objects are combined to form larger structures.

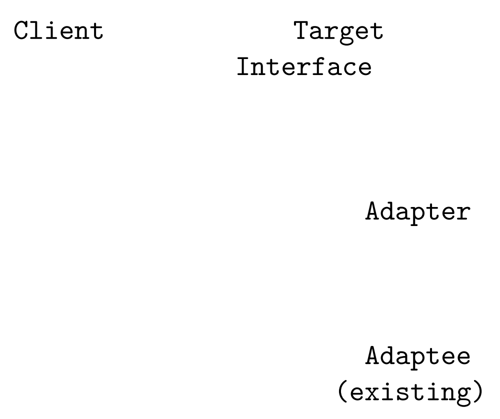
Adapter Pattern

Intent: Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.

When to Use:

- You want to use an existing class with an incompatible interface
- You're integrating with third-party code you can't modify
- You need to create a reusable class that cooperates with unrelated classes

Structure:



Implementation:

```
// Existing interface our code uses
public interface MediaPlayer {
    void play(String filename);
}

// Existing class with incompatible interface (third-party library)
public class AdvancedVideoPlayer {
    public void playMp4(String filename) {
```

```java
        System.out.println("Playing MP4: " + filename);
    }

    public void playVlc(String filename) {
        System.out.println("Playing VLC: " + filename);
    }
}

// Adapter makes AdvancedVideoPlayer compatible with MediaPlayer
public class VideoPlayerAdapter implements MediaPlayer {
    private AdvancedVideoPlayer advancedPlayer;

    public VideoPlayerAdapter() {
        this.advancedPlayer = new AdvancedVideoPlayer();
    }

    @Override
    public void play(String filename) {
        if (filename.endsWith(".mp4")) {
            advancedPlayer.playMp4(filename);
        } else if (filename.endsWith(".vlc")) {
            advancedPlayer.playVlc(filename);
        } else {
            throw new UnsupportedOperationException("Format not supported");
        }
    }
}

// Usage
MediaPlayer player = new VideoPlayerAdapter();
player.play("movie.mp4");  // Works through adapter
```

Decorator Pattern

Intent: Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.

When to Use:

- Add responsibilities to individual objects without affecting other objects
- Responsibilities can be withdrawn
- Extension by subclassing is impractical or impossible

Structure:

```
        Component (interface)
```

```
 + operation(): void





ConcreteComponent      Decorator (abstract)

 + operation()       - component: Component
             + operation()





              DecoratorA          DecoratorB

              + operation()       + operation()
              + addedBehavior()   + addedBehavior()
```

Implementation:

```java
// Component interface
public interface Coffee {
    String getDescription();
    double getCost();
}

// Concrete component
public class SimpleCoffee implements Coffee {
    @Override
    public String getDescription() {
        return "Simple Coffee";
    }

    @Override
    public double getCost() {
        return 2.00;
    }
}

// Base decorator
public abstract class CoffeeDecorator implements Coffee {
    protected Coffee coffee;
    public CoffeeDecorator(Coffee coffee) {
        this.coffee = coffee;
    }

    @Override
    public String getDescription() {
        return coffee.getDescription();
    }

    @Override
    public double getCost() {
        return coffee.getCost();
    }
}

// Concrete decorators
public class MilkDecorator extends CoffeeDecorator {
    public MilkDecorator(Coffee coffee) {
        super(coffee);
    }

    @Override
    public String getDescription() {
        return coffee.getDescription() + ", Milk";
    }

    @Override
    public double getCost() {
        return coffee.getCost() + 0.50;
    }
}

public class SugarDecorator extends CoffeeDecorator {
    public SugarDecorator(Coffee coffee) {
        super(coffee);
    }

    @Override
    public String getDescription() {
        return coffee.getDescription() + ", Sugar";
    }

    @Override
    public double getCost() {
        return coffee.getCost() + 0.25;
    }
}
```

```java
// Usage - decorators can be stacked
Coffee coffee = new SimpleCoffee();
coffee = new MilkDecorator(coffee);
coffee = new SugarDecorator(coffee);

System.out.println(coffee.getDescription());  // Simple Coffee, Milk, Sugar
System.out.println(coffee.getCost());         // 2.75
```

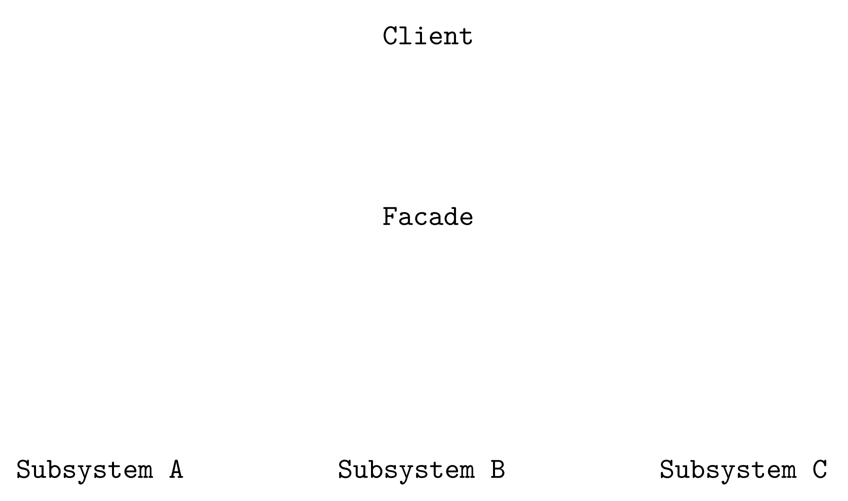
Facade Pattern

Intent: Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.

When to Use:

- You want to provide a simple interface to a complex subsystem
- There are many dependencies between clients and implementation classes
- You want to layer your subsystems

Structure:



Implementation:

```java
// Complex subsystem classes
public class CPU {
    public void freeze() { System.out.println("CPU: Freezing"); }
    public void jump(long position) { System.out.println("CPU: Jumping to " + position); }
    public void execute() { System.out.println("CPU: Executing"); }
}
```

```
public class Memory {
    public void load(long position, byte[] data) {
        System.out.println("Memory: Loading data at " + position);
    }
}

public class HardDrive {
    public byte[] read(long lba, int size) {
        System.out.println("HardDrive: Reading " + size + " bytes from " + lba);
        return new byte[size];
    }
}

// Facade provides simple interface
public class ComputerFacade {
    private CPU cpu;
    private Memory memory;
    private HardDrive hardDrive;

    private static final long BOOT_ADDRESS = 0x0000;
    private static final long BOOT_SECTOR = 0x001;
    private static final int SECTOR_SIZE = 512;

    public ComputerFacade() {
        this.cpu = new CPU();
        this.memory = new Memory();
        this.hardDrive = new HardDrive();
    }

    // Simple interface hiding complex boot sequence
    public void start() {
        cpu.freeze();
        byte[] bootData = hardDrive.read(BOOT_SECTOR, SECTOR_SIZE);
        memory.load(BOOT_ADDRESS, bootData);
        cpu.jump(BOOT_ADDRESS);
        cpu.execute();
        System.out.println("Computer started successfully!");
    }
}

// Usage - client only needs to know about facade
ComputerFacade computer = new ComputerFacade();
computer.start();
```

4.4.3 Behavioral Patterns

Behavioral patterns deal with communication between objects—how objects interact and distribute responsibility.

Strategy Pattern

Intent: Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.

When to Use:

- Many related classes differ only in their behavior
- You need different variants of an algorithm
- An algorithm uses data that clients shouldn't know about
- A class defines many behaviors as conditional statements

Structure:

```
          Context

 - strategy: Strategy

 + setStrategy(s: Strategy)
 + executeStrategy(): void




      Strategy (interface)

 + execute(): void






ConcreteA    ConcreteB    ConcreteC

+ execute()  + execute()  + execute()
```

Implementation:

```java
// Strategy interface
public interface PaymentStrategy {
    void pay(double amount);
}

// Concrete strategies
public class CreditCardPayment implements PaymentStrategy {
    private String cardNumber;
    private String cvv;

    public CreditCardPayment(String cardNumber, String cvv) {
        this.cardNumber = cardNumber;
        this.cvv = cvv;
    }

    @Override
    public void pay(double amount) {
        System.out.println("Paid $" + amount + " with credit card " +
            cardNumber.substring(cardNumber.length() - 4));
    }
}

public class PayPalPayment implements PaymentStrategy {
    private String email;

    public PayPalPayment(String email) {
        this.email = email;
    }

    @Override
    public void pay(double amount) {
        System.out.println("Paid $" + amount + " via PayPal (" + email + ")");
    }
}

public class CryptoPayment implements PaymentStrategy {
    private String walletAddress;

    public CryptoPayment(String walletAddress) {
        this.walletAddress = walletAddress;
    }

    @Override
    public void pay(double amount) {
        System.out.println("Paid $" + amount + " in crypto to " +
            walletAddress.substring(0, 8) + "...");
    }
}

// Context
public class ShoppingCart {
    private List<Item> items = new ArrayList<>();
    private PaymentStrategy paymentStrategy;

    public void addItem(Item item) {
        items.add(item);
    }

    public void setPaymentStrategy(PaymentStrategy strategy) {
        this.paymentStrategy = strategy;
    }

    public void checkout() {
        double total = items.stream()
            .mapToDouble(Item::getPrice)
            .sum();
        paymentStrategy.pay(total);
    }
}

// Usage - switch strategies at runtime
ShoppingCart cart = new ShoppingCart();
cart.addItem(new Item("Book", 29.99));
cart.addItem(new Item("Pen", 4.99));

cart.setPaymentStrategy(new CreditCardPayment("4111111111111111", "123"));
cart.checkout();  // Paid $34.98 with credit card 1111

cart.setPaymentStrategy(new PayPalPayment("user@email.com"));
cart.checkout();  // Paid $34.98 via PayPal (user@email.com)
```

Observer Pattern

Intent: Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

When to Use:

- Changes to one object require changing others, and you don't know how many objects need to change
- An object should notify other objects without knowing who they are
- You need to implement event handling systems

Structure:

```
      Subject (interface)

  + attach(o: Observer)
  + detach(o: Observer)
  + notify(): void



      ConcreteSubject                        Observer (interface)

  - state                                + update(): void
  - observers: List<Observer>

  + getState(): State
  + setState(s: State): void                 ConcreteObserver

                                             + update(): void
```

Implementation:

```java
// Observer interface
public interface Observer {
    void update(String message);
}

// Subject interface
public interface Subject {
    void attach(Observer observer);
    void detach(Observer observer);
    void notifyObservers();
}

// Concrete Subject
public class NewsAgency implements Subject {
    private String news;
    private List<Observer> observers = new ArrayList<>();

    @Override
    public void attach(Observer observer) {
        observers.add(observer);
    }

    @Override
    public void detach(Observer observer) {
        observers.remove(observer);
    }

    @Override
    public void notifyObservers() {
        for (Observer observer : observers) {
            observer.update(news);
        }
    }

    public void setNews(String news) {
        this.news = news;
        notifyObservers();
    }
}

// Concrete Observers
public class NewsChannel implements Observer {
    private String name;

    public NewsChannel(String name) {
        this.name = name;
    }

    @Override
    public void update(String news) {
        System.out.println(name + " received news: " + news);
    }
}

public class MobileApp implements Observer {
    @Override
    public void update(String news) {
        System.out.println("Mobile notification: " + news);
    }
}

// Usage
NewsAgency agency = new NewsAgency();
NewsChannel cnn = new NewsChannel("CNN");
NewsChannel bbc = new NewsChannel("BBC");
MobileApp app = new MobileApp();

agency.attach(cnn);
agency.attach(bbc);
agency.attach(app);
agency.setNews("Breaking: New discovery on Mars!");
// Output:
// CNN received news: Breaking: New discovery on Mars!
// BBC received news: Breaking: New discovery on Mars!
// Mobile notification: Breaking: New discovery on Mars!

agency.detach(bbc);
agency.setNews("Update: Weather forecast changed");
// Output:
// CNN received news: Update: Weather forecast changed
// Mobile notification: Update: Weather forecast changed
```

Template Method Pattern

Intent: Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.

When to Use:

- You want to implement the invariant parts of an algorithm once and leave variable parts to subclasses
- You want to control subclass extensions
- Common behavior among subclasses should be factored and localized in a common class

Structure:

```
       AbstractClass

 + templateMethod(): void
   - step1()
   - step2()  // abstract
   - step3()  // abstract
   - step4()
 # step1(): void
 # step2(): void {abstract}
 # step3(): void {abstract}
 # step4(): void




  ConcreteClassA          ConcreteClassB

 # step2(): void         # step2(): void
 # step3(): void          # step3(): void
```

Implementation:

```
// Abstract class with template method
public abstract class DataProcessor {

    // Template method - defines the algorithm skeleton
    public final void process() {
        readData();
        processData();
        writeData();
        cleanup();
    }

    // Common step - same for all subclasses
    private void readData() {
        System.out.println("Reading data from source...");
    }

    // Abstract steps - must be implemented by subclasses
    protected abstract void processData();
    protected abstract void writeData();

    // Hook method - optional override, has default implementation
    protected void cleanup() {
        System.out.println("Standard cleanup...");
    }
}

// Concrete implementation for CSV
public class CSVProcessor extends DataProcessor {

    @Override
    protected void processData() {
        System.out.println("Parsing CSV data, validating fields...");
    }

    @Override
    protected void writeData() {
        System.out.println("Writing to CSV output file...");
    }
}

// Concrete implementation for JSON
public class JSONProcessor extends DataProcessor {
    @Override
    protected void processData() {
        System.out.println("Parsing JSON objects, transforming structure...");
    }

    @Override
    protected void writeData() {
        System.out.println("Writing to JSON output file...");
    }

    @Override
    protected void cleanup() {
        System.out.println("Closing JSON streams and freeing memory...");
    }
}

// Usage
DataProcessor csvProcessor = new CSVProcessor();
csvProcessor.process();
// Reading data from source...
// Parsing CSV data, validating fields...
// Writing to CSV output file...
// Standard cleanup...

DataProcessor jsonProcessor = new JSONProcessor();
jsonProcessor.process();
// Reading data from source...
// Parsing JSON objects, transforming structure...
// Writing to JSON output file...
// Closing JSON streams and freeing memory...
```

4.4.4 Design Patterns Summary

Pattern	Category	Intent
Singleton	Creational	Ensure one instance with global access
Factory Method	Creational	Defer instantiation to subclasses
Builder	Creational	Construct complex objects step by step
Adapter	Structural	Convert interface to expected interface
Decorator	Structural	Add responsibilities dynamically
Facade	Structural	Provide simple interface to complex subsystem
Strategy	Behavioral	Encapsulate interchangeable algorithms
Observer	Behavioral	Notify dependents of state changes
Template Method	Behavioral	Define algorithm skeleton, defer steps

4.5 The Software Architecture Document (SAD)

A **Software Architecture Document** (SAD) communicates the architectural decisions for a system. It serves as a reference for developers, a communication tool for stakeholders, and a record of design rationale.

4.5.1 Purpose of the SAD

The SAD serves multiple purposes:

Communication: Explains the architecture to all stakeholders—developers, managers, operations, security teams.

Decision Record: Documents what was decided, why, and what alternatives were considered.

Onboarding: Helps new team members understand the system structure.

Evolution Guide: Provides context for future architectural changes.

Compliance: Satisfies documentation requirements in regulated industries.

4.5.2 SAD Structure

While formats vary, a typical SAD includes:

```
1. Introduction
   1.1 Purpose
   1.2 Scope
   1.3 Definitions, Acronyms, Abbreviations
   1.4 References

2. Architectural Goals and Constraints
   2.1 Technical Goals
   2.2 Business Goals
   2.3 Constraints

3. Architectural Representation
   3.1 Architectural Style
   3.2 Architectural Views

4. Architectural Views
   4.1 Logical View (Class/Component diagrams)
   4.2 Process View (Activity/Sequence diagrams)
   4.3 Development View (Package/Module organization)
   4.4 Physical View (Deployment diagrams)
   4.5 Use Case View (Use Case diagrams)

5. Quality Attributes
```

```
   5.1 Performance
   5.2 Scalability
   5.3 Security
   5.4 Reliability
   5.5 Maintainability

6. Design Decisions
   6.1 Decision 1: [Title]
        - Context
        - Decision
        - Rationale
        - Consequences
   6.2 Decision 2: [Title]
   ...

7. Size and Performance Targets

8. Quality Assurance

Appendices
   A. Glossary
   B. Architecture Decision Records (ADRs)
```

4.5.3 The 4+1 View Model

Philippe Kruchten's **4+1 View Model** is a popular way to organize architectural views:

```
                             Use Case
                               View
                           (Scenarios)






    Logical                 Process                 Development
      View                    View                      View
  (Functionality)         (Concurrency)            (Organization)






                           Physical
```

```
                                View
                            (Deployment)
```

Logical View: The object-oriented decomposition. Shows packages, classes, and their relationships. Addresses functional requirements.

Process View: The run-time behavior. Shows processes, threads, and their interactions. Addresses concurrency, distribution, and performance.

Development View: The static organization of software in its development environment. Shows modules, layers, and packages. Addresses build, configuration management.

Physical View: The mapping of software onto hardware. Shows nodes, networks, and deployment. Addresses availability, reliability, performance.

Use Case View (+1): The scenarios that tie other views together. Shows how the architecture supports key use cases.

4.5.4 Architecture Decision Records (ADRs)

Architecture Decision Records are a lightweight way to document individual architectural decisions. Each ADR captures one decision in a standardized format.

ADR Template:

```
# ADR-001: Use PostgreSQL as Primary Database

## Status
Accepted

## Context
We need a database for storing user data, orders, and product information.
The system needs to support complex queries, transactions, and eventual
scaling to millions of records.

## Decision
We will use PostgreSQL as our primary database.

## Rationale
- Strong ACID compliance for transactional integrity
- Excellent JSON support for semi-structured data
- Proven scalability (read replicas, partitioning)
- Team has existing PostgreSQL expertise
- Open source with strong community support
- Cloud providers offer managed PostgreSQL services

## Alternatives Considered
```

```
### MySQL
- Similar capabilities but less robust JSON support
- Team has less experience

### MongoDB
- Better for truly unstructured data
- Weaker transaction support
- Would require learning new paradigms

### DynamoDB
- Excellent scalability but vendor lock-in
- Limited query flexibility
- Higher cost at our scale

## Consequences

### Positive
- Reliable transactions for order processing
- Flexible schema evolution with JSON columns
- Easy to find developers with experience

### Negative
- Need to manage database operations (or use managed service)
- Eventual consistency challenges if we add read replicas
- May need sharding strategy for very high scale

## Date
2024-01-15

## Authors
- Jane Developer
- John Architect
```

Benefits of ADRs:

- Decisions are documented when made, preserving context
- New team members can understand why things are the way they are
- Enables revisiting decisions when circumstances change
- Creates a decision log over time
- Encourages explicit, deliberate decision-making

4.6 Making Architectural Decisions

Good architecture doesn't emerge by accident. It results from deliberate decisions made with awareness of trade-offs.

4.6.1 Factors in Architectural Decisions

Functional Requirements: What must the system do? Some functionality naturally suggests certain architectures.

Quality Attributes: Non-functional requirements like performance, scalability, security, and maintainability drive many architectural choices.

Constraints: Technology constraints, budget, timeline, team skills, regulatory requirements.

Business Context: Organizational structure, build vs. buy decisions, time to market pressures.

Technical Context: Existing systems, integration requirements, infrastructure.

4.6.2 Common Trade-offs

Performance vs. Maintainability: Optimized code is often harder to maintain. Caching improves performance but adds complexity.

Consistency vs. Availability: In distributed systems, you often can't have both perfect consistency and continuous availability (CAP theorem).

Flexibility vs. Simplicity: More abstraction and indirection enable flexibility but increase complexity.

Security vs. Usability: Stronger security measures often make systems harder to use.

Build vs. Buy: Custom solutions fit exactly but take time; third-party solutions are faster but may not fit perfectly.

Monolith vs. Microservices: Monoliths are simpler to develop and deploy; microservices offer better scalability and team autonomy but add complexity.

4.6.3 Evaluating Architectures

How do you know if an architecture is good? Consider these evaluation approaches:

Scenario Analysis: Walk through key scenarios (both typical and exceptional) to see how the architecture handles them.

Quality Attribute Analysis: For each quality attribute, assess how the architecture supports it.

Risk Assessment: Identify the biggest risks and how the architecture addresses them.

Architecture Trade-off Analysis Method (ATAM): A formal evaluation method that identifies sensitivity points, trade-offs, and risks.

Prototype/Spike: Build a minimal implementation to validate technical feasibility.

4.7 Chapter Summary

Software architecture is the foundation on which successful systems are built. It defines the high-level structure, establishes patterns for communication and organization, and addresses the quality attributes that matter most to stakeholders.

Key takeaways from this chapter:

- **Software architecture** encompasses the fundamental structures of a system and the decisions that are hard to change later. Good architecture enables systems to meet their quality requirements.
- **Architectural styles** like layered architecture, MVC, microservices, and event-driven architecture provide templates for organizing systems. Each has strengths and trade-offs.
- **The SOLID principles** guide class and module design: Single Responsibility, Open/Closed, Liskov Substitution, Interface Segregation, and Dependency Inversion.
- **Design patterns** are reusable solutions to common problems. Creational patterns (Singleton, Factory, Builder) address object creation. Structural patterns (Adapter, Decorator, Facade) address object composition. Behavioral patterns (Strategy, Observer, Template Method) address object interaction.
- **The Software Architecture Document** communicates architectural decisions to stakeholders. The 4+1 view model organizes multiple perspectives on the architecture.
- **Architecture Decision Records** document individual decisions with their context, rationale, and consequences.
- **Architectural decisions** involve trade-offs. Good architects make these trade-offs explicitly and document their reasoning.

4.8 Key Terms

Term	Definition
Software Architecture	The fundamental structures of a software system and the discipline of creating them
Architectural Style	A named collection of architectural decisions common in a given context
Layered Architecture	Architecture organizing system into horizontal layers

Term	Definition
MVC	Model-View-Controller; separates data, presentation, and control logic
Microservices	Architecture of small, independent, communicating services
Event-Driven Architecture	Architecture based on production and consumption of events
SOLID	Five design principles for maintainable object-oriented software
Design Pattern	A reusable solution to a common software design problem
Creational Pattern	Pattern dealing with object creation
Structural Pattern	Pattern dealing with object composition
Behavioral Pattern	Pattern dealing with object interaction
SAD	Software Architecture Document
ADR	Architecture Decision Record
4+1 View Model	Architectural documentation using four views plus scenarios

---

4.9 Review Questions

1. What is software architecture, and why is it important? How does it differ from software design?
2. Compare and contrast layered architecture and microservices architecture. What are the trade-offs, and when would you choose each?
3. Explain the MVC pattern. How do Model, View, and Controller interact? What are the benefits of this separation?
4. Describe each of the SOLID principles and provide an example of how each improves software design.
5. What is the Single Responsibility Principle? Identify a violation in code you've written or seen, and explain how you would refactor it.
6. Explain the Dependency Inversion Principle. How does dependency injection help implement this principle?
7. Compare the Factory Method and Builder patterns. When would you use each?
8. How does the Strategy pattern differ from a simple if-else chain? What are the benefits of using Strategy?
9. Describe the Observer pattern and give three real-world examples where it would be appropriate.

