COMP433 Software Engineering

Requirements Engineering

Part 1: foundations, requirement types, and the language of requirements

Hisham Ihshaish

Department of Computer Science, Birzeit University

Sections 5 and 6 | Semester 2, 2025/26

Lecture objectives

By the end of this session, you should be able to:

1. Distinguish user, system, and software-specification requirements, and explain who reads each.
2. Tell functional, non-functional, and domain requirements apart, and give examples of each.
3. Refine a high-level user requirement into a set of detailed, testable system requirements.
4. Recognise when a non-functional requirement is a goal versus a verifiable statement, and write the verifiable form.
5. Identify the practical problems that domain requirements introduce into a project.

What is requirements engineering?

Requirements engineering is the process of establishing

- the services (or functionalities) that the customer requires from a system, and
- the constraints under which the system operates and is developed.

And what is a requirement?

A requirement is a description of a system service or constraint generated during the requirements engineering process.

In other words: requirements are the answers to two questions, written down so everyone agrees.

What should the system do?
Under what constraints should it do it?

What is a requirement? The dual nature

Requirements may range from a high-level abstract statement of a service, to a detailed mathematical functional specification.

This is inevitable, because requirements serve two different purposes:

Basis for a bid

Used by candidate suppliers to estimate scope and cost.

- Must be open to interpretation, so different bidders can propose different solutions.
- Tends to be high-level, abstract, in natural language.

Basis for the contract

Once a supplier is chosen, the contract pins down exactly what will be built.

- Must be precise, defined in detail, with no room for re-interpretation.
- Tends to be specific, structured, sometimes formal.

Both of these are called requirements. The level of detail depends on what they are for.

Three levels of abstraction

The same system is described at three increasingly detailed levels, each written for a different audience.

User requirements

Written for customers

Statements in natural language, plus simple diagrams, of the services the system provides and the constraints under which it operates.

System requirements

Written for customers and as a contract between client and contractor

A structured document setting out detailed descriptions of the system services and constraints. Each user requirement is decomposed into one or more system requirements.

Software specification

Written for developers

A detailed software description that can serve as a basis for the design and the implementation. Adds the technical detail needed to build the system.

Running example: the Mentcare system

A patient information system for mental health care

What it does

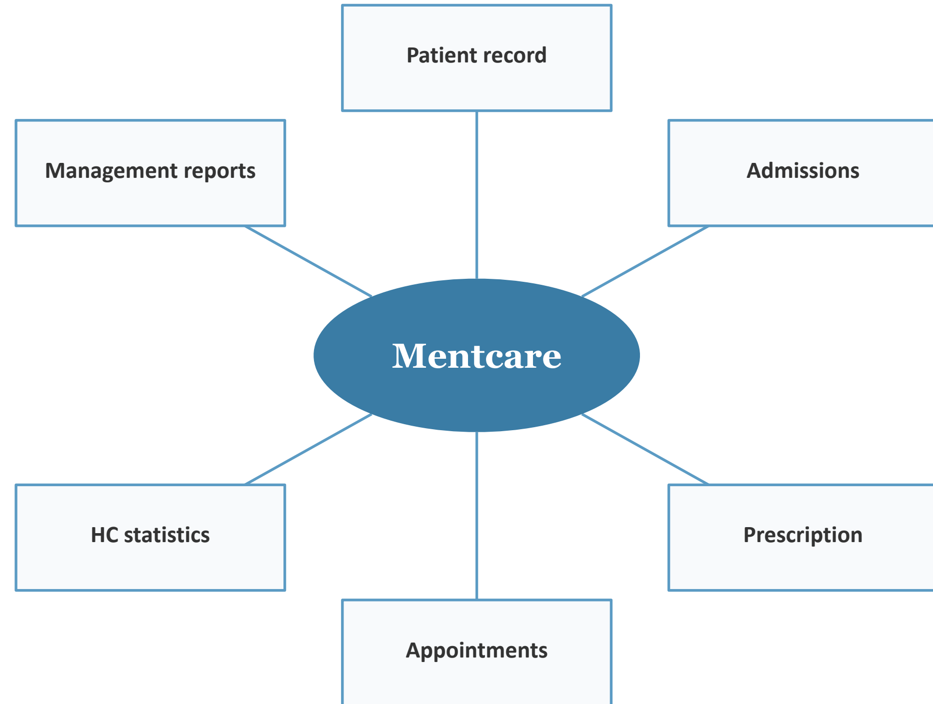
Records patient details, consultations, and treatments for mental-health clinics. Generates letters, reports, and management statistics.

Why this case study

Sommerville's main case study in the 10th edition. We will use it for most worked examples in this and the next two lectures so you can build up a coherent picture of one system.

Note

Older editions call this system MHC-PMS. It is the same system; the 10th edition renamed it.



System context: Mentcare and the systems it interacts with

User requirements

User requirements describe the actions or abilities the system must provide to its users to support their business needs and services.

They cover both functional and non-functional requirements, expressed at a level a non-technical user can read.

Properties of a good user requirement

- Written in natural language, supplemented with simple tables or diagrams when helpful.
- Understandable by stakeholders who do not have detailed technical knowledge.
- Stated in terms of the user's goal, not the system's internals.
- Free of premature design choices: it says **what**, not **how**.

User requirements: Mentcare examples

A non-exhaustive list. Each line is one user requirement, written for the customer to read.

UR1	The Mentcare system **shall** generate monthly management reports showing the cost of drugs prescribed by each clinic and doctor during that month.
UR2	The Mentcare system **should** generate monthly reports showing the details of admitted and discharged patients in each clinic.
UR3	The Mentcare system **should** generate monthly reports showing the details of deceased patients in each clinic.
UR4	The system administrator **shall** be able to register and create user accounts for new staff members of the Mentcare system.
UR5	Staff members **shall** be able to log in to the Mentcare system using a registered username and password.
UR6	Authorised staff (Receptionist, Nurse) shall be able to register a patient and create a patient medical record.
UR7	Authorised staff (Receptionist, Nurse) shall be able to activate and deactivate a patient medical record.
UR8	Authorised staff (Receptionist, Manager, Doctor, Nurse) shall be able to view a patient's personal information.
UR9	Authorised staff (Doctor, Nurse) shall be able to view and edit a patient's medical record.
UR10	The Mentcare system shall generate clinical reports showing the clinical information of patients with active treatment cases.

