Software Engineering 2025–26

Lecture 2: Requirements and Specifications

Or: How do we decide what to build?

Prof. Robert Harle

Department of Computer Science and Technology
University of Cambridge

Easter Term

The Danger of Perfection

"There is nothing quite so useless as doing with great efficiency something that should not be done at all."
– Peter Drucker

- Engineers love solving technical puzzles.
- Building a perfectly secure, infinitely scalable system is worthless if it doesn't solve the user's actual problem.
- We must separate the *Problem Space* from the *Solution Space*.

The Business Goals (The "Why")

The Problem Space

- Why are we funding this project?
- Example: "Our current warehouse packing process is too slow, causing delayed shipments and unhappy customers."
- The "Why" is purely focused on business value, user needs, and market realities. It is technology-agnostic.

The Technical Implementation (The "What")

The Solution Space

- What are we going to build to solve the "Why"?
- Example: "We will build a tablet-based barcode scanning app that calculates optimal walking routes for warehouse staff."
- The "What" introduces technology, architecture, and engineering constraints.

The Translation Gap

Bridging the Divide

- Stakeholders (business owners, clients) know the "Why", but often speak in vague, contradictory terms.
- Engineers know the "How", but often lack domain expertise in the client's business.
- **Requirements Engineering** is the formal process of bridging this gap.

Requirements Elicitation: Interviews

How do we find out what they want?

- **Interviews:** Sitting down with stakeholders.
- *Pros:* Good for initial scoping and understanding high-level business goals.
- *Cons:* Users often don't know what they actually want, or they describe a solution instead of their problem.
- "If I had asked people what they wanted, they would have said faster horses." – (Apocryphally attributed to Henry Ford)

Requirements Elicitation: Ethnography

Watching Users Work

- **Ethnography / Observation:** Embedding yourself in the user's environment.
- Why is this powerful? Users develop invisible workarounds.
- Example: You notice a nurse has a sticky note with passwords on her monitor because the current login system logs her out too fast.
- She wouldn't mention this in an interview because she considers it "normal."

The Problem with Natural Language

Ambiguity is the enemy of engineering.

- Human language is naturally ambiguous, context-dependent, and imprecise.
- Code is binary, deterministic, and merciless.
- If a requirement is ambiguous, the programmer will unconsciously make an assumption to fill the gap. That assumption is usually wrong.

Example: The Vague Requirement

Bad Requirement

"The search system should be fast and easy to use."

Why is this bad?

- What does "fast" mean? 100 milliseconds? 2 seconds?
- What does "easy to use" mean? How do you test that?
- If the client rejects the software saying "it's too slow," you have no contractual defense.

Example: The Testable Requirement

Good Requirement

"The system shall return search results for a database of 1 million records in under 500 milliseconds for 95% of queries under normal load."

Why is this good?

- It is quantifiable.
- It is testable (you can write an automated script to verify this).
- It sets a clear boundary for "done".

The Cost of Late Fixes

Boehm's Curve

- Fixing a misunderstood requirement during the *design phase* costs 1x.
- Fixing it during *coding* costs 5x.
- Fixing it during *testing* costs 10x.
- Fixing it *after deployment* (in production) costs 100x+.

Measure twice, cut once.

Defining Functional Requirements (FRs)

Functional Requirement

A statement of a specific action or behaviour the system must perform. It describes **what** the system should do.

- Think of these as the verbs of your system.
- They define the direct inputs, processing, and outputs.
- If the system doesn't do this, it is fundamentally broken.

Examples: Functional Requirements

Context: An E-Commerce Website

- "The system shall allow a user to add an item to a digital shopping cart."
- "The system shall calculate the total cost of items in the cart, including VAT."
- "The system shall send a confirmation email to the user upon successful payment."

Defining Non-Functional Requirements (NFRs)

Non-Functional Requirement

A constraint on the operation of the system. It describes **how well** the system should perform its functions.

- Think of these as the adjectives or adverbs of your system.
- They define quality attributes, performance bounds, and architectural constraints.
- Crucially, NFRs often dictate the architecture of the entire system.

The "-ilities" (NFR Categories)

NFRs are often colloquially referred to as the "-ilities":

- **Scalability:** Can it handle 10x the users?
- **Reliability:** How often does it crash?
- **Maintainability:** How hard is it to update the code?
- **Portability:** Can we move it from AWS to Google Cloud?
- **Usability:** Can a novice learn it in 5 minutes?

Performance Metrics as NFRs

Latency vs. Throughput

- **Latency:** How long does one operation take? (e.g., "Page load time must be < 2s").
- **Throughput:** How many operations can happen at once? (e.g., "System must process 5,000 transactions per second").
- You often have to design differently to optimize for one over the other.

Security as an NFR

Constraints on Data and Access

- Security is almost always an NFR, but it generates functional requirements.
- *NFR:* "The system must comply with GDPR data protection laws."
- *Resulting FR:* "The system shall provide a button for the user to delete their account and all associated data."

