Software engineering explanation

🗀 Fichier chargé

Please explain each concept covered in these slides do not miss details and be simple and efficient please

Lecture des documents interrompue

Perfect 👍 The slides you uploaded — **"Software Engineering Lecture 1" by Okba Tibermacine (Sept 2025)** — cover the *foundations of software engineering*.
Below is a **complete, simple, and efficient explanation of every concept** in the slides (from start to end, no detail missed).

🧩 1. Programming vs Software Engineering

Programming

Writing code to make something work (e.g., a simple "Hello World").
It focuses on **making the program run**, not on longevity or maintenance.

Software Engineering

Goes far beyond coding. It's about **designing, building, and maintaining** software that:

- Has lasting value
- Can be changed easily
- Is understandable by others

Key differences

Aspect	Programming	Software Engineering
Focus	Make it work	Make it work, right, and maintainable
Approach	Ad-hoc	Systematic, planned
Goal	Quick result	Quality, scalability, maintainability

Aspect	Programming	Software Engineering
Includes	Only code	Code + documentation + testing + processes

Example

Changing “Hello World” → “Hello All” may seem simple, but real engineers must consider:

- Requirements change
- Cost and business implications
- Ethical support and maintenance
- Scalability and future updates

🏗️ 2. Why Software Engineering?

Importance

> “Our civilization runs on software.” — *Bjarne Stroustrup*

Software powers everything — banking, transport, healthcare, government, education.

Software failures are costly

68% of software projects fail (cancelled, over budget, late, or unusable).

Example 1: Ariane 5 Rocket (1996)

- $500M explosion caused by a **data conversion bug (64-bit → 16-bit)**.
- Lesson: even small code errors are catastrophic without engineering discipline.

Example 2: UK NHS NPfIT (2002–2011)

- £12B lost, 9 years wasted.
- Poor planning, unclear requirements, management failure.

Conclusion: Software engineering provides structure to prevent such disasters.

🧠 3. Definition and Historical Context

Definition

> "Systematic, disciplined, and quantifiable approach to development, operation, and maintenance of software."

- **Systematic** → organized methods
- **Disciplined** → follow rules and standards
- **Quantifiable** → measure progress, cost, quality

History

- 1960s: hardware was advancing, but software was chaotic → *software crisis*.
- **1968 NATO Conference** introduced "Software Engineering" as a formal discipline.

Software Crisis: inability to develop reliable, efficient programs on time.

🧩 4. No Silver Bullet (Frederick Brooks, 1987)

> There is **no magical solution** that makes software development easy, fast, and error-free.

Two kinds of difficulties

Essential Difficulties (inherent)

1. **Complexity** – enormous interconnections between components.
2. **Conformity** – must adapt to changing laws, environments, and constraints.
3. **Changeability** – constant need for new features and updates.
4. **Invisibility** – software is abstract; you can't *see* its structure like a car or bridge.

Accidental Difficulties (solvable)

- Bad IDEs, poor documentation, buggy compilers, weak frameworks.

✅ *We can reduce accidental difficulties, but essential ones always remain.*

📚 5. Topics of Study in Software Engineering (SWEBOK 2024)

IEEE's **Software Engineering Body of Knowledge** defines **18 key areas**:

1. Software Requirements
2. Software Architecture
3. Software Design
4. Software Construction (coding)

5. Software Testing
6. Software Engineering Operations
7. Software Maintenance
8. Software Configuration Management
9. Software Engineering Management
10. Software Process
11. Software Models & Methods
12. Software Quality
13. Software Security
14. Professional Practice
15. Software Economics
16. Computing Foundations
17. Mathematical Foundations
18. Engineering Foundations

📋 6. Key Activities Explained

Requirements Engineering

Defines **what** the software should do and **how** it should behave.

Steps:

- Elicit → gather from users
- Analyze → resolve conflicts
- Document → specify clearly
- Validate → confirm with stakeholders

Types

- **Functional** → system features (e.g., transfer money)
- **Non-functional** → system qualities (e.g., secure, fast, usable)

Software Design

Defines structure of the system (classes, modules, interactions).