System requirements

System requirements give detailed descriptions of the specific needs of each user requirement, so they can be used as a basis for designing the system and as part of the system contract.

Each user requirement typically expands into several system requirements.

Properties of a good system requirement

- Detailed enough to be unambiguous and testable.
- Traceable back to a user requirement, so we know why each one exists.
- Written using natural language, supported by structured tables or simple system models when needed.
- Free of design decisions that the developer should be free to make.

Think this way: *name fields*, *formats*, *error conditions*, *units of measurement*, *ranges*, *dates*, *role-based privileges*

Anything that is left ambiguous becomes a developer assumption, and developer assumptions become bugs.

From user requirement to system requirements (1)

One UR, five SRs. Same intent, different precision.

UR1	The Mentcare system shall generate monthly reports showing the cost of drugs prescribed by each clinic and doctor during that month.

expanded into

SR1.1	On the last working day of each month, a summary of the drugs prescribed, their cost, and the prescribing clinics **shall** be generated.
SR1.2	The system shall automatically generate the report for **printing after 17:30** on the last working day of the month.
SR1.3	A report shall be created for each clinic, listing individual drug names, total number of prescriptions, number of doses prescribed, and total cost of the prescribed drugs.
SR1.4	If drugs are available in different dose units (e.g. 10 mg, 20 mg), separate reports shall be created for each dose unit.
SR1.5	Access to all cost reports shall be restricted to authorised users listed on a management access control list.

From user requirement to system requirements (2)

Two URs, several SRs. Notice how account creation and login decompose.

UR4	The system administrator shall be able to register and create user accounts for new staff members of the Mentcare system.
SR4.1	The system shall enable the system administrator to create an account containing the user's full name and address.
SR4.2	The system shall require a valid email address for every account; account creation must be rejected without one.
SR4.3	Usernames shall be a minimum of 8 alphanumeric characters and uniquely identify the user.
SR4.4	Passwords shall be at least 12 characters and include at least three of: lowercase, uppercase, digit, special character.
SR4.5	Passwords shall be stored using a salted, slow hash (bcrypt or argon2id); plaintext storage is prohibited.
SR4.6	Password resets shall be sent only to the user's verified email address.

UR5	Staff members shall be able to log in to the Mentcare system using a registered username and password.
SR5.1	The system shall grant access only after the username and password match a stored, hashed credential.
SR5.2	After five consecutive failed login attempts within ten minutes, the account shall be temporarily locked.

Who reads which document?

Each level of requirement has a different audience. Write at the level your reader can verify.

User requirements	**Audience:** Client managers, system end-users, client engineers, contractor managers, system architects **Why:** *The customer needs to read these to confirm scope. The contractor's senior staff use them to plan.*
System requirements	**Audience:** System end-users, client engineers, system architects, software developers **Why:** *The contract-level document. Both sides reference these to settle disputes about what should have been built.*
Software design specification	**Audience:** Client engineers (sometimes), system architects, software developers **Why:** *The implementation-level document. Mostly read by the development team; clients rarely look at it.*

Part 2

Functional, non-functional, and domain requirements

A second way to classify requirements: by what they are about, rather than who they are written for.

Three categories of requirement

Every requirement falls into one of these three categories. Knowing which helps us write it well and verify it.

Functional

What the system does

Statements of services the system should provide, how it should react to particular inputs, and how it should behave in particular situations.

Example

The system shall display a patient's appointments for the next 14 days.

Non-functional

How well the system does it

Constraints on the services and functions offered: timing, reliability, response time, storage, standards, the development process. Often apply to the whole system, not individual features.

Example

Patient lookups shall return within 2 seconds at the 95th percentile.

Domain

What the application area demands

Requirements that come from the application domain and reflect its characteristics. May add new functional or non-functional requirements, or new constraints on existing ones.

Example

All clinical access shall be logged in line with national mental-health-records regulations.

Functional requirements

Statements of services the system should provide, how the system should react to particular inputs, and how the system should behave in particular situations.

Notes on functional requirements

- They depend on the type of software, the expected users, and the system context.
- Functional user requirements are typically high-level statements of what the system should do.
- Functional system requirements should describe each service in detail: inputs, outputs, exception cases.
- If functional requirements are imprecise, the same statement can be implemented in two incompatible ways.

Functional requirements: Mentcare examples

Three functional requirements at system-requirement level. Notice how each pins down inputs, behaviour, or constraints.

FR-a	A user shall be able to search the appointments lists for all clinics that they have access to. *The phrase "that they have access to" closes the imprecision Sommerville calls out.*
FR-b	The Mentcare system shall generate, each day, for each clinic, a list of patients who are expected to attend appointments that day. *Pins down frequency (each day), grouping (each clinic), and content (expected attendees).*
FR-c	Each staff member using the system shall be uniquely identified by an 8-digit employee number. *Pins down a constraint that other requirements (audit logging, role-based access) will depend on.*

Non-functional requirements

Constraints on the services or functions offered by the system. They define system properties such as reliability, response time, storage, and process or platform standards.

Often apply to the system as a whole, not to individual features.

Why NFRs can matter more than FRs

- If a functional requirement is missing, a feature is missing.
- If a non-functional requirement is missed, the whole system can become unusable.
- *An aircraft control system that gives the right output 30 seconds late is not a slow aircraft control system; it is a crashed aircraft.*

NFR classification: Product, Organisational, External

Three sources of non-functional requirements. The source determines who you negotiate with to change them.