Usability as an NFR

Constraints on Human Interaction

- Usability metrics must be specific to be useful.
- "The system must be intuitive" is meaningless.
- "A new warehouse employee must be able to successfully scan and dispatch a package with less than 15 minutes of training." (Testable).

The Reality of Trade-offs

You cannot have it all.

- NFRs constantly fight each other. Engineering is the art of balancing them.
- **Security vs. Usability:** A 24-character password requirement with 2FA is highly secure, but terrible for user usability.
- **Performance vs. Maintainability:** Writing core logic in hand-optimized Assembly language improves performance, but destroys maintainability.

Documenting NFRs Testably

Fit Criteria Every NFR must have a "fit criterion"—a measurement to prove it was met.

- **Scale:** "Supports 10,000 concurrent websocket connections."
- **Availability:** "99.99% uptime per month (allows for $\approx$ 4.3 minutes of downtime)."
- **Accessibility:** "Complies with WCAG 2.1 AA standards for screen readers."

Class Exercise

Functional

or

Non-Functional?

Class Exercise: Q1

Context: University Library System

"The system shall allow a student to reserve a book that is currently checked out."

Class Exercise: Q1

Context: University Library System

"The system shall allow a student to reserve a book that is currently checked out."

Answer: Functional Requirement. (It describes a specific action the system performs).

Class Exercise: Q2

Context: University Library System

"The search functionality must return results in under 2 seconds, even with 500 concurrent users."

Class Exercise: Q2

Context: University Library System

"The search functionality must return results in under 2 seconds, even with 500 concurrent users."

Answer: Non-Functional Requirement. (Performance/Scalability constraint).

Class Exercise: Q3

Context: University Library System

"Database passwords must be stored using bcrypt hashing with a salt factor of 12."

Class Exercise: Q3

Context: University Library System

"Database passwords must be stored using bcrypt hashing with a salt factor of 12."

Answer: Non-Functional Requirement. (Implementation/Security constraint).

Why do we model?

The Power of Abstraction

- You cannot show a stakeholder 10,000 lines of Java code to explain how the system works.
- We use models to abstract away complexity and communicate architecture clearly.
- A good model hides exactly enough detail to make the core concept understandable.

Introduction to UML

Unified Modeling Language (UML)

- Developed in the 1990s as a standardized way to visualize system design.
- Contains 14 different diagram types (e.g., Use Case, Class, Sequence, State Machine).
- Historically, companies tried to generate entire codebases directly from highly detailed UML diagrams. (This largely failed).

Modern UML: Sketches vs. Blueprints

How we use UML today

- **As Blueprints:** Rarely used today. This means defining every single method, variable, and type before coding. Too rigid.
- **As Sketches:** The modern approach. Engineers use informal UML on whiteboards to explain architectural concepts, ignoring strict syntax rules in favor of speed and clarity.

Class Diagrams: Static Structure

Class Diagram

Shows the static structure of a system by outlining its classes, their attributes, operations (methods), and the relationships among objects.

- The backbone of Object-Oriented Design.
- Shows *what* exists in the system, not *when* things happen.

Class Diagrams: The Box

Anatomy of a Class Node

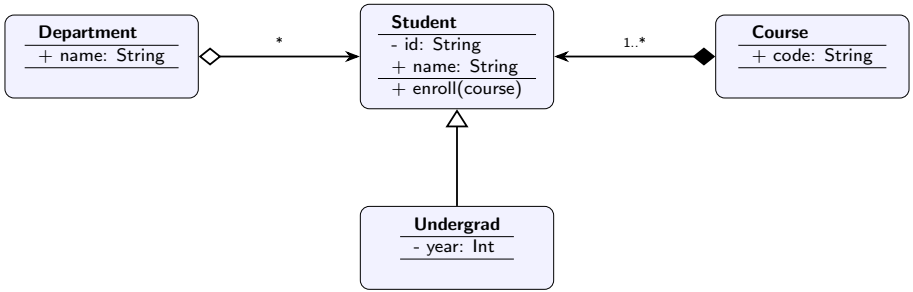
- **Top Compartment:** Class Name (e.g., `Customer`).
- **Middle Compartment:** Attributes / Variables (e.g., `+ email: String`).
- **Bottom Compartment:** Methods / Operations (e.g., `+ verifyPassword(pwd): Boolean`).
- *Visibility modifiers:* + (Public), – (Private), # (Protected).

Class Diagrams: Relationships

How classes interact

- **Association:** A general connection (Line).
- **Inheritance:** "Is-a" relationship (Arrow with hollow triangle). E.g., `Undergrad` inherits from `Student`.
- **Aggregation:** "Has-a" relationship (Hollow diamond). E.g., `Department` has `Professors`.
- **Composition:** Strict "Has-a" relationship where the child dies if the parent dies (Solid diamond). E.g., `Building` has `Rooms`.

Class Diagram: Visual Structure



Example: University Management System

Sequence Diagrams: Dynamic Behavior

Sequence Diagram

An interaction diagram that shows how processes operate with one another and in what order.

- Focuses on the message flow over time.
- Excellent for modeling network calls, API interactions, and authentication flows.

