Software Engineering

Engr. Mbiarrambang Alain

2026-05-15

Table of contents

1 Course Overview

1.1 Course Title

Software Engineering I

1.2 Credit Value

6 Credits

1.3 Duration

14–16 Weeks

1.4 Weekly Breakdown

Week	Topic
1	Introduction to Software Engineering
2	Professional Software Development
3	Software Processes
4	Software Process Models

Week	Topic
5	Agile Software Development
6	Requirements Engineering
7	Requirements Elicitation & Validation
8	UML & System Modeling
9	Behavioral & Structural Modeling
10	Architectural Design
11	Object-Oriented Design
12	Software Testing
13	Software Evolution & Maintenance
14	Ethics, Review & Mini Project Defense

2 INTRODUCTION TO SOFTWARE ENGINEERING

2.1 Learning Objectives

At the end of this chapter, students should be able to:

- Define software engineering
- Explain why software engineering is important
- Differentiate software engineering from computer science
- Identify software system types
- Understand software engineering ethics
- Explain professional software development

2.2 What is Software Engineering?

Software engineering is an engineering discipline concerned with all aspects of software production, from initial specification through development, validation, and evolution (maintenance). It is a systematic way of producing software that balances quality, cost, schedule, and customer needs.

> **i** In simple terms, software engineering is:
>
> A systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software.
> A useful way to remember the definition is:

- **software** is not only the program code;
- **engineering** means working systematically under constraints;
- **software engineering** means producing useful software professionally, not casually.

2.3 Key Idea

Software engineering is NOT just coding.

It includes:

- Requirements gathering
- Analysis
- Design
- Testing
- Deployment
- Maintenance
- Project management
- Documentation
- Quality assurance

2.4 Why Software Engineering Matters

Modern society depends heavily on software:

- Banking systems
- Mobile money
- Healthcare systems
- Aviation
- Telecommunications
- Transportation
- Education
- Government systems

Without proper engineering:

- Systems fail
- Lives may be lost
- Financial loss
- Operational downtime
- Safety hazards
- Privacy breaches
- Legal consequences

- Public distrust

2.5 Real-World Example — Cameroon

A university wants a student management system. The initial request may sound simple: "store student records." In reality, the system may need:

- Role-based access
- Grading workflows
- Payment integration
- Reporting
- Backups
- Data privacy
- Audit trails
- Future changes to grading policy

This is exactly why software engineering exists.

2.6 Characteristics of Good Software

Attribute	Meaning	Why it matters
Maintainability	Easy to change and extend	Software inevitably evolves
Dependability and security	Reliable, safe, and protected against misuse	Users must trust the system
Efficiency	Uses resources well	Low cost and good performance
Acceptability	Usable and suited to its users	A system is useless if people reject it

2.6.1 Maintainability

Maintainability is one of the most important software qualities because change is unavoidable. A system must be designed so that corrections, enhancements, and adaptations can be made without breaking everything else.

2.6.2 Dependability and security

Dependability includes:

- reliability
- availability
- safety
- security

A payroll system that leaks salaries fails security.

A medical system that crashes during consultation fails availability.

A braking system that gives wrong commands fails safety.

2.6.3 Efficiency

Efficiency includes:

- response time
- throughput
- memory use
- processor use
- network use

Efficiency is not only about speed. A system can be "fast" but still wasteful.

2.6.4 Acceptability

A system is acceptable when users can actually use it effectively. This includes:

- usability
- understandable interface
- compatibility with work practices
- accessibility
- training requirements

> **!** Important
>
> Good software is not only correct. It must also be useful, safe, maintainable, and acceptable to real users.

2.7 Software Engineering vs Computer Science vs System Engineering

2.7.1 vs Computer Science

Computer science focuses on the theories and principles underlying computation. Software engineering focuses on the practical production of working software under real-world constraints.

This means:

- computer science asks, "What can be computed?"
- software engineering asks, "How do we build it so people can use it reliably?"

They overlap, but the purpose is different.

2.7.2 vs System Engineering

System engineering deals with the whole computer-based system:

- hardware
- software
- networks
- people
- procedures
- organization
- deployment

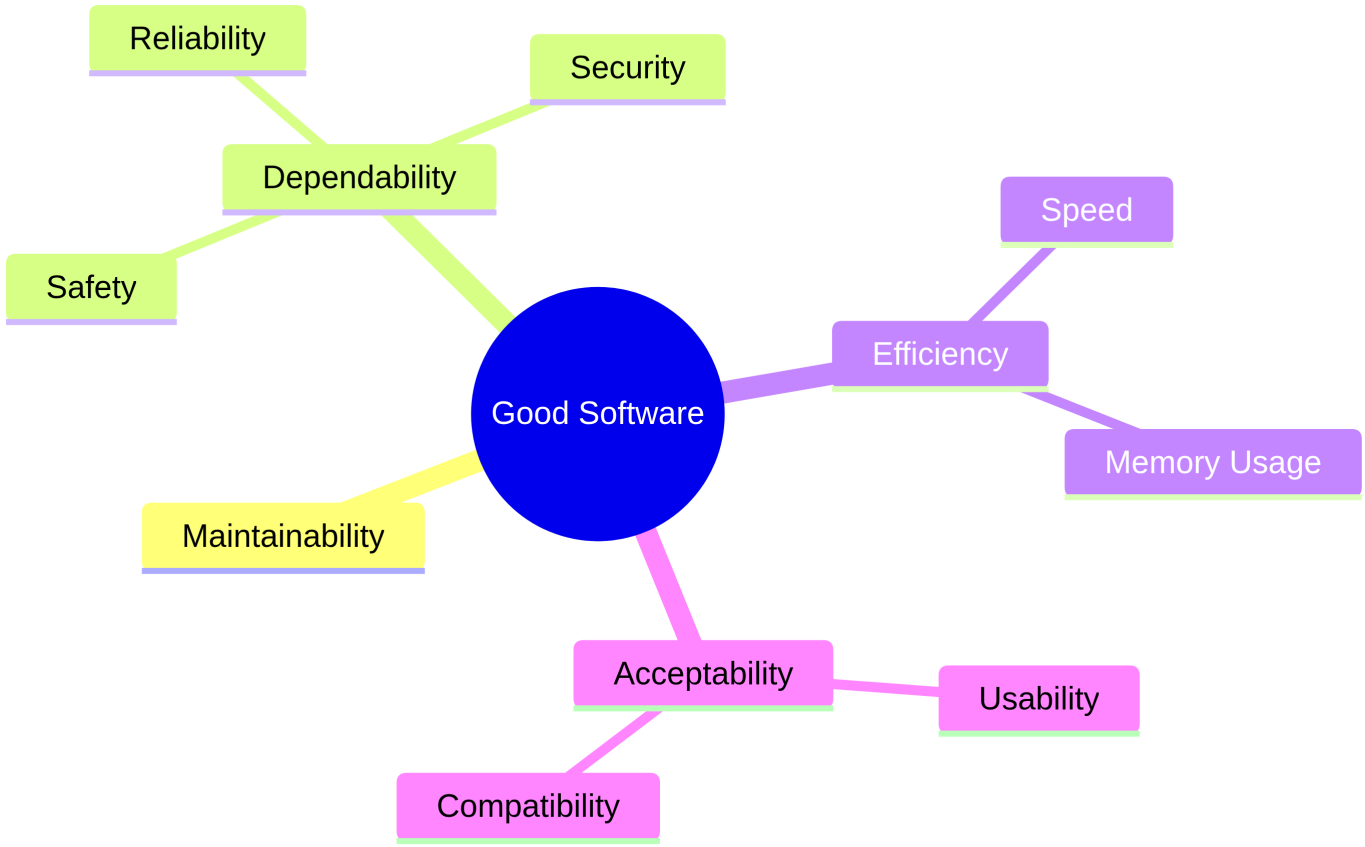
Software engineering is a subset of system engineering focused on the software component.

