Software Engineering: Software Requirements Analysis

@overview

In this module, you will learn how to understand, capture, analyse, document, and prioritise software requirements. You will explore why requirements are central to successful projects, how to work with stakeholders to discover what they really need, and how to structure and prioritise those needs using frameworks such as FURPS+ and MoSCoW. By the end of this module, you will be able to produce a well-organised, high-quality set of requirements for a software system.

Lesson Preparation

For this lesson, it is helpful (but not essential) to have:

- A simple example system in mind (e.g. an online library, a mobile banking app, or an AI-based recommendation system).
- Some experience working in or observing a software project (e.g. coursework, internship, or hobby project).
- Pen and paper or a text editor to draft your own requirements.

---

1. Introduction: What Are Software Requirements?

Software requirements describe **what** a system should do and the **constraints** under which it must operate. They capture the needs of customers, users, regulators, and other stakeholders in a way that both business and technical people can understand.

- A **requirement** is a statement about a needed capability, behaviour, quality, or constraint.
- Requirements form the basis for **design**, **implementation**, **testing**, and **acceptance**.
- Poor or missing requirements are a major cause of project failure (overruns, rework, unhappy users).

1.1 Goals, Requirements, and Design

It is useful to distinguish between:

- **Business goals**: High-level outcomes the organisation wants (e.g. "Reduce manual processing time by 50%").

- **Requirements**: Observable capabilities that support those goals (e.g. "The system shall automatically classify incoming requests within 10 seconds").
- **Design decisions**: How the system is built to satisfy requirements (e.g. "Use a microservice architecture with a message queue").

In requirements analysis, we focus mainly on the **requirements** layer: making sure we understand the problem and desired behaviour before diving into technical solutions.

---

2. The Role of Requirements in the SDLC

Requirements sit near the beginning of the software development lifecycle (SDLC), but they influence all later stages:

- **Planning**: Estimation and scheduling rely on understanding what needs to be built.
- **Architecture and design**: The system structure must support the required behaviour and qualities.
- **Implementation**: Developers need clear, unambiguous requirements to implement correctly.
- **Testing**: Test cases are derived from requirements (especially functional and non-functional).
- **Deployment and maintenance**: Requirements guide roll-out, user training, and future enhancements.

Both **plan-driven** (e.g. waterfall) and **agile** approaches depend on requirements:

- In **waterfall**, requirements are often documented in a detailed Software Requirements Specification (SRS) early in the project.
- In **agile**, requirements are expressed more flexibly as **user stories** and evolve iteratively, but the need for clarity, prioritisation, and traceability remains.

---

3. Types of Requirements

Requirements can be categorised in several ways. Two of the most important categories are:

- **Functional Requirements (FRs)**
- **Non-functional Requirements (NFRs)** (also called quality attributes)

3.1 Functional Requirements

Functional requirements describe the **behaviours**, **services**, and **functions** the system must provide.

- They define how the system should react to inputs and how it should behave in particular situations.
- They can often be expressed as "The system shall …".

Examples (Online Library System)

- FR1: The system shall allow registered users to search for books by title, author, or ISBN.
- FR2: The system shall allow librarians to add new books to the catalogue.
- FR3: The system shall send an email reminder to users 3 days before a book is due.

3.2 Non-functional Requirements (NFRs)

Non-functional requirements describe **qualities** of the system and **constraints** on how it is built or operated.

- They influence user satisfaction, robustness, performance, and maintainability.
- They can be more difficult to discover and articulate, but are just as important as functional requirements.

Common NFR categories include:

- Performance (e.g. response time, throughput)
- Reliability and availability
- Security and privacy
- Usability and accessibility
- Maintainability and supportability
- Portability
- Legal and regulatory compliance

  Later in this module, we will structure these systematically using the **FURPS+** model.

3.3 Other Requirement Categories

You may also encounter:

- **User requirements**: High-level statements of what users need (often in natural language).
- **System requirements**: Detailed descriptions of system functions and constraints; more technical.
- **Domain requirements**: Requirements arising from the specific application domain (e.g. medical, finance, aerospace).

- **Interface requirements**: Requirements about interactions with external systems, devices, or APIs.

---

4. The Requirements Engineering Process

Requirements engineering is a **process**, not a single step. A common high-level process is:

1. **Elicitation**
2. **Analysis and negotiation**
3. **Specification**
4. **Validation**
5. **Management (and change control)**

4.1 Elicitation

Elicitation is about **discovering** requirements:

- Understand the problem domain and business context.
- Identify stakeholders and their goals.
- Gather initial requirements via interviews, workshops, observation, etc. (see Section 5).