Sequence Diagrams: Components

Anatomy of a Sequence Diagram

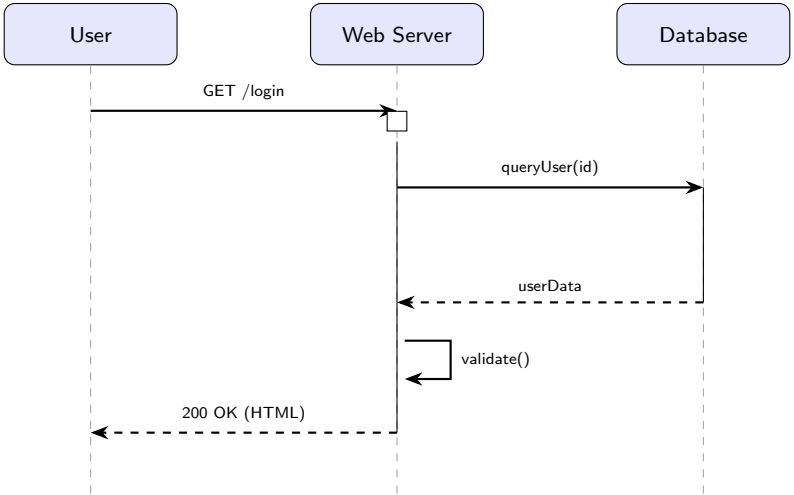
- **Lifelines:** Vertical dashed lines representing the lifespan of an object or actor.
- **Messages:** Horizontal arrows passing between lifelines (e.g., `HTTP GET /login`).
- Time flows from top to bottom.

Sequence Diagrams: Activation

Showing Processing Time

- **Activation Boxes:** Thin rectangles on lifelines indicating that an object is actively processing a task.
- **Return Messages:** Dashed arrows indicating a response (e.g., `200 OK` or a JSON payload).
- Perfect for mapping out asynchronous behavior or database queries.

Sequence Diagram: Visual Flow



Example: User Login Authentication Flow

When to use which diagram?

Use the right tool for the job.

- Need to show the database schema and entity relationships? → **Class Diagram**.
- Need to show how OAuth2 login works between the user, your server, and Google? → **Sequence Diagram**.
- Need to show what states a shopping cart can be in (Empty, Active, CheckedOut)? → **State Machine Diagram**.

Anti-pattern: Over-engineering Models

Beware of Analysis Paralysis

- Spending 3 weeks making the perfect UML diagram in Visio is usually a waste of time.
- By the time you finish the diagram, the requirements will have changed.
- **Rule of thumb:** If the diagram fits on a whiteboard and the team understands it, it is good enough. Start coding.

The Product Manager (PM)

Who manages the requirements?

- In modern software companies, requirements are driven by **Product Managers** (PMs).
- They are the intersection of Business, UX, and Technology.
- They own the "Why" and the "What". Engineers own the "How".

PM: Voice of the Customer

The PM's Responsibility

- Conduct user research and interviews.
- Analyze market trends and competitor software.
- Shield the engineering team from changing business whims by filtering requests.
- Prioritize features to maximize business value.

PM vs. Engineering Manager (EM)

A crucial distinction in industry

- **Product Manager (PM):** Decides that we need to build a new payment gateway to capture European customers. (Focuses on product value).
- **Engineering Manager (EM):** Decides which engineers will build it, oversees the code architecture, and manages the career growth of the developers. (Focuses on technical execution and team health).

Prioritization

We can't build everything.

- Stakeholders will always want 100 features. The budget only allows for 20.
- PMs and Engineers must work together to prioritize requirements.
- We need structured frameworks to avoid prioritizing based on "who shouts the loudest."

The MoSCoW Method

A common prioritization framework:

- **Must have:** The product is completely unviable without this. (e.g., Login system).
- **Should have:** Highly important, but we can launch a V1 without it if absolutely necessary. (e.g., Password reset email).
- **Could have:** Nice to have if we finish early. (e.g., Dark mode).
- **Won't have:** Out of scope for this release. Let's not talk about it right now.

The Product Requirements Document (PRD)

The Source of Truth

- The PRD is a living document (often in Confluence or Google Docs) authored by the PM.
- It contains the background, business goals, user personas, and the list of Functional and Non-Functional Requirements.
- Engineers read the PRD to understand what to build.

Requirement Traceability

How do we know we built what we promised?

- **Traceability:** The ability to trace a requirement from its origin (PRD) to its implementation (Code) to its verification (Test).
- Example: Requirement #42 $\rightarrow$ Pull Request #105 $\rightarrow$ Integration Test `test_req_42()`.
- Highly regulated industries (aviation, medical) require strict Traceability Matrices.

Summary of Lecture 2

- Understand the "Why" before designing the "What".
- Ambiguous requirements are the root of most software project failures.
- **Functional Requirements** are actions (verbs); **Non-Functional Requirements** are constraints and qualities (adverbs).
- Use UML (Class and Sequence diagrams) as whiteboard sketches to communicate architecture.
- PMs manage requirements; Engineers execute them.