10. What is the purpose of a Software Architecture Document? Who are its audiences, and what do they need from it?

4.10 Hands-On Exercises

Exercise 4.1: Identifying Architectural Styles

For each of the following systems, identify which architectural style(s) would be most appropriate and explain why:

1. A personal blog website
2. Netflix's streaming service
3. A banking system processing transactions
4. A real-time multiplayer game
5. An IoT system monitoring factory equipment

Exercise 4.2: SOLID Refactoring

The following code violates SOLID principles. Identify which principles are violated and refactor the code:

```java
public class Report {
    private String content;
    private String format;

    public void generateReport(Database db, String query) {
        // Get data from database
        ResultSet data = db.execute(query);

        // Format the report
        if (format.equals("PDF")) {
            content = formatAsPDF(data);
        } else if (format.equals("HTML")) {
            content = formatAsHTML(data);
        } else if (format.equals("CSV")) {
            content = formatAsCSV(data);
        }

        // Save to file
        FileSystem.write("/reports/output", content);

        // Send email notification
        EmailClient.send("admin@company.com", "Report generated", content);
    }
}
```

Exercise 4.3: Implementing Design Patterns

Implement the following:

1. **Factory Pattern**: Create a `ShapeFactory` that creates different shapes (Circle, Rectangle, Triangle) based on input parameters.

2. **Decorator Pattern**: Create a `Notification` system where notifications can be decorated with additional delivery methods (SMS, Email, Slack) stacked together.

3. **Observer Pattern**: Create a weather monitoring system where `WeatherStation` notifies multiple displays (CurrentConditions, Statistics, Forecast) when measurements change.

Exercise 4.4: Architecture Analysis

For your semester project:

1. Identify the primary quality attributes that matter most (e.g., performance, security, maintainability)
2. Choose an architectural style and justify your choice
3. Identify at least two design patterns you will use and explain where and why
4. Document any trade-offs you're making

Exercise 4.5: Software Architecture Document

Create a Software Architecture Document for your semester project, including:

1. Introduction and goals
2. Architectural style and rationale
3. Component/Package diagram showing major components
4. At least one sequence diagram for a key scenario
5. Deployment view (how will the system be deployed?)
6. At least two Architecture Decision Records (ADRs) for major decisions

Exercise 4.6: Pattern Recognition

Identify which design pattern is being used in each of the following code snippets:

Snippet A:

```
public class Logger {
    private static Logger instance;

    private Logger() {}

    public static Logger getInstance() {
        if (instance == null) {
            instance = new Logger();
        }
        return instance;
    }
}
```

Snippet B:

```
public interface SortStrategy {
    void sort(int[] array);
}

public class QuickSort implements SortStrategy { ... }
public class MergeSort implements SortStrategy { ... }

public class Sorter {
    private SortStrategy strategy;

    public void setStrategy(SortStrategy strategy) {
        this.strategy = strategy;
    }

    public void performSort(int[] array) {
        strategy.sort(array);
    }
}
```

Snippet C:

```
public interface Beverage {
    double cost();
}

public class Espresso implements Beverage {
    public double cost() { return 1.99; }
}

public class MilkDecorator implements Beverage {
    private Beverage beverage;
    public MilkDecorator(Beverage beverage) {
        this.beverage = beverage;
    }

    public double cost() {
        return beverage.cost() + 0.50;
    }
}
```

4.11 Further Reading

Books:

- Martin, R. C. (2017). *Clean Architecture: A Craftsman's Guide to Software Structure and Design*. Prentice Hall.
- Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1994). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley.
- Richards, M. & Ford, N. (2020). *Fundamentals of Software Architecture*. O'Reilly Media.
- Newman, S. (2021). *Building Microservices* (2nd Edition). O'Reilly Media.
- Bass, L., Clements, P., & Kazman, R. (2012). *Software Architecture in Practice* (3rd Edition). Addison-Wesley.

Online Resources:

- Refactoring Guru - Design Patterns: https://refactoring.guru/design-patterns
- Martin Fowler's Architecture Guide: https://martinfowler.com/architecture/
- Microsoft Architecture Guides: https://docs.microsoft.com/en-us/azure/architecture/
- ADR GitHub Organization: https://adr.github.io/

References

Bass, L., Clements, P., & Kazman, R. (2012). *Software Architecture in Practice* (3rd Edition). Addison-Wesley.

Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., & Stal, M. (1996). *Pattern-Oriented Software Architecture Volume 1: A System of Patterns*. Wiley.

Fowler, M. (2002). *Patterns of Enterprise Application Architecture*. Addison-Wesley.

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1994). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley.

Kruchten, P. (1995). The 4+1 View Model of Architecture. *IEEE Software*, 12(6), 42-50.

Martin, R. C. (2000). Design Principles and Design Patterns. Retrieved from http://www.objectmentor.com/

Martin, R. C. (2017). *Clean Architecture: A Craftsman's Guide to Software Structure and Design.* Prentice Hall.

Newman, S. (2021). *Building Microservices* (2nd Edition). O'Reilly Media.

Richards, M., & Ford, N. (2020). *Fundamentals of Software Architecture: An Engineering Approach.* O'Reilly Media.

Chapter 5: UI/UX Design and Prototyping

Learning Objectives

By the end of this chapter, you will be able to:

- Explain the principles of human-centered design and their importance in software development
- Distinguish between User Interface (UI) and User Experience (UX) design
- Apply UX heuristics to evaluate and improve interface designs
- Create wireframes and prototypes at varying levels of fidelity
- Design responsive interfaces that work across different devices and screen sizes
- Implement accessibility best practices to create inclusive software
- Develop a UI style guide for consistent design across an application
- Use modern prototyping tools to communicate design ideas

5.1 Understanding UI and UX

You've probably used software that felt frustrating—confusing menus, buttons that didn't look clickable, forms that lost your data, error messages that made no sense. You've also used software that felt effortless—intuitive navigation, clear feedback, tasks accomplished with minimal friction. The difference isn't accidental. It's the result of thoughtful design.

User Experience (UX) and **User Interface (UI)** design are the disciplines that create this difference. While often mentioned together, they address different aspects of how users interact with software.

5.1.1 What Is User Experience (UX)?

User Experience encompasses all aspects of a user's interaction with a product, service, or company. It's about how the product feels to use—whether it's satisfying, frustrating, efficient, or confusing.

UX design asks questions like:

- What problem is the user trying to solve?
- What steps must users take to accomplish their goals?
- How do users feel during and after using the product?
- What obstacles prevent users from succeeding?

- How can we make the experience more efficient and enjoyable?

UX extends beyond the screen. It includes:

- The user's first impression when discovering the product
- The onboarding experience for new users
- The day-to-day experience of regular use
- Error handling and recovery
- Customer support interactions
- The experience of leaving or canceling

UX Design Activities:

- User research and interviews
- Creating user personas
- Journey mapping
- Information architecture
- Interaction design
- Usability testing
- Analyzing user behavior data

5.1.2 What Is User Interface (UI)?

User Interface design focuses on the visual and interactive elements users directly interact with—buttons, icons, typography, colors, layouts, animations, and more.

UI design asks questions like:

- What should users see on this screen?
- How should interactive elements look and behave?
- What visual hierarchy guides users' attention?
- How do colors, fonts, and spacing create the right mood?
- How does the interface communicate its state?

UI design is about making the interface:

- **Visually appealing**: Aesthetically pleasing and aligned with brand identity
- **Clear**: Users understand what they're seeing
- **Consistent**: Similar elements look and behave similarly
- **Responsive**: Interface provides feedback for user actions

UI Design Activities:

- Visual design (colors, typography, imagery)
- Layout and composition
- Icon and button design
- Motion and animation design
- Creating design systems and style guides
- Responsive design for multiple screen sizes

5.1.3 The Relationship Between UX and UI

UX and UI are deeply interconnected:

```
                      User Experience (UX)


                     User Interface (UI)


        Visual         Interactive       Layout
        Design          Elements





     User          Information      Interaction       Usability
    Research      Architecture        Design           Testing
```

A useful analogy: If your product were a house, UX would be the architecture—the floor plan, the flow between rooms, how living in it feels. UI would be the interior design—the paint colors, the furniture choices, the light fixtures.

You can have:

- Good UX with poor UI: The product works well but looks outdated or unappealing
- Good UI with poor UX: The product looks beautiful but is frustrating to use
- Good UX and UI: The product is both effective and delightful (the goal!)

5.1.4 Why Software Engineers Need Design Skills

You might wonder why a software engineering course covers design. Can't designers handle this? In practice:

Many teams don't have dedicated designers. Startups, small companies, and internal tools often rely on developers to make design decisions.

Design affects technical decisions. How you structure your code depends on what the interface needs to do. Understanding design helps you build better systems.

Better communication with designers. Even with dedicated designers, understanding their work improves collaboration.

Design thinking improves problem-solving. The empathy and iteration central to design make you a better engineer overall.

Users don't separate code from design. Users experience the whole product. Poor design undermines excellent code.

5.2 Human-Centered Design

Human-Centered Design (HCD) is an approach that grounds the design process in information about the people who will use the product. Rather than designing based on assumptions or technical constraints, HCD starts with understanding users' needs, behaviors, and contexts.

5.2.1 Principles of Human-Centered Design

1. Focus on People

Design begins with understanding users—not technology, not business requirements, but the humans who will interact with the system.

This means:

- Observing users in their natural environment
- Understanding their goals, frustrations, and contexts
- Recognizing that users are experts in their own needs
- Designing for real people, not idealized users

2. Find the Right Problem

Many failed products solve the wrong problem beautifully. HCD invests in problem definition before jumping to solutions.

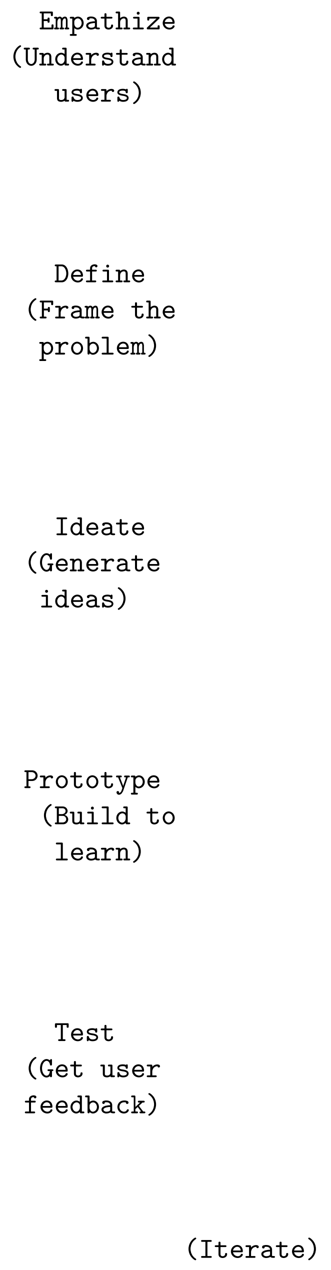
> "If I had an hour to solve a problem, I'd spend 55 minutes thinking about the problem and 5 minutes thinking about solutions." — attributed to Albert Einstein

3. Think Systemically

Problems exist within systems. A solution that fixes one issue might create others. HCD considers the broader context and ripple effects of design decisions.

4. Iterate Relentlessly

Perfect solutions rarely emerge fully formed. HCD embraces iteration—designing, testing, learning, and refining in cycles.


Empathize (Understand users)
Define (Frame the problem)
Ideate (Generate ideas)
Prototype (Build to learn)
Test (Get user feedback)
(Iterate)

5. Prototype to Learn

Prototypes aren't mini-products; they're thinking tools. Build prototypes to answer questions and test assumptions, not to show off solutions.

5.2.2 Understanding Users

Before designing, you need to understand who you're designing for. Several techniques help build this understanding:

User Research Methods:

Method	Description	When to Use
Interviews	One-on-one conversations	Deep understanding of needs and motivations
Surveys	Questionnaires to many users	Quantitative data, validating hypotheses
Observation	Watching users in context	Understanding actual behavior
Usability Testing	Users attempt tasks	Evaluating existing designs
Analytics	Behavioral data from usage	Understanding patterns at scale
Card Sorting	Users organize information	Designing information architecture

User Personas

A **persona** is a fictional character representing a user type. Personas help teams maintain empathy for users throughout the project.

Example Persona:

```
PERSONA: Sarah Chen


Demographics:
• 34 years old
• Marketing Manager at mid-size company
• Lives in suburban Chicago
• Uses iPhone, MacBook, and iPad

Goals:
• Coordinate marketing campaigns across her team of 6
• Track project progress without micromanaging
• Meet deadlines reliably
• Reduce time spent in status meetings

Frustrations:
• Information scattered across email, Slack, and spreadsheets
• Surprises about project delays discovered too late
• Team members working on outdated versions of documents
• Too many tools that don't integrate well

Tech Comfort: Moderate
• Comfortable with common apps (Office, Google Workspace)
• Prefers intuitive tools over powerful-but-complex ones
• Willing to learn new tools if the benefit is clear
```

```
  Quote: "I just want to know what's happening without having to ask."
```

User Journey Maps

A **journey map** visualizes the user's experience over time, including their actions, thoughts, emotions, and pain points.

```
USER JOURNEY: New Customer Making First Purchase


Stage:     DISCOVER    →    EVALUATE    →    PURCHASE    →    RECEIVE    →    USE

Actions:   • Sees ad    • Browses       • Creates       • Tracks       • Opens
           • Visits       products        account         shipping       package
             site       • Reads         • Enters        • Waits        • Uses
           • Browses      reviews         payment                        product
                        • Compares

Thinking:  "This looks  "Will this     "Is this       "When will    "Is this what
           interesting"  work for me?"   secure?"       it arrive?"   I expected?"

Feeling:    Curious     Uncertain     Anxious       Impatient    or

Pain       • Slow       • Not enough  • Long         • No          • Instructions
Points:      loading      reviews       checkout       tracking      unclear
           • Cluttered  • Hard to     • Account      • Delayed     • Missing
             homepage     compare       required       delivery      pieces

Opportu-   • Fast,      • Rich        • Guest        • Real-time   • Quick
nities:      clean        product       checkout       updates       start
             first        info          option       • Proactive     guide
             impression • Easy          available      communication
                          comparison
```

5.2.3 Defining the Problem

Good problem definition is half the solution. A well-framed problem focuses the team and prevents wasted effort on the wrong issues.

Problem Statement Format:

> [User] needs a way to [accomplish goal] because [insight from research], but [current obstacle].

Example:

> Marketing managers need a way to **track their team's project progress in real-time** because **they're accountable for deadlines they can't directly control**, but **current tools require manually asking for updates**, which is time-consuming and often provides outdated information.

"How Might We" Questions:

Transform problem statements into opportunity questions:

- How might we help managers see project status without asking?
- How might we surface delays before they become crises?
- How might we reduce the friction of status updates for team members?

These questions open possibilities without constraining solutions.

5.3 UX Design Heuristics

Heuristics are rules of thumb—general principles that guide design decisions without prescribing specific solutions. Jakob Nielsen's 10 Usability Heuristics, developed in the 1990s, remain the most influential framework for evaluating interface design.

5.3.1 Nielsen's 10 Usability Heuristics

1. Visibility of System Status

The system should always keep users informed about what is going on through appropriate feedback within a reasonable time.

Good Examples:

- Progress bars during file uploads
- "Saving…" indicator in document editors
- Real-time character count in text fields with limits
- Loading spinners during data fetches

Bad Examples:

- Submitting a form with no feedback
- Processes running with no indication of progress
- Buttons that don't respond to clicks

Implementation:

```
Uploading document...

              67%

2 of 3 files uploaded
Estimated time remaining: 12 seconds

[Cancel]
```

2. Match Between System and the Real World

The system should speak the user's language, using words, phrases, and concepts familiar to the user rather than system-oriented terms. Follow real-world conventions, making information appear in a natural and logical order.

Good Examples:

- Shopping cart icon for e-commerce
- "Trash" or "Recycle Bin" for deleted items
- Calendar interfaces that look like calendars
- Using "Save" instead of "Persist to Database"

Bad Examples:

- Technical jargon in user interfaces ("SQLException occurred")
- Arbitrary icons without clear meaning
- Navigation that doesn't match user mental models

Tip: Use the same terminology users use when describing their tasks. If users say "customers," don't call them "accounts" in the interface.

3. User Control and Freedom

Users often choose system functions by mistake and need a clearly marked "emergency exit" to leave the unwanted state without going through an extended process.

Good Examples:

- Undo/Redo functionality (Ctrl+Z, Ctrl+Y)
- "Cancel" buttons on forms and dialogs
- "Go back" option in wizards
- Gmail's "Undo send" feature
- Clear navigation to return to previous states

Bad Examples:

- No way to cancel a long-running operation
- Destructive actions without confirmation
- Wizards with no back button
- Modal dialogs without close buttons

4. Consistency and Standards

Users should not have to wonder whether different words, situations, or actions mean the same thing. Follow platform conventions.

Types of Consistency:

- **Internal consistency**: The same action works the same way throughout your application
- **External consistency**: Your application follows conventions users know from other applications
- **Visual consistency**: Similar elements look similar

Good Examples:

- Blue underlined text for links
- "X" in top corner to close dialogs
- Ctrl+S to save (on Windows)
- Swipe to delete (on mobile)

Bad Examples:

- Different button styles for the same type of action
- Non-standard icons for common functions
- Inconsistent placement of navigation elements

5. Error Prevention

Even better than good error messages is a careful design that prevents problems from occurring in the first place.

Types of Error Prevention:

- **Constraints**: Prevent invalid input (date pickers, dropdowns)
- **Suggestions**: Autocomplete reduces typos
- **Defaults**: Sensible defaults reduce decisions
- **Confirmations**: Confirm destructive actions

Good Examples:

```
Delete project "Annual Report 2024"?

This will permanently delete:
• 47 tasks
• 12 documents
• 156 comments

This action cannot be undone.

Type "Annual Report 2024" to confirm:
        [Cancel]    [Delete Project]
```

Bad Examples:

- Free-form date entry instead of date pickers
- Delete buttons without confirmation
- Allowing form submission with known invalid data

6. Recognition Rather Than Recall

Minimize user memory load by making objects, actions, and options visible. Users should not have to remember information from one part of the interface to another.

Good Examples:

- Dropdown menus showing all options
- Recent files and search history
- Autocomplete showing previous entries
- Icons with labels (not icons alone)
- Showing examples of expected input formats

Bad Examples:

- Requiring users to remember codes or IDs
- Icons without labels
- Empty forms without hints about expected format
- Requiring memorization of keyboard shortcuts

7. Flexibility and Efficiency of Use

Accelerators—invisible to novice users—may speed up interaction for expert users. Allow users to tailor frequent actions.

Good Examples:

- Keyboard shortcuts for common actions
- Customizable toolbars and dashboards
- "Recent" and "Favorites" lists
- Bulk operations for power users
- Templates for common tasks

Implementation Example:

```
  File    Edit    View    Help

  New           Ctrl+N
  Open          Ctrl+O
```

```
Save          Ctrl+S
Save As...    Ctrl+Shift+S

Recent Files      report-final.docx
                    presentation-v2.pptx
                    budget-2024.xlsx
```

8. Aesthetic and Minimalist Design

Interfaces should not contain information that is irrelevant or rarely needed. Every extra unit of information competes with relevant information and diminishes their relative visibility.

Principles:

- Remove unnecessary elements
- Prioritize important information visually
- Use progressive disclosure (show more on demand)
- White space is not wasted space

Good Examples:

- Google's search homepage (minimal)
- Progressive disclosure in settings (basic → advanced)
- Contextual toolbars that show relevant tools

Bad Examples:

- Cluttered dashboards showing everything at once
- Dense forms with rarely-used fields always visible
- Excessive decorative elements

9. Help Users Recognize, Diagnose, and Recover from Errors

Error messages should be expressed in plain language (no codes), precisely indicate the problem, and constructively suggest a solution.

Good Error Message:

```
     Couldn't save your changes

Your internet connection was lost.
Your changes have been saved locally
and will sync when you're back online.

[Retry Now]    [Work Offline]
```

Bad Error Messages:

- "Error 500: Internal Server Error"
- "An error occurred"
- "Invalid input"
- "Operation failed. Contact administrator."

Error Message Checklist:

☒ Says what went wrong (specifically)
☒ Uses plain language (no technical jargon)
☒ Suggests how to fix it
☒ Offers an action the user can take

10. Help and Documentation

Even though it's better if the system can be used without documentation, it may be necessary to provide help and documentation. Such information should be easy to search, focused on the user's task, list concrete steps, and not be too large.

Good Examples:

- Contextual help (? icons next to complex fields)
- Searchable help documentation
- Interactive tutorials and onboarding
- Tooltips explaining interface elements

Help Types:

```
API Rate Limit  [?]

  1000



   Maximum API requests per hour per user.
    Higher limits may affect server performance.
    Learn more about rate limiting →
```

5.3.2 Applying Heuristics: Heuristic Evaluation

A **heuristic evaluation** is a usability inspection method where evaluators examine an interface and judge its compliance with recognized usability principles.

How to Conduct a Heuristic Evaluation:

1. **Prepare**: Gather the interface (or prototype) and the heuristic checklist

2. **Evaluate individually**: Each evaluator reviews the interface alone, noting violations
3. **Rate severity**: Assign severity ratings to each issue

Severity Rating Scale:

- 0 = Not a usability problem
- 1 = Cosmetic problem; fix if time permits
- 2 = Minor problem; low priority
- 3 = Major problem; important to fix
- 4 = Catastrophic; must fix before release

4. **Aggregate**: Combine findings from all evaluators
5. **Prioritize**: Address issues by severity and frequency

Heuristic Evaluation Template:

#	Issue Description	Heuristic Violated	Severity	Recommendation
1	No feedback after form submission	#1 Visibility of System Status	3	Add success/error message
2	Technical error messages shown to users	#9 Error Recovery	3	Translate to plain language
3	Delete button has no confirmation	#5 Error Prevention	4	Add confirmation dialog
4	Icons without labels	#6 Recognition vs Recall	2	Add tooltips or labels

---

5.4 Wireframing and Prototyping

Design is an iterative process, and prototypes are the tools that enable iteration. They let you explore ideas, communicate concepts, and test assumptions—all before investing in full implementation.

5.4.1 The Prototyping Spectrum

Prototypes exist on a spectrum from low-fidelity sketches to high-fidelity interactive simulations:

Low Fidelity High Fidelity

Sketches	Wireframes	Mockups	Prototypes
Paper/ whiteboard rough ideas	Digital, grayscale, layout focus	Visual design with colors/fonts	Interactive, clickable, simulates behavior
Minutes	Hours	Hours-Days	Days-Weeks
Explore concepts	Define structure	Refine aesthetics	Validate interactions

Low-fidelity prototypes are quick and cheap. They're good for exploring many ideas and getting early feedback without emotional attachment.

High-fidelity prototypes look and feel like real products. They're better for testing detailed interactions and getting reactions to visual design.

5.4.2 Sketching

Sketching is the fastest way to explore ideas. Don't worry about artistic ability—rough sketches communicate ideas effectively.

Why Sketch?