Product

Requirements specifying that the delivered product must behave in a particular way.

Includes

Efficiency, dependability, security, usability.

Organisational

Requirements that follow from the customer's organisational policies and procedures.

Includes

Environmental, operational, development.

External

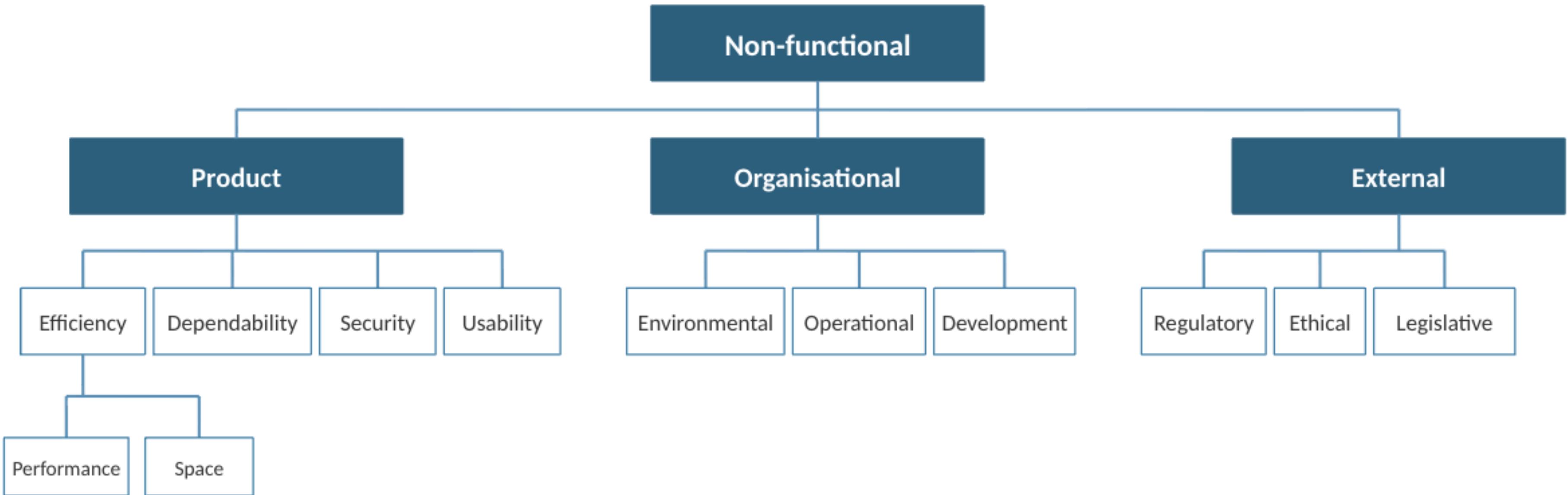
Requirements that arise from factors external to the system and its development process.

Includes

Regulatory, ethical, legislative.

NFR types (Sommerville, 10th edition)

The full tree. Each leaf is a category of non-functional requirement we may need to write.



Note: *older editions (and many older slide decks) used a different leaf set: Reliability and Portability under Product; Delivery, Implementation, and Standards under Organisational; Interoperability, Ethical, and Legislative under External. The 10th edition reorganised these into the structure above.*

NFR examples in Mentcare

Product *P-3.C.8*	The Mentcare system shall be available to all clinics during normal working hours (Mon-Fri, 08:30-17:30). Downtime within normal working hours shall not exceed five seconds in any one day. *Constrains availability and downtime - a dependability requirement.*
Organisational *O-5.4.3*	Users of the Mentcare system shall authenticate themselves using their health authority identity card. *Imposed by the customer's IT policy - an operational requirement.*
External *E-7.2.3*	The system shall implement patient privacy provisions as set out in regulation HStan-03-2006-priv. *Imposed by external regulation - a regulatory or legislative requirement.*

Goals vs verifiable non-functional requirements

Vague intentions are useful for direction, but useless as contracts. The job of the requirements engineer is to convert one into the other.

Goal	Verifiable NFR
Usability	
The Mentcare system should be easy to use by medical staff and should be organised in such a way that user errors are minimised.	Medical staff shall be able to use all Mentcare functions after four hours of training. After this training, the average number of errors made by experienced users shall not exceed two per hour of system use.
Security	
The Mentcare system should be secure. Patient information must not leak.	All client-server traffic shall use TLS 1.3 or above. Failed login attempts shall be rate-limited to five per ten minutes per account. All read/write access to patient records shall be logged with user ID, timestamp, and record ID.

Metrics for non-functional requirements

Each property below has standard ways of being measured. Pick one when you write a verifiable NFR.

Property	Measure
Speed	Processed transactions per second; user/event response time; screen refresh time.
Size	Mbytes; memory footprint; storage required.
Ease of use	Training time; number of help frames.
Reliability	Mean time to failure; probability of unavailability; rate of failure occurrence; availability.
Robustness	Time to restart after failure; percentage of events causing failure; probability of data corruption on failure.
Portability	Percentage of target-dependent statements; number of target systems.

Domain requirements

Requirements derived from the application domain. They describe characteristics and features that reflect the domain in which the system operates.

Two examples

Train control system

A train control system shall take into account the braking characteristics in different weather conditions.

Medical (Mentcare)

The Mentcare system shall enforce confidentiality rules in accordance with national mental-health-records practices, including restricted access to records flagged as forensic.

Why domain requirements are hard

Two recurring problems make domain requirements harder to capture than functional or non-functional requirements.

Understandability

- Domain requirements are expressed in the language of the application domain.
- Software engineers may not have the domain vocabulary, so they may not understand what is being asked.
- *A simple-sounding sentence may hide deep domain rules.*

Implicitness

- Domain specialists know the area so well that they take the rules for granted.
- They do not think to mention rules they consider obvious.
- *Important constraints can therefore be left unstated, only to surface during testing or after release.*

Mitigation: spend time in the domain. Read its documents. Shadow its experts. Ask them to walk through an end-to-end case.