Outputs: initial list of goals, requirements, constraints, and assumptions.

4.2 Analysis and Negotiation

Analysis refines raw requirements into a consistent, complete, and feasible set:

- Identify and resolve conflicts between stakeholder needs.
- Detect overlaps, duplicates, and gaps.
- Clarify ambiguous statements.
- Prioritise requirements (e.g. with **MoSCoW**).
- Check feasibility against budget, technology, time.

Outputs: structured and prioritised requirements, trade-offs agreed with stakeholders.

4.3 Specification

Specification is about documenting requirements clearly and in a form that can be used by designers, developers, and testers:

- Natural language requirements (often numbered lists).
- Use cases, user stories, scenarios.
- Diagrams (e.g. UML: use case, activity, state, sequence diagrams).
- Models (e.g. data models, domain models).

Outputs: a **Software Requirements Specification (SRS)** or equivalent backlog of well-defined requirements.

4.4 Validation

Validation checks whether the requirements are **correct and complete**:

- Do they reflect what stakeholders really want?
- Are there missing or conflicting requirements?
- Are they testable and realistic?

Common validation techniques:

- Reviews and inspections.
- Prototyping and mock-ups.
- Walkthroughs with users.
- Test case derivation to check testability.

4.5 Management and Change Control

Requirements are not static. As the project progresses, requirements may change because:

- Stakeholders better understand their needs.
- Regulations change.
- New technology becomes available.
- Constraints (budget, time) shift.

Requirements management includes:

- Versioning and baselining requirements.
- Tracking changes and their impact.
- Maintaining **traceability** (e.g. linking requirements to design, code, and tests).
- Communicating changes to the team.

---

5. Stakeholders and Elicitation Techniques

5.1 Identifying Stakeholders

Stakeholders are people or organisations who have an interest in the system:

- End users (different user roles).
- Customers and sponsors.
- Business owners and managers.
- Operations and support teams.
- Regulators and auditors.
- Other systems or teams that integrate with your system.

Missing important stakeholder groups is a common cause of late surprises and rework.

5.2 Common Elicitation Techniques

Different techniques suit different contexts. Often, a **mix** is best.

- **Interviews**
  - One-to-one or small groups.
  - Use open questions ("How do you do this task today?") and closed questions ("Do you need this daily or weekly?").
  - Good for deep understanding and exploring tacit knowledge.
- **Workshops and focus groups**
  - Bring multiple stakeholders together.
  - Good for identifying conflicts, negotiating priorities, and building shared understanding.
  - Techniques: brainstorming, storyboarding, collaborative sketching.
- **Observation (shadowing)**
  - Watch users performing their tasks in real environments.
  - Reveals workarounds and real constraints that people may not mention in interviews.
- **Document analysis**
  - Examine existing forms, procedures, reports, regulations, legacy systems.
  - Useful for understanding current processes and constraints.
- **Questionnaires and surveys**
  - Efficient for large stakeholder groups.
  - Best for quantifiable questions (e.g. "How often do you use feature X?").
- **Prototyping**
  - Build simple mock-ups or interactive prototypes.
  - Helps stakeholders understand and refine requirements, especially UI/UX.

  In AI projects, prototypes are often used to explore model behaviour with real data, revealing new functional and non-functional requirements (e.g. explainability, fairness).

---

6. Documenting Requirements

6.1 Natural Language Requirements

Natural language is widely used because it is accessible to non-technical stakeholders. To avoid ambiguity:

- Use **consistent templates**, such as:

    - “The system shall <do something> <under condition>.”
- Avoid vague terms like “user-friendly”, “fast”, “secure” without quantification.
- Use active voice and clear subjects:
    - Prefer: “The system shall log every failed login attempt.”
    - Avoid: “Every failed login attempt will be logged.”

Example (Ambiguous vs. Improved)

- Ambiguous: “The system should be fast.”
- Improved: “The system shall display search results within 2 seconds for 95% of queries under normal load.”

6.2 Use Cases

A **use case** describes a sequence of interactions between an actor (user or external system) and the system to achieve a goal.

Typical elements:

- Use case name and ID.
- Primary actor.
- Goal.
- Preconditions and postconditions.
- Main success scenario.
- Alternative and exception flows.

Example (Simplified Use Case)

- Use Case ID: UC1
- Name: Borrow Book
- Actor: Registered User
- Goal: Borrow a book from the online library.
- Main steps:
    1. User searches for a book.
    2. System displays matching books.
    3. User selects a book and clicks “Borrow”.
    4. System checks availability.
    5. System reserves the book and confirms to the user.