> **Why this distinction matters**
>
> A hospital information system is not just software. It includes:
>
> - servers
> - users
> - legal rules
> - clinical processes
> - databases
> - backup policies
> - security controls
>
> If you ignore the wider system, the software may fail even if the code is technically correct.

2.8 Software Engineering vs Computer Science

Computer Science	Software Engineering
Theory-focused	Practical-focused
Algorithms	System delivery
Mathematical foundations	Engineering process
Computation models	Real-world software production

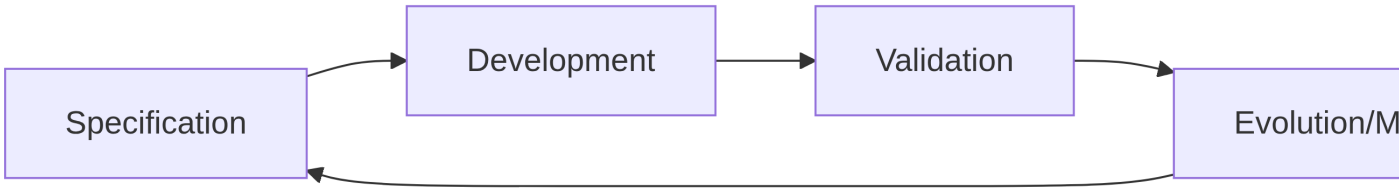
2.9 Mermaid Diagram — Essential Attributes



2.10 Software Process

A software process is a sequence of activities that leads to the production of a software product. The four fundamental process of activities:

1. Software specification
2. Software development
3. Software validation
4. Software evolution/maintenance



2.10.1 Explanation

- **Specification** answers: what should the system do?
- **Development** answers: how do we build it?
- **Validation** answers: did we build the right thing?
- **Evolution/Maintenance** answers: how should it change after delivery?

> Tip
>
> These four activities always exist in some form, even if the process looks different from project to project.

2.10.1.1 Specification

Defines:

- What the system should do
- Constraints
- Functional requirements
- Non-functional requirements

2.10.1.2 Development

Activities:

- Design
- Programming

- Implementation

2.10.1.3 Validation

Ensures:

- Software works correctly
- Meets customer needs

Includes:

- Testing
- Reviews
- Verification

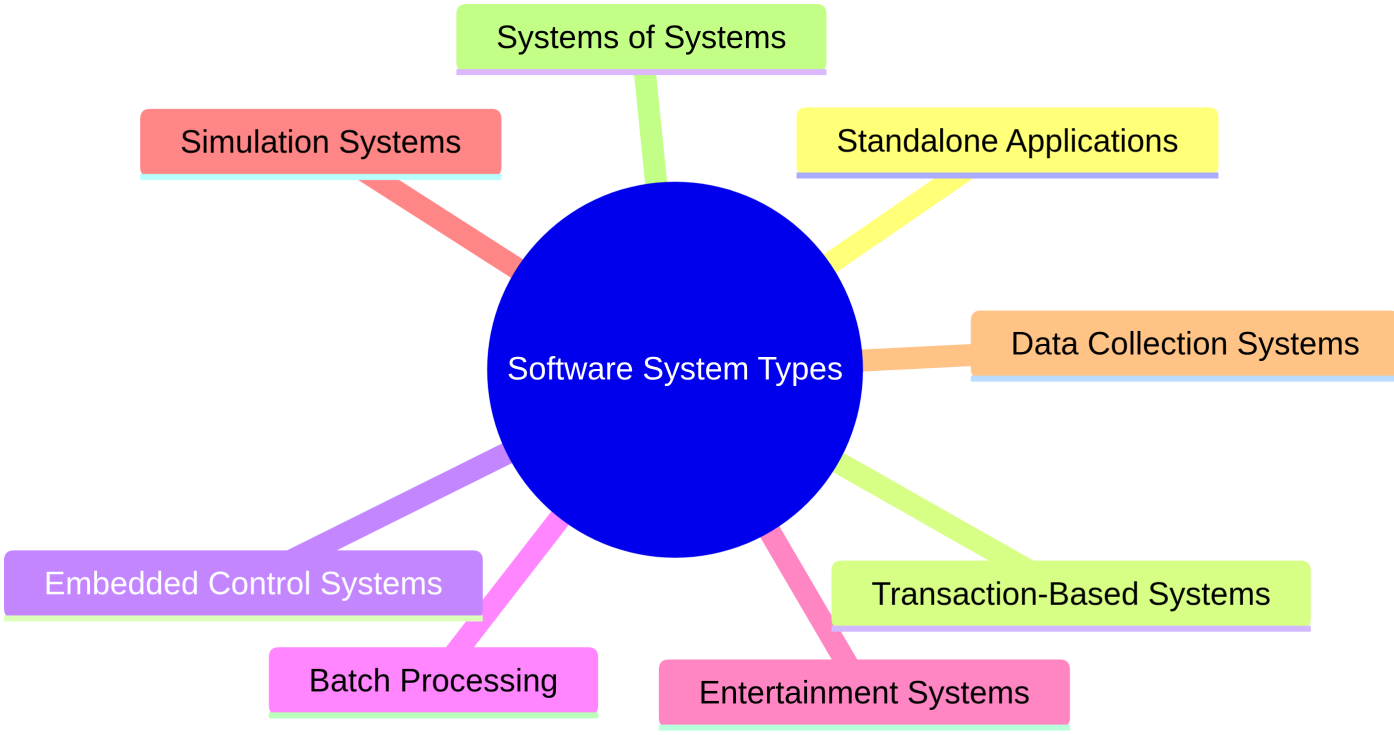
2.10.1.4 Evolution/Maintenance

Software changes over time due to:

- Business changes
- Technology changes
- Bug fixes
- Feature updates

> ⚠ Software diversity
>
> No single process or method works for all systems. The right approach depends on the software type.
> Examples:
>
> - embedded control systems need strict specification and testing
> - web systems often need iterative development
> - business systems often need fast adaptation
> - safety-critical systems need rigorous verification

2.11 Types of Software Systems



- Standalone applications
  - Examples:
    - Microsoft Word
    - VLC Media Player
- Interactive transaction-based applications
  - Examples:
    - Facebook
    - Jumia
    - Amazon
- Embedded control systems
  - Examples:
    - Car ABS systems
    - Microwave ovens
- Batch processing systems

    - Examples:
        * Payroll systems
        * Utility billing systems

- Entertainment systems
    - Examples:
        * Video games
        * Mobile games

- Modeling and simulation systems
    - Examples:
        * Weather forecasting systems
        * Traffic simulation systems

- Data collection systems
    - Examples:
        * Weather monitoring systems

- Systems of systems
    - Examples:
        * Airlines reservation system

2.12 Software Engineering Challenges

2.12.1 Heterogeneity

Different devices and platforms.

2.12.2 Business and Social Change

Requirements evolve rapidly.

2.12.3 Security and Trust

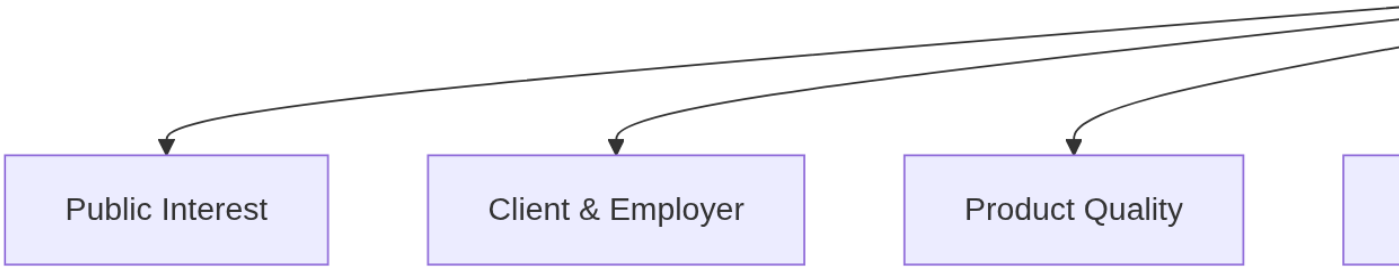
Systems must be dependable.

