SOFTWARE DESIGN & DEVELOPMENT METHODOLOGIES

UNIT 1: Foundations of Software Engineering, Software Process Models, and Software Requirements Specification

1.1 Introduction

Software now runs essential sectors like banking, healthcare, transportation, education, and government, so it must be systematically engineered rather than treated as simple programming. Software development is not just coding; it involves understanding requirements, designing solutions, estimating costs, ensuring quality, testing, and maintaining systems. Poor execution of these steps can lead to unreliable, costly, or unmaintainable software.

This chapter introduces the importance of software engineering, explains software as a product, compares development life cycle models, and builds understanding of requirements and the Software Requirements Specification (SRS). It aims to help students clearly understand concepts, compare models, and explain how real-world requirements are captured.

1.2 The Need for Software Engineering

1.2.1 Why software engineering became necessary

In the early days of computing, many programs were small. A single person could understand the whole problem, write the code, test it informally, and make changes without much difficulty. As computer systems became larger and more critical, this informal approach failed.

Large systems introduced new difficulties:

- The number of features increased,
- Many developers worked together,
- Users from different departments had conflicting needs,
- Systems had to continue working for years,
- Failures became costly or dangerous,
- Maintenance took more effort than initial development.

This led to what is often called the **software crisis**: projects were over budget, late, low in quality, and hard to maintain. Software engineering emerged as the disciplined response to this problem.

1.2.2 What problems does software engineering solve?

Software engineering helps reduce chaos by providing methods, processes, documentation, quality practices, and management techniques. It aims to make software development:

- more predictable,
- more controllable,
- more measurable,
- more reliable,
- easier to maintain,
- better aligned with user needs.

A useful analogy is building construction. A simple hut may be built informally, but a hospital or airport cannot be built without drawings, standards, inspections, and project planning. Software engineering treats software systems with the same seriousness.

1.2.3 Real-world consequences of poor engineering

When software is poorly engineered, the consequences may include:

- Wrong output,
- Data loss,
- System crashes,
- Security breaches,
- Customer dissatisfaction,
- Financial loss,
- Legal liability,
- Safety risks.

You should understand that software engineering is not “extra paperwork.” It is a way of preventing expensive mistakes.

1.3 Definition of Software Engineering

Software engineering is the systematic, disciplined, and measurable application of engineering principles to the development, operation, maintenance, and evolution of software.

This definition contains several important ideas.

1.3.1 Systematic

Software development should follow an organized method rather than random trial and error.

1.3.2 Disciplined

There must be rules, standards, reviews, documentation, and quality checks.

1.3.3 Measurable

Good engineering relies on evidence. Teams track cost, defects, schedule, test coverage, effort, and performance.

1.3.4 Full life cycle

Software engineering is not limited to coding. It includes:

1. Requirements,
2. Design,
3. Implementation,
4. Testing,
5. Deployment,
6. Maintenance,
7. Improvement.

1.3.5 NOTE:

You should be able to distinguish between:

- **Programming**: writing code,
- **Software engineering**: applying an entire engineering process to build quality software.

1.4 Characteristics of Software

Software is different from physical products. Understanding this helps you compare software with hardware.

i. Software is intangible

You cannot touch software the way you touch a machine. You can store it on physical media, but the software itself is a logical product.

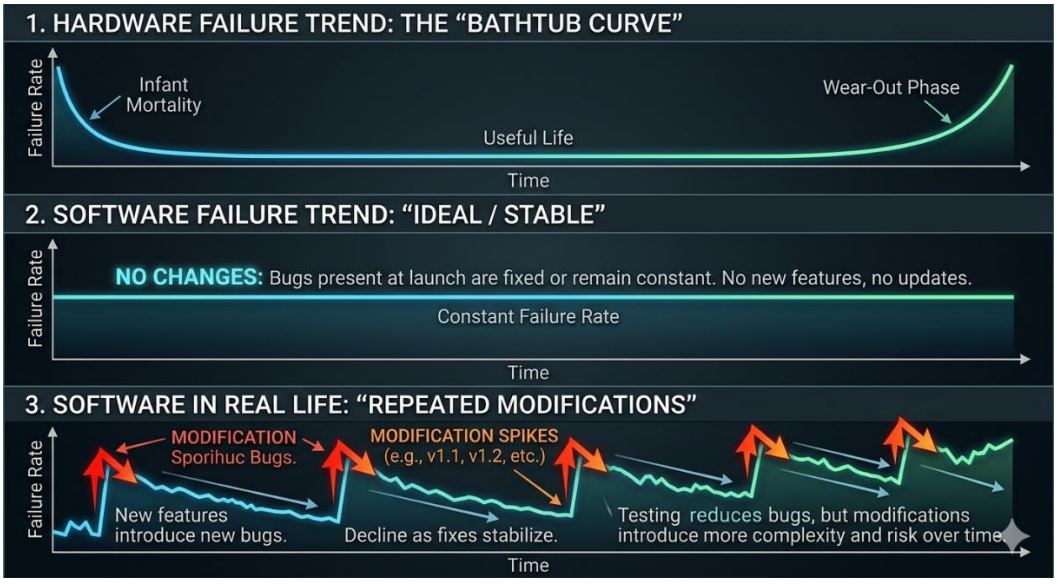
ii. Software is developed, not manufactured in the traditional sense