Activity: classify these requirements

In pairs, classify each statement on three axes. Five minutes, then we walk through the answers.

For each statement, decide: (a) user vs system requirement, (b) functional vs non-functional, (c) goal vs verifiable

S1	Doctors shall be able to view a patient's full medical record.
S2	The system should be highly available.
S3	Read access to any patient record shall be logged with user ID, timestamp, and record ID, and the log shall be retained for at least three years.

Summary and next session

Summary, in five points

1. Requirements engineering establishes services and constraints. Requirements are descriptions of those services and constraints.
2. Three abstraction levels: user requirements, system requirements, software specification. Each has a different audience and a different precision.
3. Three categories: functional (what), non-functional (how well), domain (what the application area demands).
4. NFRs sit in three buckets - product, organisational, external - and start as goals that must be turned into verifiable, measurable statements.
5. Domain requirements are hard because they are expressed in domain language and are often left implicit. Mitigate with glossaries and walkthroughs.

Next: the RE process and elicitation

Feasibility study, requirements elicitation and analysis, requirements specification, and requirements validation. Elicitation techniques: interviews, ethnography, scenarios, and use cases. Reading: Sommerville Chapter 4, sections 4.4 and 4.5.

COMP433

Software Engineering

Requirements engineering

Part 2: characteristics, processes, elicitation, validation, documentation

Hisham Ihshaish

Department of Computer Science, Birzeit University

Semester 2 (Spring) 2025/2026 | Sections 5 and 6

Requirements characteristics

What makes a requirement good, and where they typically go wrong

What makes a requirement good?

Six characteristics. You will see these again when we validate requirements.

Correctness

The requirement is part of the actual needs of the system, not implied or derived beyond scope.

Unambiguity

It can be interpreted in only one way by every reader.

Completeness

All services the customer needs are described, including externally imposed ones.

Consistency

No requirement conflicts with or contradicts another.

Verifiability

It uses a measure that can be objectively tested. You cannot evaluate what you cannot measure.

Traceability

Each requirement has a unique identifier and can be cross-referenced throughout the project.

Correctness and unambiguity: 'search'

Same one-line requirement, two failure modes. Watch how meaning slips.

Requirement 2.3 *A user shall be able to search the appointments lists for all clinics.*

Correctness

What the user actually needs:

Search for a patient name across all appointments, in all clinics.

What an implied requirement might smuggle in:

Also search by Date of Birth, Address, and other patient details, within a specific clinic.

Unambiguity

User intention:

Search a patient name across all appointments in all clinics at once.

Developer interpretation A:

User picks a clinic first, then searches within it.

Developer interpretation B:

User enters details once; the system searches all available clinics.

Completeness and consistency

The two properties that are easiest to demand and hardest to achieve in practice.

Completeness

All services required by the customer are described, including externally imposed or implied requirements.

Examples of often-omitted requirements:

- What happens on system failure
- What happens on out-of-range inputs
- Audit logging and traceability for sensitive actions
- Regulatory or legal requirements that 'everyone knows'

Consistency

There are no conflicts or contradictions in the descriptions of the system services.

Typical inconsistencies:

- Different stakeholders ask for opposing behaviours
- A non-functional requirement contradicts a functional one (e.g., logging vs anonymity)
- Two requirements describe the same service in incompatible ways
- Requirement evolves over time without removing the older version

In practice: *for large systems, producing a fully complete and fully consistent requirements document is almost impossible. The goal is to be complete and consistent enough that the residual risk is acceptable.*

Traceability and verifiability

If you cannot measure it, you cannot evaluate it. If you cannot trace it, you cannot test it.

Traceability

Each requirement has a unique identifier and is referenced consistently throughout the project: in specification, design, code, tests, and documentation.

Used for: change impact analysis, audit, verification, acceptance.

Verifiability

Each requirement uses an objective measure that can be tested. Vague language (fast, easy, reliable) is replaced with units and thresholds.

Rule of thumb: a requirement is verifiable when a tester knows, without asking, whether the system passes or fails it.

Worked example

R2.3 *A user shall be able to search the appointments lists for all clinics. Response time for search results shall not exceed 10 seconds for any given search.*

Traceable: the ID R2.3 lets us link this requirement to its parent UR, to test cases, and to the implementing module. **Verifiable:** the 10-second threshold gives the tester an objective pass/fail criterion.

Requirements and design

The textbook story is clean; reality leaks between the two activities.

In principle

Requirements state

what the system should do.

Design describes

how it does it.

In a strict plan-driven world, you finish requirements before you start design.

In practice

Requirements and design are interleaved. Three common reasons:

- The system architecture itself may shape the requirements (what is possible).
- The system must inter-operate with others, creating design-level constraints.
- A specific design (a technology, a framework) may be a domain requirement.

Iterative and agile processes make this overlap explicit; plan-driven processes hide it.

The requirements engineering process

Four interleaved sub-processes. RE in practice is iterative, not linear.

Four RE sub-processes

Generic activities common to every RE process. In practice they interleave; they do not run sequentially.

1

Requirements elicitation

Working with stakeholders to discover the application domain, the services the system should provide, and its constraints. Often called requirements discovery.

2

Requirements analysis

Understanding the implications of discovered requirements: classifying, organising, prioritising, negotiating. Output: the agreed specification.

3

Requirements validation

Checking that the requirements actually define the system the customer wants, against the six characteristics.

4

Requirements management

Managing changes to requirements as they evolve through the lifetime of the project and the system.

Elicitation: who do we talk to?

Elicitation is the part of RE that cannot be done at your desk. It is conversation, observation, and reading.

Definition. *Requirements elicitation (or discovery) involves technical staff working with customers to find out about the application domain, the services the system should provide, and its operational constraints.*

Stakeholders: who has a stake in this system?

End-users
The people who interact with the system day to day.

Managers
Those who oversee the user groups and procure the system.