From this use case, we can derive multiple functional requirements and test cases.

6.3 User Stories and Acceptance Criteria (Agile)

In agile contexts, requirements are often expressed as **user stories**:

- Template: “As a <type of user>, I want <some goal> so that <some benefit>.”

User stories should be accompanied by **acceptance criteria** that specify when the story is considered done.

Example

- User Story:
  As a registered user, I want to **receive email reminders before my books are due** so that I **avoid late fees**.
- Acceptance Criteria:
  - The system sends a reminder email 3 days before the due date.
  - If a book is renewed, the reminder schedule is updated.
  - Users can opt out of reminder emails in their profile settings.

Good user stories are often described by the **INVEST** properties: - Independent, Negotiable, Valuable, Estimable, Small, Testable.

---

7. Qualities of Good Requirements

High-quality requirements share several properties:

- **Correct**: Reflect real stakeholder needs.
- **Complete**: Cover all significant situations and constraints (within scope).
- **Consistent**: No conflicts between requirements.
- **Unambiguous**: Interpreted the same way by different readers.
- **Verifiable**: There is a clear way to test whether the requirement is satisfied.
- **Feasible**: Achievable within technical, budget, and time constraints.
- **Prioritised**: Relative importance is clear (e.g. using MoSCoW).

Example (Improving Verifiability)

- Vague: "The interface shall be intuitive."
- Verifiable: "New users shall be able to complete the registration process without external help within 3 minutes in 90% of usability test sessions."

---

8. The FURPS+ Model

The **FURPS+** model is a way to structure non-functional requirements and certain constraints. It was introduced by Hewlett-Packard and is widely used in software engineering.

FURPS stands for:

- **F**unctionality
- **U**sability
- **R**eliability

- **P**erformance
- **S**upportability

The + refers to additional categories such as **implementation**, **interface**, **operations**, **packaging**, **legal**, etc.

8.1 FURPS+ Overview

Category	Description	Example (Online Library)
Functionality	Features, capabilities, security, business rules	Role-based access control, recommendation engine
Usability	Human factors, UI, help, documentation	Consistent navigation, accessibility for screen readers
Reliability	Availability, fault tolerance, recoverability	99.5% uptime, automatic recovery from server failure
Performance	Response time, throughput, resource usage	Search results within 2 seconds, supports 500 users online
Supportability	Testability, maintainability, configurability, extensibility	Modular architecture, configuration via admin UI
+ (Plus)	Implementation, interfaces, operations, packaging, legal	Must run on Linux, GDPR compliance, REST API for partners

8.2 F – Functionality

Functionality includes:

- Feature sets and capabilities.
- Security requirements (authentication, authorisation, auditing).
- Business rules and domain-specific logic.

Examples

- The system shall enforce role-based access control (user, librarian, admin).
- The system shall log all book borrow and return operations for 2 years.
- The recommendation engine shall suggest at least 5 books based on the user's borrowing history.

8.3 U – Usability

Usability captures how easy and pleasant the system is to use:

- User interface design and consistency.
- Learnability and discoverability.
- Accessibility (e.g. WCAG compliance).
- Help, tutorials, and documentation.

Examples

- First-time users shall be able to search and borrow a book without training.
- The system shall support keyboard navigation and screen readers.
- An online help guide shall be available from every page.

8.4 R – Reliability

Reliability is about the system's ability to operate correctly over time:

- Availability (uptime).
- Mean time between failures (MTBF).
- Error rates and fault tolerance.
- Backup and recovery strategies.

Examples

- The system shall have at least 99.5% availability during opening hours.
- No more than 1 in 10,000 transactions shall be lost due to system errors.
- In case of server failure, the system shall recover within 5 minutes without data loss.

8.5 P – Performance

Performance covers response times, throughput, and resource usage:

- How quickly the system responds to actions.
- How many concurrent users or transactions it can handle.
- Resource consumption (CPU, memory, network).

Examples

- Search queries shall return results within 2 seconds for 95% of requests under normal load.
- The system shall support at least 500 concurrent users browsing the catalogue.
- Generating a monthly usage report shall take no more than 30 seconds.

8.6 S – Supportability

Supportability relates to how easily the system can be maintained, tested, and extended:

- Modularity and code quality.
- Configurability (without code changes).
- Diagnostics and logging.
- Internationalisation and localisation.

Examples

- Configuration of email server settings shall be possible via an admin UI, without code changes.