A chair can be manufactured again and again, and each additional chair requires materials and factory work. Software is different. Creating the first version is costly, but making copies is very cheap.

iii. Software does not wear out like hardware

A physical machine degrades with use. Software usually does not "wear out" physically. However, software appears to deteriorate because environments change, user needs change, and modifications introduce defects.

A simple representation:



The lesson is that software quality often drops because of poor changes, not because of physical aging.

iv. Software is often custom-built

Although reusable components are increasingly common, many systems still contain special features designed for one organization, one context, or one workflow.

v. Software is highly complex

Unlike many physical products, software may contain huge numbers of logical conditions, interfaces, data structures, and hidden interactions.

vi. Software is continually changed

Laws change, devices change, user expectations change, and security threats change. Therefore, software evolution is a normal part of software life.

vii. Software must conform to external environments

It must work with hardware, networks, databases, regulations, browsers, operating systems, and user practices.

For example,
Consider a ***college result management system***. Even after the first release, the system may need updates because:

- grading rules change,
- new departments are added,
- report formats change,
- security policies are updated,
- mobile access becomes necessary.

This shows why maintenance and requirements clarity matter.

1.5 Software Myths

Software myths are false beliefs that cause poor decisions in projects.

Software myths can be grouped into three broad categories:

1. **Management myths**,
2. **Customer myths**,
3. **Developer myths**.

1.5.1 Management myths

A common myth is:

"If the project is late, add more programmers and it will catch up."

This is often false, because new people need training, communication overhead increases, and existing staff must spend time helping them.

Another myth:

"We already have standards, so quality is guaranteed."

Standards help, but they do not replace skilled analysis, design, review, and testing.

1.5.2 Customer myths

A common customer myth is:

"A general statement of objectives is enough to start coding; details can be added later."

This is dangerous. Ambiguous requirements produce rework, cost increases, and wrong functionality.

Another myth:

"Because software is flexible, changes are easy and cheap."

Some changes are easy, but others affect architecture, interfaces, test cases, documents, database design, and schedules.

1.5.3 Developer myths

A common myth is:

"Once the program works, the job is finished."

Actually, delivery is often the beginning of maintenance work.

Another myth:

"The only deliverable that matters is working code."

This is false because projects also require:

- Requirement documents,
- Design records,
- Test cases,
- User manuals,
- Maintenance information.

1.6 Program versus Software Product

This distinction is extremely important and often misunderstood.

1.6.1 What is a program?

A **program** is a set of instructions written to solve a problem or perform a task. A student writing a C program to sort numbers is creating a program.

1.6.2 What is a software product?

A **software product** is much broader. It includes not only code, but also:

- requirements,
- design,
- documentation,
- test cases,
- error handling,
- packaging,
- installation support,
- user guidance,
- maintenance support.

1.6.3 Key differences

Aspect	Program	Software Product
Purpose	Solves a task	Serves users in real environments
User base	Often limited	Usually broader and more varied
Documentation	Minimal or absent	Essential
Testing	Informal possible	Systematic and extensive
Maintenance	Limited	Long-term
Quality expectations	May be basic	Must meet standards
Deployment	Often local	Real-world installation/use

1.6.4 Example

A student writes a Python script to calculate library fines. That is a program.

A complete **Library Management System** for a college, with user login, issue/return tracking, fine calculation, audit logs, reports, backups, user manuals, and support documentation, is a software product.

NOTE: If asked to differentiate,

A program is mainly code written for a specific task, while a software product is an engineered deliverable consisting of code plus documentation, testing, support material, and quality features for real users.