2.13 Software Engineering Ethics

Software engineering is a profession with ethical duties. Key ethical responsibilities include:

- Confidentiality
- Competence
- Intellectual property respect
- Avoiding computer misuse

2.13.1 ACM/IEEE Ethical Principles



- act in the public interest
- serve clients and employers honestly
- deliver high-quality products
- maintain professional judgment
- promote ethical management
- protect the reputation of the profession
- support colleagues
- keep learning

> **Warning**
>
> A technically brilliant solution can still be unethical if it harms people, violates privacy, or misuses data.

2.13.2 Ethical dilemma example

Suppose a company asks an engineer to hide a defect in a safety-critical system to meet a deadline. The engineer faces a conflict between loyalty to the employer and responsibility

to the public. **Software engineering ethics requires careful judgment, not blind obedience.**

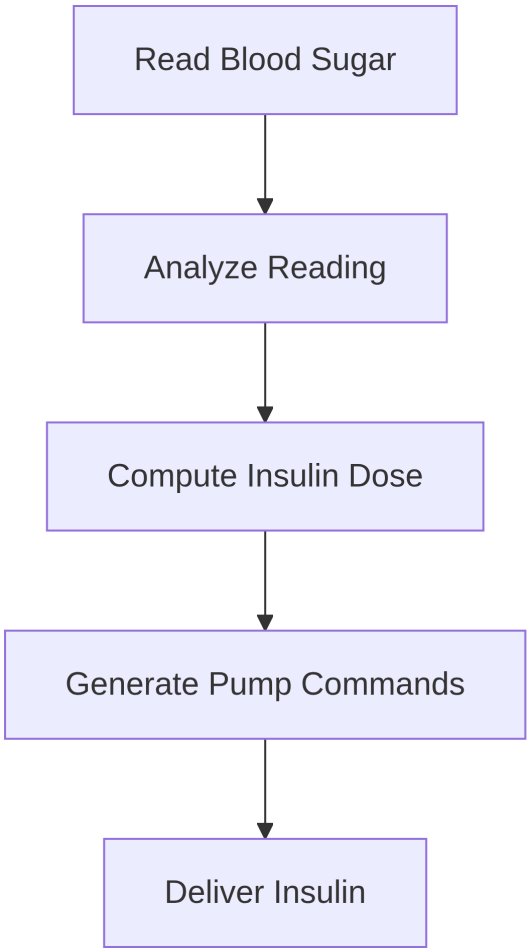
2.14 Case Study 1 — Insulin Pump System

A safety-critical embedded system. This is a medical embedded system that reads a sensor, computes insulin needs, and controls a pump. It is safety-critical because failure can harm or kill the user.

2.14.1 Key Requirements

- Reliability
- Availability
- Safety
- Real-time response

2.14.2 Activity Diagram — Insulin Pump



2.15 Lab 1 — Understanding software as a system

2.15.1 Goal

Distinguish code from software system.

2.15.2 Task

Choose one system in your environment, for example:

- university portal

- hospital records system
- mobile money app
- bus ticketing system

2.15.2.1 Students must:

1. Identify stakeholders
2. Identify system requirements
3. Identify risks
4. Identify software quality attributes

2.15.2.2 Describe:

1. its users
2. its data
3. its documentation needs
4. its maintenance needs
5. its quality risks

2.15.2.3 Deliverable

A 2-page report plus one simple diagram.

2.16 Review Questions

1. Define software engineering.
2. Differentiate software engineering from programming.
3. Explain four software process activities.
4. Discuss software engineering ethics.
5. Explain why maintainability is important.

---

3 Software Processes

3.1 What is a Software Process?

A software process is the organized set of activities that produce a software system. It is not just coding steps; it is the full lifecycle from idea to evolution.

3.2 Why processes matter

Without a process:

- work becomes chaotic
- requirements are forgotten
- testing is delayed
- changes break the system
- teams cannot coordinate

With a process:

- responsibilities are clearer
- quality can be controlled
- progress can be measured
- changes can be managed

3.3 Generic Process Activities

The textbook identifies these core process activities:

- requirements
- design
- implementation
- testing and validation
- deployment
- maintenance

3.3.1 Specification

This activity defines what the software must do and under what conditions. It includes:

- requirements elicitation
- requirements analysis
- requirements documentation

3.3.2 Design and implementation

This activity transforms requirements into a working system. It includes:

- architectural design
- detailed design
- coding
- integration

3.3.3 Testing and Validation

Validation answers the question: "Are we building the right system?"

It includes:

- reviews
- unit testing
- integration testing
- system testing
- acceptance testing

3.3.4 Deployment

Deployment is responsible for making the software available to users.

It includes:

- Environment Configuration
- Artifact Distribution
- Release Management
- Database Migration
- Continuous Deployment
- Monitoring and Logging
- Rollback Planning

3.3.5 Maintenance

Evolution continues after delivery. Software changes because:

- users request new features
- laws change
- technology changes
- defects are discovered

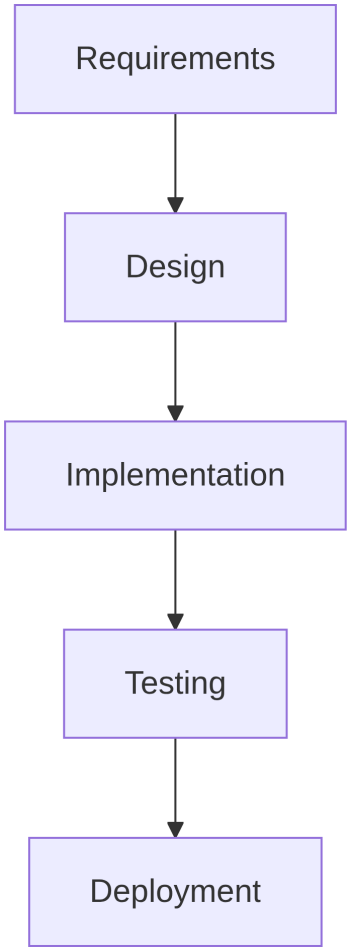
- performance needs evolve

3.4 Process Models

A process model is a simplified description of a process from one perspective. It helps us understand and compare development approaches.

3.4.1 Waterfall Model

The waterfall model is sequential.



3.4.1.1 Strengths

- Simple to understand
- Strong documentation
- Useful when requirements are stable

3.4.1.2 Weaknesses

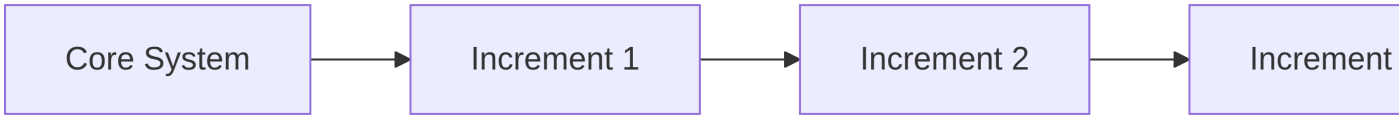
- Difficult to accommodate change
- Late feedback/testing
- Expensive if requirements are misunderstood
- Risky for uncertain projects

3.4.1.3 Best suited for

- Systems with stable, well-understood requirements
- Highly regulated projects
- Some critical systems

3.4.2 Incremental Development

In incremental development, the system is built in small usable parts.