- **Provided Interfaces** → what a module offers
- **Required Interfaces** → what it depends on

Architectural Design → higher-level structure (packages, subsystems, data flow).

Software Construction

Actual **implementation (coding)** phase.
Involves algorithms, frameworks, tools, documentation, AI-assisted development, etc.

Software Testing

Running code with selected inputs to check if output matches expectations.

> "Testing shows the presence, not the absence, of bugs." — Dijkstra

Types:

- Unit (class)
- Integration (modules)
- Performance (speed)
- Usability (user-friendliness)

Two purposes:

- **Verification** → built correctly
- **Validation** → built what user wanted

Software Maintenance

Post-delivery modifications:

1. **Corrective** – fix bugs
2. **Preventive** – prevent future issues
3. **Adaptive** – adjust to new OS/hardware
4. **Refactoring** – improve design
5. **Evolutionary** – add new features

Example: *Y2K fix* (DD-MM-YY → DD-MM-YYYY).

Project Management

Plan, organize, lead, and control project resources.

Phases: Planning → Organizing → Executing → Monitoring → Closing

Brooks' Law:

> "Adding manpower to a late project makes it later."
> More people = more communication = more delay.

⚙️ 7. Software Processes

A **software process** = structured sequence of activities from idea → delivery.

Types

Plan-driven	Agile
Rigid, documented	Flexible, adaptive
Fixed requirements	Evolving requirements
Example: Waterfall	Example: Scrum, XP

Choose based on stability of requirements.

🏆 8. Goals & Qualities of a Good Software Product

Goal

Deliver high-quality software:

- That meets real needs
- On time
- Within budget

Qualities

1. **Correctness** – works as intended
2. **Usability** – easy to use
3. **Reliability** – consistent and accurate
4. **Maintainability** – easy to modify
5. **Security** – protected from attacks

6. **Efficiency** – runs fast and light
7. **Scalability** – handles growth
8. **Portability** – runs on multiple platforms
9. **Reusability** – parts usable elsewhere
10. **Adaptability** – flexible for new needs

💡 9. Principles of Software Engineering

No universal laws — but many **guidelines** exist.

Davis's 15 Principles (simplified)

1. Quality first
2. High-quality is achievable
3. Deliver early versions
4. Understand the problem before requirements
5. Evaluate design options
6. Use the right process
7. Different languages for different phases
8. Minimize gap between real & digital world
9. Tools are secondary to good techniques
10. Correctness > speed
11. Review and inspect code
12. Good management > technology
13. People = key to success
14. Follow with care
15. Take responsibility

Additional advice:

- Design for change
- Don't test your own code
- Plan to throw one version away

🧮 10. Classification of Software Systems (Bertrand Meyer)

Type	Description	Examples	Engineering Level
A (Acute)	Life-critical	Rockets, planes, medical devices	Extremely rigorous
B (Business)	Affects finances or operations	ERP, banking systems, websites	Standard SE practices
C (Casual)	Low-impact, small scale	School projects, one-time scripts	Minimal engineering

💡 *Match engineering effort to system criticality.*

👥 11. Roles in a Software Engineering Team

Role	Key Responsibility
Project Manager	Plans, tracks, coordinates resources
Product Manager	Defines product vision & roadmap
Software Architect	Designs high-level system structure
UX/UI Designer	Designs user interface and experience
Software Developer	Writes and maintains code
QA Engineer	Tests and ensures quality
DevOps Engineer	Manages deployment and infrastructure
Business Analyst	Gathers and documents requirements

Team success = collaboration + communication + shared goals

🧭 Summary

Concept	Core Idea
Programming vs SE	SE is systematic, long-term, and maintainable
Why SE	Software runs the world; failures cost millions
Definition	Systematic, disciplined, measurable development
No Silver Bullet	No magical solution; focus on essential challenges
SWEBOK	18 domains covering all SE areas
Processes	Plan-driven vs Agile
Qualities	Correct, usable, secure, efficient
Principles	Quality first, plan early, manage well
Classification	Type A/B/C systems → adapt engineering level
Team Roles	Collaboration across specialized positions

Would you like me to make this into a **formatted study summary PDF** (with sections, bullet lists, and examples) for easy revision?