1.7 Software Development Life Cycle Models

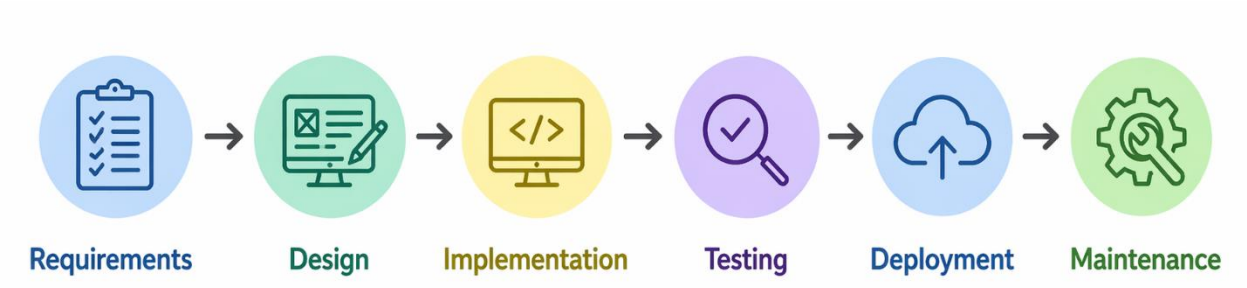
There are several life cycle models: Waterfall, Prototyping, Spiral, Iterative Enhancement, Agile, and Object-Oriented approaches, with comparison of advantages and disadvantages.

1.7.1 What is a Software Development Life Cycle (SDLC) model?

An SDLC model is a structured way to organize the major activities of software development. It tells us:

- what stages are performed,
- in what order,
- how feedback happens,
- when testing occurs,
- how changes are handled.

Typical activities include:



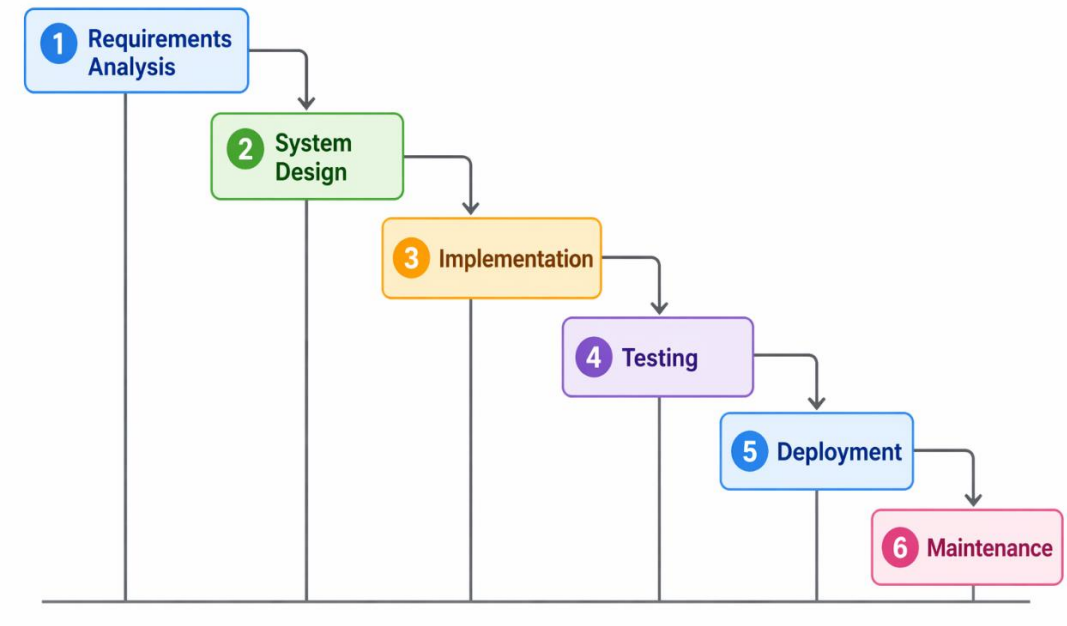
Different models arrange these activities differently.

1.8 Waterfall Model

1.8.1 Introduction

The Waterfall model is one of the earliest and most well-known software process models. It follows a largely sequential flow, where one phase must be completed before the next begins.

1.8.2 Waterfall Model Diagram



1.8.3 Main idea

The model assumes that requirements can be understood clearly at the beginning, and the project can move step-by-step through distinct stages.

Waterfall Model Steps

1. Requirements Analysis

This is the first stage where the development team finds out **what the users need** from the system. The team collects information from users, managers, and stakeholders through interviews, observation, forms, or meetings.

Example:

For a college registration system, students may need to register courses online, while administrators need reports.

2. System Design

In this stage, the team plans **how the system will be built**. They decide the system structure, database design, user interface, modules, security, and how different parts will work together.

Example:

For the college system, designers decide how the login page, course selection page, and database will be arranged.

3. Implementation

This is the coding stage. Developers convert the design into actual software using programming languages. Each module or part of the system is created according to the design.

Example:

Programmers write code for student login, course registration, and report generation.

4. Testing

In this stage, the software is checked to find errors or defects. Testing ensures that the system works according to the requirements and produces correct results.

Example:

Testers check whether a student can register only for available courses and whether wrong login details are rejected.

5. Deployment

Deployment means delivering and installing the completed system for real users. The software is placed in the actual working environment where users can start using it.