3.4.2.1 Strengths

- Early delivery
- Feedback from users
- Lower risk
- Easier to accommodate change

3.4.2.2 Weaknesses

- Architecture may weaken if poorly managed
- Integration may become complex
- Repeated refactoring may be needed

3.4.3 Reuse-Oriented Development

Reuse-oriented development builds systems by integrating existing software components rather than developing everything from scratch.

Examples of reused items:

- libraries
- frameworks
- open source components
- COTS systems
- APIs

3.4.3.1 Benefits

- Faster development
- Lower cost
- Access to mature components

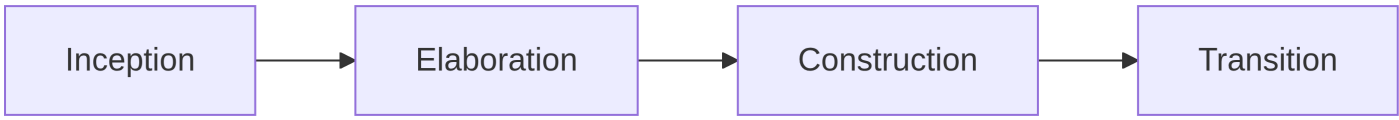
3.4.3.2 Risks

- Hidden incompatibilities
- Vendor lock-in
- Quality uncertainty
- Security concerns

3.4.4 Rational Unified Process (RUP)

RUP is an iterative process framework with phases:

- Inception
- Elaboration
- Construction
- Transition



3.4.4.1 What RUP tries to do

- Reduce project risk early
- Support iteration
- Allow architectural planning
- Combine structure with adaptability

3.5 Process activities in more detail

Processes also include supporting activities such as:

- Documentation
- Configuration management
- Project management
- Quality assurance

3.5.1 Roles in a process

- Project manager
- Programmer
- Analyst
- Tester
- Configuration manager
- Customer representative

> **i** Preconditions and postconditions
>
> A process activity often requires conditions before it begins and produces conditions after it ends.
> Example:
>
> - Before architectural design starts, requirements should be reviewed and approved
> - After it ends, an architectural model should exist and be reviewed

3.5.2 Coping with change

Change is unavoidable. Good processes are designed to cope with change rather than pretend it will not happen.

3.5.2.1 Common sources of change

- Customers refine their needs
- Market conditions change
- Technology changes
- Legislation changes
- Risks become visible only later

3.5.2.2 Coping strategies

- Iterative delivery
- Modular design
- Change control
- Traceability
- Feedback cycles
- Prototyping

3.5.3 Plan-driven versus Agile processes

3.5.3.1 Plan-driven processes

All process activities are planned in advance and progress is measured against the plan.

Useful when:

- Requirements are stable
- Risk is high
- Documentation is legally required
- Many teams must coordinate

3.5.3.2 Agile processes

Planning is incremental, and change is accepted as normal.

Useful when:

- Requirements are uncertain
- Time-to-market matters
- Customers can participate actively
- Teams are small and collaborative

> **!** Important
>
> Agile and plan-driven **are not** enemies. The key idea is balance. Different systems need different mixes of discipline and flexibility.

3.6 LAB 2 — Process Modeling

3.6.1 Task

For each system below, choose a suitable process model and justify your choice:

- ATM
- school portal
- health app
- weather station

Students should:

- Choose a process model
- Justify the choice
- Draw diagrams

3.6.2 Output

A written comparison of:

- waterfall
- incremental
- reuse-oriented
- RUP

Include risks and reasons.

---

4 Agile Software Development

4.1 What is Agile?

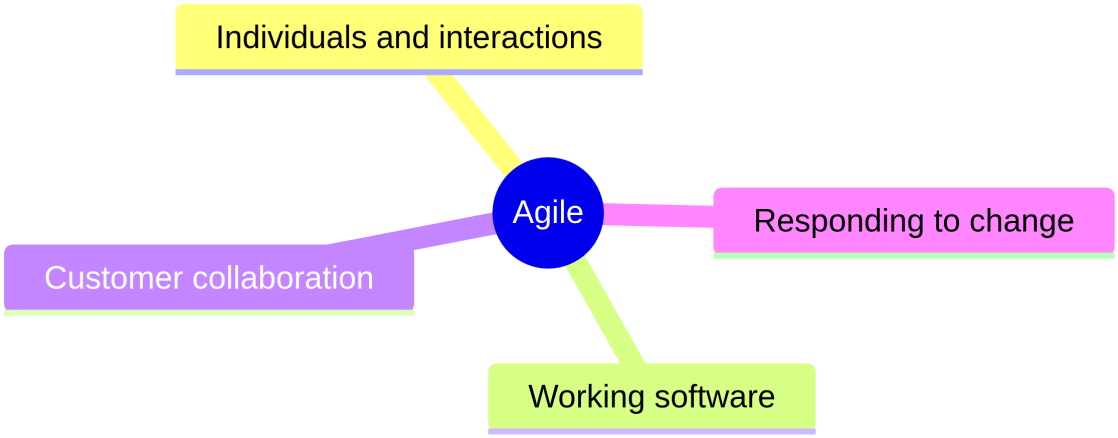
Agile methods arose because traditional heavy processes were often too slow and too bureaucratic for rapidly changing business software. They focus on:

- Collaboration
- Customer feedback
- Rapid delivery
- Adaptability

4.2 Agile Manifesto

The agile manifesto 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



4.2.1 Important interpretation

This does not mean that processes, documentation, contracts, and plans are useless. It means they should not dominate the real goal: **building software that works**.

4.3 Plan-driven and Agile development

4.3.1 Plan-driven strengths

- clearer governance
- more predictable documentation
- suited to regulated work
- easier coordination for large teams

4.3.2 Agile strengths

- faster value delivery
- frequent feedback
- easier response to changing needs
- better visibility of working progress

4.3.3 Agile vs Plan-Driven

Agile	Plan-Driven
Flexible	Structured
Iterative	Sequential
Fast feedback	Heavy documentation
Customer collaboration	Contract-focused

4.4 When agile is hard

Agile is harder when:

- users are not available
- safety certification is strict
- many dependencies exist
- architecture must be heavily constrained

4.5 Agile project management

Agile project management organizes work into short iterations or sprints.

Important ideas:

- product backlog

- sprint backlog
- sprint review
- retrospective
- prioritization by value

4.6 Extreme Programming (XP)

XP is one of the best-known agile methods. It emphasizes engineering discipline as well as agility.

4.6.1 Key practices

- user stories
- short iterations
- test-driven development
- pair programming
- continuous integration
- refactoring
- collective code ownership
- sustainable pace

4.6.2 Why XP matters

XP is not "no process." It is a process designed to keep code clean, feedback fast, and change affordable.

4.6.3 Pair programming

Two developers work at one workstation:

- one writes code
- one reviews continuously

Benefits:

- fewer defects

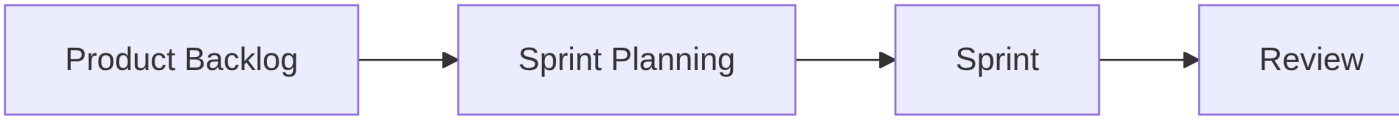
- knowledge sharing
- better design decisions
- mentoring

4.6.4 Test-driven development in XP

Tests are written before code. This improves design clarity and regression safety.