- The system shall provide logs with timestamps and correlation IDs for all API requests.
- Adding a new language shall require only translation of external resource files.

8.7 The "+" in FURPS+

The plus part covers additional constraints:

- **Implementation**: Technology stack, programming languages, frameworks.
  - Example: The backend shall be implemented in Python and run on Linux servers.
- **Interfaces**: Interfaces to other systems, hardware, or devices.
  - Example: The system shall expose a RESTful API for integration with external catalogues.
- **Operations**: Operational and deployment constraints.
  - Example: The system shall support rolling updates with zero downtime.
- **Packaging**: How the system is delivered.
  - Example: The system shall be deployable as Docker containers.
- **Legal / Licensing / Regulatory**:
  - Example: The system shall comply with GDPR for all EU users.

  FURPS+ helps ensure that important non-functional and constraint areas are not forgotten. During analysis, you can use FURPS+ as a checklist.

8.8 FURPS+ Example: Online Course Platform

Consider an **online course platform** (similar to an LMS):

- **Functionality**
  - The system shall allow instructors to create and publish courses.
  - The system shall allow students to enrol, track progress, and receive completion certificates.
  - The system shall integrate with payment providers for paid courses.
- **Usability**
  - The platform shall be accessible on desktops, tablets, and mobiles.
  - The UI shall support dark mode and standard keyboard shortcuts.
  - A tutorial course shall be available to new users.
- **Reliability**
  - The platform shall achieve 99.8% uptime.
  - No single user data loss is acceptable due to system failures.
  - Daily backups shall be taken and retained for 30 days.
- **Performance**
  - Course pages shall load within 3 seconds for 95% of page views.
  - The system shall handle 2,000 concurrent learners watching videos.

  - Video playback shall adapt automatically to network bandwidth.
- **Supportability**
  - Instructors shall be able to duplicate a course to create a new version.
  - System logs shall be searchable by course ID and user ID.
  - Admins shall be able to toggle features (e.g. comments) in configuration.
- **+ (Implementation / Interfaces / Legal)**
  - The platform shall support single sign-on (SSO) with the university's identity provider.
  - The platform shall comply with GDPR and store data in EU-based data centres.
  - The system shall expose APIs for exporting course completion data to external analytics tools.

---

9. MoSCoW Prioritisation

MoSCoW is a widely used technique for prioritising requirements, especially in agile and time-boxed projects.

MoSCoW stands for:

- **M**ust have
- **S**hould have
- **C**ould have
- **W**on't have (this time)

9.1 Categories

- **Must have**
  - Essential for a viable solution.
  - Without these, the system is not acceptable.
  - Often related to core business needs, safety, or legal requirements.
- **Should have**
  - Important but not vital.
  - Workarounds may exist, but they are often inconvenient.
- **Could have**
  - Desirable but not necessary.
  - Provide nice-to-have improvements or enhancements.
  - Usually implemented if time and budget allow.
- **Won't have (this time)**
  - Explicitly out of scope for the current release.
  - May be revisited in future versions.
  - Helps manage expectations and avoid scope creep.

9.2 Applying MoSCoW: Steps

1. **List all candidate requirements** (functional, non-functional, constraints).
2. **Discuss with stakeholders**:
   - What is absolutely essential for success?
   - What can be deferred or replaced with workarounds?
3. **Assign each requirement** to one of the four categories.
4. **Review for balance**:
   - If almost everything is marked "Must", challenge and renegotiate.
5. **Revisit regularly**:
   - Priorities may evolve as more is learned.

9.3 Example: Online Library System

Suppose we have the following requirements:

1. Users can search the catalogue.
2. Users can borrow and return books.
3. Librarians can add and remove books.
4. Users receive email reminders before due dates.
5. Users can rate and review books.
6. The system supports dark mode.
7. The system integrates with an external university database.
8. The system supports multilingual user interfaces.

A possible MoSCoW classification for the **first release**:

- **Must have**
  - R1: Users can search the catalogue.
  - R2: Users can borrow and return books.
  - R3: Librarians can add and remove books.
  - R7: The system integrates with the external university database.
- **Should have**
  - R4: Users receive email reminders before due dates.
  - R8: The system supports multilingual user interfaces.
- **Could have**
  - R5: Users can rate and review books.
  - R6: The system supports dark mode.
- **Won't have (this time)**
  - Any additional social features (e.g. following other users) that are nice but not currently planned.

  Combining MoSCoW with FURPS+ can help ensure that both functional and non-functional requirements are prioritised appropriately.

---

10. Case Study: AI-Based Recommendation System

To connect this to AI-related systems, consider an **AI-based recommendation engine** for an online learning platform.