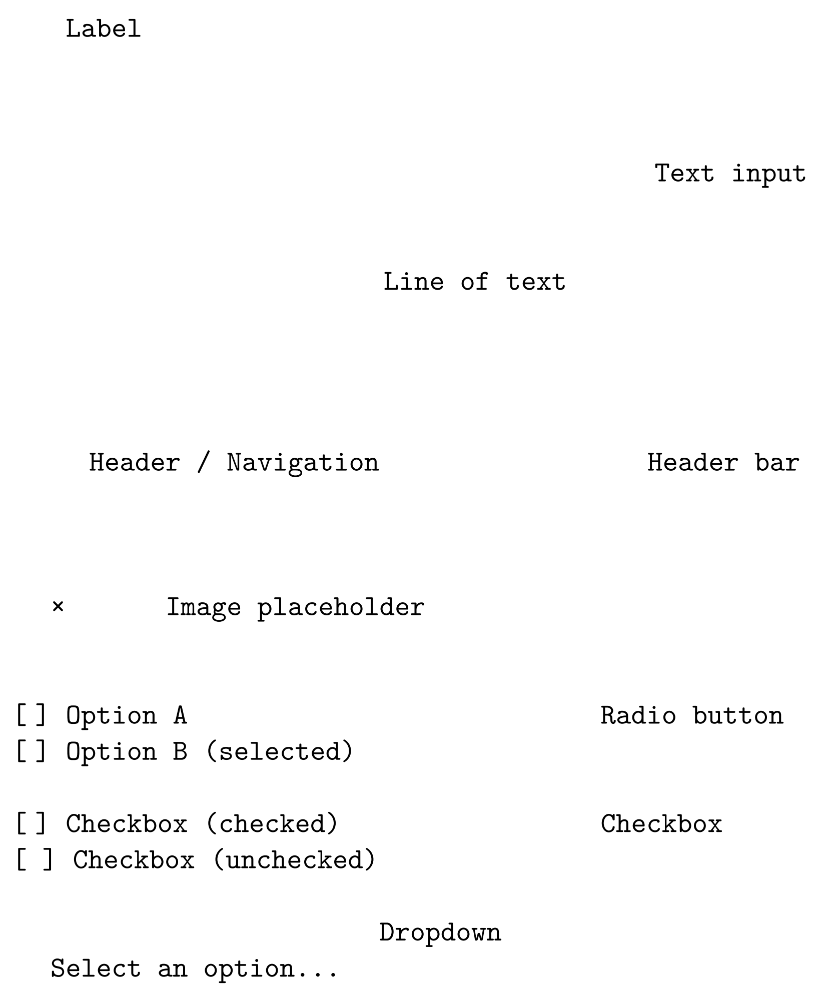
- Extremely fast (seconds to minutes)
- No software needed
- Easy to iterate—just flip to a new page
- Encourages divergent thinking
- No emotional attachment to rough sketches

Sketching Tips:

- Use pen (not pencil) to avoid erasing—just draw again
- Draw multiple variations quickly
- Annotate with notes explaining behavior
- Use standard UI conventions (boxes for buttons, lines for text)
- Include arrows and notes for interactions

Basic UI Sketching Vocabulary:

Button



5.4.3 Wireframes

Wireframes are low-fidelity representations of a user interface that show structure, layout, and content hierarchy without visual design details.

Wireframe Characteristics:

- Grayscale (no colors)
- Placeholder content (Lorem ipsum, gray boxes for images)
- Focus on layout and information hierarchy
- Basic UI elements without styling
- Annotations explaining functionality

Purpose of Wireframes:

- Define content and structure
- Establish layout and spacing

- Plan navigation and user flows
- Communicate functionality to stakeholders
- Serve as blueprint for visual design

Wireframe Example - Dashboard:

```
                                                    [?] [ ] [  User  ]
   Logo     Dashboard    Projects    Team    Settings


  Welcome back, Sarah!                                   [+ New Project]


     Active Projects        Tasks Due           Team Activity

         12                     5                    23
                              Today               Updates


  Recent Projects                                         [View All →]

     [×] Website Redesign          Progress:        75%
         Due: Oct 15    Team: 4    Status: On Track

     [×] Mobile App v2             Progress:        40%
         Due: Nov 30    Team: 6    Status: At Risk

     [×] Q4 Marketing Campaign     Progress:        20%
         Due: Dec 1     Team: 3    Status: On Track


  My Tasks                                                [View All →]

     [ ] Review design mockups              Due: Today
     [ ] Approve budget proposal            Due: Today
     [ ] Team standup meeting               Due: Tomorrow
     [ ] Review Q3 report                   Due: Oct 10
```

Wireframe Example - Mobile Login:

```
   Back
```

```
         Logo


     Welcome Back
   Sign in to continue

   Email

     sarah@example.com


   Password

     ••••••••••     []


   [ ] Remember me


         Sign In


     Forgot password?



   Don't have an account?
        Sign Up
```

5.4.4 Interactive Prototypes

Interactive prototypes add behavior to wireframes or mockups. Users can click, tap, and navigate as if using a real application.

Levels of Interactivity:

Click-through prototypes: Pages linked together. Clicking a button navigates to another page. Good for testing navigation and flow.

Interactive prototypes: Include form interactions, animations, conditional logic. Good for testing detailed interactions.

Functional prototypes: Working code (often simplified). Real data, real logic, limited scope. Good for technical validation.

What to Test with Prototypes:

- **Navigation**: Can users find their way around?
- **Comprehension**: Do users understand what they're seeing?
- **Task completion**: Can users accomplish specific tasks?
- **Expectations**: Does the design match user mental models?
- **Desirability**: Do users like the design?

5.4.5 Prototyping Tools

Modern tools make creating prototypes faster than ever:

Low-Fidelity / Wireframing:

- **Balsamiq**: Intentionally sketch-like wireframes
- **Whimsical**: Flowcharts and wireframes
- **Excalidraw**: Hand-drawn style diagrams (free)
- **Paper and pen**: Still valid!

High-Fidelity / Design:

- **Figma**: Industry standard, collaborative, free tier (highly recommended)
- **Sketch**: Mac-only, popular with designers
- **Adobe XD**: Part of Adobe ecosystem
- **Framer**: Design with code-like interactions

Code-Based Prototyping:

- **HTML/CSS**: For web interfaces
- **React with Storybook**: Component-based prototyping
- **SwiftUI Previews**: iOS prototyping

Choosing a Tool:

Need	Recommended Tool
Quick exploration	Paper, Whimsical
Shareable wireframes	Figma, Balsamiq
High-fidelity mockups	Figma, Sketch
Complex interactions	Figma, Framer
Team collaboration	Figma
Developer handoff	Figma

For this course, **Figma** is recommended because it's free, web-based, collaborative, and industry-standard.

5.5 Visual Design Fundamentals

While this isn't a graphic design course, understanding visual design principles helps you create more effective interfaces—and communicate better with designers.

5.5.1 Visual Hierarchy

Visual hierarchy guides users' attention to the most important elements first. It's achieved through size, color, contrast, position, and spacing.

Hierarchy Techniques:

Size: Larger elements attract attention first

```
BIG HEADLINE                        ← Eye goes here first
Smaller subheading                  ← Then here
Body text that provides more        ← Then here
detail about the content...
```

Color and Contrast: High-contrast elements stand out

```
      Primary Action              ← High contrast button


     Secondary Action             ← Lower contrast

       Tertiary link               ← Lowest emphasis
```

Position: Top-left (in LTR languages) gets attention first; center draws focus

White Space: Isolated elements with surrounding space appear more important

5.5.2 Typography

Typography significantly impacts readability and tone.

Font Categories:

- **Serif** (Times, Georgia): Traditional, formal, good for body text in print
- **Sans-serif** (Arial, Helvetica, Inter): Modern, clean, good for screens
- **Monospace** (Courier, Fira Code): Technical, code, data
- **Display** (decorative fonts): Headlines only, use sparingly

Typography Best Practices:

- **Limit font families**: 1-2 per project
- **Establish hierarchy**: Different sizes for headings, subheadings, body
- **Line length**: 50-75 characters per line for readability
- **Line height**: 1.4-1.6 for body text
- **Contrast**: Ensure sufficient contrast against background

Type Scale Example:

```
H1: 32px / Bold      "Page Title"
H2: 24px / Semibold  "Section Heading"
H3: 20px / Semibold  "Subsection"
Body: 16px / Regular "Paragraph text that users will read..."
Small: 14px / Regular "Helper text, captions"
Tiny: 12px / Regular "Legal text, timestamps"
```

5.5.3 Color

Color creates mood, guides attention, and communicates meaning.

Color Purposes in UI:

- **Primary color**: Brand identity, primary actions
- **Secondary color**: Supporting elements
- **Neutral colors**: Text, backgrounds, borders
- **Semantic colors**: Success (green), warning (yellow), error (red), info (blue)

Color Accessibility:

- Don't rely on color alone to convey meaning
- Ensure sufficient contrast ratios (WCAG guidelines)
- Test with color blindness simulators

Contrast Ratios (WCAG 2.1):

- Normal text: 4.5:1 minimum
- Large text (18px+ or 14px+ bold): 3:1 minimum
- UI components: 3:1 minimum

Simple Color Palette:

```
PRIMARY

  50      100      500      700      900
 (light)           (main)            (dark)


NEUTRAL

 White   Gray     Gray     Gray     Gray     Black
          100      300      500      700


SEMANTIC

 Success   Warning   Error     Info
 (green)   (yellow)  (red)    (blue)
```

5.5.4 Layout and Spacing

Consistent spacing creates visual rhythm and makes interfaces feel polished.

Spacing Systems:

Use a consistent scale (e.g., multiples of 4px or 8px):

- 4px, 8px, 12px, 16px, 24px, 32px, 48px, 64px

Grid Systems:

Grids provide structure for layout:

- **Column grids**: Common for web (12-column is standard)
- **Modular grids**: Rows and columns for complex layouts
- **Baseline grids**: Align text across columns

Layout Principles:

Alignment: Elements should align with each other

```
Good:                           Bad:

 Label                         Label
```

```
   Input                         Input


 Label                         Label

   Input                       Input
```

Proximity: Related elements should be closer together

```
  Billing Address
                      ← These belong together
    Street


    City       State    Zip

                                     ← Gap separates sections
  Shipping Address
                      ← New group
    Street
```

5.6 Responsive Design

Responsive design creates interfaces that adapt to different screen sizes and devices. Rather than building separate versions for desktop, tablet, and mobile, responsive design uses flexible layouts that reflow and resize.

5.6.1 Why Responsive Design Matters

Users access the web from many devices:

- Desktop computers (various screen sizes)
- Laptops
- Tablets (portrait and landscape)
- Smartphones (various sizes)
- Smart TVs
- Wearables

Building separate versions for each is impractical. Responsive design handles this variety with a single codebase.

5.6.2 Responsive Design Principles

1. Fluid Layouts

Use percentages and flexible units instead of fixed pixels:

```css
/* Fixed - doesn't adapt */
.container {
    width: 960px;
}

/* Fluid - adapts to screen */
.container {
    width: 90%;
    max-width: 1200px;
}
```

2. Flexible Images

Images should scale within their containers:

```css
img {
    max-width: 100%;
    height: auto;
}
```

3. Media Queries

Apply different styles based on screen characteristics:

```css
/* Base styles (mobile-first) */
.sidebar {
    width: 100%;
}

/* Tablet and up */
@media (min-width: 768px) {
    .sidebar {
        width: 30%;
        float: left;
    }
}

/* Desktop and up */
@media (min-width: 1024px) {
    .sidebar {
        width: 25%;
    }
}
```

5.6.3 Breakpoints

Breakpoints are the screen widths where layout changes occur. Common breakpoints:

Breakpoint	Target Devices
< 576px	Small phones
576px - 768px	Large phones, small tablets
768px - 1024px	Tablets
1024px - 1200px	Small desktops, laptops
> 1200px	Large desktops

Mobile-First vs. Desktop-First:

Mobile-first: Start with mobile styles, add complexity for larger screens

```css
/* Mobile styles (default) */
.nav { display: none; }

/* Larger screens */
@media (min-width: 768px) {
    .nav { display: flex; }
}
```

Desktop-first: Start with desktop styles, simplify for smaller screens

```css
/* Desktop styles (default) */
.nav { display: flex; }

/* Smaller screens */
@media (max-width: 767px) {
    .nav { display: none; }
}
```

Mobile-first is generally recommended because:

- Forces prioritization of essential content
- Progressive enhancement (add features vs. remove them)
- Mobile usage continues to grow

5.6.4 Responsive Patterns

Column Drop:

Multi-column layout stacks into single column on small screens:

```
Desktop:                    Mobile:

    A       B        C             A

                                   B

                                     C
```

Layout Shifter:

Layout reorganizes more dramatically across breakpoints:

```
Desktop:                    Mobile:

          Header                 Header


  Nav         Content          Content


                      Nav
```

Off-Canvas:

Navigation hidden off-screen on mobile, slides in when activated:

```
Desktop:                    Mobile (menu closed):    Mobile (menu open):

 Logo  Nav Nav Nav  User      Logo    User          Nav    Content
                                        (dimmed)
                                                     Nav
        Content                 Content
                                                     Nav
```

5.6.5 Touch Considerations

Mobile interfaces require touch-friendly design:

Touch Target Sizes:

- Minimum 44x44 points (Apple) or 48x48dp (Google)
- Adequate spacing between targets
- Important actions should have larger targets

Touch Gestures:

- Tap: Primary interaction
- Swipe: Navigate, delete, reveal actions
- Pinch: Zoom
- Long press: Context menus, selection

Mobile-Specific Patterns:

- Bottom navigation (thumb-friendly)
- Pull to refresh
- Swipe actions on list items
- Floating action buttons

5.7 Accessibility

Accessibility means designing products that can be used by people with disabilities. This includes users who are blind or have low vision, deaf or hard of hearing, have motor impairments, or have cognitive disabilities.

5.7.1 Why Accessibility Matters

It's the right thing to do. Everyone deserves access to digital services.

It's often legally required. Many jurisdictions mandate accessibility (ADA in the US, EN 301 549 in EU).

It improves usability for everyone. Accessibility features like clear language, keyboard navigation, and good contrast benefit all users.

It expands your audience. Over 1 billion people worldwide have disabilities.

5.7.2 WCAG Guidelines

The **Web Content Accessibility Guidelines (WCAG)** are the international standard for web accessibility. WCAG 2.1 organizes guidelines under four principles (POUR):

Perceivable: Information must be presentable in ways users can perceive

- Text alternatives for images
- Captions for video
- Sufficient color contrast
- Content adapts to different presentations

Operable: Interface components must be operable

- Keyboard accessible
- Enough time to read and use content

- No seizure-inducing content
- Navigable (clear structure, findable content)

Understandable: Information and operation must be understandable

- Readable text
- Predictable behavior
- Help users avoid and correct mistakes

Robust: Content must be robust enough for diverse user agents

- Compatible with assistive technologies
- Valid, well-structured code

Conformance Levels:

- **Level A**: Minimum accessibility (must meet)
- **Level AA**: Standard target for most sites (should meet)
- **Level AAA**: Highest accessibility (enhanced, not always possible)

5.7.3 Common Accessibility Requirements

1. Alternative Text for Images

Screen readers can't see images; they read alt text instead.

```html
<!-- Informative image -->
<img src="chart.png" alt="Sales increased 25% from Q1 to Q2">

<!-- Decorative image (empty alt) -->
<img src="decorative-line.png" alt="">

<!-- Complex image (link to description) -->
<img src="complex-diagram.png" alt="System architecture diagram"
     aria-describedby="diagram-description">
<div id="diagram-description">
    Detailed description of the diagram...
</div>
```

2. Keyboard Navigation

Everything should be operable with keyboard alone:

- Tab: Move between interactive elements
- Enter/Space: Activate buttons/links
- Arrow keys: Navigate within components
- Escape: Close dialogs/menus

```html
<!-- Good: Native button is keyboard accessible -->
<button onclick="submit()">Submit</button>

<!-- Bad: Div requires extra work for accessibility -->
<div onclick="submit()">Submit</div>

<!-- If you must use div, add keyboard support -->
<div role="button" tabindex="0"
     onclick="submit()"
     onkeydown="if(event.key==='Enter') submit()">
    Submit
</div>
```

3. Focus Indicators

Users must see which element has keyboard focus:

```css
/* Don't remove focus outlines without replacement */
/* Bad: */
:focus { outline: none; }

/* Good: Custom focus style */
:focus {
    outline: 2px solid #0066cc;
    outline-offset: 2px;
}

/* Or use focus-visible for mouse/keyboard distinction */
:focus-visible {
    outline: 2px solid #0066cc;
}
```

4. Color Contrast

Text must have sufficient contrast against background:

```
Good Contrast:                Poor Contrast:


   Dark text on light            Light gray on white
   background (7:1)              background (1.5:1)
```

Tools to check contrast:

- WebAIM Contrast Checker

- Figma accessibility plugins
- Browser developer tools

5. Form Labels

Every form input needs an associated label:

```html
<!-- Good: Label explicitly associated -->
<label for="email">Email address</label>
<input type="email" id="email" name="email">

<!-- Good: Label wraps input -->
<label>
    Email address
    <input type="email" name="email">
</label>

<!-- Bad: No label association -->
<span>Email address</span>
<input type="email" name="email">
```

6. Semantic HTML

Use HTML elements for their intended purpose:

```html
<!-- Good: Semantic structure -->
<header>
    <nav>
        <ul>
            <li><a href="/">Home</a></li>
            <li><a href="/about">About</a></li>
        </ul>
    </nav>
</header>
<main>
    <article>
        <h1>Article Title</h1>
        <p>Content...</p>
    </article>
</main>
<footer>
    <p>Copyright 2024</p>
</footer>

<!-- Bad: Divs for everything -->
<div class="header">
    <div class="nav">
        <div class="nav-item">Home</div>
        <div class="nav-item">About</div>
    </div>
</div>
<div class="content">
    <div class="title">Article Title</div>
    <div>Content...</div>
</div>
```

7. ARIA When Needed

ARIA (Accessible Rich Internet Applications) adds semantic information when HTML alone isn't sufficient:

```
<!-- Tab interface -->
<div role="tablist">
    <button role="tab" aria-selected="true" aria-controls="panel1">
        Tab 1
    </button>
    <button role="tab" aria-selected="false" aria-controls="panel2">
        Tab 2
    </button>
</div>
<div role="tabpanel" id="panel1">Content 1</div>
<div role="tabpanel" id="panel2" hidden>Content 2</div>

<!-- Live region for dynamic updates -->
<div aria-live="polite" aria-atomic="true">
    <!-- Screen reader announces when this content changes -->
    Your cart has been updated
</div>
```

ARIA Rules:

1. Don't use ARIA if HTML can do it
2. Don't change native semantics (unless necessary)
3. All interactive ARIA controls must be keyboard accessible
4. Don't use `role="presentation"` or `aria-hidden="true"` on focusable elements
5. All interactive elements must have an accessible name

5.7.4 Accessibility Testing

Automated Testing:

- axe DevTools (browser extension)
- WAVE (web accessibility evaluator)
- Lighthouse (built into Chrome)
- ESLint accessibility plugins

Manual Testing:

- Navigate with keyboard only
- Use a screen reader (VoiceOver, NVDA, JAWS)
- Zoom to 200%
- Test with high contrast mode
- Verify focus order makes sense

Accessibility Checklist:

- ☐ All images have appropriate alt text
- ☐ Color contrast meets WCAG AA (4.5:1 for text)
- ☐ All functionality available via keyboard
- ☐ Focus indicator is visible
- ☐ Form inputs have labels
- ☐ Errors are clearly described
- ☐ Page has proper heading structure
- ☐ Links have descriptive text
- ☐ Dynamic content is announced to screen readers
- ☐ No content flashes more than 3 times per second

5.8 Design Systems and Style Guides

A **design system** is a collection of reusable components, guided by clear standards, that can be assembled to build any number of applications. A **style guide** documents these standards.

5.8.1 Why Design Systems Matter

Consistency: Users learn patterns once and apply them everywhere.

Efficiency: Don't redesign buttons for every project. Reuse proven solutions.

Quality: Components are refined over time, incorporating accessibility and usability improvements.

Scalability: Large teams can work independently while maintaining consistency.

Communication: Shared vocabulary between designers and developers.

5.8.2 Components of a Design System

Design Tokens:

The smallest elements—colors, typography, spacing, shadows:

```
COLORS

primary-50:    #E3F2FD
primary-100:   #BBDEFB
primary-500:   #2196F3  (main)
primary-700:   #1976D2
primary-900:   #0D47A1

TYPOGRAPHY

font-family-primary:    "Inter", sans-serif
font-family-mono:       "Fira Code", monospace

font-size-xs:     12px
font-size-sm:     14px
font-size-base:   16px
font-size-lg:     18px
font-size-xl:     20px
font-size-2xl:    24px
font-size-3xl:    32px

SPACING

space-1:    4px
space-2:    8px
space-3:    12px
space-4:    16px
space-5:    20px
space-6:    24px
space-8:    32px
space-10:   40px
space-12:   48px
space-16:   64px

SHADOWS

shadow-sm:    0 1px 2px rgba(0,0,0,0.05)
shadow-md:    0 4px 6px rgba(0,0,0,0.1)
shadow-lg:    0 10px 15px rgba(0,0,0,0.1)

BORDER RADIUS
```

```
radius-sm:    4px
radius-md:    8px
radius-lg:    16px
radius-full:  9999px
```

UI Components:

Reusable building blocks:

```
BUTTONS


Primary Button

    Primary Action      Background: primary-500
              Text: white
                         Padding: space-3 space-6
                         Border-radius: radius-md

Secondary Button

   Secondary Action     Background: transparent
              Border: 1px solid primary-500
                         Text: primary-500

Destructive Button

       Delete           Background: error-500
              Text: white

Button States:
- Default
- Hover (darken background 10%)
- Active (darken background 20%)
- Focused (add focus ring)
- Disabled (50% opacity, no pointer events)

FORM INPUTS


Text Input

 Placeholder text

Border: 1px solid gray-300
Border-radius: radius-md
Padding: space-3 space-4
```

```
States:
- Default: gray-300 border
- Focused: primary-500 border + shadow
- Error: error-500 border
- Disabled: gray-100 background

CARDS



   Card Title


   Card content goes here with
   supporting text and information.

   [Secondary]  [Primary Action]

Background: white
Border-radius: radius-lg
Shadow: shadow-md
Padding: space-6
```

Patterns:

Common UI patterns built from components:

```
NAVIGATION PATTERN


Top Navigation (Desktop)

  [Logo]    Nav Item    Nav Item    Nav Item        [Avatar  ]


Mobile Navigation

  [ ]              [Logo]        [ ]


FORM PATTERN


Standard Form Layout

   Form Title
   Subtitle or instructions
```

```
Label

  Input

Helper text

Label

  Input


           [Cancel]  [Submit]
```

5.8.3 Creating a Style Guide

A style guide documents your design system. For your project, include:

1. Introduction

- Purpose of the style guide
- How to use it
- Where to find resources (Figma files, code repositories)

2. Design Principles

- Core values guiding design decisions
- Example: "Clarity over decoration," "Accessibility first"

3. Brand

- Logo usage
- Voice and tone

4. Visual Foundation

- Color palette (with accessibility notes)
- Typography scale
- Spacing system
- Iconography
- Imagery guidelines

5. Components

- Each component with:
  - Visual examples (all states)
  - Usage guidelines (when to use/not use)
  - Specifications (sizes, colors, spacing)
  - Code examples (if technical)

  - Accessibility requirements

6. **Patterns**

- Common UI patterns
- Page layouts
- Navigation patterns
- Form patterns

5.8.4 Style Guide Example

Here's a condensed style guide structure:

```
# TaskFlow Style Guide

## 1. Design Principles

1. **Clarity First**: Every element should have a clear purpose
2. **Respectful of Time**: Minimize steps, reduce friction
3. **Accessible to All**: WCAG AA compliance minimum
4. **Consistent Experience**: Same patterns throughout

## 2. Colors

### Primary Palette
| Name         | Hex      | Usage                      |
|-------------|---------|---------------------------|
| Primary      | #2563EB  | Interactive elements       |
| Primary Dark| #1D4ED8  | Hover states               |
| Primary Light| #DBEAFE| Backgrounds, highlights    |

### Semantic Colors
| Name    | Hex      | Usage             |
|--------|---------|------------------|
| Success | #10B981 | Success states    |
| Warning | #F59E0B | Warning states    |
| Error   | #EF4444 | Error states      |
| Info    | #3B82F6 | Info states       |

### Neutrals
| Name       | Hex      | Usage              |
|-----------|---------|-------------------|
| Gray 900   | #111827 | Primary text       |
| Gray 600   | #4B5563 | Secondary text     |
| Gray 400   | #9CA3AF | Disabled, hints    |
| Gray 200   | #E5E7EB | Borders            |
| Gray 50    | #F9FAFB | Backgrounds        |
## 3. Typography

**Font Family**: Inter (sans-serif)

| Style    | Size | Weight | Line Height | Usage        |
|----------|------|--------|-------------|--------------|
| H1       | 32px | 700    | 1.2         | Page titles  |
| H2       | 24px | 600    | 1.3         | Section heads|
| H3       | 20px | 600    | 1.4         | Subsections  |
| Body     | 16px | 400    | 1.5         | Body text    |
| Small    | 14px | 400    | 1.5         | Helper text  |
| Caption  | 12px | 400    | 1.4         | Labels       |

## 4. Spacing

Base unit: 4px

| Token    | Value | Usage                     |
|----------|-------|---------------------------|
| xs       | 4px   | Tight spacing             |
| sm       | 8px   | Related elements          |
| md       | 16px  | Default spacing           |
| lg       | 24px  | Section spacing           |
| xl       | 32px  | Large gaps                |
| 2xl      | 48px  | Page sections             |

## 5. Components

### Buttons

**Primary Button**
- Background: Primary (#2563EB)
- Text: White
- Padding: 12px 24px
- Border-radius: 6px
- States: Hover (Primary Dark), Disabled (50% opacity)

**Usage**: Use for primary actions. One per section maximum.

**Accessibility**:
- Minimum touch target 44x44px
- Focus ring: 2px offset, Primary color

[Continue for each component...]
```

5.9 Prototyping in Practice: Using Figma

Figma is the industry-standard tool for UI design and prototyping. This section provides a practical introduction.