4.7 Scrum Framework

Scrum is a lightweight Agile framework designed to help teams deliver high-value, complex products iteratively and incrementally. It promotes collaboration, accountability, and continuous improvement, breaking large projects into small, manageable pieces delivered in short, time-boxed cycles called "sprints" (typically 2–4 weeks).



4.7.1 Key Components of Scrum

- **Scrum Team**: Small, self-organizing teams consisting of a Product Owner, Scrum Master, and Developers.
- **Scrum Events (Meetings)**: Sprint Planning, Daily Scrum (stand-up), Sprint Review, and Sprint Retrospective.
- **Scrum Artifacts**: Product Backlog (the list of work), Sprint Backlog (selected tasks for the current sprint), and the Increment (the usable product at the end of the sprint).

4.7.2 How It Works

- **Plan**: The Product Owner creates a prioritized list (Product Backlog), and the team picks tasks for the sprint.
- **Act**: Developers work on tasks during the sprint, with a daily meeting to synchronize.
- **Review**: The team demonstrates the completed work (Increment) and gathers feedback.
- **Improve**: The team holds a retrospective to identify improvements for the next cycle.

4.7.3 Why Scrum is useful

Scrum gives teams a rhythm. It helps them focus, inspect progress, and adapt.

4.8 Scaling agile methods

Agile works very well for small and medium teams. Scaling it to large systems is harder because of:

- coordination overhead
- architecture dependencies
- multiple teams
- compliance requirements
- integration complexity

4.8.1 Practical lesson

Large systems often need a hybrid approach:

- agile delivery inside teams
- architectural governance across teams
- stronger documentation where needed

> Tip
>
> Scrum is based on empiricism—meaning decisions are made based on observation and experience—and relies on transparency, inspection, and adaptation.

4.9 LAB 3 — Agile Simulation

4.9.1 Task

Form a small team and simulate one sprint for a student portal.

Roles:

- product owner
- developer
- tester
- Scrum master

4.9.2 Deliverables

- product backlog
- sprint backlog
- task board
- sprint review notes
- retrospective notes

---

5 Requirements Engineering

5.1 Definition

Requirements engineering is the process of discovering, analyzing, documenting, validating, and managing what a system should do.

It is one of the most important parts of software engineering because a project cannot succeed if it builds the wrong system.

5.2 Types of requirements

5.2.1 Functional requirements

These describe the services or functions the system must provide.

Examples:

- the system shall allow users to log in
- the system shall record student grades
- the system shall calculate monthly salaries

5.2.2 Non-functional requirements

These describe constraints or qualities.

Examples:

- response time
- security
- reliability
- usability

- legal compliance

> **Note**
>
> Functional requirements describe *what* the system does. Non-functional requirements describe *how well* or *under what constraints* it must do it.

5.2.3 Domain requirements

These come from the application domain itself.

Example:

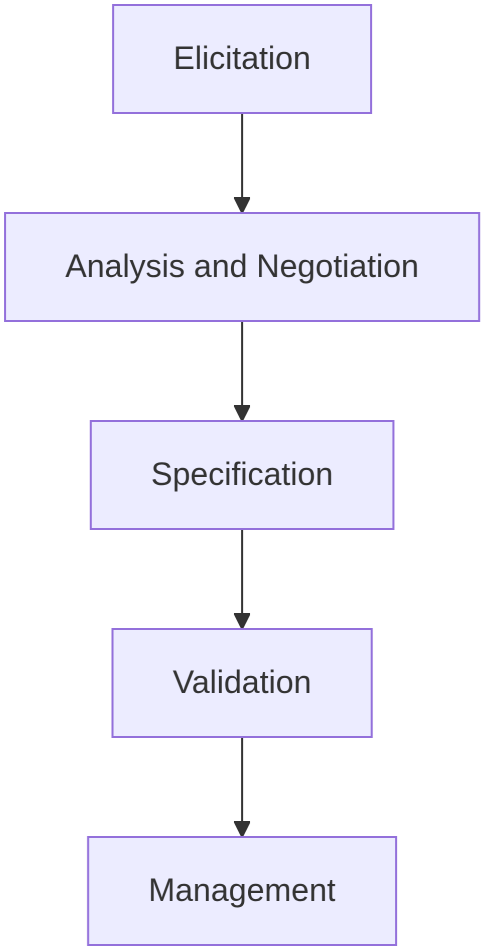
A medical system must follow healthcare rules and privacy regulations.

5.3 Why requirements are difficult

Requirements are often unclear because:

- users may not know exactly what they need
- business processes change
- stakeholders disagree
- some needs are implicit
- some requirements conflict

5.4 Requirements Engineering Process



5.4.1 Elicitation

Gather requirements from stakeholders.

Techniques:

- interviews
- observation
- questionnaires
- workshops
- scenarios
- prototyping

5.4.2 Analysis and negotiation

Check for:

- conflicts
- ambiguity
- missing requirements
- unrealistic requirements

5.4.3 Specification

Write requirements in a clear, organized form.

5.4.4 Validation

Confirm that requirements are:

- correct
- complete
- consistent
- feasible
- testable

5.4.5 Management

Keep requirements under control as they change.

5.5 Requirements documents

A good software requirements document should be:

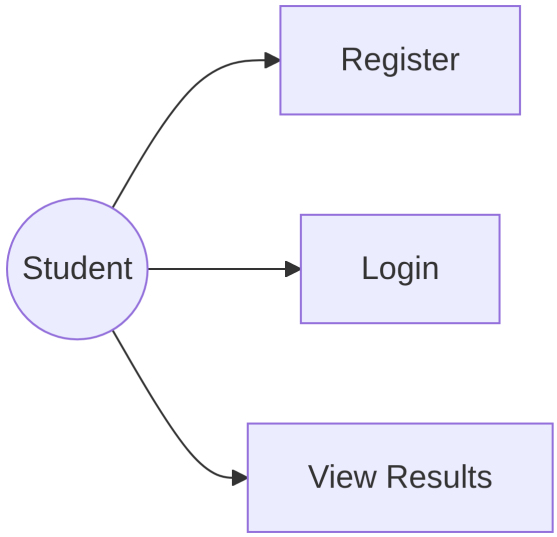
- clear
- complete
- consistent
- structured
- understandable by both users and developers

Typical contents:

- introduction
- system overview

- functional requirements
- non-functional requirements
- use cases
- assumptions
- constraints
- glossary

5.6 Use Case Diagram Example



5.7 Elicitation in real life

5.7.1 Scenario

A hospital wants a patient appointment system.

- Doctors ask for advanced scheduling rules.
- Receptionists want fast booking.
- Managers want reports.
- Patients want SMS reminders.

These stakeholders may not agree, so the analyst must negotiate and prioritize.

5.8 Validation methods

Requirements can be validated by:

- reviews
- prototyping
- test-case generation
- consistency checking
- stakeholder walkthroughs

5.9 Requirements management

Requirements change for many reasons:

- regulations
- customer priorities
- technology
- budgets
- discovered errors

> Tip
>
> The key is to track change, not pretend it will not happen.

5.10 LAB 4 — Requirements Engineering Practice

5.10.1 Task

Write a mini SRS for a:

- library system
- attendance system
- clinic appointment system

Include:

- 10 functional requirements
- 5 non-functional requirements
- 3 use cases
- assumptions
- constraints

---

6 System Modeling

6.1 Definition

System modeling is the process of creating abstract, visual, or mathematical representations of a system to understand, analyze, and communicate its structure, behavior, and requirements. It helps engineers and developers visualize different viewpoints—such as interactions, behavior, and structure—using notations like Unified Modeling Language (UML) to improve design and identify errors early.