Maintenance engineers
Those who will keep the system running after delivery.

Domain experts
Specialists in the application area (medicine, finance, law).

External bodies
Regulators, trade unions, partner organisations, suppliers.

The customer
The organisation paying for and accepting the system.

Requirements analysis

Analysis comes after discovery. You take raw findings and turn them into something developers can work from.

Definition. *Analysis is the process of understanding customer requirements and their implications. Technical staff work iteratively with customers on the discovered requirements:* ***classifying****,* ***organising****,* ***negotiating****,* ***prioritising****. The output is the agreed* ***requirements specification document*** *(SRS).*

Stakeholders in the Mentcare patient information system

Patients
Whose information is recorded in the system.

Doctors
Responsible for assessing and treating patients.

Nurses
Coordinate consultations and administer treatments.

Receptionists
Manage patient appointments and front desk.

IT staff
Install and maintain the system.

Auditors and regulators
Verify compliance with privacy and clinical standards.

Why analysis is hard

Four recurring problems above. The bottom diagram shows how analysis loops, not lines.

Stakeholders don't know what they want

They describe what exists, not what they need.

They use their own language

Domain vocabulary; two people may use the same word for different things.

Different stakeholders conflict

Doctors want speed; auditors want logging. You will mediate.

Requirements keep changing

The business changes while you are analysing.

Analysis is iterative: the four stages cycle until requirements stabilise.

Why analysis is hard

Four recurring problems above. The bottom diagram shows how analysis loops, not lines.

Stakeholders don't know what they want

They describe what exists, not what they need.

They use their own language

Domain vocabulary; two people may use the same word for different things.

Different stakeholders conflict

Doctors want speed; auditors want logging. You will mediate.

Requirements keep changing

The business changes while you are analysing.

What this means for the project

For Phase 2 of the projects, "requirements analysis" is the activity in which you:

1. Take the raw notes from interviewing the customer group (elicitation done)
2. Sort each statement: F, NF, or domain
3. Decide which are user-level (broad) and which are system-level (precise)
4. Write user requirements in natural language
5. Decompose each user requirement into one or more system requirements
6. Apply the six validation characteristics as a sanity check before moving on

The deliverable is the **SRS** -which contains both layers.

Requirements management planning

Before a project begins, you decide how requirements will be managed throughout its lifetime.

Requirements identification

Every requirement gets a unique ID. The ID lets you cross-reference with related requirements, with design, with test cases, with code.

Change management process

A defined set of activities that assess the impact and cost of a proposed change before accepting it. Often involves a change-control board.

Traceability policies

What relationships are tracked? Requirement-to-requirement, requirement-to-design, requirement-to-test. The deeper the trace, the more expensive the upkeep.

Tool support

From dedicated requirements-management tools (DOORS, Jama) to lighter options (spreadsheets, simple databases). Pick what your team will actually maintain.

Elicitation techniques

How to discover requirements: interviews, scenarios, use cases, ethnography.

Discovery techniques: an overview

Elicitation is non-trivial; you cannot just ask 'what should the system do?'. Five techniques, often used in combination.

Technique	Description
Interviewing	Open, closed, or focused conversations with stakeholders.
Scenario generation	Concrete real-life narratives of how the system is used.
Use case analysis	Scenario-based UML technique identifying actors and interactions.
Ethnography	Observational technique: spend time watching people work.
Prototyping	Build a partial system; let users react to something concrete.

Use cases *are introduced briefly here; full coverage with diagrams comes in coming lectures…on the UML modelling block.*

Interviewing: types and approach

Formal or informal conversations with stakeholders. Part of almost every RE process.

Types of interview

Open
Various issues are explored with the stakeholder, with few pre-defined questions.

Closed
Based on a pre-determined list of questions. Easier to compare across stakeholders.

Focused
Conducted with a cluster of stakeholders together, exploring a defined topic.

Effective interviewing

Be open-minded.	Drop pre-conceived ideas about what the system should do. Listen first, propose later.
Prompt to get the conversation going.	Use a springboard question, a draft requirement, or a low-fidelity prototype as a discussion anchor.

Interviews: running an effective meeting

The mechanics matter. A poorly run meeting wastes the stakeholder's time and yours.

1. Follow cultural introduction protocols at the start. Build trust before extracting information.
2. First meeting: aim to understand the business and its context, before getting into specific services.
3. Assign a chair at the start to keep time and steer the discussion.
4. Define an agenda with clear outcomes for each item; share it in advance if possible.
5. Set timescales per item and stick to them.
6. Capture actions and decisions during the meeting: who is responsible, by when.
7. Summarise actions and decisions verbally at the end before closing.

Interviews in practice: strengths and limits

Normally a mix of closed and open-ended. Good for some things, weak for others.

Where interviews work well

- Getting an overall understanding of what stakeholders do.
- Learning how stakeholders interact with current systems or processes.
- Discovering stated needs the stakeholder is aware of.
- Building trust and relationships with stakeholders for later phases.

Where interviews struggle

- Capturing domain knowledge: the engineer may not understand specialised vocabulary.
- Surfacing tacit knowledge: stakeholders may not know what they know, or assume it is obvious.
- Revealing actual workflow when it differs from official process.
- Eliciting requirements that only emerge under failure or unusual conditions.

Implication: *for domain or tacit requirements, combine interviews with ethnography or scenarios.*

Scenarios: real-life narratives of system use

Scenarios make the abstract concrete. They are easy for stakeholders to read and react to.

Definition. *A scenario is a real-life example of how a system can be used. It is told as a narrative, not as a list of requirements.*

A complete scenario includes:

#	Element	Description
1	**Starting situation**	What is true before the scenario begins: actors, system state, preconditions.
2	**Normal flow of events**	The expected sequence of actions and system responses, end to end.
3	**What can go wrong**	Exception paths, failure modes, recovery actions.
4	**Concurrent activities**	Other things happening at the same time that affect or are affected by the scenario.
5	**End state**	What is true when the scenario completes: outcomes, persistent changes, follow-on actions.