5.9.1 Figma Basics

Key Concepts:

- **Canvas**: Infinite workspace where you design
- **Frames**: Containers for your designs (like artboards)
- **Layers**: Objects stack in layers (like Photoshop)
- **Components**: Reusable elements
- **Variants**: Different versions of a component
- **Auto Layout**: Flexible, responsive containers
- **Prototyping**: Link frames to create interactive flows

Essential Tools:

Tool	Shortcut	Purpose
Move	V	Select and move objects
Frame	F	Create frames/containers
Rectangle	R	Draw rectangles
Ellipse	O	Draw circles/ellipses
Line	L	Draw lines
Text	T	Add text
Pen	P	Draw custom shapes
Hand	H (hold Space)	Pan around canvas
Zoom	Scroll or Z	Zoom in/out

5.9.2 Creating a Simple Wireframe in Figma

Step 1: Set Up Frame

1. Press F for Frame tool
2. Select a device preset (e.g., "Desktop" 1440x900)
3. Name your frame in the layers panel

Step 2: Add Structure

1. Draw rectangles (R) for major areas:
   - Header
   - Sidebar
   - Main content
   - Footer

2. Use gray fills (#E5E7EB for backgrounds, #9CA3AF for placeholders)

Step 3: Add Content Placeholders

1. Text tool (T) for headings and labels
2. Rectangles with X pattern for image placeholders
3. Lines for text content

Step 4: Create Components

1. Select a reusable element (e.g., a button)
2. Right-click → "Create Component" (or Ctrl/Cmd + Alt + K)
3. Use instances of the component throughout your design

Step 5: Add Prototyping Interactions

1. Switch to Prototype tab (right panel)
2. Select an element
3. Drag the connection handle to the target frame
4. Set interaction (e.g., "On Click → Navigate to")
5. Press Play button to preview

5.9.3 Figma Tips for Beginners

Organization:

- Name your layers meaningfully
- Group related elements (Ctrl/Cmd + G)
- Use pages for different sections of your project
- Create a "Components" page for your design system

Efficiency:

- Copy styles: Select object with desired style, then use eyedropper
- Duplicate: Alt + drag
- Align/distribute: Use toolbar or right-click
- Auto Layout: Makes responsive containers (Shift + A)

Collaboration:

- Share link for viewing/editing
- Leave comments on designs
- Use branches for major changes
- Export assets for development (PNG, SVG, CSS)

5.10 Chapter Summary

UI/UX design is essential to creating software that users actually want to use. Good design isn't about making things pretty—it's about understanding users and creating interfaces that help them accomplish their goals efficiently and pleasantly.

Key takeaways from this chapter:

- **UX design** focuses on the overall experience—how users feel when using a product. **UI design** focuses on the visual and interactive elements they directly interact with.
- **Human-centered design** starts with understanding users through research, empathy, and observation. Personas and journey maps help teams maintain focus on user needs.
- **Nielsen's 10 Usability Heuristics** provide a framework for evaluating and improving interfaces. They address visibility, consistency, error prevention, and more.
- **Prototyping** ranges from paper sketches to interactive simulations. Different fidelities serve different purposes in the design process.
- **Visual design fundamentals**—hierarchy, typography, color, and layout—create interfaces that communicate clearly and guide user attention.
- **Responsive design** creates interfaces that adapt to different screen sizes using fluid layouts, flexible images, and media queries.
- **Accessibility** ensures products work for users with disabilities. WCAG provides guidelines organized around perceivability, operability, understandability, and robustness.
- **Design systems and style guides** document reusable components and standards, enabling consistency and efficiency across projects.
- **Figma** is the industry-standard tool for UI design and prototyping, enabling designers and developers to collaborate on interface designs.

5.11 Key Terms

Term	Definition
User Experience (UX)	The overall experience a user has when interacting with a product
User Interface (UI)	The visual and interactive elements users interact with
Human-Centered Design	Design approach that focuses on understanding and addressing user needs
Persona	Fictional character representing a user type
Journey Map	Visualization of user experience over time
Heuristic	Rule of thumb for evaluating design quality

Term	Definition
Wireframe	Low-fidelity representation of interface layout
Prototype	Interactive model for testing design concepts
Responsive Design	Design that adapts to different screen sizes
Breakpoint	Screen width where layout changes occur
Accessibility	Design that can be used by people with disabilities
WCAG	Web Content Accessibility Guidelines
Design System	Collection of reusable components with documented standards
Style Guide	Documentation of design standards and patterns
Design Token	Smallest element of a design system (color, spacing, etc.)

5.12 Review Questions

1. Explain the difference between UX and UI design. How do they relate to each other?
2. What is human-centered design? Describe its key principles and why they matter for software development.
3. Choose three of Nielsen's 10 Usability Heuristics and explain each with an example of how a violation would affect users.
4. What is the difference between low-fidelity and high-fidelity prototypes? When would you use each?
5. Explain the mobile-first approach to responsive design. What are its advantages?
6. What is WCAG, and what are its four main principles? Give one specific guideline for each principle.
7. Why do keyboard navigation and focus indicators matter for accessibility?
8. What is a design system, and what are its benefits for software development teams?
9. Explain the concept of visual hierarchy. What techniques can you use to create it?
10. You're designing a checkout flow for an e-commerce site. How would you apply error prevention principles to reduce user mistakes?

5.13 Hands-On Exercises

Exercise 5.1: Heuristic Evaluation

Choose a website or application you use regularly. Conduct a heuristic evaluation:

1. Review the interface against Nielsen's 10 heuristics
2. Identify at least 10 usability issues
3. Rate each issue's severity (0-4)
4. Provide specific recommendations for improvement
5. Document your findings in a report

Exercise 5.2: User Persona Creation

For your semester project:

1. Identify your primary user type(s)
2. Create at least 2 detailed personas including:
    - Demographics and background
    - Goals and motivations
    - Frustrations and pain points
    - Technology comfort level
    - A representative quote
3. Create a user journey map for one key task

Exercise 5.3: Paper Prototyping

Before using digital tools:

1. Sketch wireframes on paper for 5-7 key screens of your project
2. Include annotations explaining functionality
3. Test your paper prototype with a classmate:
    - Give them a task to complete
    - "Swap" paper screens as they navigate
    - Note where they get confused
4. Iterate based on feedback

Exercise 5.4: Digital Wireframes

Using Figma (or similar tool):

1. Create wireframes for your project's main screens
2. Include:
    - Navigation structure

    - Content layout
    - Key UI elements (buttons, forms, etc.)
    - Placeholder content

3. Link wireframes into an interactive prototype
4. Test with at least 2 users and document findings

Exercise 5.5: Style Guide Creation

Create a style guide for your project including:

1. **Color Palette**

    - Primary, secondary, and accent colors
    - Semantic colors (success, error, warning, info)
    - Neutral/gray scale
    - Accessibility notes for each color combination

2. **Typography**

    - Font families
    - Size scale (H1-H6, body, small)
    - Weight and line-height specifications

3. **Spacing System**

    - Base unit and scale

4. **Component Documentation** (at least 5 components):

    - Buttons (with all states)
    - Input fields (with states)
    - Cards
    - Navigation
    - One additional component of your choice

Exercise 5.6: Accessibility Audit

Conduct an accessibility review of your project prototype:

1. Use an accessibility checker tool (axe, WAVE)
2. Test keyboard navigation manually
3. Check color contrast for all text
4. Verify all images have alt text
5. Test with a screen reader (even briefly)
6. Document issues found and fixes needed
7. Create an accessibility compliance checklist for your project

Exercise 5.7: Responsive Design

For one screen of your project:

1. Design the mobile version (375px wide)
2. Design the tablet version (768px wide)
3. Design the desktop version (1440px wide)
4. Document what changes at each breakpoint
5. Explain your responsive design decisions

5.14 Further Reading

Books:

- Krug, S. (2014). *Don't Make Me Think, Revisited: A Common Sense Approach to Web Usability* (3rd Edition). New Riders.
- Norman, D. (2013). *The Design of Everyday Things* (Revised Edition). Basic Books.
- Cooper, A., Reimann, R., Cronin, D., & Noessel, C. (2014). *About Face: The Essentials of Interaction Design* (4th Edition). Wiley.
- Lidwell, W., Holden, K., & Butler, J. (2010). *Universal Principles of Design* (Revised Edition). Rockport.

Online Resources:

- Nielsen Norman Group: https://www.nngroup.com/articles/
- Laws of UX: https://lawsofux.com/
- WebAIM Accessibility: https://webaim.org/
- Figma Learn: https://help.figma.com/
- A11y Project: https://www.a11yproject.com/
- Material Design Guidelines: https://material.io/design

Tools:

- Figma: https://www.figma.com/ (free tier available)
- WebAIM Contrast Checker: https://webaim.org/resources/contrastchecker/
- axe DevTools: Browser extension for accessibility testing
- Stark: Figma plugin for accessibility

References

Cooper, A., Reimann, R., Cronin, D., & Noessel, C. (2014). *About Face: The Essentials of Interaction Design* (4th Edition). Wiley.

IDEO. (2015). *The Field Guide to Human-Centered Design*. IDEO.org.

Krug, S. (2014). *Don't Make Me Think, Revisited: A Common Sense Approach to Web Usability* (3rd Edition). New Riders.

Nielsen, J. (1994). *Usability Engineering*. Morgan Kaufmann.

Nielsen, J. (1994). 10 Usability Heuristics for User Interface Design. Nielsen Norman Group. Retrieved from https://www.nngroup.com/articles/ten-usability-heuristics/

Norman, D. (2013). *The Design of Everyday Things* (Revised Edition). Basic Books.

W3C. (2018). Web Content Accessibility Guidelines (WCAG) 2.1. Retrieved from https://www.w3.org/TR/WCAG

Wroblewski, L. (2011). *Mobile First*. A Book Apart.Sample content for 06-ui-ux.qmd

Chapter 6: Agile Methodologies and Project Management

Learning Objectives

By the end of this chapter, you will be able to:

- Explain the philosophy and values behind the Agile movement
- Compare and contrast Scrum, Kanban, and Extreme Programming (XP)
- Implement Scrum practices, including sprint planning, daily standups, reviews, and retrospectives
- Apply Kanban principles to visualize and optimize workflow
- Estimate work using story points and measure team velocity
- Use project management tools like GitHub Projects and Jira effectively
- Break down projects into epics, stories, and tasks
- Create and manage a sprint backlog and Kanban board
- Adapt Agile practices to different team sizes and contexts

6.1 The Agile Revolution

In the late 1990s, software development was in crisis. Projects routinely failed—delivered late, over budget, or not at all. The dominant approach, often called "big design up front" or Waterfall, required extensive planning and documentation before any code was written. By the time software was delivered, requirements had changed, and the product no longer met user needs.

A group of software practitioners who had been experimenting with lighter, more iterative approaches came together in February 2001 at a ski resort in Snowbird, Utah. They emerged with the **Agile Manifesto**, a document that would reshape how the world builds software.

6.1.1 The Agile Manifesto

The manifesto articulates four core values:

Individuals and interactions over processes and tools

Working software over comprehensive documentation

Customer collaboration over contract negotiation

Responding to change over following a plan

The manifesto explicitly notes: "While there is value in the items on the right, we value the items on the left more."

This is a crucial nuance. Agile doesn't reject processes, documentation, contracts, or plans. It prioritizes their counterparts when trade-offs must be made.

Unpacking the Values:

Individuals and interactions over processes and tools: The best processes and tools can't compensate for poor communication or unmotivated people. Focus on building collaborative teams and enabling effective communication.

Working software over comprehensive documentation: Documentation that nobody reads adds no value. Working software that users can actually use provides real feedback. This doesn't mean "no documentation"—it means documentation that serves a purpose.

Customer collaboration over contract negotiation: Traditional contracts tried to specify everything upfront, then hold parties accountable to that specification. But requirements evolve. Agile favors ongoing collaboration where customers and developers work together toward shared goals.

Responding to change over following a plan: Plans become obsolete. Markets shift, users provide feedback, technologies evolve. Rather than fighting change, Agile embraces it as an opportunity to deliver more value.

6.1.2 The Twelve Principles

Behind the manifesto are twelve principles that guide Agile practice:

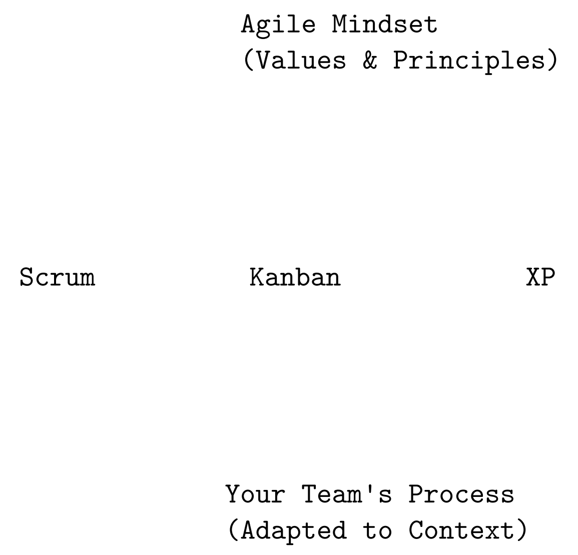
1. **Satisfy the customer** through early and continuous delivery of valuable software.
2. **Welcome changing requirements**, even late in development. Agile processes harness change for the customer's competitive advantage.
3. **Deliver working software frequently**, from a couple of weeks to a couple of months, with a preference to the shorter timescale.
4. **Business people and developers must work together** daily throughout the project.
5. **Build projects around motivated individuals.** Give them the environment and support they need, and trust them to get the job done.
6. **Face-to-face conversation** is the most efficient and effective method of conveying information.
7. **Working software is the primary measure of progress.**

8. **Agile processes promote sustainable development.** The sponsors, developers, and users should be able to maintain a constant pace indefinitely.
9. **Continuous attention to technical excellence** and good design enhances agility.
10. **Simplicity**—the art of maximizing the amount of work not done—is essential.
11. **The best architectures, requirements, and designs emerge from self-organizing teams.**
12. **At regular intervals, the team reflects** on how to become more effective, then tunes and adjusts its behavior accordingly.

6.1.3 Agile Is a Mindset, Not a Methodology

A common misconception is that "Agile" is a specific process you can install. It's not. Agile is a set of values and principles—a mindset. Specific methodologies like Scrum, Kanban, and XP are implementations of that mindset, each with different practices suited to different contexts.

Organizations that adopt Agile practices without embracing Agile values often fail to see benefits. They do "Agile theater"—standups that are status reports, sprints that are just short Waterfall phases, retrospectives that lead to no changes. True agility requires genuine commitment to the underlying principles.



6.1.4 Why Agile Works

Agile works because it aligns with fundamental truths about software development:

Requirements are uncertain. Users often don't know what they want until they see it. Agile delivers working software frequently, enabling early feedback and course correction.

Complexity defies prediction. Software projects are complex adaptive systems. Small changes can have large effects. Agile embraces empiricism—making decisions based on observation rather than prediction.

People matter. Software is built by humans, and human factors dominate project outcomes. Agile focuses on team dynamics, motivation, and sustainable pace.

Change is constant. Markets evolve, competitors move, users learn. Organizations that can respond quickly to change have a competitive advantage. Agile makes change a feature, not a bug.

6.2 Scrum: A Framework for Agile Development

Scrum is the most widely adopted Agile framework. Originally described by Ken Schwaber and Jeff Sutherland, Scrum provides a lightweight structure for teams to deliver complex products iteratively.

Scrum is named after the rugby formation where a team works together to move the ball down the field. Like rugby, Scrum emphasizes teamwork, adaptability, and continuous forward progress.

6.2.1 The Scrum Framework Overview

Scrum organizes work into fixed-length iterations called **sprints**, typically two weeks long. Each sprint produces a potentially shippable product increment.

```
                         SCRUM FRAMEWORK


ROLES              EVENTS                 ARTIFACTS

• Product Owner    • Sprint               • Product Backlog
• Scrum Master     • Sprint Planning      • Sprint Backlog
• Developers       • Daily Scrum          • Increment
                   • Sprint Review
                   • Sprint Retrospective
```

The Sprint Cycle:

```
 Product        Sprint          Daily        Sprint        Sprint
 Backlog       Planning         Scrum        Review     Retrospective
```

```
What               SPRINT (2-4 weeks)             Increment



                        Sprint Backlog
                   (Committed work for sprint)
```

6.2.2 Scrum Roles

Scrum defines three roles, each with distinct responsibilities:

Product Owner

The Product Owner is responsible for maximizing the value of the product. They are the single voice of the customer within the team.

Responsibilities:

- Maintains and prioritizes the Product Backlog
- Ensures the backlog is visible, transparent, and understood
- Makes decisions about what to build and in what order
- Accepts or rejects work completed by the team
- Communicates with stakeholders about progress and priorities

The Product Owner must be empowered to make decisions. A committee of stakeholders or a Product Owner who must get approval for every decision slows the team down.

Scrum Master

The Scrum Master is responsible for the Scrum process itself. They help the team understand and apply Scrum effectively.

Responsibilities:

- Facilitates Scrum events (planning, standups, reviews, retrospectives)
- Removes impediments that block the team
- Coaches the team on Scrum practices
- Protects the team from external interruptions
- Helps the organization understand and adopt Scrum

The Scrum Master is not a project manager. They don't assign tasks or manage the team. They serve the team by enabling effective Scrum practice.

Developers

The Developers are the people who do the work of building the product. Despite the name, this includes anyone contributing to the increment—developers, testers, designers, analysts, and others.

Responsibilities:

- Estimate work and commit to sprint goals
- Self-organize to accomplish sprint work
- Deliver a potentially shippable increment each sprint
- Participate in all Scrum events
- Hold each other accountable for quality and commitments

Scrum teams are cross-functional—they have all skills needed to deliver the increment without depending on people outside the team.

Team Size:

Scrum works best with small teams. The Scrum Guide recommends 10 or fewer people. Larger groups should split into multiple Scrum teams.

```
                        SCRUM TEAM


     Product          Scrum               Developers
      Owner           Master             (3-9 people)

   • Backlog        • Process           • Designer
   • Priority       • Coach             • Developer
   • Value          • Facilitate        • Developer
   • Decide         • Remove            • Developer
                      obstacles         • QA Engineer
```

6.2.3 Scrum Artifacts

Product Backlog

The Product Backlog is an ordered list of everything that might be needed in the product. It's the single source of requirements.

Characteristics:

- Owned and maintained by the Product Owner

- Ordered by value, risk, priority, and necessity
- Items at the top are more detailed than items at the bottom
- Continuously refined (grooming/refinement)
- Never complete—it evolves as the product and market evolve

Backlog items typically include:

- Features
- Bug fixes
- Technical work
- Knowledge acquisition (spikes)

Example Product Backlog:

```
PRODUCT BACKLOG - TaskFlow                              Owner: Sarah

Rank  Item                                          Points   Status

 1    User can create and edit tasks                  5      Ready
 2    User can assign tasks to team members           3      Ready
 3    User can set due dates with reminders           5      Ready
 4    User can view tasks on Kanban board             8      Ready
 5    User can receive email notifications            5      Needs
                                                             refinement
 6    User can filter and search tasks                8      Needs
                                                             refinement
 7    Admin can manage team membership                5      Rough
 8    User can attach files to tasks                  8      Rough
 9    Integration with Google Calendar                13     Rough
10    Mobile app (iOS)                                ?      Idea
11    Mobile app (Android)                            ?      Idea
...   ...
```

Sprint Backlog

The Sprint Backlog is the set of Product Backlog items selected for the sprint, plus a plan for delivering them.

Characteristics:

- Created during Sprint Planning
- Owned by the Developers
- Represents the team's commitment for the sprint
- Updated daily as work progresses
- Visible to all stakeholders

The Sprint Backlog includes:

- Selected user stories
- Tasks to complete each story
- Estimated hours remaining (optional)

Example Sprint Backlog:

```
SPRINT 3 BACKLOG                          Sprint Goal: Core Task CRUD



  Story: User can create and edit tasks (5 pts)
  Status: In Progress

  Tasks:
  [x] Design task creation form (Designer, 4h)
  [x] Create Task model and database schema (Dev A, 3h)
  [~] Implement create task API endpoint (Dev B, 4h) - In Progress
  [ ] Build task creation UI component (Dev C, 6h)
  [ ] Implement edit task functionality (Dev B, 4h)
  [ ] Write unit tests (QA, 3h)
  [ ] Write integration tests (QA, 2h)



  Story: User can assign tasks to team members (3 pts)
  Status: Not Started

  Tasks:
  [ ] Add assignee field to Task model (Dev A, 2h)
  [ ] Create user assignment dropdown component (Dev C, 4h)
  [ ] Implement assignment API (Dev B, 3h)
  [ ] Add assignment notification (Dev A, 2h)
  [ ] Write tests (QA, 2h)



  Story: User can set due dates with reminders (5 pts)
  Status: Not Started


Sprint Capacity: 60 hours | Committed: 52 hours | Remaining: 38 hours
```

Increment

The Increment is the sum of all Product Backlog items completed during a sprint, plus all previous increments. It must be in usable condition—meeting the team's Definition of Done—regardless of whether the Product Owner decides to release it.

Definition of Done (DoD)

The Definition of Done is a shared understanding of what "complete" means. It ensures transparency and quality.

Example Definition of Done:

- Code complete and peer-reviewed
- Unit tests written and passing
- Integration tests passing
- Documentation updated
- No known bugs
- Meets acceptance criteria
- Deployed to staging environment
- Product Owner accepted

6.2.4 Scrum Events

Scrum prescribes five events, each with a specific purpose. These events create regularity and minimize the need for ad-hoc meetings.

The Sprint

The Sprint is a container for all other events. It's a fixed time-box (typically 2 weeks) during which the team works to deliver a potentially shippable increment.

Sprint Rules:

- Fixed duration (don't extend sprints)
- No changes that endanger the Sprint Goal
- Quality standards don't decrease
- Scope may be clarified and renegotiated with Product Owner
- A new sprint begins immediately after the previous one ends

Sprint Planning

Sprint Planning kicks off the sprint. The team decides what to work on and how to accomplish it.

Duration: Up to 8 hours for a one-month sprint (proportionally less for shorter sprints)

Part 1: What can be done this sprint?

- Product Owner presents top-priority items
- Team discusses and asks clarifying questions
- Team selects items they believe they can complete
- Team crafts a Sprint Goal

Part 2: How will the work be accomplished?