6.2 Why model software systems?

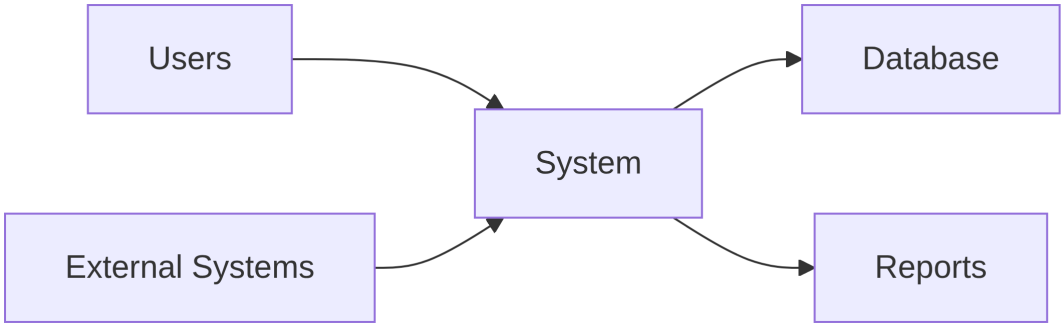
A model is a simplified representation of a system used to understand, analyze, or communicate design.

Models help engineers:

- reduce complexity
- communicate ideas
- validate requirements
- detect missing information
- document system behavior

6.3 Context models

A context model shows the system and its environment.



6.3.1 Why context matters

A system never exists alone. It interacts with users, databases, devices, and other systems.

6.4 Types of Models

6.4.1 Interaction models

Interaction models show how users and systems interact.

Common forms:

- use case diagrams
- sequence diagrams
- communication diagrams

6.4.1.1 Use case idea

Use cases describe how an actor uses the system to achieve a goal.

Example actors:

- student
- lecturer
- admin
- patient
- doctor

6.4.2 Structural models

Structural models describe the static organization of the system.

Typical UML diagram:

- class diagram
- component diagrams

6.4.2.1 Example concept

A student has a name, matric number, and may register for multiple courses.

6.4.3 Behavioral models

Behavioral models describe dynamic behavior over time.

Typical diagrams:

- state diagrams
- activity diagrams

6.4.3.1 Example

A course registration request may move through:

- created
- pending approval
- approved
- rejected

6.5 Model-driven engineering

Model-driven engineering emphasizes creating models that can be transformed into implementation.

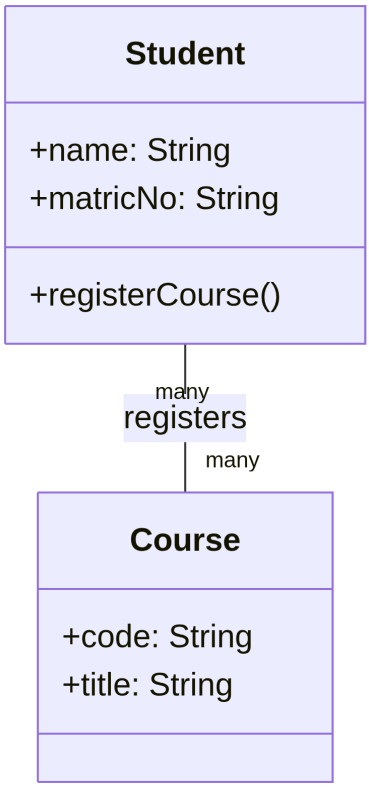
Benefits:

- higher-level thinking
- less manual coding
- better consistency
- stronger traceability

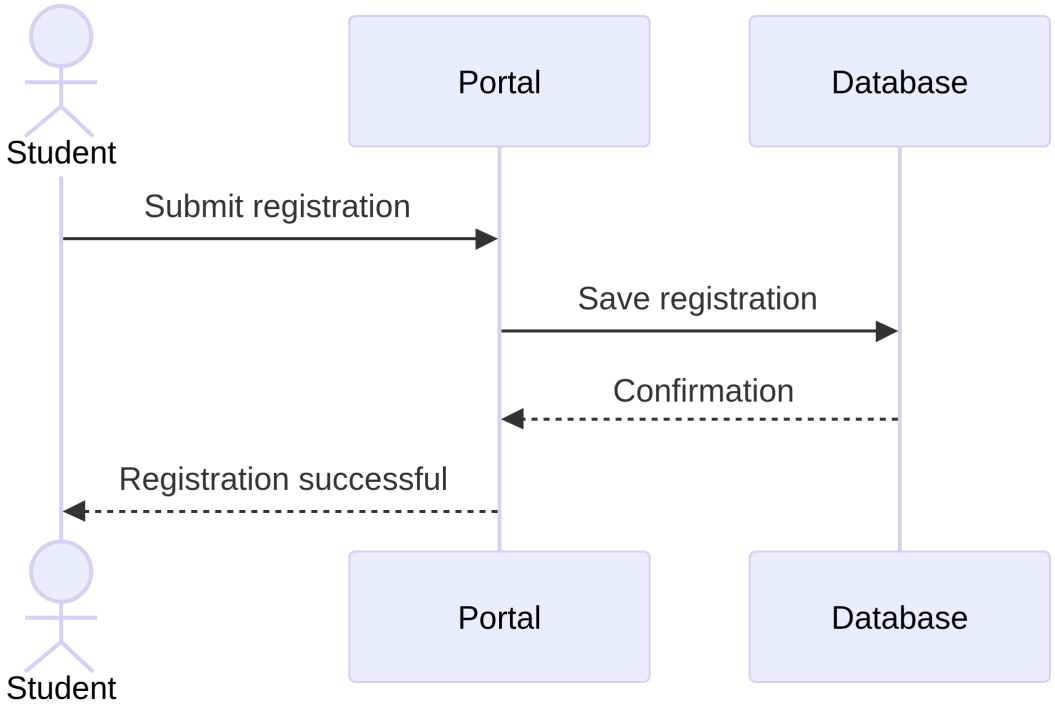
6.6 UML Diagrams

UML is useful because it gives a standard language for discussing software structure and behavior.

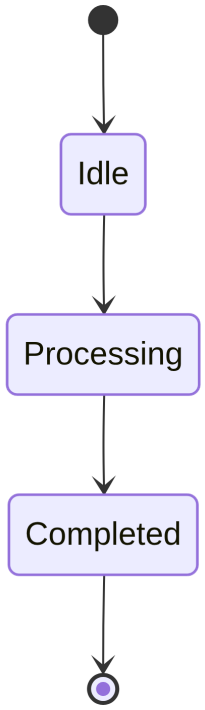
6.6.1 Simple class diagram example



6.6.2 Simple sequence diagram example



6.6.3 Simple state diagram example



6.7 LAB 5 — Build models for a small system

6.7.1 Task

Choose one:

- ATM
- clinic appointment system
- online bookstore

Create:

- context diagram
- use case diagram
- class diagram
- sequence diagram
- activity diagram
- state diagram

---

7 Architectural Design

7.1 What architecture is

Software architecture is the high-level organization of a software system: its major components, the relationships between them, and the principles governing their design and evolution.

Architecture is important because it determines whether the system can meet its:

- performance goals
- security goals
- availability goals
- safety goals
- maintainability goals

7.2 Architecture and requirements

Architectural design is closely tied to requirements engineering. In practice, some architectural decisions are made early because they strongly affect whether the system can satisfy its non-functional requirements.

7.2.1 Example

If security is critical, the architecture may need layers and strong access control.

If performance is critical, the architecture may need fewer distributed calls and localized computation.