Scenario example: medical history

How a new patient's medical history is captured in the Mentcare system, end to end.

Scenario

Start

A new patient with no record on file is registered at reception. The receptionist is logged in.

Flow

The receptionist creates a patient record and a new medical history. The doctor opens the history during consultation and adds diagnostic notes.

Exception

If the patient already has a record at another clinic, the system merges histories under a single patient ID after the patient confirms identity.

Concurrent

While the history is being captured, an automated check runs against the national insurance database to verify cover.

End

A complete, indexed medical history is stored. Subsequent visits append to it. The receptionist prints a summary if requested.

Successful outputs of the scenario

Yes	Patient record created with unique ID
Yes	Medical history attached to the record
Yes	Insurance verification recorded
No	Failure case: duplicate ID across clinics not resolved automatically

Requirement extracted: *the system shall flag potential duplicate patient records across clinics and route them for manual confirmation.*

Use cases: scenarios in UML

Use cases are the UML formalisation of scenarios. We will draw them later, the concept.

A use case is a scenario-based UML technique that identifies the **actors** in an interaction and describes the **interaction** itself. A set of use cases describes all possible interactions with the system.

From scenario to use case

- A scenario is one narrative path through a use case.
- A use case captures all paths: the normal flow and its variations.
- A use case has a name, actors, and a goal.
- Multiple scenarios feed into the same use case.

Representation

- High-level: a UML use case diagram (actor figures, oval use cases, lines).
- Detailed: a structured textual description or a table.
- Behavioural: a **sequence diagram**, refining the use case step by step.
- We will cover all three in coming lectures; the UML modelling block.

Ethnography: watching, not asking

An observational technique. A researcher spends time inside the work setting and learns by being present.

Definition. *Ethnography is an observational technique used to understand operational processes and derive requirements that support them. A social scientist spends considerable time observing and analysing how people actually work. Stakeholders do not have to explain or articulate their work.*

Ethnography is particularly effective for:

Discovering actual workflows

How people really work, as opposed to how business processes claim they should work.

Cooperative work patterns

Requirements that emerge from how teams coordinate and how activities depend on each other.

Complex coordination

Work environments where stakeholders cannot articulate the coordination they perform unconsciously.

Cost: *ethnography is expensive in time and expertise. Use it for systems where tacit, cooperative, or domain-rich knowledge is critical.*

Choosing a technique: a worked decision

Tutorial 3 question 2. Two parts of one system have very different stakeholder structures. Pick techniques accordingly.

Scenario

A complex integrated manufacturing and ordering system. The manufacturing part has many fragmented stakeholders across a large area, who individually cannot articulate the process. The ordering part has fewer, more direct stakeholders.

Manufacturing side

Many stakeholders, fragmented, with tacit coordination.

Recommended techniques

- Ethnography: observe how actual coordination happens.
- Focused interviews with clusters of stakeholders.
- Scenarios constructed jointly with workers.

Ordering side

Fewer stakeholders, direct, can articulate requirements.

Recommended techniques

- Open and closed interviews with each stakeholder.
- Use case analysis with the stakeholders themselves.
- Prototyping: build a draft and iterate on feedback.

Requirements validation

Are these the right requirements? Six checks, three techniques.

Why validation matters

Requirements errors are the most expensive errors in software. Find them now, not after delivery.

Definition. *Requirements validation is concerned with demonstrating that the requirements define the system the customer really wants. It is not the same as verification, which checks that the system meets its specification.*

Cost of a late requirements error

Up to 100x the cost of fixing an implementation error.

Sommerville's rule of thumb: a requirement error discovered after delivery costs up to a hundred times more to fix than the same error caught during requirements analysis. The cost grows with each phase: design, implementation, testing, deployment, maintenance.

Validation is therefore done deliberately, not casually, before the requirements document is signed off.

Validation checks for every requirement

These are the questions you ask of each requirement in your document. Tutorial 3 question 1(c) applies them.

#	Check	Question
1	**Correctness/Validity**	*Does the system provide the functions that best support the customer's needs?*
2	**Unambiguity**	*Can any requirement be interpreted in more than one way?*
3	**Consistency**	*Are there requirements that conflict with each other?*
4	**Completeness**	*Are all functions required by the customer included?*
5	**Verifiability**	*Can each requirement be verified and validated objectively?*
6	**Traceability**	*Is each requirement traceable through a unique identifier?*
7	**Realism/Feasibility**	*Can the requirements be implemented within the time, budget, team skills, and technology available?*

Validation techniques

Three complementary techniques. Use more than one for high-risk requirements.

Requirements reviews

A systematic manual analysis of the requirements document by a small team. Different reviewers focus on different concerns (technical, customer, domain). Reviews are scheduled and minuted.

Prototyping

Build an executable model of the system, even if partial, and use it to elicit reactions. Stakeholders can spot misunderstandings in minutes that interviews would take weeks to reveal.

Test-case generation

Develop tests that the system would have to pass to satisfy each requirement. If you cannot write a test, the requirement is not verifiable. Forces clarity early.

Practical advice: *for your project Phase 2, use reviews (your group reviews each requirement together) and test-case generation (one acceptance test per system requirement). Prototyping comes in Phase 3 alongside design.*

The requirements document

Structure, notation, and writing practice.

The requirements document: structure

The agreed statement of system requirements. Two reference structures: IEEE 830 (concise) and Sommerville (expanded).

IEEE 830 structure

Generic, instantiated per system.