- Team breaks selected items into tasks

- Team estimates task effort
- Team commits to the Sprint Backlog

Sprint Planning Agenda Example:

```
SPRINT 3 PLANNING


Time: Monday 9:00 AM - 1:00 PM (4 hours)
Attendees: Full Scrum Team

AGENDA


9:00 - 9:15    Review Sprint 2 outcomes and current velocity
               Previous velocity: 18 points
               Capacity this sprint: Full team available

9:15 - 10:30   PART 1: Select Sprint Backlog Items
               • Product Owner presents priorities
               • Team asks clarifying questions
               • Team selects items (targeting ~18-20 points)

10:30 - 10:45  Break

10:45 - 11:15  Define Sprint Goal
               • What business value will we deliver?
               • How will we know we succeeded?

11:15 - 12:45  PART 2: Plan the Work
               • Break stories into tasks
               • Identify dependencies
               • Assign initial owners (optional)
               • Identify risks and unknowns

12:45 - 1:00   Confirm Commitment
               • Review Sprint Backlog
               • Team confirms they can commit
               • Scrum Master confirms understanding

OUTPUT

• Sprint Goal: "Users can create, edit, and manage basic tasks"
• Sprint Backlog: 5 stories, 18 points
• Task breakdown for all stories
```

Daily Scrum (Standup)

The Daily Scrum is a brief daily meeting for the Developers to synchronize and plan the day's work.

Duration: 15 minutes maximum Time: Same time and place every day Attendees: Developers (Scrum Master facilitates, Product Owner may attend)

Traditional Format (Three Questions):

1. What did I accomplish yesterday?
2. What will I work on today?
3. What obstacles are in my way?

Alternative Format (Walking the Board):

- Review each item on the sprint board
- What's needed to move it forward?
- Who's working on what?

Daily Scrum Best Practices:

DO:

- Stand up (keeps it short)
- Start on time, even if people are missing
- Focus on sprint goal progress
- Identify blockers immediately
- Keep it under 15 minutes

DON'T:

- Give detailed status reports
- Solve problems during standup (take offline)
- Make it a report to the Scrum Master
- Skip days
- Let it become a complaint session

Example Daily Scrum:

```
DAILY SCRUM - Tuesday 9:00 AM


ALICE:
"Yesterday I completed the task creation API. Today I'm starting
on the edit endpoint. No blockers."

BOB:
"I'm still working on the task form component. Should finish
today. I have a question about validation rules-I'll grab Sarah
after standup."

CAROL:
```

```
"Finished unit tests for the model layer. Starting integration
tests today. Blocked on needing access to the staging database-
can someone help?"

SCRUM MASTER:
"I'll get Carol database access right after this. Anything else?
No? Let's get to work."

Duration: 4 minutes
```

Sprint Review

The Sprint Review is held at the end of the sprint to inspect the increment and adapt the Product Backlog.

Duration: Up to 4 hours for a one-month sprint Attendees: Scrum Team plus invited stakeholders

What happens:

- Team demonstrates completed work
- Stakeholders provide feedback
- Product Owner discusses the backlog
- Group collaborates on what to do next
- Discussion of timeline, budget, and capabilities

What it's NOT:

- A formal presentation
- A sign-off meeting
- Just a demo (it's interactive)

Sprint Review Agenda Example:

```
SPRINT 3 REVIEW


Time: Friday 2:00 PM - 3:30 PM
Attendees: Scrum Team + Marketing Lead, Customer Success Lead

AGENDA


2:00 - 2:10    Welcome and Sprint Overview
               • Sprint Goal: Core Task CRUD
               • What we committed to vs. what we delivered

2:10 - 2:45    Demo of Completed Work
               • Task creation (Alice)
               • Task editing (Bob)
               • Task assignment (Carol)
               • Due date setting (Alice)

               [Interactive-stakeholders can try features]

2:45 - 3:00    Discussion and Feedback
               • What do you like?
               • What would you change?
               • What questions do you have?

3:00 - 3:15    Product Backlog Review
               • What's coming next sprint?
               • Any priority changes based on feedback?
               • New items to add?

3:15 - 3:30    Release Discussion
               • Are we on track for beta launch?
               • What risks do we see?

OUTPUT

• Feedback captured
• Backlog updates identified
• Stakeholder alignment
```

Sprint Retrospective

The Sprint Retrospective is held after the Sprint Review and before the next Sprint Planning. The team inspects how the sprint went and identifies improvements.

Duration: Up to 3 hours for a one-month sprint Attendees: Scrum Team only (safe space for candid discussion)

Purpose:

- Inspect the last sprint (people, relationships, process, tools)
- Identify what went well
- Identify what could be improved
- Create a plan for implementing improvements

Retrospective Formats:

Start-Stop-Continue:

- What should we START doing?
- What should we STOP doing?
- What should we CONTINUE doing?

Glad-Sad-Mad:

- What made us GLAD?
- What made us SAD?

- What made us MAD?

4Ls:

- What did we LIKE?
- What did we LEARN?
- What did we LACK?
- What do we LONG FOR?

Sailboat:

```
           Island (Goals)


     Boat (Team)


    Anchor (What slows us down)

    Wind (What propels us forward)

    Rocks (Risks ahead)
```

Example Retrospective Output:

```
SPRINT 3 RETROSPECTIVE


WHAT WENT WELL

• Completed all committed stories
• Great collaboration between frontend and backend
• New testing approach caught bugs early
• Stakeholder feedback was very positive

WHAT COULD BE IMPROVED

• Sprint planning ran long (5 hours instead of 4)
• Two stories had unclear acceptance criteria
• Staging environment was unstable
• Daily standups started late several times

ACTION ITEMS

1. [HIGH] Refine stories before sprint planning
   Owner: Product Owner
   Due: Before Sprint 4 planning
```

```
2. [MEDIUM] Set up staging environment monitoring
   Owner: Carol
   Due: Sprint 4, Day 3

3. [LOW] Start standup alarm at 8:58 AM
   Owner: Scrum Master
   Due: Immediately
```

6.2.5 Sprint Metrics

Velocity

Velocity is the amount of work a team completes in a sprint, measured in story points. It's used for planning future sprints.

```
Sprint History:
Sprint 1: 15 points
Sprint 2: 18 points
Sprint 3: 21 points
Sprint 4: 16 points
Sprint 5: 20 points

Average Velocity: 18 points
```

Important: Velocity is a planning tool, not a performance metric. Using velocity to compare teams or pressure teams to "increase velocity" undermines its usefulness.

Burndown Chart

A burndown chart shows work remaining versus time. It helps visualize sprint progress.

```
Story Points
Remaining

 20
        - - - - - - - - - - - - Ideal Burndown
 15

 10

  5

  0
                      Days
         1    2    3    4    5    6
```

If actual burndown is above the ideal line, the team is behind. If below, they're ahead.

Burnup Chart

A burnup chart shows cumulative work completed. It's useful for seeing scope changes.

```
Story Points
                                          Scope (may change)
 50

 40

 30

 20

 10

  0
                         Days
          1     2     3     4     5     6
```

6.3 Kanban: Continuous Flow

While Scrum organizes work into sprints, **Kanban** focuses on continuous flow. Originating from Toyota's manufacturing system, Kanban was adapted for software development by David Anderson in the mid-2000s.

6.3.1 Kanban Principles

Kanban is built on four foundational principles:

1. Start with what you do now

Kanban doesn't prescribe roles, ceremonies, or artifacts. It overlays on your existing process to make it visible and improve it incrementally.

2. Agree to pursue incremental, evolutionary change

Rather than wholesale transformation, Kanban favors small, continuous improvements. This reduces resistance and risk.

3. Respect the current process, roles, and responsibilities

Don't throw everything out. Preserve what works while improving what doesn't.

4. Encourage acts of leadership at all levels

Improvement ideas can come from anywhere. Empower everyone to identify and implement improvements.

6.3.2 Kanban Practices

1. Visualize the Workflow

Make work visible using a Kanban board. Each column represents a stage in your workflow.

```
                            KANBAN BOARD

  Backlog        To Do    In Progress    Review           Done


  Task 8      Task 5      Task 3      Task 1      Task A
                           Alice       Carol


  Task 9      Task 6      Task 4                   Task B
                            Bob


  Task 10                                            Task C



     (∞)          (3)          (3)          (2)              (∞)


                    WIP Limits shown in parentheses
```

2. Limit Work in Progress (WIP)

WIP limits cap how many items can be in each stage simultaneously. This prevents overload and improves flow.

Why WIP limits matter:

- Reduces context switching
- Exposes bottlenecks
- Encourages finishing before starting
- Improves cycle time
- Increases focus

```
Without WIP Limits:               With WIP Limits:
```

```
In Progress: 12 items           In Progress: 3 items (Limit: 3)
• Many half-finished            • Fewer items, more focus
• Lots of context switching     • Items finish faster
• Nothing actually finishing    • Problems visible immediately
• Problems hidden in pile       • Team swarms to unblock
```

3. Manage Flow

Monitor and optimize the flow of work through the system. Track metrics, identify bottlenecks, and make improvements.

4. Make Policies Explicit

Document the rules governing how work flows through the system:

- Definition of Ready (when can work enter a column?)
- Definition of Done (when can work leave a column?)
- WIP limits
- Prioritization rules
- Blocked item policies

5. Implement Feedback Loops

Regular cadences for review and adaptation:

- Daily standups
- Replenishment meetings (add work to board)
- Delivery planning
- Service delivery review
- Operations review
- Risk review
- Strategy review

6. Improve Collaboratively, Evolve Experimentally

Use the scientific method:

- Observe current state
- Form hypotheses about improvements
- Experiment with changes
- Measure results
- Adopt what works

6.3.3 Kanban Board Design

Columns should reflect your actual workflow. Common patterns:

Simple Board:

```
Backlog → In Progress → Done
```

Development Board:

```
Backlog → Ready → Development → Code Review → Testing → Done
```

Board with Explicit Buffer:

```
Backlog → Ready → Development → Dev Done → Testing → Done

         (Buffer between development and testing)
```

Board with Swimlanes:

```

           To Do   In Progress   Review   Done

 Urgent                                       ← Priority
                               Lanes
 Normal

 Low
```

6.3.4 Kanban Metrics

Cycle Time

Cycle time is how long an item takes from start to finish—the time between "work started" and "work completed."

```
Task A: Started Monday 9 AM → Completed Wednesday 4 PM
Cycle Time: 2.3 days

Average Cycle Time: Sum of all cycle times / Number of items
```

Lower cycle time means faster delivery. Track cycle time over time to measure improvement.

Lead Time

Lead time is the total time from request to delivery—including time waiting in the backlog.

```
Request      Start           Complete
←     Lead Time (total)
              ←  Cycle Time
```

Throughput

Throughput is the number of items completed in a given time period.

```
Week 1: 12 items completed
Week 2: 15 items completed
Week 3: 11 items completed

Average Throughput: 12.7 items/week
```

Cumulative Flow Diagram (CFD)

A CFD shows the quantity of items in each state over time. It reveals bottlenecks and flow problems.

```
Items

 50                      Done

 40                     Review

 30                   In Progress

 20                 To Do

 10              Backlog

  0
                         Weeks
         1    2    3    4    5    6    7
```

The vertical distance between bands represents WIP. The horizontal distance represents cycle time. Widening bands indicate bottlenecks.

6.3.5 Scrum vs. Kanban

Aspect	Scrum	Kanban
Cadence	Fixed sprints (1-4 weeks)	Continuous flow
Roles	Product Owner, Scrum Master, Developers	No prescribed roles
Planning	Sprint Planning at start of sprint	Continuous (just-in-time)
Change	No changes during sprint	Change anytime
Metrics	Velocity, Burndown	Cycle time, Throughput
Commitment	Sprint backlog commitment	No commitment beyond WIP
Best for	Product development, cross-functional teams	Operations, maintenance, varied work

Scrumban:

Many teams combine elements of both:

- Kanban board with visualized flow
- WIP limits
- Sprint cadence for planning and review
- Retrospectives for improvement

6.4 Extreme Programming (XP)

Extreme Programming (XP) is an Agile methodology that emphasizes technical excellence and team practices. Created by Kent Beck in the late 1990s, XP "turns the dials to 10" on good software practices.

6.4.1 XP Values

Communication: Team members communicate face-to-face frequently. Problems are surfaced immediately. Knowledge is shared, not hoarded.

Simplicity: Do the simplest thing that could possibly work. Don't build for requirements you don't have yet. Simplify code through refactoring.

Feedback: Get feedback quickly and often. Short iterations, continuous integration, pair programming, and customer involvement all provide rapid feedback.

Courage: Make difficult decisions. Refactor mercilessly. Throw away code that doesn't work. Tell customers the truth about estimates.

Respect: Team members respect each other's contributions. Everyone's input matters. People are not resources.

6.4.2 XP Practices

XP defines twelve practices organized into four categories:

Fine-Scale Feedback:

Pair Programming: Two developers work together at one workstation. One "drives" (types), the other "navigates" (reviews). Pairs switch frequently. Benefits include knowledge sharing, better design, and fewer bugs.

Planning Game: Customers and developers collaborate on release and iteration planning. Customers define features; developers estimate. Simple, cards-based planning.

Test-Driven Development (TDD): Write a failing test before writing code. Write just enough code to pass the test. Refactor. Repeat. This produces well-tested, well-designed code.

Whole Team: The customer (or customer representative) is part of the team, available daily to answer questions, provide feedback, and make decisions.

Continuous Process:

Continuous Integration: Developers integrate code frequently (multiple times per day). Each integration is verified by automated tests. This catches problems early and keeps the codebase stable.

Refactoring: Continuously improve code structure without changing behavior. Remove duplication. Improve clarity. Keep the codebase healthy.

Small Releases: Release small increments frequently. This provides feedback, delivers value early, and reduces risk.

Shared Understanding:

Coding Standards: The team agrees on coding conventions and follows them. Anyone can work on any code. The codebase looks like one person wrote it.

Collective Code Ownership: Anyone can modify any code. This spreads knowledge and enables flexibility. Code reviews and pair programming support this.

Simple Design: Design for current requirements, not imagined future needs. The simplest design is easiest to understand, test, and modify.

System Metaphor: A shared story of how the system works. A metaphor helps the team communicate and reason about the system consistently.

Developer Welfare:

Sustainable Pace: Work at a pace you can maintain indefinitely. No death marches. Tired developers make mistakes and burn out.

6.4.3 The TDD Cycle

Test-Driven Development follows a simple cycle:

```
1. RED
Write a failing
test




2. GREEN
Write code to
pass the test
```

```
3. REFACTOR
Improve the code
(tests still pass)
```

Example TDD Session:

```python
# Step 1: RED - Write a failing test
def test_calculate_total_for_empty_cart():
    cart = ShoppingCart()
    assert cart.calculate_total() == 0

# Run tests: FAIL - ShoppingCart doesn't exist

# Step 2: GREEN - Write minimal code to pass
class ShoppingCart:
    def calculate_total(self):
        return 0

# Run tests: PASS

# Step 3: REFACTOR - Nothing to refactor yet

# Step 1: RED - Write next failing test
def test_calculate_total_for_single_item():
    cart = ShoppingCart()
    cart.add_item(Product("Widget", 9.99), quantity=1)
    assert cart.calculate_total() == 9.99

# Run tests: FAIL - add_item doesn't exist

# Step 2: GREEN - Write code to pass
class ShoppingCart:
    def __init__(self):
        self.items = []

    def add_item(self, product, quantity):
        self.items.append((product, quantity))

    def calculate_total(self):
        return sum(product.price * qty for product, qty in self.items)
```

```
# Run tests: PASS

# Continue cycle...
```

6.4.4 When to Use XP Practices

Not all XP practices are appropriate for all situations:

Practice	High Value When…	Less Valuable When…
Pair Programming	Complex problems, knowledge transfer needed, quality critical	Simple tasks, team very experienced, remote-only team
TDD	New code, complex logic, long-lived codebase	Prototypes, UI-heavy work, exploratory work
Continuous Integration	Multiple developers, frequent changes	Solo developer, stable codebase
Refactoring	Growing codebase, changing requirements	Throwaway code, frozen requirements
Collective Ownership	Cross-functional teams, bus factor concerns	Specialized expertise areas

---

6.5 Estimation: Story Points and Planning

Estimation is notoriously difficult in software development. How long will a feature take? It depends on countless factors, many unknown at estimation time. Agile approaches estimation differently—focusing on relative sizing rather than absolute time predictions.

6.5.1 Why Traditional Estimation Fails

Traditional estimation asks: "How many hours/days will this take?" This approach fails because:

Uncertainty: Early in a project, we don't know enough to estimate precisely. Later, we know more but have less flexibility.

Anchoring: Once someone states an estimate, others anchor to it. The first number biases all subsequent discussion.

Pressure: Estimates become commitments, then deadlines. Teams pad estimates defensively or face blame when reality differs from prediction.

Individual variation: A task that takes Alice two hours might take Bob eight. Whose estimate do we use?

Hidden complexity: Software has unknown unknowns. We discover complexity during implementation, not during estimation.

6.5.2 Story Points

Story points are a relative measure of effort, complexity, and uncertainty. Rather than estimating in hours, teams assign point values that represent how "big" a story is relative to other stories.

Key Characteristics:

- **Relative, not absolute**: "Story A is about twice as big as Story B"
- **Team-specific**: Points aren't comparable across teams
- **Include uncertainty**: Bigger point values include higher uncertainty
- **Not tied to time**: A 2-point story doesn't mean 2 hours or 2 days

The Fibonacci Sequence:

Most teams use a modified Fibonacci sequence for point values:

```
1, 2, 3, 5, 8, 13, 21, (40, 100)
```

Why Fibonacci? The gaps between numbers grow larger at higher values, reflecting the increasing uncertainty of larger work items. You can reasonably distinguish a 2-point story from a 3-point story, but distinguishing 21 points from 22 points is meaningless.

Reference Story:

Teams establish a reference story—a well-understood, medium-sized piece of work assigned a middle value (often 3 or 5 points). All other stories are sized relative to this reference.

```
Reference: "User can log in with email/password" = 3 points

Now estimate:
• "User can reset password via email" → Similar complexity → 3 points
• "User can view their profile" → Simpler → 2 points
• "User can upload profile photo" → More complex (file handling) → 5 points
• "User can log in with SSO (Google, Microsoft)" → Much more complex → 8 points
```

6.5.3 Planning Poker

Planning Poker is a consensus-based estimation technique that avoids anchoring and encourages discussion.

How It Works:

1. Each team member has cards with point values (1, 2, 3, 5, 8, 13, 21, ?)
2. The Product Owner presents a story
3. Team asks clarifying questions
4. Everyone simultaneously reveals their estimate

5. If estimates differ significantly, discuss and re-estimate
6. Repeat until consensus

```
PLANNING POKER SESSION


Story: "User can filter tasks by status, assignee, and due date"

Round 1:
Alice: 5    Bob: 8    Carol: 8    Dave: 5    Eve: 13

Discussion:
Eve: "I'm thinking about the complexity of combining multiple filters"
Alice: "I assumed we'd reuse our existing filter component"
Bob: "Good point, but we need new database queries for each filter type"
Eve: "Oh, if we're reusing components, that does simplify things"

Round 2:
Alice: 5    Bob: 8    Carol: 5    Dave: 5    Eve: 8

Discussion:
Carol: "Changed my mind-filter component does help"
Bob: "I'm still at 8 because of the query complexity"

Final: Team agrees on 8 (erring toward higher due to query work)
```

Planning Poker Benefits:

- Everyone participates
- No anchoring (simultaneous reveal)
- Differences trigger valuable discussions
- Builds shared understanding
- Fun and engaging

6.5.4 Velocity and Capacity Planning

Velocity is the average number of story points a team completes per sprint. It's calculated from historical data.

```
Sprint 1: 21 points completed
Sprint 2: 18 points completed
Sprint 3: 24 points completed
Sprint 4: 20 points completed
Sprint 5: 22 points completed

Average Velocity: 21 points per sprint
```

Using Velocity for Planning:

If average velocity is 21 points, the team should plan for approximately 21 points next sprint. Some teams use a range:

- Pessimistic: 18 points (worst recent sprint)
- Expected: 21 points (average)
- Optimistic: 24 points (best recent sprint)

Velocity Adjustments:

Adjust for known factors:

- Team member on vacation: Reduce proportionally
- New team member: Expect lower velocity initially
- Technical debt paydown sprint: Reduce feature velocity

Important Cautions:

- Don't compare velocity across teams (points aren't standardized)
- Don't use velocity as a performance metric (teams will game it)
- Velocity stabilizes after 3-5 sprints; early sprints are unreliable
- Changing team composition resets velocity

6.5.5 Estimation Alternatives

Some teams abandon points entirely:

Story Counting:

If stories are consistently small (well-refined), just count them. "We complete about 8 stories per sprint." This works when stories are similar in size.

T-Shirt Sizing:

Use qualitative sizes: XS, S, M, L, XL. Simpler than points but still enables relative comparison. Often converted to points for tracking:

```
XS = 1 point
S  = 2 points
M  = 5 points
L  = 8 points
XL = 13 points (should probably be split)
```

#NoEstimates:

Some teams avoid estimation entirely, focusing instead on:

- Breaking work into small, similarly-sized pieces
- Counting completed items
- Using cycle time for forecasting

This works best with mature teams and well-refined backlogs.

6.6 Project Management Tools

Modern project management tools support Agile workflows with digital boards, backlog management, and reporting.

6.6.1 GitHub Projects

GitHub Projects provides project management directly integrated with GitHub's issues and pull requests.

Setting Up a GitHub Project:

1. Navigate to your repository (or organization)
2. Click "Projects" tab → "New Project"
3. Choose a template (Board, Table, or Roadmap)
4. Customize columns to match your workflow

Board View:

```
TaskFlow Development                                        + Add view

    Backlog         Ready          In Progress     Review
                                                                Done

#23 SSO Login    #18 Password    #15 Task CRUD   #12 User       #10
enhancement      reset           @alice          profiles       #8
                 3 points        5 points        @carol         #7
#24 File                                         3 points       #5
attachments      #19 Due date    #16 Task                       #3
                 reminders       assignment                     #1
#25 Calendar     5 points        @bob
integration                      3 points

+ Add item       + Add item      + Add item      + Add item
```

Table View:

```
Title                  Status       Assignee   Points   Sprint   Labels

Task CRUD              In Progress  @alice     5        Sprint3  feature
Task assignment        In Progress  @bob       3        Sprint3  feature
User profiles          Review       @carol     3        Sprint3  feature
Password reset         Ready        -          3        Sprint3  feature
Due date reminders     Ready        -          5        Sprint3  feature
SSO Login              Backlog      -          8        -        feature
File attachments       Backlog      -          8        -        feature
```

GitHub Projects Features:

- **Custom Fields**: Add story points, sprints, priorities
- **Automation**: Auto-move items when issues close or PRs merge
- **Filtering**: Filter by assignee, label, milestone, custom fields
- **Grouping**: Group items by sprint, assignee, or status
- **Iterations**: Track sprint cycles
- **Insights**: Burnup charts and progress tracking

Connecting Issues to Projects:

```markdown
# Issue Template Example

## User Story
As a [user type], I want to [action] so that [benefit].

## Acceptance Criteria
- [ ] Criterion 1
- [ ] Criterion 2
- [ ] Criterion 3

## Technical Notes
- Relevant implementation details

## Story Points: 5
## Sprint: Sprint 3
```

6.6.2 Jira

Jira is the most widely used Agile project management tool, especially in enterprise environments.

Key Jira Concepts:

- **Project**: Container for all issues related to a product or initiative
- **Issue Types**: Story, Bug, Task, Epic, Subtask

- **Workflow**: States and transitions (To Do → In Progress → Done)
- **Board**: Scrum or Kanban visualization
- **Sprint**: Time-boxed iteration (Scrum)
- **Backlog**: Ordered list of work items

Jira Board Example:

```
 TASKFLOW SPRINT 3                                  Sprint ends in 5 days

   TO DO (5)     IN PROGRESS (3)    IN REVIEW (1)       DONE (4)



   TASK-45       TASK-42       TASK-40       TASK-38
  Filter UI      Due dates       User profile    Login page
                Alice         Carol         Done



   TASK-46       TASK-43                               TASK-39
  Date bug       Assignments                             Task model
             Bob                           Done



   TASK-47       TASK-44                               TASK-41
  Sort tasks     Task CRUD                               DB schema
                Dave                          Done



 = Story Point      = High Priority      = Story      = Bug
```

Jira Reports:

- **Burndown Chart**: Work remaining vs. time
- **Velocity Chart**: Points completed per sprint
- **Sprint Report**: Summary of sprint completion
- **Cumulative Flow**: Work in each state over time
- **Control Chart**: Cycle time analysis

6.6.3 Other Tools

Trello: Simple, visual board-based tool. Good for small teams and simple workflows. Less feature-rich than Jira but easier to learn.