🗁 Fichier chargé

Please explain each concept covered in these slides do not miss details and be simple and efficient please

Lecture des documents interrompue

Perfect 👍 You uploaded **"Software Engineering – Lecture 2: Software Processes (Part 1)"** by **Okba Tibermacine (October 7 2025)**.
Below is a **complete, clear, and efficient explanation** of **every concept** in these slides — nothing skipped.

🧩 1. Revision from Last Week

Definition of Software Engineering

> A discipline that integrates processes, methods, and tools to analyze, design, develop, test, and maintain software systems.
> First used in 1968 at the NATO Conference to address the **software crisis**.

Goal of Software Development

Build **high-quality software** that meets **real customer needs** — not just working code.

Davis's Principles (1994)

Key professional guidelines:

- Quality first
- Give products early to customers
- Understand the problem before requirements
- Evaluate design alternatives
- Use appropriate process models
- Manage well — people > tools
- Inspect code and plan for change

Qualities of Industrial-Strength Software

- Correctness
- Usability
- Reliability

- Robustness
- Efficiency
- Security
- Portability
- Maintainability

The Software Lifecycle

Steps: **Requirements → Design → Implementation → Testing → Deployment → Maintenance**

Real-World vs Ideal Process

In theory, development follows a clean, sequential flow.
In practice, it's messy and iterative — changes, bugs, and delays happen.

Process Types

Plan-Driven	Agile
Planned upfront, heavy documentation	Incremental, lightweight docs
Fixed requirements	Requirements evolve
Example: Waterfall	Example: Scrum, XP

🧠 2. Software Processes and Activities

What is a Software Process?

A **structured set of activities** used to build software.
It defines *who* does *what*, *when*, and *how* to turn **requirements into a working system**.

Four Components of a Process

1. **Activities** – actions done (e.g., coding, testing)
2. **Roles** – responsibilities (developer, tester, etc.)
3. **Artifacts/Deliverables** – tangible outputs (docs, code, reports)
4. **Workflows** – the order and dependencies between activities

---

Process Activity

A **unit of work** performed by a role that creates or modifies artifacts.

- **Pre-condition:** What must be true before starting (e.g., requirements complete)
- **Post-condition:** What must be true after completion (e.g., design reviewed)
- Each activity defines:
  - Deliverable
  - Duration
  - Resources
  - Dependencies

Generic Activities in Any Software Process

1. Requirements specification
2. Design and planning
3. Coding / implementation
4. Testing
5. Integration
6. Maintenance

Umbrella Activities

Support tasks done throughout the project:

- **Project tracking & control** – monitor progress
- **Risk management** – anticipate and mitigate risks
- **Quality assurance (SQA)** – ensure standards are met
- **Technical reviews** – check design/code quality
- **Measurement & metrics** – collect data to improve processes

---

Tasks

Smaller **work units** contributing to an activity.
Examples:

- Implement feature X
- Fix a bug

- Send invoices
- Collect user data
- Hire a translator

Rules:

- Usually assigned to one person
- Can be reassigned or split into sub-tasks
- Each task should have a **due date**

Task Statuses

Common stages:

1. **Backlog** – idea or pending work
2. **In Plan** – selected for execution
3. **Started** – being worked on
4. **Completed** – finished, awaiting review
5. **To Deliver** – ready for deployment
6. **To Accept** – pending client approval
7. **Closed** – done & accepted
8. **Trash** – dropped/ignored

💡 *Workflows differ across organizations.*

Tools for Tracking Tasks

- **Jira**
- **Trello**
- **Asana**
  They visualize project progress and highlight delays or blockers.

Policy: A task should only be marked *completed* after **review** by someone other than its creator.

⚙️ 3. Software Process Models

What Is a Process Model?

A **representation** of how software development activities are structured and ordered.