7.3 6.3 Architectural views

A system can be described from different perspectives:

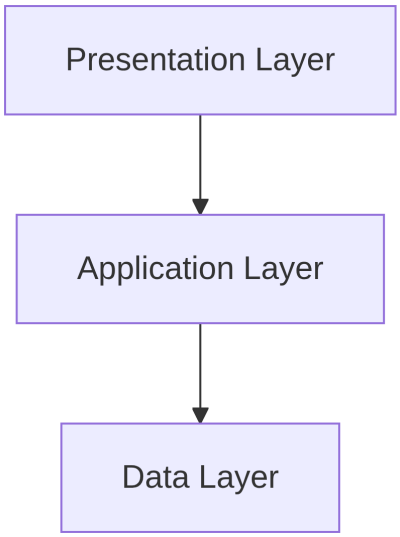
- conceptual view
- process view
- development view
- physical view

These views help different stakeholders understand different aspects of the system.

7.4 Architectural patterns

Patterns are reusable solutions to recurring architecture problems.

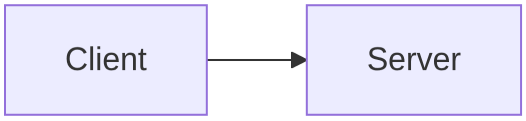
7.4.1 Layered architecture



Benefits:

- separation of concerns
- easier maintenance
- clearer security boundaries

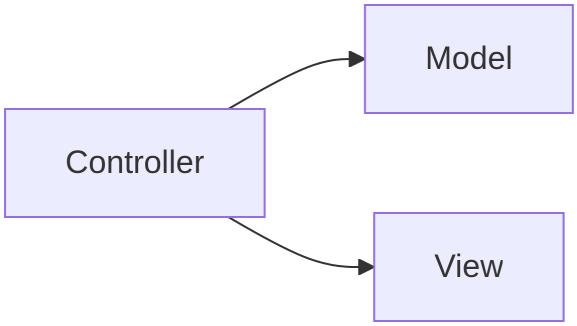
7.4.2 Client-server architecture



Used in:

- web applications
- enterprise systems
- database-based services

7.4.3 MVC Architecture



7.4.4 Repository architecture

A central data store is shared by multiple components.

Used in:

- compilers
- information systems
- integrated data platforms

7.5 Architectural trade-offs

Architecture involves compromise.

For example:

- large components may improve performance
- small components may improve maintainability
- redundancy may improve availability but increase cost

7.6 Application architectures

Common application architectures include:

- transaction processing systems
- language processing systems
- data-centered systems
- event-driven systems

7.6.1 Transaction processing example

A banking system receives a transaction, validates it, updates accounts, and records the result.

7.7 Architectural Design Decisions

Consider:

- Scalability
- Security
- Performance
- Maintainability

7.8 LAB 6 — Architecture Design

7.8.1 Task

Design the architecture of one of the following:

- hospital management system
- mobile money platform
- e-commerce site

Include:

- major components
- data flow
- architectural pattern choice
- non-functional justification

---

8 Design and Implementation

8.1 Design and implementation in software engineering

This chapter extends architecture into detailed design and implementation. Design turns architectural ideas into classes, interfaces, modules, and algorithms. Implementation turns those ideas into working code.

8.2 Object-Oriented Design

OO design is based on:

- Encapsulation
- Inheritance
- Polymorphism
- Abstraction

8.2.1 Why OO design matters

OO design helps manage complexity by organizing software around objects and responsibilities.

8.3 Design principles

Good design is:

- modular
- reusable
- understandable
- testable
- low in coupling
- high in cohesion

8.3.1 Cohesion and coupling

- **high cohesion** means each module does one focused job
- **low coupling** means modules depend on each other as little as possible

These are essential to maintainability.

8.4 Design Patterns

Design patterns are reusable solutions to common design problems.

8.4.1 Singleton

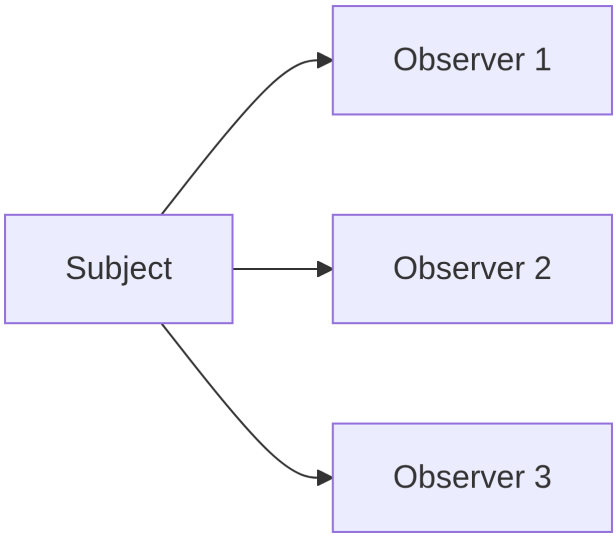
Ensures only one instance of a class exists.

8.4.2 Factory

Creates objects without exposing the exact creation logic.

8.4.3 Observer

Allows one object to notify others when its state changes.



8.5 Benefits of OO

8.5.1 Clean Code Principles

- Meaningful names
- Small functions
- DRY principle
- SOLID principles

8.5.2 Open source development

Open source software is widely used because it enables:

- reuse
- collaboration
- transparency
- rapid improvement

But teams must also manage:

- dependency risks
- licensing issues
- maintenance responsibilities
- security review

8.6 Implementation issues

Implementation is not only coding. It includes:

- language choice
- coding standards
- version control
- naming conventions
- error handling
- build scripts
- integration
- reuse of libraries

8.7 LAB 7 — OOP Design

8.7.1 Task

Design a class structure for a library system with:

- Book
- Member
- Loan
- Librarian

Explain:

- responsibilities
- relationships
- design patterns you may use

---

9 Software Testing

9.1 Definition:

Software testing is the systematic process of evaluating and verifying that a software application functions correctly, meets specified requirements, and is free of defects. Its primary goal is to identify bugs or gaps in functionality before the product reaches the end user, ensuring high quality and reliability.

9.2 Categories of Testing

Testing is generally divided into two main execution methods and two primary focuses:

- **Manual Testing**: Testers execute test cases manually, acting as end-users to find UI issues or usability flaws without automated scripts.
- **Automation Testing**: Using specialized tools (like Selenium or Playwright) to run repetitive test scripts quickly and efficiently.
- **Functional Testing**: Verifies what the system does—ensuring features like login or payment processing work according to business requirements.
- **Non-Functional Testing**: Checks how the system performs—evaluating speed, security, and stability under heavy loads.

9.3 Why testing matters

Testing finds defects before software is delivered. It is a critical validation activity.

Testing does not prove correctness, but it increases confidence that the software behaves as required.

9.4 Levels of Testing

9.4.1 Unit testing

Tests small parts of the system in isolation.

9.4.2 Integration testing

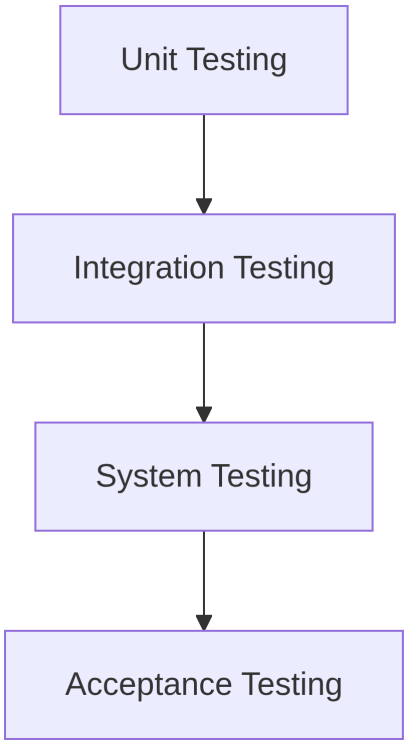
Tests interactions between components.

9.4.3 System testing

Tests the complete integrated system.