10.1 High-Level Description

- The system recommends courses and learning materials to students based on their past activity, preferences, and performance.
- It is integrated into the main platform and runs on real user data.

10.2 Sample Requirements

Functional

- FR1: The system shall generate a list of at least 5 recommended courses for each active user each day.
- FR2: The system shall update recommendations after users complete a course.
- FR3: The system shall provide an explanation (e.g. "Because you completed Course X") for each recommendation.

Non-functional (Using FURPS+)

- Functionality:
  - Must support course recommendations for undergraduate, postgraduate, and professional courses.
  - Must filter out courses that are not available in the user's region.
- Usability:
  - Recommendations shall be displayed in a dedicated "Recommended for You" section on the dashboard.
  - Users shall be able to hide or dismiss recommendations they find irrelevant.
- Reliability:
  - The recommendations service shall have an availability of at least 99%.
  - If the AI model fails, the system shall fall back to a simple popularity-based recommendation.
- Performance:
  - Generating recommendations for a single user shall take no more than 500 ms on average under normal load.
  - The system shall handle up to 10,000 recommendation requests per minute.
- Supportability:
  - It shall be possible to deploy updated models without downtime.
  - The system shall log model versions used for each recommendation for later analysis.
- – (Legal / Ethics / Interfaces):

  - The system shall comply with privacy regulations and allow users to opt out of personalised recommendations.
  - The system shall expose an internal API for retrieving recommendations from other platform components.

10.3 MoSCoW Prioritisation Example

For the first release:

- **Must have**
  - Daily recommendations (FR1).
  - Basic explanations for recommendations (FR3).
  - Privacy compliance and opt-out options.
  - Performance requirement of max 500 ms per recommendation under normal load.
- **Should have**
  - Real-time updates after course completion (FR2).
  - Logging of model versions.
- **Could have**
  - Advanced explanations (e.g. feature contributions, visualisations).
  - User ability to give feedback (like/dislike) on recommendations.
- **Won't have (this time)**
  - Cross-platform recommendations across multiple partner universities.
  - Complex fairness constraints for specific research studies (planned for a future release).

---

11. Common Pitfalls and Best Practices

11.1 Common Pitfalls

- **Vague requirements**:
  - "Fast", "user-friendly", "secure" without measurable criteria.
- **Hidden assumptions**:
  - Assuming users have high-speed internet, or that all data is clean.
- **Missing stakeholders**:
  - Ignoring support teams, operations, or regulators.
- **Over-engineering**:
  - Adding many "Could have" features as "Must have" without justification.
- **Poor traceability**:
  - Hard to know which code and tests implement a given requirement.

11.2 Best Practices

- Use **structured templates** for requirements (e.g. "The system shall …").

- Combine **multiple elicitation techniques** to get a more complete picture.
- Regularly perform **reviews and walkthroughs** with stakeholders.
- Use **FURPS+** as a checklist for non-functional requirements and constraints.
- Use **MoSCoW** (or a similar scheme) to **prioritise** requirements explicitly.
- Maintain **traceability** between requirements, design, implementation, and tests.
- Keep requirements **up to date** as changes are agreed.

---

12. Activities

Try the following activities to apply what you have learned.

Activity 1: Identify Requirements for a Simple System

Choose a simple system, for example:

- A mobile task management app.
- A campus room booking system.
- A simple chatbot for answering FAQs.

Tasks:

1. Identify at least 8–10 functional requirements.
2. Identify at least 6–8 non-functional requirements.
3. Classify the non-functional requirements using **FURPS+**.
4. Prioritise all requirements using **MoSCoW**.

Activity 2: Rewrite Ambiguous Requirements

Given the following vague requirements, rewrite them to be clearer and testable:

1. "The system should be easy to use."
2. "The system must be secure."
3. "Reports should be generated quickly."

For each, specify:

- Measurable criteria.
- Relevant FURPS+ category.

Activity 3: Simple Use Case and User Stories

For the same system you chose in Activity 1:

1. Write one detailed **use case** (name, actor, goal, main flow).
2. Derive at least three **user stories** from this use case, each with acceptance criteria.

Additional Resources

- Ian Sommerville, *Software Engineering* – Chapters on Requirements Engineering.
- Karl Wiegers & Joy Beatty, *Software Requirements*.
- IEEE 830 / ISO/IEC/IEEE 29148 – Standards for Software Requirements Specifications.
- Articles and guides on FURPS+ and MoSCoW prioritisation from reputable software engineering sources.

Recap

@recap