- Describes **what happens**, **when**, and **in what sequence**.
- Shows process **flow** — can be **linear**, **iterative**, or **parallel**.

Example Forms:

- **Linear**: Step 1 → Step 2 → Step 3
- **Iterative**: Feedback loops; repeated improvements
- **Parallel**: Multiple tasks done simultaneously

Why Use a Process Model?

- To stay **organized** and coordinate work systematically.
- Ensures **quality** and consistent progress.
  Even a **solo developer** benefits from it for planning and tracking.

🧩 4. Plan-Driven vs Agile (Reminder)

Plan-Driven	Agile
Predetermined plan	Incremental, flexible
Heavy documentation	Lightweight documentation
Suited for stable requirements	Suited for changing requirements

---

🧱 5. Simple Process Models

1. Code-and-Fix Model

- Developers start coding **immediately**, no formal design or documentation.
- Whenever errors appear, they fix them on the fly.

Problems:

- No structure, no documentation
- Difficult maintenance
- Low quality and high cost
- Not suitable for large projects
- Developers themselves must test (no specs for testers)

Used only for very small or experimental projects.

---

2. No-Code Process Model

- Uses **drag-and-drop platforms** with pre-built components (e.g., **Bubble**, visual programming).
- Fast development, no traditional coding.

Limitations:

- Poor scalability (bad for large apps)
- Limited custom logic
- Vendor lock-in (dependent on platform)

Good for: small prototypes or startup MVPs.
Not good for: enterprise or performance-critical systems.

💧 6. Traditional Methodologies

A. Waterfall Model

Concept:
Sequential process — move to next phase only when the previous one is complete.

Phases:
Requirements → Design → Implementation → Testing → Deployment → Maintenance

Features:

1. Sequential and structured
2. Document-driven
3. Quality controlled
4. Strict planning and milestones

Strengths:

- Works for well-understood problems
- Easy to manage (clear deliverables)
- Good documentation trail
- Prevents endless requirement changes

Weaknesses:

- Inflexible — no changes once started
- No working software until late
- Minimal user feedback
- Unrealistic separation between specification and design

When to Use:

- Embedded or critical systems (where safety = priority)
- Large, stable projects

When *Not* to Use:

- Unclear or changing requirements
- Innovative projects needing iterations

Boehm's Cost of Change (1981):
Fixing bugs becomes exponentially more expensive later in the cycle.

B. Incremental Model

- Breaks software into **increments** (modules or versions).
- Start with core features, then add more in later versions.
- Each increment uses a mini-waterfall cycle.

Strengths:

- Reduces complexity (divide & conquer)
- Continuous customer feedback
- Progressive delivery

Weaknesses:

- Higher cost due to repeated testing and integration
- Harder long-term planning
- Risk of breaking production with new increments

C. Prototyping Model

- Build a **quick, rough version** (prototype) to clarify unclear requirements.
- Customers test the prototype and give feedback.
- Once needs are understood, the prototype is **discarded**, and the real system is built.

Tools:

- Paper sketches, **Figma**, **Balsamiq**, **Proto.io**, etc.

Strengths:

- Better requirement understanding
- Reduces miscommunication

Weaknesses:

- Costly and time-consuming
- Clients may underestimate real effort (they think final system will be as easy to build)

D. Spiral Model (Barry Boehm, 1986)

Combines **Waterfall**, **Prototyping**, and **Iterative** models.
It's a **risk-driven approach** — the main goal is to manage risk through multiple spirals (cycles).

Four Quadrants in Each Spiral:

1. **Planning:** gather requirements, set objectives
2. **Risk Analysis:** identify and mitigate risks
3. **Engineering:** design, code, test the product
4. **Evaluation:** get user feedback

Common Risks:

- Technical (wrong tech, performance)
- Schedule (delays)
- Cost (budget overruns)
- Resource (staff or skills)
- Business (market changes)
- Security (data breaches)

Advantages:

- Excellent risk management
- Flexible and adaptive
- Early feedback, supports large projects
- Prototypes help reduce uncertainty

Disadvantages:

- Complex and expensive
- Needs expert risk analysts
- Not suited for small/simple projects

Other Traditional Models

- **V-Model:** Verification and Validation in parallel
- **RAD (Rapid Application Development):** emphasizes speed and quick prototypes

General Weakness of Traditional Models:

- Long development cycles
- Hard to adapt to changing requirements
- Heavy documentation
- Overreliance on “heroic” programmers

🚀 7. What’s Next?