Asana: Task and project management with multiple views (list, board, timeline, calendar). Good for cross-functional teams.

Linear: Modern, fast issue tracking built for software teams. Keyboard-driven, GitHub integration, clean interface.

Azure DevOps: Microsoft's integrated platform including boards, repos, pipelines, and test plans. Good for Microsoft-ecosystem teams.

Notion: Flexible workspace that can be configured for project management. Good for teams wanting to combine docs and project tracking.

Choosing a Tool:

Factor	Consider…
Team size	Simple tools for small teams; robust tools for large
Integration needs	GitHub integration, CI/CD, communication tools
Complexity	Simple workflows → simple tools; complex → robust tools
Budget	Free tiers vary; enterprise features cost money
Learning curve	Time available for training
Reporting needs	Basic → simple tools; detailed metrics → robust tools

For student projects, **GitHub Projects** is often sufficient and integrates naturally with your code repository.

6.7 Breaking Down Work: Epics, Stories, and Tasks

Effective Agile teams break work into appropriately sized pieces. The hierarchy typically flows from large (Epics) to small (Tasks).

6.7.1 The Work Hierarchy

```
INITIATIVE / THEME
"Improve user engagement"
(Strategic goal, spans multiple releases)

   EPIC
     "User Task Management"
     (Large feature, spans multiple sprints)

        USER STORY
          "User can create tasks"
```

```
              (Deliverable value, completable in one sprint)

                 TASK
                   "Implement create task API endpoint"
                   (Technical work, hours to complete)

                      SUBTASK
                        "Write unit tests for validation"
                        (Small piece of a task)
```

6.7.2 Epics

Epics are large bodies of work that span multiple sprints. They're too big to complete in one iteration and must be broken down into smaller stories.

Characteristics:

- Takes weeks or months to complete
- Includes multiple user stories
- May span multiple teams
- Represents a significant feature or capability

Example Epics:

```
EPIC: User Authentication System
  Story: User can register with email/password
  Story: User can log in with credentials
  Story: User can reset forgotten password
  Story: User can update password
  Story: Admin can manage user accounts
  Story: User can log in with Google SSO
  Story: User can enable two-factor authentication

EPIC: Task Management
  Story: User can create tasks
  Story: User can edit tasks
  Story: User can delete tasks
  Story: User can assign tasks to team members
  Story: User can set due dates
  Story: User can add labels to tasks
  Story: User can set task priority
  Story: User can add comments to tasks
```

6.7.3 User Stories (Review)

As covered in Chapter 2, user stories follow the format:

> As a [type of user], I want [capability] so that [benefit].

Well-Sized Stories:

Stories should be small enough to complete in one sprint—ideally in a few days. The INVEST criteria guide good stories:

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

Splitting Large Stories:

When a story is too large, split it using these patterns:

By workflow step:

```
Original: User can purchase a product

Split into:
• User can add items to cart
• User can view cart
• User can enter shipping address
• User can enter payment information
• User can review and confirm order
```

By business rule:

```
Original: User can search for products

Split into:
• User can search by product name
• User can search by category
• User can filter by price range
• User can sort search results
```

By data variation:

```
Original: User can log in

Split into:
• User can log in with email/password
• User can log in with Google
• User can log in with Microsoft
```

By operation (CRUD):

```
Original: User can manage tasks

Split into:
• User can create tasks
• User can view tasks
• User can edit tasks
• User can delete tasks
```

By user type:

```
Original: User can view reports

Split into:
• Team member can view their own reports
• Manager can view team reports
• Admin can view all reports
```

6.7.4 Tasks

Tasks are the technical activities required to complete a story. Unlike stories, tasks describe *how* rather than *what*.

Characteristics:

- Estimated in hours (typically 1-8 hours)
- Assigned to individuals
- Technical in nature
- Not directly valuable to users (stories are)

Example Story with Tasks:

```
STORY: User can create tasks (5 points)

TASKS:

 Task                                                       Hours   Assignee
```

```
  Create Task database model and migration                  2     Alice
  Implement CreateTask API endpoint                         4     Alice
  Write API endpoint validation                             2     Alice
  Create task creation form component                       4     Bob
  Implement form validation (client-side)                   2     Bob
  Connect form to API                                       2     Bob
  Write unit tests for Task model                           2     Carol
  Write integration tests for API                           3     Carol
  Write E2E test for task creation flow                     2     Carol
  Update API documentation                                  1     Alice

  TOTAL                                                    24
```

6.7.5 Definition of Ready and Definition of Done

Definition of Ready (DoR):

Criteria that must be met before a story enters a sprint:

```
DEFINITION OF READY


A story is Ready when:
  Story is written in user story format
  Acceptance criteria are defined
  Story has been estimated
  Dependencies are identified
  UX designs are complete (if applicable)
  Technical approach is understood
  Story is small enough for one sprint
  Product Owner is available to answer questions
```

Definition of Done (DoD):

Criteria that must be met before a story is considered complete:

```
DEFINITION OF DONE


A story is Done when:
  All acceptance criteria are met
  Code is written and committed
  Code has been peer-reviewed
  Unit tests are written and passing
  Integration tests are passing
  Documentation is updated
```

```
Code is deployed to staging
QA has verified the feature
Product Owner has accepted the work
No known bugs remain
```

6.8 Running Effective Agile Meetings

Agile meetings are collaborative working sessions, not status reports. Effectiveness depends on preparation, facilitation, and follow-through.

6.8.1 Meeting Anti-Patterns

The Status Report Standup:

```
Each person reports to the Scrum Master
No interaction between team members
Runs long because people give detailed updates
Feels like micromanagement
```

The Endless Planning:

```
Goes hours over time-box
Debates implementation details
No clear outcome
Team disengages
```

The Blameful Retrospective:

```
Focus on who made mistakes
Defensive atmosphere
Same issues discussed repeatedly
No action items
```

6.8.2 Facilitating Effective Standups

Structure (15 minutes max):

```
DAILY STANDUP FACILITATION


BEFORE:
• Start exactly on time
• Stand up (keeps it short)
• Face the board, not the Scrum Master

DURING:
• Walk the board right-to-left (focus on finishing)
  OR each person answers three questions
• Keep updates brief (30-60 seconds each)
• Note blockers but don't solve them
• Note discussions needed but defer them

AFTER:
• Immediately address blockers
• Schedule follow-up discussions
• Update the board

FACILITATION TIPS:
• "Let's save that discussion for after standup"
• "What's blocking this from moving forward?"
• "Who can help with this today?"
• "Let's keep moving-we can discuss offline"
```

6.8.3 Facilitating Effective Retrospectives

Structure (1-2 hours):

```
RETROSPECTIVE FACILITATION


1. SET THE STAGE (5-10 min)
   • Welcome, state purpose
   • Review working agreements
   • Check-in activity (how are people feeling?)

2. GATHER DATA (15-20 min)
   • What happened during the sprint?
   • Use a retrospective format (Start/Stop/Continue, 4Ls, etc.)
   • Silent brainstorming, then share
```

```
3. GENERATE INSIGHTS (15-20 min)
   • Why did these things happen?
   • Group related items
   • Identify patterns and root causes

4. DECIDE WHAT TO DO (15-20 min)
   • What specific actions will we take?
   • Limit to 1-3 actions (more won't get done)
   • Assign owners and due dates

5. CLOSE (5-10 min)
   • Summarize action items
   • Appreciate the team
   • Quick feedback on the retro itself
```

Example Retrospective Flow:

```
SPRINT 4 RETROSPECTIVE


FORMAT: Sailboat

          Island (Our Goals)
         "Ship MVP by end of month"


      Team (Current Position)


    Anchors (What's slowing us down?)
      • Unclear requirements on 2 stories
      • Staging environment unstable
      • Too many meetings

    Wind (What's pushing us forward?)
      • Great collaboration this sprint
      • New CI pipeline saving time
      • Customer feedback very helpful

    Rocks (Risks ahead?)
      • Key developer vacation next sprint
      • Integration with payment system unknown

ACTION ITEMS:
1. [Product Owner] Refine next sprint stories by Thursday
2. [DevOps] Fix staging stability by Monday
3. [Scrum Master] Audit meeting calendar, propose reductions
```

6.8.4 Remote/Hybrid Agile

Many teams now work remotely or hybrid. Agile practices need adaptation:

Remote Standup Tips:

- Video on (builds connection)
- Use a shared board everyone can see
- Strict time-keeping (easier to run long remotely)
- Consider async standups for distributed time zones

Remote Retrospective Tips:

- Use digital whiteboarding tools (Miro, FigJam, MURAL)
- More structure (harder to read the room remotely)
- Breakout rooms for small group discussions
- Extra attention to psychological safety

Async Standups:

```
ASYNC STANDUP (via Slack/Teams)



Post by 9:30 AM local time:

1. What did you complete yesterday?
2. What are you working on today?
3. Any blockers?

Example:
@alice:
1.  Completed Task CRUD API
2.  Starting task assignment feature
3.  None

@bob:
1.  Fixed date picker bug
2.  Working on form validation
3.  Blocked: Need design clarification on error states

Thread replies for questions or offers to help.
```

6.9 Adapting Agile to Your Context

Agile isn't one-size-fits-all. Teams must adapt practices to their specific context.

6.9.1 Team Size Considerations

Solo Developer:

- Kanban often works better than Scrum
- No standups needed (but planning and review still valuable)
- Personal Kanban board for visualization
- Time-boxed work sessions (Pomodoro)

Small Team (2-4):

- Lighter ceremonies
- Roles may overlap (developer might also be Product Owner)
- Simple tools sufficient
- Stand-ups can be very quick

Medium Team (5-9):

- Full Scrum works well
- All roles dedicated
- More structure needed
- Better tooling helpful

Large Team (10+):

- Split into multiple teams
- Need coordination mechanisms (Scrum of Scrums, scaled frameworks)
- More formal processes
- Robust tooling essential

6.9.2 Project Type Considerations

New Product Development:

- High uncertainty → Scrum's iterative approach
- Frequent pivots → Short sprints
- Heavy user involvement

Maintenance/Operations:

- Continuous flow → Kanban
- Unpredictable work → WIP limits
- Mix of planned and unplanned work

Fixed-Deadline Projects:

- Release planning critical
- Velocity tracking for forecasting
- Scope management key

Research/Experimental:

- Time-boxed experiments (spikes)
- High uncertainty acknowledgment
- Learning over delivery

6.9.3 Scaling Agile

When multiple teams work on the same product, coordination frameworks help:

Scrum of Scrums:

- Representatives from each team meet daily/weekly
- Share progress, dependencies, and blockers
- Coordinate across teams

SAFe (Scaled Agile Framework):

- Comprehensive scaling framework
- Program Increments (8-12 weeks)
- Multiple teams, roles, and ceremonies
- Works for very large organizations

LeSS (Large-Scale Scrum):

- Minimal additional process
- Multiple teams, one Product Backlog
- Shared Sprint Review and Retrospective

Spotify Model:

- Squads (small teams)
- Tribes (groups of related squads)
- Chapters (people with same skills across squads)
- Guilds (communities of interest)

For most student projects, scaling isn't needed. Focus on core Agile practices first.

6.10 Chapter Summary

Agile methodologies revolutionized software development by embracing change, valuing people, and delivering working software frequently. Understanding these practices is essential for modern software engineers.

Key takeaways from this chapter:

- **The Agile Manifesto** established values prioritizing individuals, working software, collaboration, and responding to change. These values guide all Agile practices.

- **Scrum** is a framework with defined roles (Product Owner, Scrum Master, Developers), events (Sprint Planning, Daily Scrum, Sprint Review, Retrospective), and artifacts (Product Backlog, Sprint Backlog, Increment).
- **Kanban** focuses on visualizing work, limiting work in progress, and optimizing flow. It's less prescriptive than Scrum and works well for continuous-flow environments.
- **Extreme Programming (XP)** emphasizes technical excellence through practices like pair programming, test-driven development, and continuous integration.
- **Story points** provide relative estimation that accounts for uncertainty. Planning Poker builds consensus while avoiding anchoring bias.
- **Velocity** measures how much work a team completes per sprint, enabling forecasting and capacity planning.
- **Project management tools** like GitHub Projects and Jira support Agile workflows with boards, backlogs, and reporting.
- **Work breakdown** flows from Epics (large features) to Stories (deliverable value) to Tasks (technical work). Definition of Ready and Definition of Done ensure quality.
- **Effective meetings** require preparation, facilitation, and follow-through. Anti-patterns like status-report standups undermine Agile benefits.
- **Adaptation** is key—teams should modify Agile practices to fit their context, team size, and project type.

6.11 Key Terms

Term	Definition
Agile	A mindset and set of values prioritizing individuals, working software, collaboration, and responding to change
Scrum	An Agile framework using sprints, defined roles, and ceremonies
Sprint	A fixed time-box (typically 2 weeks) for delivering an increment
Product Backlog	Ordered list of everything that might be needed in the product
Sprint Backlog	Items selected for the sprint plus a plan for delivering them
Increment	The sum of all completed items, in usable condition
Velocity	Story points completed per sprint, used for planning

Term	Definition
Kanban	A method focusing on visualizing work, limiting WIP, and managing flow
WIP Limit	Maximum items allowed in a workflow stage
Cycle Time	Time from work started to work completed
Story Points	Relative measure of effort, complexity, and uncertainty
Epic	Large body of work spanning multiple sprints
Definition of Done	Shared criteria for when work is complete
Retrospective	Meeting to inspect the process and identify improvements
Planning Poker	Consensus-based estimation technique

6.12 Review Questions

1. Explain the four values of the Agile Manifesto. How does each value translate into practical behavior for software teams?

2. Compare and contrast Scrum and Kanban. When would you choose one over the other?

3. Describe the three Scrum roles and their responsibilities. Why is it important that the Product Owner is a single person rather than a committee?

4. What is the purpose of the Daily Scrum? How does it differ from a traditional status meeting?

5. Explain the concept of story points. Why are they preferred over time-based estimates?

6. What is velocity, and how should it (and shouldn't it) be used?

7. Describe the Sprint Retrospective. What makes a retrospective effective versus ineffective?

8. What is a WIP limit, and why is it important in Kanban?

9. Explain the difference between Epics, User Stories, and Tasks. How do they relate to each other?

10. What is the Definition of Done, and why is it important for teams to have one?

6.13 Hands-On Exercises

Exercise 6.1: Product Backlog Creation

For your semester project:

1. Identify at least 3 Epics that represent major features
2. Break each Epic into 5-8 User Stories
3. Write each story in the "As a… I want… so that…" format
4. Add acceptance criteria to each story
5. Order the backlog by priority

Exercise 6.2: Story Point Estimation

Conduct a Planning Poker session:

1. Select a reference story and assign it 3 points
2. As a team, estimate at least 10 stories
3. Document any significant discussions that arose
4. Calculate total backlog size in points
5. Estimate how many sprints to complete the backlog (assuming a reasonable velocity)

Exercise 6.3: Sprint Planning

Plan your first sprint:

1. Determine sprint length (recommend 2 weeks)
2. Estimate team capacity (available hours × focus factor)
3. Select stories from the top of the backlog totaling your target velocity
4. Break each story into tasks
5. Create a Sprint Goal
6. Document your Sprint Backlog

Exercise 6.4: Kanban Board Setup

Set up a project board in GitHub Projects:

1. Create columns matching your workflow:
    - Backlog
    - Ready (refined, ready to start)
    - In Progress
    - In Review
    - Done
2. Add WIP limits to appropriate columns

3. Create cards for all your user stories
4. Move cards to appropriate columns based on current status
5. Add labels for Epics, priority, and type

Exercise 6.5: Sprint Simulation

Simulate running a sprint:

1. Conduct a Sprint Planning meeting (30-60 minutes)
2. For one week, conduct daily standups (async is fine):
    - Post daily updates
    - Track blockers
    - Move cards on your board
3. At the end of the week, conduct:
    - Sprint Review (demonstrate completed work)
    - Sprint Retrospective (identify improvements)
4. Document lessons learned

Exercise 6.6: Retrospective Facilitation

Practice facilitating a retrospective:

1. Choose a retrospective format (Start-Stop-Continue, 4Ls, Sailboat)
2. Prepare materials (digital whiteboard or physical supplies)
3. Facilitate a 45-minute retrospective with your team or classmates
4. Generate at least 3 specific, actionable improvements
5. Assign owners and deadlines
6. Reflect on what made the retrospective effective or challenging

Exercise 6.7: Agile Sprint Plan Document

Create a formal Sprint Plan document including:

1. Project overview and Sprint Goal
2. Team capacity calculation
3. Sprint Backlog with stories and tasks
4. Risk and dependency identification
5. Definition of Done for the sprint
6. Meeting schedule (standups, review, retrospective)
7. Success criteria

6.14 Further Reading

Books:

- Schwaber, K., & Sutherland, J. (2020). *The Scrum Guide.* Scrum.org. (Free download)
- Sutherland, J. (2014). *Scrum: The Art of Doing Twice the Work in Half the Time.* Crown Business.
- Anderson, D. (2010). *Kanban: Successful Evolutionary Change for Your Technology Business.* Blue Hole Press.
- Beck, K. (2004). *Extreme Programming Explained: Embrace Change* (2nd Edition). Addison-Wesley.
- Cohn, M. (2005). *Agile Estimating and Planning.* Prentice Hall.
- Derby, E., & Larsen, D. (2006). *Agile Retrospectives: Making Good Teams Great.* Pragmatic Bookshelf.

Online Resources:

- The Scrum Guide: https://scrumguides.org/
- Agile Manifesto: https://agilemanifesto.org/
- Mountain Goat Software (Scrum resources): https://www.mountaingoatsoftware.com/
- Kanban University: https://kanban.university/
- GitHub Projects Documentation: https://docs.github.com/en/issues/planning-and-tracking-with-projects

---

References

Anderson, D. J. (2010). *Kanban: Successful Evolutionary Change for Your Technology Business.* Blue Hole Press.

Beck, K. (2004). *Extreme Programming Explained: Embrace Change* (2nd Edition). Addison-Wesley.

Beck, K., et al. (2001). Manifesto for Agile Software Development. Retrieved from https://agilemanifesto.org/

Cohn, M. (2005). *Agile Estimating and Planning.* Prentice Hall.

Derby, E., & Larsen, D. (2006). *Agile Retrospectives: Making Good Teams Great.* Pragmatic Bookshelf.

Grenning, J. (2002). Planning Poker or How to Avoid Analysis Paralysis while Release Planning. Retrieved from https://wingman-sw.com/articles/planning-poker

Schwaber, K., & Sutherland, J. (2020). *The Scrum Guide.* Retrieved from https://scrumguides.org/

Sutherland, J. (2014). *Scrum: The Art of Doing Twice the Work in Half the Time.* Crown Business.

Chapter 7: Version Control Workflows

Learning Objectives

By the end of this chapter, you will be able to:

- Explain the importance of structured version control workflows in team environments
- Compare and contrast major branching strategies including Gitflow, GitHub Flow, and trunk-based development
- Create, manage, and merge branches effectively using Git
- Write meaningful pull requests that facilitate effective code review
- Conduct thorough, constructive code reviews
- Resolve merge conflicts confidently and correctly
- Maintain repository hygiene through proper documentation and conventions
- Choose appropriate branching strategies for different project contexts

---

7.1 Why Version Control Workflows Matter

In Chapter 1, we introduced Git and GitHub as tools for tracking changes and collaborating on code. But knowing Git commands is only the beginning. When multiple developers work on the same codebase simultaneously, chaos can ensue without agreed-upon workflows. Who can commit to which branch? How do changes get reviewed? What happens when two people modify the same file?

A **version control workflow** is a set of conventions and practices that define how a team uses version control. It answers questions like:

- How do we organize our branches?
- How do changes move from development to production?
- Who reviews code, and when?
- How do we handle releases and hotfixes?

7.1.1 The Cost of Poor Version Control

Without structured workflows, teams encounter predictable problems:

Integration nightmares: Developers work in isolation for weeks, then try to merge everything at once. Massive conflicts result, and subtle bugs slip through as incompatible changes collide.

Unstable main branch: Without protection, broken code gets committed directly to main. The build fails. Nobody can deploy. Everyone's blocked.

Lost work: Without proper branching, experimental changes get mixed with stable code. Rolling back becomes impossible without losing good work too.

No accountability: Without code review, bugs slip into production. Nobody catches security vulnerabilities, performance problems, or architectural violations until they cause real damage.

Release chaos: Without clear release processes, teams don't know what's deployed where. Hot-fixes go to the wrong version. Customers get inconsistent experiences.

7.1.2 What Good Workflows Provide

Structured workflows address these problems:

Isolation: Developers work on separate branches, insulated from each other's in-progress changes. Integration happens deliberately, not accidentally.

Stability: The main branch stays deployable. Broken code never reaches it because changes must pass tests and review first.

Traceability: Every change is linked to a purpose—a feature, a bug fix, a task. History tells the story of why the code evolved.

Quality: Code review catches bugs, shares knowledge, and maintains standards. Multiple eyes improve quality.

Confidence: Clear processes mean everyone knows what to do. Deployments become routine, not risky adventures.

7.2 Understanding Git Branching

Before exploring workflows, let's deepen our understanding of Git branching—the foundation on which all workflows build.

7.2.1 What Is a Branch?

A **branch** in Git is simply a lightweight movable pointer to a commit. When you create a branch, Git creates a new pointer; it doesn't copy any files. This makes branching fast and cheap.

`main`

`feature`

In this diagram:

- Each is a commit
- `main` points to the latest commit on the main line
- `feature` points to the latest commit on the feature branch
- The branches share history up to where they diverged

7.2.2 HEAD: Where You Are

HEAD is a special pointer that indicates your current position—which branch (and commit) you're working on.

`HEAD`

`main`

`feature`

When you checkout a different branch, HEAD moves:

```
git checkout feature
```

`main`

feature

HEAD

7.2.3 Branch Operations

Creating a Branch:

```
# Create a new branch
git branch feature-login

# Create and switch to a new branch
git checkout -b feature-login

# Modern alternative (Git 2.23+)
git switch -c feature-login
```

Switching Branches:

```
# Traditional
git checkout main

# Modern alternative
git switch main
```

Listing Branches:

```
# List local branches
git branch

# List all branches (including remote)
git branch -a

# List with last commit info
git branch -v
```

Deleting Branches:

```
# Delete a merged branch
git branch -d feature-login

# Force delete an unmerged branch
git branch -D experimental-feature
```

7.2.4 Merging Branches

Merging combines the changes from one branch into another. Git supports several merge strategies.

Fast-Forward Merge:

When the target branch hasn't diverged, Git simply moves the pointer forward:

```
Before:
    main







                    feature

After git checkout main && git merge feature:

                       main





                    feature
```

No merge commit is created—history stays linear.

Three-Way Merge:

When branches have diverged, Git creates a merge commit with two parents:

```
Before:
              main
```

```
                    feature

After git checkout main && git merge feature:

                        main


                    (merge commit)





                    feature
```

Merge Commands:

```
# Merge feature into current branch (main)
git checkout main
git merge feature-login

# Merge with a commit message
git merge feature-login -m "Merge feature-login into main"

# Abort a merge in progress
git merge --abort
```

7.2.5 Rebasing

Rebasing rewrites history by moving commits to a new base. Instead of a merge commit, rebase creates a linear history.

```
Before:
              main








                    feature
```

```
After git checkout feature && git rebase main:

                   main



                ' '  '



                          feature
```

The commits on feature are recreated (’ indicates new commits with same changes but different hashes).