Example:

The college registration system is uploaded to the college server and students begin using it.

6. Maintenance

After deployment, the system is monitored and improved. Maintenance includes fixing bugs, updating features, improving performance, and adapting the system to new requirements.

Example:

If the college adds a new department or changes registration rules, the system must be updated.

1.8.4 Advantages

The Waterfall model is easy to understand and manage. It works reasonably well when:

- requirements are stable,
- the technology is familiar,
- the scope is clear,
- documentation is important,
- the domain is regulated.

Its structure also makes progress tracking simpler because milestones are clearly defined.

1.8.5 Disadvantages

Its biggest weakness is inflexibility. Real-world projects often change after development starts. If the team discovers missing or incorrect requirements late, going back is costly.

Because testing happens relatively late, defects in understanding may remain hidden for a long time.

1.8.6 Best use cases

Waterfall is better suited to projects where:

- requirements are well known,
- heavy documentation is required,
- safety or compliance processes favor
- changes are rare,
- formal phase approvals.

Example:

Suppose a college wants a simple payroll system and rules are fixed by policy. If requirements are clear and unlikely to change quickly, a sequential process may work.

Note:

You should not say Waterfall is “bad.” It is more accurate to say it is **less suitable for changing requirements**.

1.9 Prototyping Model

1.9.1 Introduction

The Prototyping model is used when requirements are unclear or users are unable to describe exactly what they need in the beginning.

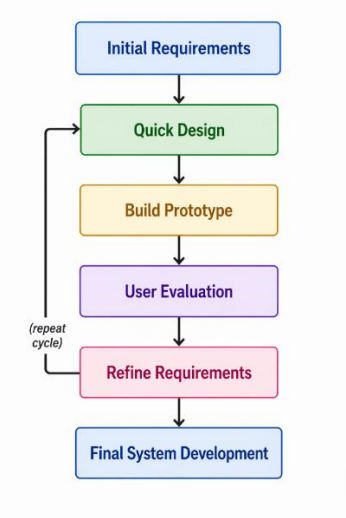
A **prototype** is an early working model of a system or a part of the system.

1.9.2 Why prototyping helps

Users often react better to something they can see than to abstract verbal descriptions. A prototype helps reveal:

- missing requirements,
- misunderstood functions,
- usability issues,
- layout preferences,
- workflow expectations.

1.9.3 Prototyping Model Diagram



1.9.4 Types of prototypes

i. **Throwaway prototype:** Built only to understand requirements, then discarded.

ii. **Evolutionary prototype:** Improved step by step until it becomes part of the final system.

1.9.5 Advantages

Prototyping improves communication with users and reduces requirement ambiguity. It is especially useful for:

- User interface intensive systems,
- Innovative products,
- Situations where users are uncertain,
- Systems with complex interactions.

1.9.6 Disadvantages

Users may think the prototype is nearly finished, even when it is only experimental. Developers may also take shortcuts in prototype code and later carry poor design into the final product.

Example

Imagine developing a **hospital appointment booking portal**. Doctors, reception staff, and patients may all have different ideas about the screens and workflow. A prototype can help finalize those requirements before full development.

1.10 Spiral Model

1.10.1 Introduction

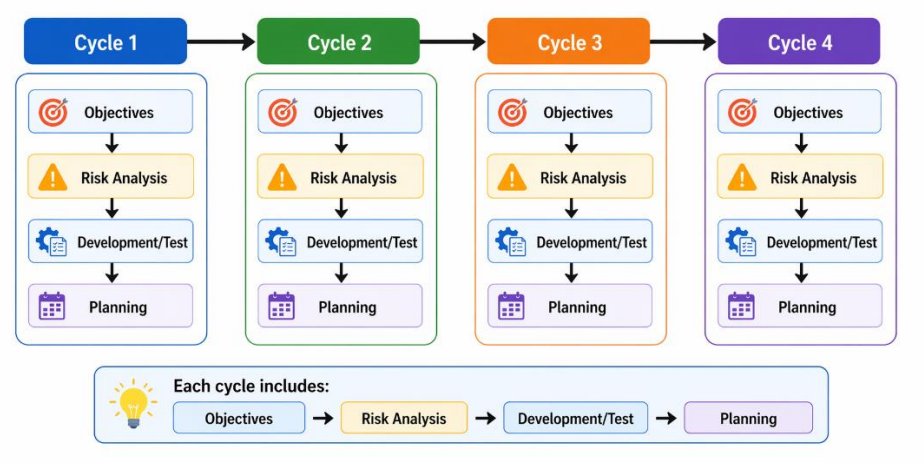
The Spiral model combines iterative development with systematic risk analysis. It is especially valuable in large, complex, high-risk projects.