Next lecture → **Modern Methodologies (Agile Models)**

📅 8. Project Information

Deadlines

- **Oct 9 2025:** Submit group project topics.
  If missing members, contact your instructor.

Team Project Phases

Phase 1 – Without LLMs (Weeks 1–11):

- No ChatGPT, Copilot, etc.
- Goal: evaluate human-only analytical and coding ability.

Phase 2 – LLM-Based Development:

- Teams swap projects and use AI tools for implementation.
- Compare: *human-only vs LLM-assisted* development.

🧭 Summary Table

Concept	Main Idea
Software Process	Structured way to build software
Components	Activities, Roles, Artifacts, Workflows
Activities	Units of work with pre/post-conditions
Umbrella Activities	Risk mgmt, quality, reviews, metrics
Tasks	Small work items with workflow & status
Process Models	Define sequence/organization of tasks
Code-and-Fix	Unstructured coding – bad for large projects
No-Code	Visual tools – fast but limited
Waterfall	Sequential, rigid, good for stable requirements
Incremental	Add features in steps
Prototyping	Build mockup to clarify needs
Spiral	Iterative + risk-driven
Traditional vs Agile	Plan-driven vs flexible
Team Projects	Human-only phase then AI-assisted phase

Would you like me to make this into a **study summary PDF** (formatted with headings, diagrams, and examples) to revise easily before exams?

🗂 Fichier chargé

Please explain each concept covered in these slides do not miss details and be simple and efficient please

Lecture des documents interrompue

Perfect 👍 You uploaded **"Software Engineering – Lecture 3: Software Processes (Part 2)"** by **Okba Tibermacine (Oct 15 2025)** — this lecture explains **Agile methodologies** in detail (Agile philosophy, RUP, Scrum, and XP).
Here's a **simple, complete, and efficient explanation** of every concept (no detail missed 👇).

🧠 1. From Last Week

- **Software process concepts:** Activity, Task, Process Model.
- **Traditional models:** Waterfall, Spiral, Incremental, Prototyping.
- **Iterative vs Incremental:**
  - *Iterative:* improve what already exists.
  - *Incremental:* add new features.
  - **Agile = both:** build new parts + improve continuously.

Real-world challenge

1. **Unclear requirements** – users don't always know what they want, and needs change.
2. **Over-documentation** – long documents that no one uses.
   Hence → Agile methods appeared.

⚙️ 2. What Is Agile?

Definition:
A family of software development methodologies that build software in **short iterations** (sprints), welcome **change**, and focus on **customer collaboration**.

Agile appeared as a reaction to heavy, plan-driven methods that were too rigid.

📜 3. Agile Manifesto (2001)

Created by **17 software engineers** (the “Agile Alliance”).

They defined 4 core values:

Agile Values	Meaning
Individuals & interactions > processes & tools	People and communication matter more than strict procedures or fancy tools.
Working software > comprehensive documentation	Delivering something that actually runs is more important than thick reports.
Customer collaboration > contract negotiation	Ongoing discussion with clients > rigid contracts.
Responding to change > following a plan	Flexibility > sticking to outdated plans.

👉 Documentation, contracts, and plans still have value — but people, software, and adaptability matter **more**.

Examples for each value

- **Individuals & interactions:** daily stand-ups > formal tickets.
- **Working software:** prototype > 200-page spec.
- **Customer collaboration:** adjust features during dev > follow fixed contract.
- **Responding to change:** change priorities mid-project > force old plan.