1. Introduction
2. General description
3. Specific requirements
4. Appendices
5. Index

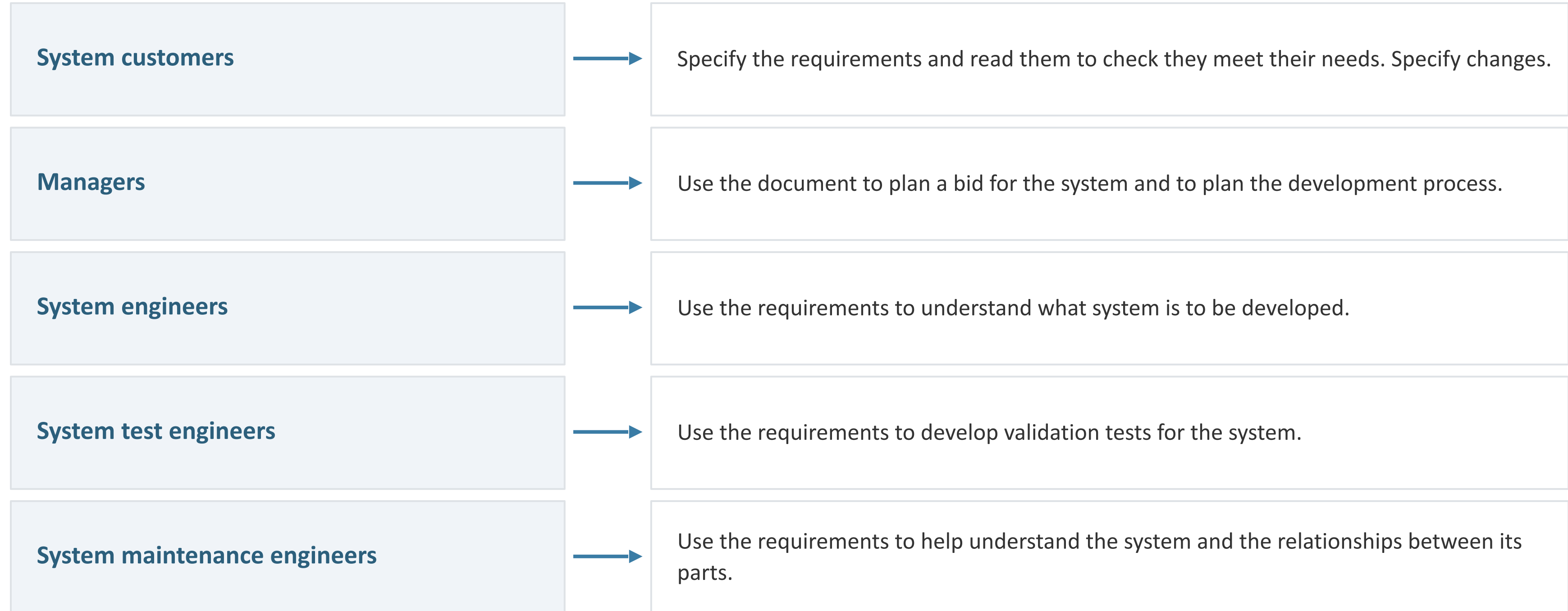
Sommerville's expanded structure

More detail; closer to a real industry SRS.

1. Preface
2. Introduction
3. Glossary
4. User requirements definition
5. System requirements specification
6. System architecture
7. System models
8. System evolution
9. Appendices and index

Who reads the requirements document?

Five distinct audiences. Each reads it differently. Write so all of them can use it.



How requirements are written

Five notations for system requirements. Plus the writing guidelines that prevent the most common mistakes.

Notations for system requirements

Natural language	Numbered sentences in plain English. One sentence, one requirement.
Structured natural language	Templates and standard forms. Each field captures one aspect (purpose, trigger, workflow, ...).
Design description languages	Programming-like but more abstract. Useful for interface specifications, rare elsewhere now.
Graphical notations	Annotated diagrams (UML use cases, activity diagrams) augmenting text.
Mathematical specifications	Finite-state machines, set theory, formal logic. Unambiguous but customers cannot read them.

Writing guidelines

- Invent a standard format and use it for every requirement.
- Use 'shall' for mandatory requirements; 'should' for desirable ones.
- Use text highlighting (bold, italic) to mark the key part of each requirement.
- Include a rationale explaining why each requirement is necessary. Avoid computer jargon.

Insulin pump: the running example

Before the structured specification, a quick look at the device we are specifying.

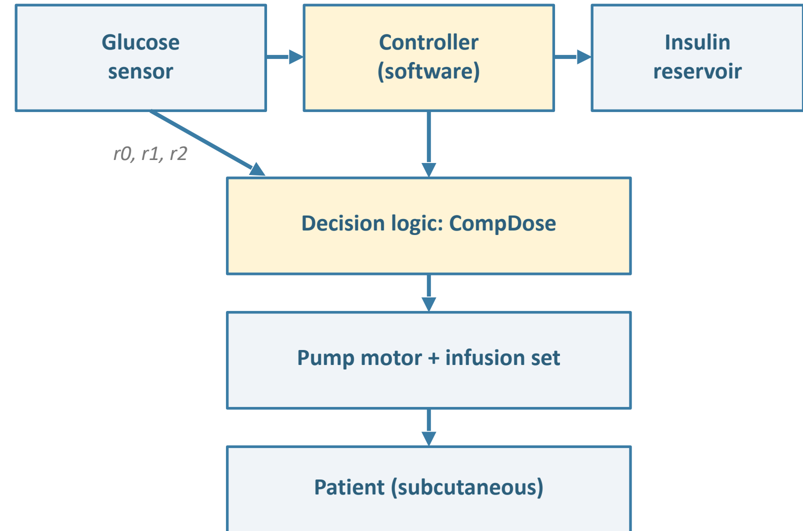
What it is

An insulin pump is a small medical device worn by patients with diabetes. It measures blood sugar and delivers insulin in small, frequent doses to keep sugar levels in a safe range. Sometimes called a virtual pancreas.

Why this example

It is safety-critical, automatic, sensor-driven, and constrained by physical limits. Each of those properties forces clear, structured requirements.

Components and flow



Variables to remember: *r2 is the current reading, r1 the previous, r0 the one before. CompDose is the computed insulin dose. We use these on the next slides.*

Structured specification: worked example

An insulin pump computes a dose every 10 minutes. Here is one requirement, fully structured.