1.10.2 Core idea

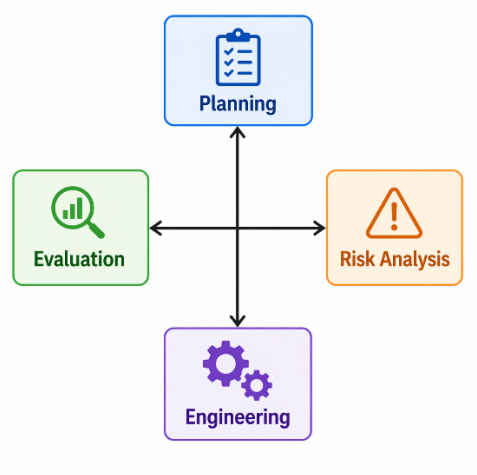
Development proceeds in repeated cycles, called spirals. In each cycle, the team:

1. Determines objectives,
2. Identifies and analyzes risks,
3. Develops and verifies the next version,
4. Plans the next iteration.

1.10.3 Conceptual diagram



A more visual text form:



1.10.4 Why risk matters

Large projects fail for many reasons:

- unclear requirements,
- cost overruns,
- immature technology,
- integration difficulty,
- performance uncertainty,
- security concerns.

The Spiral model addresses these by making risk analysis central.

1.10.5 Advantages

- Strong focus on risk management
- Suitable for large and complex projects
- Supports iterative refinement
- Allows customer feedback over time

1.10.6 Disadvantages

- More complex to manage
- Requires expertise in risk analysis
- Can be expensive for small projects
- Not ideal when risks are small and scope is simple

Example

A national-level digital identity system or large banking platform may involve security, scalability, legal compliance, and integration risks. A risk-driven process is more appropriate here than a simple linear model.

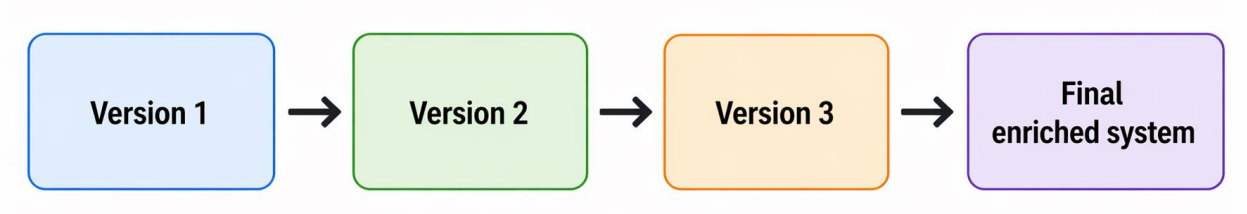
1.11 Iterative Enhancement Model

1.11.1 Introduction

In the Iterative Enhancement model, the system is built and improved in repeated increments. Rather than delivering the whole system at once, the team delivers a basic version and then adds enhancements.

1.11.2 Main logic

The product grows through iterations:



Each version includes analysis, design, coding, and testing for the selected features.

1.11.3 Benefits

This model allows early delivery of useful functionality. Users do not need to wait until the entire system is complete. Feedback from early versions can improve later ones.

1.11.4 Challenges

The architecture must be planned carefully so future additions fit properly. If the team keeps adding features without a strong design foundation, the system may become messy.

Example

A college portal might first release:

- Iteration 1: student login and profile,
- Iteration 2: course registration,
- Iteration 3: attendance view,
- Iteration 4: fee payment and reports.

1.12 Agile Model

1.12.1 Introduction

Agile development is not a single rigid method but a family of approaches that emphasize adaptability, collaboration, frequent delivery, and close customer involvement.

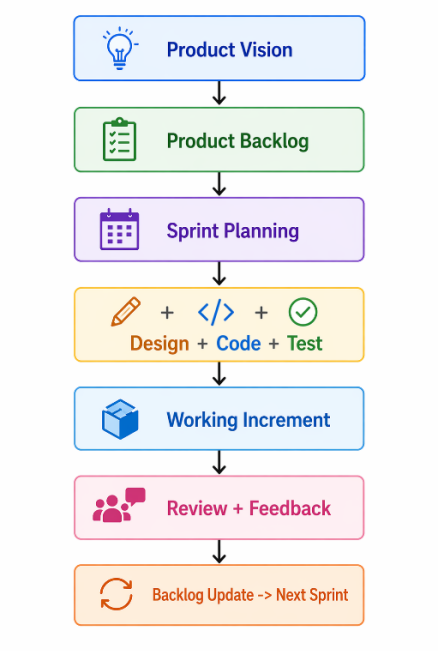
1.12.2 Basic philosophy

Agile recognizes that change is normal. Instead of trying to predict every detail far in advance, Agile teams work in short cycles and continuously refine the product.