9.4.4 Acceptance testing

Checks whether the system meets user needs.



9.5 Development testing

This happens during development and is often done by developers themselves.

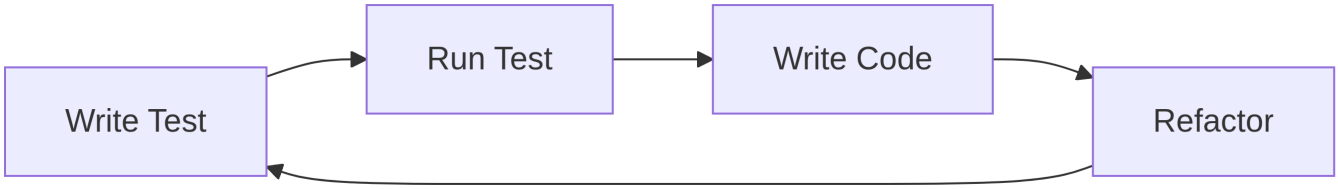
Purpose:

- catch defects early
- localize failures
- support refactoring

9.6 Test-Driven Development (TDD)

In TDD, the cycle is:

1. write a failing test
2. write minimal code to pass it
3. refactor



TDD is valuable because it makes requirements more concrete and encourages clean design.

9.7 Release testing

Release testing checks whether the software is ready for deployment to real users.

It focuses on:

- functionality
- reliability
- security
- performance
- compatibility

9.8 User testing

Users validate whether the system is acceptable in real use conditions. This includes:

- usability testing
- alpha testing
- beta testing

9.9 Testing strategy in practice

Good testing combines:

- developer testing
- automated tests
- regression tests
- user acceptance tests

9.10 Black Box vs White Box Testing

Black Box	White Box
External behavior	Internal structure
No code knowledge	Requires code knowledge

9.11 Test Cases

9.11.1 Definition

A test case is a set of conditions, steps, and variables used by testers to determine if a specific feature of a software application works correctly. It acts as a detailed instruction manual for validating a single functional requirement.

9.11.2 Key Components of a Test Case

To be effective, a test case must be clear, repeatable, and structured with the following elements:

- Test Case ID: A unique identifier (e.g., TC_LOGIN_01) for tracking purposes.
- Description: A clear summary stating what is being tested.
- Pre-conditions: Requirements that must be met before starting (e.g., user must be registered).
- Test Steps: Execution steps written in sequential order.
- Test Data: The specific inputs needed (e.g., usernames, passwords, file types).
- Expected Result: The correct, intended behavior of the system.
- Actual Result: What the system actually does during execution (filled out during testing).
- Status: Marked as Pass, Fail, or Blocked based on the results.

9.11.3 Practical Example: User Login Functionality

Field	Details
Test Case ID	TC_AUTH_01
Description	Verify successful login with valid credentials.
Pre-conditions	User has an active, registered account.
Test Steps	1. Navigate to the login page. 2. Enter a valid email. 3. Enter a valid password. 4. Click the "Submit" button.
Test Data	Email: test@example.com, Password: Password123
Expected Result	User is redirected to the dashboard, and a welcome message appears.

9.11.4 Best Practices for Writing Test Cases

- **Keep it Simple**: Write clear steps so any tester can execute them without guessing.
- **Focus on One Thing**: Test exactly one condition or flow per test case.
- **Include Negative Paths**: Test invalid inputs (e.g., wrong passwords) to check error handling.
- **Ensure Reusability**: Design cases that can be reused for future regression testing.

> **!** Important
>
> Test quality is not only about the number of tests. It is about whether the tests actually expose important defects.

9.12 LAB 8 — Software Testing

9.12.1 Task

Create tests for a login system.

Include:

- valid login
- invalid password
- empty username
- locked account
- repeated failed login attempts

Write test cases with expected results.

---

10 Software Evolution

10.1 Why software evolves

Software evolves because:

- requirements change
- bugs are found
- laws change
- technology changes
- users discover new needs
- systems must integrate with new services

10.2 Evolution process

10.3 Types of Maintenance

10.3.1 Corrective maintenance

Fixes defects.

10.3.2 Adaptive maintenance

Adapts software to a new environment.

10.3.3 Perfective maintenance

Improves performance or usability.

10.3.4 Preventive maintenance

Reduces future risks and technical debt.

10.4 Program evolution dynamics

Systems often become more complex over time unless they are actively refactored. Without care, systems accumulate:

- technical debt
- poor structure
- outdated dependencies
- hard-to-change code

10.5 Legacy systems

Legacy systems are older systems that are still valuable and still in use.

Common problems:

- poor documentation
- obsolete technology
- limited support
- difficult integration
- high maintenance cost

10.5.1 Typical evolution decision

An organization may choose to:

- keep the system as-is
- wrap it
- migrate it
- replace it
- re-engineer it

10.6 Software Evolution Process

Maintenance and business value

Maintenance is not secondary work. In many organizations, maintenance consumes more resources than initial development. That is why design for change matters from the beginning.

10.7 LAB 9 — Legacy System Analysis

10.7.1 Task

Choose one old system in a school, clinic, church, company, or office.

Analyze:

- current purpose
- current problems
- risks
- maintenance costs
- possible modernization strategy

---

11 Final Project

11.1 Build a Complete Software Engineering Project

Design a complete system using the Part I lifecycle:

1. define the problem
2. gather requirements
3. choose a process model
4. draw UML models
5. propose an architecture
6. design key classes

7. define a testing strategy
8. explain how the system will evolve

11.1.1 Suggested topics

- student portal
- clinic appointment system
- bus reservation system
- mobile payment app
- inventory management system

> **!** Important
>
> In real software engineering work, the best solutions are rarely perfect. They are the ones that are well-reasoned, testable, maintainable, and adaptable.

---

12 Consolidated revision guide

12.1 Examination Preparation Topics

Students should master:

- Software engineering principles
- Process models
- Agile methodologies
- UML diagrams
- Requirements engineering
- Architecture
- Testing strategies
- Software maintenance
- Ethics

12.2 Must-know definitions

- **Software engineering**: engineering discipline for software production
- **Software process**: structured set of development activities
- **Agile**: iterative, collaborative, change-friendly development
- **Requirements engineering**: discovering and managing system requirements
- **Model**: simplified representation used to understand a system
- **Architecture**: high-level structure of a software system
- **Testing**: checking a system to find defects and validate behavior
- **Evolution**: changing software after delivery

12.3 Exam traps to avoid

1. Saying software engineering is just coding
2. Confusing functional and non-functional requirements
3. Treating architecture as the same as detailed design
4. Thinking testing proves correctness
5. Ignoring ethics and professionalism
6. Claiming one process model is always best

12.4 Short exam-style answers

12.4.1 Why is software engineering needed?

Because software systems are complex, expensive to change, and critical to modern society, so they must be developed systematically.

12.4.2 Why are requirements important?

Because the system cannot be successful unless it solves the right problem and satisfies stakeholders.

12.4.3 Why is architecture important?

Because architecture determines whether the system can meet its performance, security, availability, and maintainability goals.

12.4.4 Why is evolution important?

Because software is used in changing environments and must be updated throughout its life.

---

13 Recommended Tools

Category	Tools
UML Modeling	StarUML, Draw.io
Version Control	Git
IDEs	VS Code, IntelliJ
Testing	JUnit, PyTest
Agile Boards	Trello, Jira
Documentation	Quarto, Markdown

---

14 Final Advice to Students

Software engineering is not simply about coding.

It is about:

- Building reliable systems
- Solving human problems
- Managing complexity
- Working in teams
- Designing for change
- Engineering software professionally

A great software engineer combines:

- Technical ability
- Communication skills
- Ethical responsibility
- Problem-solving capability
- Continuous learning