Rebase Commands:

```
# Rebase current branch onto main
git rebase main

# Interactive rebase (edit, squash, reorder commits)
git rebase -i main

# Abort a rebase in progress
git rebase --abort

# Continue after resolving conflicts
git rebase --continue
```

Merge vs. Rebase:

Aspect	Merge	Rebase
History	Preserves true history	Creates linear history
Merge commits	Creates merge commits	No merge commits
Conflict resolution	Once per merge	Once per rebased commit
Safety	Safe for shared branches	Don’t rebase shared branches
Traceability	Shows when branches joined	Hides branch structure

The Golden Rule of Rebasing:

> Never rebase commits that have been pushed to a public/shared branch.

Rebasing rewrites history. If others have based work on commits you rebase, their history diverges from yours, causing major problems.

7.3 Branching Strategies

A **branching strategy** defines how teams organize and use branches. Different strategies suit different team sizes, release cycles, and risk tolerances.

7.3.1 Gitflow

Gitflow, introduced by Vincent Driessen in 2010, is a comprehensive branching model designed for projects with scheduled releases.

Branch Types:

```
                          GITFLOW BRANCHES


  MAIN (main/master)
  • Production-ready code
  • Tagged with version numbers
  • Only receives merges from release and hotfix branches

  DEVELOP (develop)
  • Integration branch for features
  • Contains latest delivered development changes
  • Features branch from and merge back to develop

  FEATURE (feature/*)
  • New features and non-emergency fixes
  • Branch from: develop
  • Merge to: develop
  • Naming: feature/user-authentication, feature/payment-gateway

  RELEASE (release/*)
  • Preparation for production release
  • Branch from: develop
  • Merge to: main AND develop
  • Naming: release/1.2.0, release/2.0.0

  HOTFIX (hotfix/*)
  • Emergency fixes for production
  • Branch from: main
  • Merge to: main AND develop
  • Naming: hotfix/critical-security-fix, hotfix/payment-bug
```

Visual Representation:

```
main
           \                     / \               /      /
            \                   /   \             /      /
release      \                  \           /      /
              \      /               \         /      /
               \    /                 \       /      /
develop
                \      /          \        /
                 \    /            \      /
feature
                                          \
                                           \
hotfix
```

Gitflow Workflow:

Starting a new feature:

```bash
# Create feature branch from develop
git checkout develop
git checkout -b feature/user-authentication

# Work on feature...
git add .
git commit -m "Add login form"

# Continue working...
git commit -m "Add authentication API"

# Finish feature
git checkout develop
git merge feature/user-authentication
git branch -d feature/user-authentication
```

Creating a release:

```bash
# Create release branch from develop
git checkout develop
git checkout -b release/1.2.0

# Bump version numbers, final testing, documentation
git commit -m "Bump version to 1.2.0"

# Finish release
git checkout main
git merge release/1.2.0
git tag -a v1.2.0 -m "Release version 1.2.0"
git checkout develop
git merge release/1.2.0

git branch -d release/1.2.0
```

Creating a hotfix:

```
# Create hotfix branch from main
git checkout main
git checkout -b hotfix/critical-security-fix

# Fix the issue
git commit -m "Fix SQL injection vulnerability"

# Finish hotfix
git checkout main
git merge hotfix/critical-security-fix
git tag -a v1.2.1 -m "Hotfix release 1.2.1"

git checkout develop
git merge hotfix/critical-security-fix

git branch -d hotfix/critical-security-fix
```

Gitflow Pros and Cons:

Pros	Cons
Clear structure for releases	Complex with many branch types
Parallel development and release	Slow for continuous deployment
Hotfix path separate from features	Merge conflicts between long-lived branches
Good for versioned software	Overhead for small teams
Well-documented, widely understood	develop can become stale

When to Use Gitflow:

- Software with explicit version releases
- Multiple versions in production
- Teams with dedicated release management
- Products requiring extensive release testing
- Compliance environments requiring audit trails

7.3.2 GitHub Flow

GitHub Flow is a simpler workflow designed for continuous deployment. Created at GitHub, it has only one rule: anything in `main` is deployable.

Branch Types:

```
                        GITHUB FLOW BRANCHES

  MAIN (main)
  • Always deployable
  • Protected-no direct commits
  • All changes come through pull requests

  FEATURE BRANCHES (descriptive names)
  • All work happens in feature branches
  • Branch from: main
  • Merge to: main (via pull request)
  • Naming: descriptive (add-user-auth, fix-payment-bug)
```

Visual Representation:

```
main
               \    /    \   /         \        /
                \  /      \ /           \      /
feature
```

GitHub Flow Process:

```
                        GITHUB FLOW PROCESS

  1. CREATE A BRANCH
     • Branch from main
     • Use descriptive name

  2. ADD COMMITS
     • Make changes
     • Commit frequently with clear messages
     • Push to remote regularly

  3. OPEN A PULL REQUEST
     • Start discussion about changes
     • Request review from teammates
     • CI runs tests automatically
```

```
4. DISCUSS AND REVIEW
   • Reviewers leave comments
   • Author addresses feedback
   • More commits as needed

5. DEPLOY (optional)
   • Deploy branch to test environment
   • Verify in production-like setting

6. MERGE
   • Merge to main after approval
   • Delete the feature branch
   • Main is deployed to production
```

GitHub Flow Commands:

```
# 1. Create a branch
git checkout main
git pull origin main
git checkout -b add-password-reset

# 2. Make changes and commit
git add .
git commit -m "Add password reset request form"
git commit -m "Add password reset email functionality"
git commit -m "Add password reset confirmation page"

# Push to remote (enables PR and backup)
git push -u origin add-password-reset

# 3. Open Pull Request (on GitHub)
# - Write description
# - Request reviewers
# - Link to issue

# 4. Address review feedback
git add .
git commit -m "Address review feedback: add rate limiting"
git push

# 5. After approval, merge via GitHub UI

# 6. Clean up locally
git checkout main
git pull origin main
git branch -d add-password-reset
```

GitHub Flow Pros and Cons:

Pros	Cons
Simple—only two branch types	No explicit release management
Fast—optimized for continuous deployment	Requires robust CI/CD
Pull requests enable code review	Less structure for versioned releases
Works great for web applications	Hotfixes indistinguishable from features
Low overhead	May need extensions for complex projects

When to Use GitHub Flow:

- Web applications with continuous deployment
- Small to medium teams
- Products without versioned releases
- Teams practicing continuous integration
- Projects prioritizing simplicity

7.3.3 Trunk-Based Development

Trunk-based development (TBD) takes simplicity further: all developers commit to a single branch (the "trunk," typically `main`), either directly or through very short-lived feature branches.

Core Principles:

```
                    TRUNK-BASED DEVELOPMENT


  1. SINGLE SOURCE OF TRUTH
     • All code integrates to main/trunk
     • No long-lived branches

  2. FREQUENT INTEGRATION
     • Integrate at least daily
     • Small, incremental changes

  3. FEATURE FLAGS
     • Hide incomplete features in production
     • Deploy code before features are complete

  4. RELEASE FROM TRUNK
     • Create release branches only if needed
     • Tag releases on trunk
```

Visual Representation:

```
With short-lived branches (< 1 day):

main
              / \        / \       |    \      /
             /   \      /   \      |     \    /
feature

Direct commits (pair programming):

main
           Alice Bob  Both  Alice Bob  Alice  Bob  Both
```

Feature Flags:

Since incomplete features merge to main, feature flags hide them from users:

```python
# Feature flag example
if feature_flags.is_enabled('new_checkout_flow', user):
    return render_new_checkout(cart)
else:
    return render_old_checkout(cart)
```

Feature flags allow:

- Deploying incomplete code safely
- Gradual rollouts (1% → 10% → 50% → 100% of users)
- Quick rollback without code changes
- A/B testing

Trunk-Based Development Commands:

```bash
# Option 1: Direct to main (with pair programming/mob programming)
git checkout main
git pull
# Make small change
git add .
git commit -m "Add email validation to signup form"
git pull --rebase  # Get others' changes
git push

# Option 2: Short-lived branch (< 1 day)
git checkout main
git pull
git checkout -b small-fix
# Make changes
```

```
git add .
git commit -m "Fix typo in error message"
git checkout main
git pull
git merge small-fix
git push
git branch -d small-fix
```

Trunk-Based Development Pros and Cons:

Pros	Cons
Continuous integration by definition	Requires strong testing discipline
No merge hell from long-lived branches	Feature flags add complexity
Faster feedback on integration issues	Direct commits require senior team
Simpler mental model	Incomplete features visible in codebase
Enables continuous deployment	Less isolation for experimental work

When to Use Trunk-Based Development:

- Teams with excellent test coverage
- Strong CI/CD pipeline
- Senior, disciplined developers
- Products requiring very fast iteration
- Organizations practicing DevOps/continuous deployment

7.3.4 Comparing Branching Strategies

Aspect	Gitflow	GitHub Flow	Trunk-Based
Branch types	5 (main, develop, feature, release, hotfix)	2 (main, feature)	1-2 (main, optional short-lived)
Complexity	High	Low	Very low
Release style	Scheduled releases	Continuous	Continuous
Integration frequency	When feature complete	At PR merge	Multiple times daily
Best for	Versioned products	Web apps	High-velocity teams
Feature isolation	High	Medium	Low (use feature flags)
CI/CD requirement	Helpful	Important	Essential

7.3.5 Choosing a Strategy

Consider these factors when choosing:

Team size and experience:

- Small/senior team → Trunk-based or GitHub Flow
- Large/mixed experience → Gitflow or GitHub Flow

Release cadence:

- Continuous deployment → GitHub Flow or Trunk-based
- Scheduled releases → Gitflow
- Multiple versions in production → Gitflow

Product type:

- Web application → GitHub Flow or Trunk-based
- Mobile app (app store releases) → Gitflow
- Packaged software → Gitflow
- Internal tools → GitHub Flow

Risk tolerance:

- High risk tolerance, fast iteration → Trunk-based
- Low risk tolerance, careful releases → Gitflow
- Balanced → GitHub Flow

For your course project, GitHub Flow is recommended—it's simple enough to learn quickly while teaching important practices like pull requests and code review.

7.4 Pull Requests

A **pull request** (PR) is a request to merge changes from one branch into another. More than just a merge mechanism, pull requests are collaboration tools that enable discussion, review, and quality control.

7.4.1 Anatomy of a Good Pull Request

```
Add user authentication system                               #142


alice wants to merge 5 commits into main from feature/user-auth



   ## Summary

   This PR adds a complete user authentication system including:
   - User registration with email verification
- Login/logout functionality
- Password reset via email
- Session management with JWT tokens

## Related Issues

Closes #98
Relates to #95, #96

## Changes

- Added User model with email, password hash, and verification
- Implemented AuthService with register, login, logout methods
- Created auth API endpoints (/register, /login, /logout, etc.)
- Added JWT middleware for protected routes
- Integrated SendGrid for verification and reset emails
- Added rate limiting to prevent brute force attacks

## Testing

- Added 45 unit tests for AuthService (100% coverage)
- Added 12 integration tests for auth endpoints
- Manual testing checklist completed (see below)

## Screenshots

[Login Form Screenshot]
[Registration Flow GIF]

## Checklist

- [x] Code follows project style guide
- [x] Tests added/updated
- [x] Documentation updated
- [x] No console.log or debug code
- [x] Tested on Chrome, Firefox, Safari
```

```
Reviewers: @bob @carol                  Labels: feature, auth
Assignees: @alice                       Milestone: MVP
```

7.4.2 Writing Effective PR Descriptions

Title:

- Clear and concise
- Describes what the PR does (not how)
- Often starts with a verb: "Add…", "Fix…", "Update…", "Remove…"

Description Template:

```
## Summary
Brief description of what this PR does and why.

## Related Issues
- Closes #123
- Relates to #456

## Changes
- Bullet points describing specific changes
- Focus on the "what" and "why"
- Group related changes together

## Testing
- How was this tested?
- Any specific testing instructions for reviewers?
- Test coverage information

## Screenshots/GIFs (if applicable)
Visual demonstration of UI changes

## Checklist
- [ ] Code follows style guide
- [ ] Tests added/updated
- [ ] Documentation updated
- [ ] Self-review completed
- [ ] Ready for review

## Notes for Reviewers
Any specific areas you'd like extra attention on?
Any known issues or trade-offs?
```

7.4.3 Pull Request Best Practices

Keep PRs Small:

Small PRs are easier to review, faster to merge, and less risky. Aim for:

- Under 400 lines of changes
- One logical change per PR
- Completable in 1-2 days

```
PR Size Guidelines:

< 100 lines    → Easy to review, quick turnaround
100-300 lines  → Reasonable, standard PR
300-500 lines  → Large, may need splitting
> 500 lines    → Too large, definitely split

Exceptions:
- Generated code
- Data migrations
- Dependency updates
```

Splitting Large PRs:

```
Instead of one giant PR:
"Add complete user management system" (2000+ lines)

Split into:
1. "Add User model and database migration" (~150 lines)
2. "Add user registration endpoint" (~200 lines)
3. "Add user login endpoint" (~200 lines)
4. "Add user profile endpoints" (~250 lines)
5. "Add user management UI" (~300 lines)

Each PR is reviewable and independently mergeable.
```

Self-Review Before Requesting Review:

Before requesting review:

1. Read through all your changes in the GitHub diff
2. Check for debugging code, console.logs, TODOs
3. Verify tests pass locally
4. Ensure documentation is updated
5. Add comments explaining non-obvious code

Respond to Feedback Promptly:

- Acknowledge comments even if you need time to address them
- Explain your reasoning if you disagree (respectfully)
- Push new commits to address feedback
- Re-request review when ready

7.4.4 Draft Pull Requests

Draft PRs indicate work-in-progress that isn't ready for review. Use them to:

- Get early feedback on approach
- Show progress to teammates
- Run CI before formal review
- Discuss design decisions

```
 [DRAFT] Add user authentication system                  #142


This pull request is still a work in progress.

## Current Status
- [x] User model
- [x] Registration endpoint
- [ ] Login endpoint (in progress)
- [ ] Password reset
- [ ] Tests

## Questions for Team
- Should we use JWT or session cookies?
- What's our password policy?

Not ready for formal review yet, but feedback on approach welcome!
```

Convert to a regular PR when ready for review.

7.5 Code Review

Code review is the practice of having team members examine code changes before they're merged. It's one of the most valuable practices in software engineering, improving code quality, spreading knowledge, and catching bugs early.

7.5.1 Why Code Review Matters

Quality Improvement:

- Catches bugs before production

- Identifies security vulnerabilities
- Ensures code meets standards
- Improves design and architecture

Knowledge Sharing:

- Spreads knowledge across the team
- New team members learn the codebase
- Senior developers mentor juniors
- No single point of failure

Accountability and Ownership:

- Multiple people understand each change
- Shared responsibility for quality
- Documentation through review comments

7.5.2 The Reviewer's Mindset

Good reviewers approach code with:

Empathy: Someone worked hard on this. Be kind and constructive.

Curiosity: Why was this approach chosen? What am I missing?

Rigor: Don't rubber-stamp. Actually read and think about the code.

Humility: The author may know something you don't. Ask questions rather than assuming bugs.

Efficiency: Don't make the author wait. Review promptly.

7.5.3 What to Look For

Functionality:

- Does the code do what it's supposed to do?
- Are edge cases handled?
- What happens with unexpected input?

Code Quality:

- Is the code readable and maintainable?
- Are names clear and meaningful?
- Is there unnecessary complexity?
- Is there duplicated code?

Design:

- Does the approach make sense?
- Does it fit with existing architecture?

- Are there better alternatives?
- Is it extensible for future needs?

Testing:

- Are there sufficient tests?
- Do tests cover edge cases?
- Are tests readable and maintainable?

Security:

- Are inputs validated?
- Is sensitive data protected?
- Are there injection vulnerabilities?
- Is authentication/authorization correct?

Performance:

- Are there obvious performance issues?
- Database queries efficient?
- Memory leaks possible?

Documentation:

- Is complex code explained?
- Are public APIs documented?
- Is README updated if needed?

7.5.4 Review Checklist

```
CODE REVIEW CHECKLIST



UNDERSTANDING
  I understand what this PR is supposed to do
  The PR description explains the changes clearly
  Related issues are linked

FUNCTIONALITY
  Code achieves the stated goal
  Edge cases are handled
  Error handling is appropriate
  No obvious bugs

DESIGN
  Approach is reasonable
  Consistent with existing patterns
  No unnecessary complexity
  Changes are in appropriate locations
```

```
CODE QUALITY
 Code is readable
 Names are clear and meaningful
 No dead/commented code
 No debugging code (console.log, etc.)
 Follows project style guide

TESTING
 Tests exist for new functionality
 Tests cover important cases
 Tests are readable
 All tests pass

SECURITY
 No obvious security issues
 Inputs are validated
 Sensitive data is handled properly

DOCUMENTATION
 Complex code is commented
 API documentation updated
 README updated if needed
```

7.5.5 Giving Feedback

Be Specific:

```
 "This code is confusing."

 "I found the logic in processOrder() hard to follow. Consider
  extracting the discount calculation into a separate function
  like calculateDiscount(order)."
```

Explain Why:

```
 "Use const instead of let here."

 "Use const instead of let here since items isn't reassigned.
  Using const signals intent and prevents accidental reassignment."
```

Ask Questions:

```
 "This is wrong."
 "I'm not sure I understand the logic here. What happens if
   the user has no orders? Wouldn't orders.length be 0, making
   the average calculation divide by zero?"
```

Offer Suggestions:

```
 "This could be better."

 "Consider using array.find() instead of the for loop:

   const user = users.find(u => u.id === targetId);

   This is more concise and expresses intent more clearly."
```

Distinguish Importance:

Use prefixes to indicate severity:

```
[blocking] - Must be fixed before merge
[suggestion] - Optional improvement
[question] - Seeking understanding
[nit] - Minor/stylistic, very optional
[praise] - Positive feedback!
```

Examples:

```
[blocking] This SQL query is vulnerable to injection. Please use
parameterized queries.

[suggestion] Consider extracting this into a helper function. It
would improve readability and allow reuse.

[question] Why did you choose to load all users into memory? With
large datasets this could be slow-was that considered?

[nit] Extra blank line here.

[praise] Great test coverage! The edge case tests are especially
thorough.
```

7.5.6 Receiving Feedback

Don't Take It Personally:

- Review is about the code, not you
- Feedback is a gift that improves your work

- Everyone's code can be improved

Understand Before Responding:

- Re-read comments to make sure you understand
- Ask for clarification if needed
- Consider the reviewer's perspective

Engage Constructively:

- Thank reviewers for their feedback
- Explain your reasoning if you disagree
- Accept valid criticism gracefully

Response Examples:

```
Reviewer: "This function is doing too much. Consider splitting it."

Response Options:

  "Good point! I've split it into validateInput() and
   processRequest(). Much cleaner now."

  "I considered splitting it, but keeping it together lets us
   do X more efficiently. Here's my reasoning... What do you think?"

  "Could you clarify what you mean? Are you suggesting splitting
   by responsibility or by... something else?"

  "It's fine how it is."
  "I don't have time to refactor this."
```

7.5.7 Review Etiquette

For Reviewers:

- Review within 24 hours (ideally same day)
- Be thorough but not pedantic
- Approve when it's good enough (not perfect)
- Follow up on addressed comments
- Thank authors for good code

For Authors:

- Respond to all comments
- Don't argue every point
- Make requested changes promptly
- Re-request review when ready
- Thank reviewers for their time

7.5.8 Automated Checks

Automated tools complement human review:

CI/CD Checks:

- Tests pass
- Build succeeds
- Code coverage maintained

Linting:

- Code style consistency
- Potential bugs (unused variables, etc.)
- Formatting

Security Scanning:

- Dependency vulnerabilities
- Code security issues

Configuring Required Checks:

```yaml
# .github/workflows/ci.yml
name: CI
on: [pull_request]

jobs:
  test:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3
      - name: Install dependencies
        run: npm install
      - name: Run linter
        run: npm run lint
      - name: Run tests
        run: npm test
      - name: Check coverage
        run: npm run coverage
```

7.6 Handling Merge Conflicts

Merge conflicts occur when Git can't automatically combine changes—usually because two branches modified the same lines of code. Conflicts are normal and manageable with the right approach.

7.6.1 Why Conflicts Happen

```
Scenario: Two developers modify the same file

Initial state (main):

  function greet(name) {
    return "Hello, " + name;
  }


Alice's branch:

  function greet(name) {
    return `Hello, ${name}!`;               ← Changed to template literal
  }


Bob's branch:

  function greet(name) {
    return "Hi, " + name;                   ← Changed "Hello" to "Hi"
  }


Alice merges first. Now when Bob tries to merge:
CONFLICT! Git doesn't know which change to keep.
```

7.6.2 Anatomy of a Conflict

When a conflict occurs, Git marks the conflicting sections:

```
function greet(name) {
<<<<<<< HEAD
  return `Hello, ${name}!`;
=======
  return "Hi, " + name;
>>>>>>> feature/bobs-greeting
}
```

Understanding the markers:

- `<<<<<<< HEAD`: Start of current branch's version
- `=======`: Divider between versions
- `>>>>>>> feature/bobs-greeting`: End, with other branch name

7.6.3 Resolving Conflicts

Step 1: Identify conflicting files

```bash
git status

# Output:
# On branch main
# You have unmerged paths.
#   (fix conflicts and run "git commit")
#
# Unmerged paths:
#   (use "git add <file>..." to mark resolution)
#       both modified:   src/greet.js
```

Step 2: Open and edit conflicting files

Remove the conflict markers and create the correct merged version:

```javascript
// Before (conflicted):
function greet(name) {
<<<<<<< HEAD
  return `Hello, ${name}!`;
=======
  return "Hi, " + name;
>>>>>>> feature/bobs-greeting
}

// After (resolved-combining both changes):
function greet(name) {
  return `Hi, ${name}!`;  // Template literal + "Hi"
}
```

Step 3: Mark as resolved and commit

```bash
# Stage the resolved file
git add src/greet.js

# Complete the merge
git commit -m "Merge feature/bobs-greeting, combine greeting changes"
```

7.6.4 Conflict Resolution Strategies

Keep Current (Ours): Discard incoming changes, keep current branch's version:

```
git checkout --ours src/greet.js
git add src/greet.js
```

Keep Incoming (Theirs): Discard current branch's changes, keep incoming version:

```
git checkout --theirs src/greet.js
git add src/greet.js
```

Manual Merge: Edit the file to combine changes appropriately (most common).

Use a Merge Tool:

```
git mergetool
```

Opens a visual merge tool (configure with `git config merge.tool`).

7.6.5 Preventing Conflicts

While conflicts can't be entirely prevented, you can minimize them:

Integrate Frequently:

```
# Before starting work each day:
git checkout main
git pull
git checkout my-feature
git merge main  # Or: git rebase main
```

Keep Branches Short-Lived: Branches that live for weeks accumulate conflicts. Merge frequently.

Communicate: If you know you're working on the same area as someone else, coordinate.