1.12.3 Common Agile features

- short iterations, often called sprints,
- regular customer feedback,
- prioritized backlog,
- working software delivered frequently,
- team collaboration,
- continuous testing and integration,
- responsiveness to change.

1.12.4 Agile workflow diagram



1.12.5 Why Agile became popular

Many projects operate in environments where:

- user needs evolve,
- market conditions change,
- technology changes rapidly,
- early feedback is valuable.

Agile helps teams adapt quickly while still delivering value.

1.12.6 Advantages

- Good for changing requirements,
- Faster delivery of usable increments,
- Strong customer interaction,
- Problems can be discovered earlier,
- Encourages team communication.

1.12.7 Disadvantages

- Difficult if customer involvement is weak,
- Scope can drift,
- Documentation may become insufficient if poorly managed,
- Large distributed teams may struggle without discipline,
- Requires skilled, collaborative team members.

Example

Suppose a startup is building a food delivery app. Customer expectations about offers, payment methods, live tracking, and restaurant filtering may change quickly. Agile is often more suitable here than Waterfall.

Note:

You should not define Agile as "no planning." Agile does include planning, but planning is done continuously and incrementally.

1.13 Object-Oriented Model

1.13.1 Introduction

The Object-Oriented model refers to developing software around **objects**, **classes**, and their interactions, often using object-oriented analysis and design principles.

1.13.2 Central idea

Instead of seeing a system only as a sequence of functions, object-oriented development represents the system as a set of interacting objects that combine data and behavior.

1.13.3 Basic concepts

- **Object**: an entity with state and behavior
- **Class**: a blueprint for objects
- **Encapsulation**: bundling data and methods together
- **Inheritance**: deriving a new class from an existing class
- **Polymorphism**: one interface, multiple implementations
- **Abstraction**: focusing on essential features

1.13.4 Example

Consider a library system:

Class: Book

Attributes: bookId, title, author, status

Methods: issue(), return(), reserve()

Class: Student

Attributes: studentId, name, course

Methods: borrowBook(), returnBook()

Class: Librarian

Attributes: staffId, name

Methods: addBook(), removeBook(), generateReport()

This kind of modelling helps represent real-world entities naturally.

1.13.5 Advantages

- Supports reuse,
- Improves maintainability,
- Matches real-world modelling,
- Encourages modular design,
- Works well for large systems.

1.13.6 Disadvantages

- Requires careful design skill,
- May be unnecessarily complex for very small problems,
- Poor class design can reduce benefits.

1.14 Comparison of Life Cycle Models

Students should be able not only to define each model, but also compare them. Below is a strong exam-oriented comparison.

Model	Best When	Major Strength	Major Weakness
Waterfall	Requirements are stable	Clear structure and documentation	Poor flexibility
Prototyping	Requirements are unclear	Helps discover user needs	Can create unrealistic expectations
Spiral	Risks are high	Strong risk management	Complex and costly
Iterative Enhancement	Early partial delivery is useful	Improves system step by step	Needs good architecture
Agile	Requirements change often	Adaptability and fast feedback	Needs active collaboration
Object-Oriented approach	Complex systems needing modularity	Reuse and maintainability	Requires design maturity

1.14.1 A decision framework

"Which model is best?" the correct answer is not one fixed model. The correct answer is:
The choice depends on project size, clarity of requirements, risk level, customer involvement, and need for flexibility.

1.14.2 Comparative example

Imagine three projects:

1. **Income tax rule calculator with fixed rules**
   Waterfall may work.

2. **New mobile app with uncertain interface expectations**
   Prototyping or Agile may work better.

3. **Defense or aerospace control software with technical uncertainty and serious risks**
   Spiral is more appropriate.

1.15 Software Requirement Analysis

1.15.1 What is a requirement?

A **requirement** is a condition or capability needed by a user or a system to solve a problem or achieve an objective. In simpler terms, a requirement states what the software must do or what constraints it must satisfy.

1.15.2 Types of requirements

1. Functional requirements

These describe services or functions the system **must** perform.

Examples:

- The system shall allow students to register for courses.
- The system shall generate semester grade reports.

2. Non-functional requirements

These describe quality attributes or constraints.

Examples:

- The system shall respond within 2 seconds for normal queries.
- Only authorized staff shall access payroll data.
- The system shall be available 99% of working hours.

3. Domain requirements

These arise from the application domain.

Example:

- The result processing system shall calculate grades according to university regulations.

1.16 Why good requirements matter

Poor requirements are one of the main causes of software failure. If requirements are wrong, even perfectly written code will produce the wrong system.

A good requirement process helps:

- reduce rework,
- improve estimates,
- improve customer satisfaction,
- reduce disputes,
- support testing,
- guide design,
- support maintenance.