Title	Compute Insulin Dose (CompDose)
Purpose	Compute the insulin dose to deliver, based on the measured blood sugar level.
Description	Computes the dose when the current measured level of sugar is in the safe zone, between 3 and 7 units.
Actors	SystemTimer (triggers the computation)
Trigger	Automatic; every 10 minutes by the SystemTimer.
Pre-condition	Insulin reservoir contains at least the maximum allowed single dose.
Workflow	1. Read current sugar level r2. 2. Read stored previous readings r0, r1. 3. Compute trend within safe zone. 4. Compute dose based on trend. 5. If dose is in allowed range (5-15 mg), deliver. Else beep. 6. Shift readings: r0 = r1; r1 = r2.
Post-condition	Previous readings replaced and stored. Dose delivered or alarm raised.

Three templates, same job

There is no single 'correct' structured template. Pick the fields that fit your system's character.

Function-oriented

Pure computations; sensor-driven loops.

Fields

- Function
- Description
- Inputs
- Source
- Outputs
- Destination

Sommerville Fig. 4.10

Action-oriented

Behavioural rules; state-changing operations.

Fields

- Action
- Requirements
- Pre-condition
- Post-condition
- Side effects

Sommerville Fig. 4.11

Use-case-oriented

User-facing services; interactions with actors.

Fields

- Title
- Purpose
- Actors
- Trigger
- Pre-condition
- Workflow
- Post-condition

Adel L3 slide 73

Tabular specification

When the requirement is a set of rules, a decision table is clearer than prose. Example: the CompDose action.

Condition	Action
Sugar level falling (r2 < r1)	CompDose = 0
Sugar level stable (r2 = r1)	CompDose = 0
Sugar increasing, rate of increase decreasing ((r2 - r1) < (r1 - r0))	CompDose = 0
Sugar increasing, rate stable or increasing ((r2 - r1) >= (r1 - r0))	CompDose = round((r2 - r1) / 4) If rounded result = 0 then CompDose = MinimumDose

When to use: *rule-based logic, decision trees, state-dependent behaviour. A decision table forces you to enumerate every case -- and reveals the gaps when you cannot.*

Activity: validate a small set

Apply the six characteristics to this set of requirements. Five minutes.

Requirement set

1. *The system shall provide a service for users (students) to register and create an account.*
2. *Users shall be able to submit queries.*
3. *The system shall adhere to the guidelines set by the ministry of higher education.*
4. *Users should be able to listen to music when using the website.*
5. *The system shall allow only the registered users to use the services of the website.*

Your task

1. For each requirement, mark which of the six characteristics it fails: correctness, unambiguity, completeness, consistency, traceability, feasibility.
2. Identify which one looks like a domain requirement, and which one looks out of scope.
3. Be ready to defend your answers when we discuss as a group.

Summary

Today, in five points.

1. A good requirement is correct, unambiguous, complete, consistent, verifiable, and traceable. Apply all six during validation.
2. RE has four interleaved sub-processes: elicitation, analysis, validation, and management. They iterate; they do not run in sequence.
3. Choose elicitation techniques to fit stakeholder structure. Interviews and use cases for direct stakeholders; ethnography for fragmented, tacit settings.
4. Validate deliberately. The cost of a late requirements error is up to a hundred times the cost of fixing it during analysis.
5. Write requirements in natural language and structured templates. Use 'shall' for mandatory, 'should' for desirable, and always include a rationale.

Next session and project links

Where today's content goes in the project, and what comes next.

In your project

- Phase 2 (Requirements analysis): write a structured SRS using the Sommerville document layout.
- Apply the six validation characteristics to every requirement before submission.
- Use the structured-template format (Title, Purpose, Trigger, Workflow, ...) for at least three of your system requirements.
- Each requirement has a unique ID and a rationale.

Lecture 7: UML modelling

- Use case diagrams: actors, use cases, include and extend relationships.
- Detailed use case descriptions: from scenarios to UML.
- Use cases as the input to design.
- Tools: Lucidchart, Astah, Enterprise Architect. No AI-generated UML.

References

Sommerville, I. (2016).
Software Engineering. 10th Edition. Pearson. Chapter 4 (Requirements engineering), sections 4.4-4.7.

Bruegge, B. and Dutoit, A. (2013).
Object-Oriented Software Engineering Using UML, Patterns, and Java. 3rd Edition. Prentice Hall. Chapter 4 (Requirements elicitation).

IEEE Std 830 (1998).
IEEE Recommended Practice for Software Requirements Specifications. Superseded; useful historical reference for SRS structure.

ISO/IEC/IEEE 29148:2018.
Systems and software engineering - Life cycle processes - Requirements engineering. The current international standard.

Taweel, A. (2025/26).
COMP433 Lecture 3: Requirements engineering. Birzeit University course materials.

COMP433 (2025/26).
Tutorial 3: Requirements engineering. Used in this lecture for the validation activity.

Boehm, B. (1981).
Software Engineering Economics. Prentice Hall. Original source for the cost-of-error figure cited in validation.

References

- Sommerville, I. (2016). Software Engineering. 10th Edition. Pearson. Chapter 4 (Requirements engineering).
- Bruegge, B. and Dutoit, A. (2013). Object-Oriented Software Engineering Using UML, Patterns, and Java. 3rd Edition. Prentice Hall. Chapter 4 (Requirements elicitation).
- IEEE (1998). IEEE Recommended Practice for Software Requirements Specifications. IEEE Std 830-1998 (superseded; useful as historical reference).
- ISO/IEC/IEEE 29148 (2018). Systems and software engineering - Life cycle processes - Requirements engineering. The current international standard.
- OWASP. Password Storage Cheat Sheet. owasp.org. Reference for modern password hashing practices.
- NIST SP 800-63B (2017, reaffirmed 2020). Digital Identity Guidelines - Authentication and Lifecycle Management.