Structure Code to Reduce Conflicts:

- Small, focused files
- Clear module boundaries
- Avoid monolithic files everyone edits

7.6.6 Conflict Resolution Example

Scenario: You're working on a feature branch and need to merge in changes from main.

```
# Your branch has been open for a few days
git checkout feature/user-profile
git status
# On branch feature/user-profile
# Your branch is ahead of 'origin/feature/user-profile' by 3 commits.

# Bring in latest main
git fetch origin
git merge origin/main

# Conflict!
# Auto-merging src/components/Header.jsx
# CONFLICT (content): Merge conflict in src/components/Header.jsx
# Automatic merge failed; fix conflicts and then commit the result.
```

Open the conflicted file:

```
// src/components/Header.jsx
import React from 'react';

function Header({ user }) {
  return (
    <header>
<<<<<<< HEAD
      <Logo size="large" />
      <nav>
        <NavLink to="/home">Home</NavLink>
        <NavLink to="/profile">Profile</NavLink>
        <NavLink to="/settings">Settings</NavLink>
      </nav>
      {user && <UserMenu user={user} />}
=======
      <Logo />
      <nav>
        <NavLink to="/home">Home</NavLink>
        <NavLink to="/dashboard">Dashboard</NavLink>
      </nav>
      {user && <Avatar user={user} />}
>>>>>>> origin/main
    </header>
  );
}
```

Analyze the conflict:

- Your branch: Added `size="large"` to Logo, added Profile/Settings links, uses UserMenu
- Main branch: Added Dashboard link, uses Avatar component

Resolve by combining:

```
// src/components/Header.jsx
import React from 'react';

function Header({ user }) {
  return (
    <header>
      <Logo size="large" />
      <nav>
        <NavLink to="/home">Home</NavLink>
        <NavLink to="/dashboard">Dashboard</NavLink>
        <NavLink to="/profile">Profile</NavLink>
        <NavLink to="/settings">Settings</NavLink>
      </nav>
      {user && <UserMenu user={user} />}
    </header>
  );
}
```

Complete the merge:

```
git add src/components/Header.jsx
git commit -m "Merge origin/main into feature/user-profile

- Combined navigation links from both branches
- Kept Logo size='large' from feature branch
- Kept UserMenu from feature branch (preferred over Avatar)"

git push
```

7.6.7 Handling Complex Conflicts

For complex conflicts, consider:

Talk to the other developer: "Hey, I see we both modified Header.jsx. Can we sync up on the intended behavior?"

Review both versions carefully:

```
# See what main changed
git diff main...origin/main -- src/components/Header.jsx

# See what your branch changed
git diff main...feature/user-profile -- src/components/Header.jsx
```

Run tests after resolving:

```
git add .
npm test  # Make sure nothing broke
git commit -m "Merge origin/main, resolve Header.jsx conflict"
```

When in doubt, ask for help: A second pair of eyes can catch mistakes in conflict resolution.

7.7 Repository Hygiene

A well-maintained repository is easier to navigate, understand, and contribute to. **Repository hygiene** encompasses conventions, documentation, and practices that keep repositories clean and professional.

7.7.1 Branch Naming Conventions

Consistent branch names communicate purpose and improve organization.

Common Patterns:

```
TYPE/DESCRIPTION

Types:
• feature/    - New features
• bugfix/     - Bug fixes
• hotfix/     - Urgent production fixes
• release/    - Release preparation
• docs/       - Documentation only
• refactor/   - Code refactoring
• test/       - Test additions/fixes
• chore/      - Maintenance tasks

Examples:
feature/user-authentication
feature/shopping-cart
bugfix/login-timeout
bugfix/date-format-error
hotfix/security-patch
release/2.1.0
docs/api-documentation
refactor/database-queries
test/payment-integration
chore/update-dependencies
```

Including Issue Numbers:

```
feature/123-user-authentication
bugfix/456-login-timeout
```

Naming Guidelines:

- Use lowercase
- Use hyphens, not underscores or spaces
- Keep names concise but descriptive
- Include ticket/issue number if available

7.7.2 Commit Message Conventions

Good commit messages explain what changed and why. They're documentation for the future.

The Seven Rules of Great Commit Messages:

1. Separate subject from body with a blank line
2. Limit the subject line to 50 characters
3. Capitalize the subject line
4. Do not end the subject line with a period
5. Use the imperative mood in the subject line
6. Wrap the body at 72 characters
7. Use the body to explain what and why vs. how

Commit Message Template:

```
<type>(<scope>): <subject>

<body>

<footer>
```

Types (Conventional Commits):

- `feat`: New feature
- `fix`: Bug fix
- `docs`: Documentation changes
- `style`: Formatting, missing semicolons, etc.
- `refactor`: Code change that neither fixes a bug nor adds a feature
- `test`: Adding or correcting tests
- `chore`: Maintenance tasks

Examples:

```
feat(auth): Add password reset functionality

Users can now reset their password via email. When a user clicks
"Forgot Password" on the login page, they receive an email with
a reset link valid for 24 hours.

- Add PasswordResetService with request and confirm methods
- Add /api/password-reset endpoints
- Add email templates for reset notification
- Add rate limiting (3 requests per hour)

Closes #234
```

```
fix(cart): Prevent negative quantities in shopping cart

Previously, users could enter negative numbers in the quantity
field, resulting in negative order totals. This commit adds
validation to ensure quantities are always >= 1.

Fixes #567
```

```
docs: Update API documentation for v2 endpoints

- Add examples for all new v2 endpoints
- Document breaking changes from v1
- Add authentication section with JWT examples
```

Short Form (for small changes):

```
fix: Correct typo in error message
style: Format code with Prettier
chore: Update dependencies
```

7.7.3 Essential Repository Files

README.md:

The README is often the first thing visitors see. It should include:

```markdown
# Project Name

Brief description of what the project does.

## Features

- Feature 1
```

~~~
- Feature 2
- Feature 3

## Getting Started

### Prerequisites

- Node.js 18+
- PostgreSQL 14+

### Installation

```bash
git clone https://github.com/username/project.git
cd project
npm install
cp .env.example .env
# Edit .env with your configuration
npm run setup
~~~

Running the Application

```
npm run dev       # Development mode
npm run build     # Production build
npm start         # Start production server
```

Documentation

- API Documentation
- Architecture Overview
- Contributing Guide

Contributing

Please read CONTRIBUTING.md for details on our code of conduct and the process for submitting pull requests.

License

This project is licensed under the MIT License - see the LICENSE file for details.

~~~
**CONTRIBUTING.md:**

```markdown
# Contributing to Project Name

Thank you for considering contributing!

## Development Setup

1. Fork the repository
2. Clone your fork
3. Create a branch: `git checkout -b feature/your-feature`
4. Make your changes
5. Run tests: `npm test`
6. Commit: `git commit -m "feat: add your feature"`
7. Push: `git push origin feature/your-feature`
8. Open a Pull Request

## Code Style

- We use ESLint and Prettier
- Run `npm run lint` before committing
- Follow the existing code style

## Commit Messages

We follow [Conventional Commits](https://conventionalcommits.org/):
- `feat:` for new features
- `fix:` for bug fixes
- `docs:` for documentation
- See COMMIT_CONVENTION.md for details

## Pull Request Process

1. Update documentation as needed
2. Add tests for new functionality
3. Ensure all tests pass
4. Get approval from at least one maintainer
5. Squash and merge

## Code of Conduct

Please be respectful and inclusive. See CODE_OF_CONDUCT.md.
~~~

LICENSE:

Always include a license. Common choices:

- MIT: Permissive, simple
- Apache 2.0: Permissive with patent protection
- GPL: Copyleft, requires derivatives to be GPL

.gitignore:

```
# Dependencies
node_modules/
vendor/

# Build outputs
dist/
build/
*.pyc
__pycache__/

# Environment files
.env
.env.local
.env.*.local

# IDE files
.vscode/
.idea/
*.swp
*.swo

# OS files
.DS_Store
Thumbs.db

# Logs
*.log
logs/

# Test coverage
coverage/

# Temporary files
tmp/
temp/
```

CHANGELOG.md:

```
# Changelog

All notable changes to this project will be documented in this file.
```

```
The format is based on [Keep a Changelog](https://keepachangelog.com/),
and this project adheres to [Semantic Versioning](https://semver.org/).

## [Unreleased]

### Added
- User profile page

### Changed
- Updated dashboard layout

## [1.2.0] - 2024-03-15

### Added
- User authentication system
- Password reset functionality
- Email notifications

### Fixed
- Date formatting bug in reports
- Memory leak in image processor

### Security
- Updated dependencies to fix CVE-2024-XXXX

## [1.1.0] - 2024-02-01

### Added
- Initial release
```

7.7.4 Branch Protection

Branch protection rules prevent accidental or unauthorized changes to important branches.

GitHub Branch Protection Settings:

```
Branch protection rule: main


  Require a pull request before merging
    Require approvals: 1
    Dismiss stale PR approvals when new commits are pushed
    Require review from code owners

  Require status checks to pass before merging
```

```
      Require branches to be up to date before merging
     Status checks:
         ci/test
         ci/lint

    Require conversation resolution before merging

    Require signed commits

    Do not allow bypassing the above settings

    Restrict who can push to matching branches

    Allow force pushes: Nobody

    Allow deletions:
```

CODEOWNERS File:

Automatically request reviews from specific people for specific paths:

```
# .github/CODEOWNERS

# Default owners for everything
* @team-lead

# Frontend owners
/src/components/ @frontend-team
/src/pages/ @frontend-team

# Backend owners
/src/api/ @backend-team
/src/services/ @backend-team

# Database changes need DBA review
/migrations/ @dba-team

# Security-sensitive files need security review
/src/auth/ @security-team
/src/crypto/ @security-team
```

7.7.5 Cleaning Up Branches

Stale branches clutter the repository. Clean them up regularly.

Deleting Merged Branches Locally:

```bash
# Delete a merged branch
git branch -d feature/completed-feature

# Delete multiple merged branches
git branch --merged main | grep -v main | xargs git branch -d
```

Deleting Remote Branches:

```bash
# Delete a remote branch
git push origin --delete feature/completed-feature

# Prune deleted remote branches locally
git fetch --prune
```

GitHub Auto-Delete:

Enable “Automatically delete head branches” in repository settings to auto-delete branches after PR merge.

7.7.6 Git Hooks

Git hooks run scripts at specific points in the Git workflow. Use them for automation and enforcement.

Common Hooks:

```
pre-commit     → Before commit is created
commit-msg     → Validate commit message
pre-push       → Before push to remote
post-merge     → After merge completes
```

Example: Pre-commit Hook for Linting:

```sh
#!/bin/sh
# .git/hooks/pre-commit

echo "Running linter..."
npm run lint

if [ $? -ne 0 ]; then
  echo "Lint failed. Please fix errors before committing."
  exit 1
fi

echo "Running tests..."
npm test
```

```
if [ $? -ne 0 ]; then
  echo "Tests failed. Please fix before committing."
  exit 1
fi

exit 0
```

Using Husky (Recommended):

Husky manages Git hooks in a way that's shareable with the team:

```
# Install Husky
npm install husky --save-dev
npx husky install

# Add to package.json
"scripts": {
  "prepare": "husky install"
}

# Add hooks
npx husky add .husky/pre-commit "npm run lint"
npx husky add .husky/pre-push "npm test"
```

.husky/pre-commit:

```
#!/usr/bin/env sh
. "$(dirname -- "$0")/_/husky.sh"

npm run lint-staged
```

Lint-Staged (Run linters only on staged files):

```
// package.json
{
  "lint-staged": {
    "*.{js,jsx,ts,tsx}": [
      "eslint --fix",
      "prettier --write"
    ],
    "*.{json,md}": [
      "prettier --write"
    ]
  }
}
```

7.8 Advanced Git Techniques

7.8.1 Interactive Rebase

Interactive rebase lets you edit, reorder, combine, or delete commits before sharing them.

```
# Rebase last 4 commits interactively
git rebase -i HEAD~4
```

This opens an editor:

```
pick abc1234 Add user model
pick def5678 Add user service
pick ghi9012 Fix typo in user model
pick jkl3456 Add user controller

# Commands:
# p, pick = use commit
# r, reword = use commit, but edit the commit message
# e, edit = use commit, but stop for amending
# s, squash = use commit, but meld into previous commit
# f, fixup = like "squash", but discard this commit's message
# d, drop = remove commit
```

Common Operations:

Squash related commits:

```
pick abc1234 Add user model
squash ghi9012 Fix typo in user model    # Combine with previous
pick def5678 Add user service
pick jkl3456 Add user controller
```

Reword a commit message:

```
reword abc1234 Add user model            # Will prompt for new message
pick def5678 Add user service
```

Reorder commits:

```
pick def5678 Add user service            # Moved up
pick abc1234 Add user model              # Moved down
pick jkl3456 Add user controller
```

Delete a commit:

```
pick abc1234 Add user model
drop def5678 Add user service            # Remove this commit
pick jkl3456 Add user controller
```

7.8.2 Cherry-Pick

Cherry-pick applies a specific commit from one branch to another.

```
# Apply a specific commit to current branch
git cherry-pick abc1234

# Apply multiple commits
git cherry-pick abc1234 def5678

# Cherry-pick without committing (stage changes only)
git cherry-pick --no-commit abc1234
```

Use Cases:

- Apply a bug fix from one branch to another
- Port specific features between branches
- Recover commits from deleted branches

7.8.3 Stashing

Stash temporarily saves uncommitted changes so you can switch contexts.

```
# Stash current changes
git stash

# Stash with a message
git stash save "WIP: working on login form"

# List stashes
git stash list

# Apply most recent stash (keeps stash)
git stash apply

# Apply and remove most recent stash
git stash pop

# Apply a specific stash
git stash apply stash@{2}

# Drop a stash
```

```
git stash drop stash@{0}

# Clear all stashes
git stash clear
```

Stash Workflow:

```
# You're working on feature-a but need to fix a bug
git stash save "WIP: feature-a progress"

git checkout main
git checkout -b hotfix/urgent-bug
# Fix the bug
git commit -m "fix: resolve urgent bug"
git checkout main
git merge hotfix/urgent-bug
git push

# Return to feature work
git checkout feature-a
git stash pop  # Restore your work
```

7.8.4 Bisect

Git bisect helps find which commit introduced a bug using binary search.

```
# Start bisect
git bisect start

# Mark current (broken) commit as bad
git bisect bad

# Mark a known good commit
git bisect good abc1234

# Git checks out a commit in the middle
# Test it, then mark:
git bisect good  # If this commit is okay
# or
git bisect bad   # If this commit has the bug

# Repeat until Git identifies the culprit

# When done, reset to original state
git bisect reset
```

Automated Bisect:

```
# Run a script to test each commit automatically
git bisect start HEAD abc1234
git bisect run npm test
```

7.8.5 Reflog

The **reflog** records every change to HEAD—even ones not in branch history. It's a safety net for recovering "lost" work.

```
# View reflog
git reflog

# Output:
# abc1234 HEAD@{0}: commit: Add new feature
# def5678 HEAD@{1}: checkout: moving from main to feature
# ghi9012 HEAD@{2}: merge feature-x: Fast-forward
# jkl3456 HEAD@{3}: reset: moving to HEAD~1
# mno7890 HEAD@{4}: commit: This commit was "lost"
```

Recovering "Lost" Commits:

```
# Oops, I hard reset and lost a commit!
git reset --hard HEAD~1

# Find it in reflog
git reflog
# mno7890 HEAD@{1}: commit: Important work

# Recover it
git checkout mno7890
# or
git cherry-pick mno7890
```

7.9 Workflow for Your Project

For your course project, here's a recommended workflow combining best practices.

7.9.1 Project Setup

```
# 1. Clone the repository
git clone https://github.com/your-team/project.git
cd project

# 2. Set up your identity
git config user.name "Your Name"
git config user.email "your.email@example.com"

# 3. Set up branch protection on GitHub
#    - Require PR before merging to main
#    - Require at least 1 approval
#    - Require status checks to pass
```

7.9.2 Daily Workflow

```
# Morning: Start fresh
git checkout main
git pull origin main

# Create feature branch
git checkout -b feature/task-assignment

# Work on feature, commit frequently
git add .
git commit -m "feat(tasks): add assignee field to task model"

git add .
git commit -m "feat(tasks): add assignment dropdown component"

# Push to remote (backup + enable PR)
git push -u origin feature/task-assignment

# If main has changed, integrate
git fetch origin
git merge origin/main
# Resolve any conflicts
git push
```

7.9.3 Pull Request Process

```
## Summary
Implements task assignment functionality, allowing users to assign
tasks to team members.
```

```
## Related Issues
Closes #45

## Changes
- Added `assigneeId` field to Task model
- Created UserDropdown component for assignment UI
- Added PATCH /tasks/:id/assign endpoint
- Added notification when task is assigned

## Testing
- Unit tests for Task model changes
- Integration tests for assignment endpoint
- Manually tested assignment flow

## Checklist
- [x] Code follows style guide
- [x] Tests added
- [x] Documentation updated
```

7.9.4 Code Review Process

As Author:

1. Open PR with detailed description
2. Request review from teammates
3. Respond to feedback promptly
4. Make requested changes
5. Re-request review when ready
6. Merge after approval

As Reviewer:

1. Review within 24 hours
2. Check functionality, design, tests
3. Leave constructive feedback
4. Approve when satisfied
5. Follow up on addressed comments

7.9.5 Branch Cleanup

```
# After PR is merged, clean up
git checkout main
git pull origin main
git branch -d feature/task-assignment
```

```
# Clean up remote tracking branches
git fetch --prune
```

7.10 Chapter Summary

Version control workflows are essential for team collaboration. The right workflow depends on your team size, release cadence, and project needs, but all good workflows share common elements: isolation through branching, quality through review, and stability through protection.

Key takeaways from this chapter:

- **Branching** enables parallel development. Git branches are lightweight pointers that allow developers to work in isolation.
- **Merging** combines work from different branches. Fast-forward merges keep linear history; three-way merges create merge commits.
- **Rebasing** rewrites history for a cleaner timeline but should never be used on shared branches.
- **Gitflow** provides structured branches for releases but adds complexity. Best for versioned software with scheduled releases.
- **GitHub Flow** simplifies to just main and feature branches with pull requests. Ideal for continuous deployment.
- **Trunk-based development** minimizes branching, with all developers integrating to main frequently. Requires excellent CI/CD and team discipline.
- **Pull requests** enable code review and discussion before merging. Good PRs are small, well-described, and focused.
- **Code review** improves quality, shares knowledge, and catches bugs. Good reviewers are specific, constructive, and empathetic.
- **Merge conflicts** are normal and manageable. Integrate frequently and communicate with teammates to minimize conflicts.
- **Repository hygiene** keeps codebases maintainable through conventions, documentation, branch protection, and cleanup.

7.11 Key Terms

Term	Definition
Branch	A lightweight movable pointer to a commit
HEAD	Special pointer indicating current branch and commit
Merge	Combining changes from one branch into another
Fast-forward merge	Moving branch pointer forward when no divergence exists
Three-way merge	Creating a merge commit when branches have diverged
Rebase	Rewriting history by moving commits to a new base
Pull Request (PR)	Request to merge changes, enabling review and discussion
Code Review	Examination of code changes by teammates
Merge Conflict	Situation where Git can't automatically combine changes
Gitflow	Branching model with main, develop, feature, release, and hotfix branches
GitHub Flow	Simple workflow with main and feature branches
Trunk-based Development	Workflow where all developers commit to main frequently
Feature Flag	Toggle to hide incomplete features in production
Branch Protection	Rules preventing unauthorized changes to branches
Git Hook	Script that runs at specific points in Git workflow

7.12 Review Questions

1. Explain the difference between merge and rebase. When would you use each?
2. Describe the Gitflow branching model. What are the five branch types, and what is each used for?
3. Compare GitHub Flow and trunk-based development. What are the key differences, and when would you choose each?
4. What makes a good pull request? List at least five characteristics.
5. Describe the code review process. What should reviewers look for, and how should they give feedback?

6. Explain what causes merge conflicts and how to resolve them.

7. What is branch protection, and why is it important? List three protection rules you might enable.

8. Describe the Conventional Commits specification. Why are consistent commit messages valuable?

9. What are Git hooks? Give two examples of how they can improve workflow.

10. You're joining a new team that doesn't have a defined Git workflow. What questions would you ask, and what workflow might you recommend?

7.13 Hands-On Exercises

Exercise 7.1: Branching Practice

Practice the Git branching basics:

1. Create a new repository or use your project
2. Create three feature branches from main
3. Make different changes in each branch
4. Merge each branch back to main using different strategies:
   - Fast-forward merge (if possible)
   - Three-way merge with merge commit
   - Squash merge
5. Examine the history with `git log --oneline --graph`

Exercise 7.2: Conflict Resolution

Practice resolving conflicts:

1. Create a branch `feature-a` and modify line 10 of a file
2. Return to main, create `feature-b`, and modify the same line differently
3. Merge `feature-a` into main
4. Attempt to merge `feature-b` into main (conflict!)
5. Resolve the conflict by combining both changes appropriately
6. Complete the merge and verify the result

Exercise 7.3: Interactive Rebase

Practice cleaning up history:

1. Make 5 commits on a feature branch, including:
    - A commit with a typo in the message
    - Two commits that should be squashed
    - One commit that should be removed
2. Use interactive rebase to:
    - Fix the typo (reword)
    - Squash the related commits
    - Remove the unnecessary commit
3. Verify the cleaned-up history

Exercise 7.4: Pull Request

Create a full pull request:

1. Create a feature branch in your project
2. Implement a small feature (or improve documentation)
3. Push the branch to GitHub
4. Open a pull request with:
    - Descriptive title
    - Summary of changes
    - Related issues (if any)
    - Testing notes
    - Checklist
5. Request review from a teammate

Exercise 7.5: Code Review

Practice code review:

1. Find a teammate's open pull request
2. Review the code using the checklist from this chapter
3. Leave at least:
    - One piece of positive feedback
    - One suggestion for improvement
    - One question about the implementation
4. Use appropriate prefixes ([suggestion], [question], etc.)

Exercise 7.6: Repository Setup

Set up repository hygiene for your project:

1. Create or update these files:
   - README.md (complete with setup instructions)
   - CONTRIBUTING.md (contribution guidelines)
   - .gitignore (appropriate for your technology)
   - CHANGELOG.md (start tracking changes)
2. Configure branch protection:
   - Require PR before merging to main
   - Require at least 1 approval
   - Require status checks (if CI configured)
3. Create a CODEOWNERS file assigning reviewers

Exercise 7.7: Git Hooks

Implement Git hooks for your project:

1. Install Husky (or set up hooks manually)
2. Create a pre-commit hook that:
   - Runs the linter
   - Runs relevant tests
3. Create a commit-msg hook that:
   - Validates commit message format
4. Test that the hooks work by:
   - Making a bad commit (should be rejected)
   - Making a good commit (should succeed)

7.14 Further Reading

Books:

- Chacon, S., & Straub, B. (2014). *Pro Git* (2nd Edition). Apress. (Free online: https://git-scm.com/book)
- Loeliger, J., & McCullough, M. (2012). *Version Control with Git* (2nd Edition). O'Reilly Media.

Articles and Guides:

- Driessen, V. (2010). A successful Git branching model. https://nvie.com/posts/a-successful-git-branching-model/
- GitHub Flow Guide: https://guides.github.com/introduction/flow/
- Trunk Based Development: https://trunkbaseddevelopment.com/
- Conventional Commits: https://www.conventionalcommits.org/
- How to Write a Git Commit Message: https://chris.beams.io/posts/git-commit/

Tools:

- GitHub Documentation: https://docs.github.com/
- Husky (Git hooks): https://typicode.github.io/husky/
- Commitlint (Commit message linting): https://commitlint.js.org/

References

Chacon, S., & Straub, B. (2014). *Pro Git* (2nd Edition). Apress. Retrieved from https://git-scm.com/book

Driessen, V. (2010). A successful Git branching model. Retrieved from https://nvie.com/posts/a-successful-git-branching-model/

GitHub. (2021). Understanding the GitHub flow. Retrieved from https://guides.github.com/introduction/flow/

Hammant, P. (2017). Trunk Based Development. Retrieved from https://trunkbaseddevelopment.com/

Conventional Commits. (2021). Conventional Commits Specification. Retrieved from https://www.conventionalcommits.org/

Sample content for 09-prototype-implementation.qmd

Sample content for 10-testing.qmd

Sample content for 11-quality-assurance.qmd

Sample content for 12-data-management-apis.qmd

Sample content for 13-devops-deployment.qmd

Sample content for 14-security.qmd

Sample content for 15-maintenance-evolution.qmd

Sample content for 16-final-integration.qmd

Sample content for appendix-project-guide.qmd

Sample content for glossary.qmd