1.16.1 Example of poor requirement

"System should be user-friendly."

This is vague. Different people will interpret it differently.

1.16.2 Better version

"The system shall allow a trained clerk to complete a student registration in no more than 3 minutes under normal working conditions."

This is clearer and more measurable.

1.17 Developing SRS from Business Requirements

1.17.1 Business requirements versus software requirements

A business requirement describes the organizational goal. A software requirement translates that goal into system behavior.

Example

Business requirement:

Reduce student registration delays at the beginning of each semester.

Derived software requirements:

- The system shall allow online course registration.
- The system shall prevent registration for full sections.
- The system shall verify prerequisite completion.
- The system shall generate a registration confirmation slip.

This conversion is extremely important in requirements engineering.

1.17.2 Classroom interpretation

Business requirements explain **why** the project exists.

Software requirements explain **what** the system must do.

1.18 Requirement Process

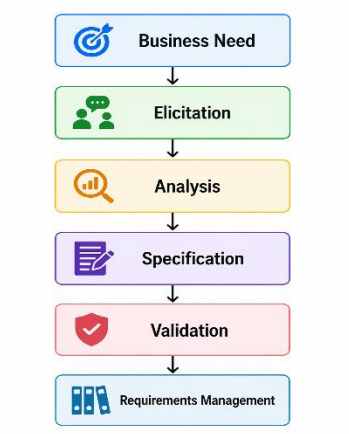
The requirement process is often called **requirements engineering**.

1.18.1 Main stages

A common requirement process includes:

1. Inception or feasibility understanding,
2. Elicitation,
3. Analysis,
4. Specification,
5. Validation,
6. Management.

1.18.2 Diagram



1.18.3 Elicitation

This means gathering requirements from stakeholders.

Methods include:

- Interviews,
- Questionnaires,
- Observation,
- Document study,
- Workshops,
- Brainstorming,
- Prototyping.

1.18.4 Analysis

In analysis, the team resolves conflicts, removes ambiguity, prioritizes needs, and checks feasibility.

1.18.5 Specification

The requirements are documented formally, usually in the SRS.

1.18.6 Validation

The team checks whether requirements are correct, complete, consistent, and testable.

1.18.7 Management

Requirements often change. Management includes version control, prioritization, traceability, and change handling.

1.19 Requirement Specification

Requirement specification means writing requirements clearly and systematically so that developers, testers, managers, and users share the same understanding.

A requirement is well specified when it is:

- Clear,
- Precise,
- Unambiguous,

- Testable,
- Relevant,
- Feasible.

1.19.1 Example of bad specification

"The system should be fast."

Problems:

- What does "fast" mean?
- For which operation?
- Under what load?

1.19.2 Improved specification

"The system shall display student records within 2 seconds for 95% of requests during normal office hours."

This is more measurable.

1.20 Desirable Characteristics of a Good SRS

This is a very important theory topic.

A good SRS should be:

i. **Correct:** It should reflect actual stakeholder needs.
ii. **Unambiguous:** Each requirement should have only one interpretation.
iii. **Complete:** All significant requirements, interfaces, constraints, and responses should be included.
iv. **Consistent:** Requirements should not conflict with one another.
v. **Ranked or prioritized:** Important requirements should be identified as essential, desirable, or optional where appropriate.
vi. **Verifiable:** It should be possible to check whether each requirement has been satisfied.
vii. **Modifiable:** The document should be organized so changes can be made without confusion.
viii. **Traceable:** Each requirement should be traceable to its source and forward to design, code, and test cases.

1.20.1 Memory aid

A good SRS must be **clear, complete, consistent, correct, and checkable**.

1.21 Components of an SRS

An SRS structure can vary by organization, but a standard academic format often contains the following.

1. **Introduction**

Includes purpose, scope, definitions, references, and overview.

2. **Overall description**

Describes product perspective, product functions, user classes, assumptions, and constraints.

3. **Specific requirements**

This is the heart of the document. It includes detailed functional and non-functional requirements.

4. **External interface requirements**

User interfaces, hardware interfaces, software interfaces, communication interfaces.

5. **Performance requirements**

Speed, capacity, throughput, response time.

6. **Design constraints**

Standards, legal constraints, hardware limitations, policy constraints.

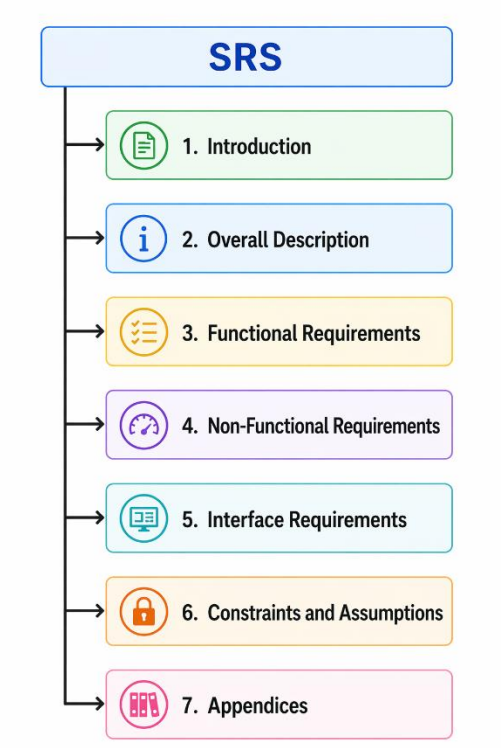
7. **Quality attributes**

Reliability, maintainability, portability, usability, security.

8. **Appendices or supporting material**

Glossary, models, references, data dictionary.

Simplified SRS template diagram



1.22 Structure of a Requirements Document

A requirements document should not be a random list of wishes. It should be logically organized.

A good sequence is:

1. Purpose of the system,
2. Scope and objectives,
3. Stakeholder context,
4. System overview,
5. Detailed functional requirements,
6. Non-functional requirements,
7. Interfaces,
8. Constraints,
9. Assumptions,
10. Acceptance criteria.

1.22.1 Why structure matters

A poorly structured document causes:

- Repeated statements,
- Contradictory requirements,
- Missing details,
- Difficult maintenance.

A well-structured document supports communication across the project.

1.23 Requirements Gathering

Requirements gathering is the practical activity of finding out what stakeholders need.

1.23.1 Who are stakeholders?

Stakeholders include anyone affected by the system:

- end users,
- managers,
- operators,
- customers,
- support staff,
- regulators,
- technical teams.

1.23.2 Gathering techniques

i. **Interview:** Useful when deep understanding is needed.
ii. **Questionnaire:** Useful for large numbers of users.
iii. **Observation:** Useful when users perform tasks they may find hard to explain.
iv. **Document analysis:** Useful when old forms, reports, policies, or manuals already exist.
v. **Workshop / Joint Application Development (JAD) session:** Useful when multiple stakeholders must agree.
vi. **Prototype:** Useful when users respond better to a visible model.

1.23.3 Example: College fee payment system

To gather requirements, the analyst may:

- interview accountants,
- observe fee counters,
- review payment rules,
- study current receipts,
- ask students about usability problems,
- Build a screen prototype for validation.

1.23.4 Common problems in gathering

- Users do not know exactly what they need,
- Stakeholders disagree,
- Hidden assumptions exist,
- Some users focus only on current system problems,
- Analysts misunderstand domain terms.

This is why elicitation must be followed by analysis and validation.

1.24 Creating a Backlog in Agile

1.24.1 What is a backlog?

A **product backlog** is an ordered list of features, fixes, enhancements, and technical tasks that may be worked on in future iterations.

It is dynamic, not frozen.

1.24.2 Backlog items

Backlog items are often written as **user stories**.

Example:

As a student, I want to view my attendance percentage so that I can know whether I meet exam eligibility.

This format highlights:

- the user,
- the desired capability,
- the value.

1.24.3 From requirement to backlog item

Formal requirement:

- The system shall allow students to view semester attendance.

Agile backlog story:

- As a student, I want to see my attendance by subject so that I can track eligibility.

1.24.4 Backlog prioritization

Items are ordered by business value, urgency, dependency, and risk.

High-priority items are implemented first.

1.24.5 Backlog example for a college portal

Priority	**Backlog Item**
High	Student login
High	Course registration
High	Prerequisite validation
Medium	Attendance dashboard
Medium	Fee payment receipt download
Low	Dark mode interface

1.24.6 Why backlog matters

The backlog provides flexibility. It lets teams handle changing requirements without losing organization.

1.25 Integrated Example: From Need to SRS to Backlog

Let us connect the whole chapter using one example.

Problem: A college faces long queues during semester course registration.

Business requirement

Reduce registration waiting time and improve accuracy.

Gathered needs

Students want online registration.

Staff want fewer manual errors.

Administrators want reports.

The system must block invalid subject combinations.

Functional requirements

- The system shall authenticate students.
- The system shall display available courses.
- The system shall verify prerequisites.
- The system shall prevent duplicate registration.
- The system shall generate confirmation receipts.

Non-functional requirements

- Response time shall be under 3 seconds for routine actions.
- Only authorized users shall access admin functions.
- The system shall maintain transaction logs.

Possible SRS section

These requirements are documented formally with interfaces, constraints, assumptions, and validation criteria.

Agile backlog transformation

- As a student, I want to log in securely.
- As a student, I want to choose courses from my department.
- As the system, I want to block courses without prerequisites.
- As an admin, I want registration reports.