💡 4. Agile Principles (Key Rephrased)

1. **Short releases and iterations** – deliver small, usable updates often.
2. **Incremental design** – don’t over-design early; refine as you learn.
3. **User involvement** – constant feedback from real users.
4. **Embrace change** – design to handle changing requirements.
5. **People not process** – trust skilled developers, not strict rules.
6. **Maintain simplicity** – build only what’s needed, avoid complexity.
7. **Sustainable pace** – no overtime → steady productivity.

🧩 5. Rational Unified Process (RUP)

A **framework by Rational Software (IBM)** combining structure + Agile ideas.
It is:

- **Use-case & requirements-driven**
- **Architecture-centric**
- **Iterative and incremental**

Four Phases of RUP

Phase	Goal
1. Inception	Define scope, vision, main use-cases, schedule, resources, and risks.
2. Elaboration	Clarify all major requirements, decide architecture & platforms, plan testing, define metrics.
3. Construction	Actual coding + testing → produce working version (beta).
4. Transition	Integrate components, finalize docs, train users, release the product.

Each phase may have several **iterations** and includes varying levels of all activities (design, test, etc.).

🌀 6. Scrum Framework

Definition

An **Agile project-management framework** for iterative, incremental delivery.
Focuses on **teamwork**, **communication**, **transparency**, and **continuous improvement** with **self-organized teams**.

Why Scrum?

- Builds product in **manageable chunks (sprints)**
- Handles **changing requirements** easily
- Ensures **fast feedback** and better morale
- Ideal for **startups and mobile/web apps**

“Scrum” term

- From **rugby** → team restart together after a stop.
- Unlike the **relay (Waterfall)** model, everyone collaborates simultaneously.

👥 7. Scrum Roles

Role	Responsibility
Product Owner (PO)	Defines vision & goals, manages Product Backlog, prioritizes items, validates increments. *Single person only!*
Scrum Master (SM)	Facilitator & coach → ensures Scrum is followed, removes impediments, organizes meetings, guides the team.
Development Team	Cross-functional professionals ($\leqslant$ 10 members) who implement, integrate, and deliver each sprint’s increment.

📦 8. Scrum Artifacts

Artifact	Description
Product Backlog	Ordered list of all desired features (user stories, epics, tasks) managed by PO.
Sprint Backlog	Subset of items + Sprint Goal + tasks selected for the current sprint (2–4 weeks).

Artifact	Description
Product Increment	The working software built during the sprint – potentially shippable version.

Types of Items

- **User Story:** *As a user, I want … so that …*
- **Task:** technical step (e.g., implement UI).
- **Epic:** large feature grouping several stories.

⏱️ 9. Scrum Events (Ceremonies)

Event	Purpose	Timebox
Sprint Planning	Decide *what* to do and *how* to do it (create Sprint Backlog & Goal).	4–8 h (one-month sprint)
Daily Scrum	15-min meeting → each dev answers: what I did / will do / blockers.	15 min
Sprint Review	Present increment to stakeholders, collect feedback, adapt Product Backlog.	⩽ 4 h
Sprint Retrospective	Team reflects on process → decide improvements for next sprint.	⩽ 3 h

---

📈 10. Scrum Burndown Chart

A graph tracking **remaining work vs time** in a sprint.
Helps predict if sprint will finish on time.
Velocity = work completed / time elapsed.

💪 11. Scrum Values

1. **Commitment** – to goals & team.
2. **Focus** – on Sprint Goal.
3. **Openness** – about work & challenges.
4. **Respect** – for everyone's abilities.
5. **Courage** – to face issues and make tough calls.

🧭 12. How to Start Scrum (Setup)

1. Choose tools (Jira, Trello, Slack).
2. Assign roles (PO, SM, Devs).
3. Create Product Backlog (user stories).
4. Schedule ceremonies with fixed times (e.g., daily 9 AM).
5. Run sprints → plan → develop → review → retro.

🍱 13. Scrum Simulation – MealPrep Pro (App)

A full example showing Scrum in action.

Phase 1 – Vision & Team

- **Vision:** Healthy-meal-planning mobile app.
- **Team:**
  - Omar (PO) – vision & backlog
  - Samir (SM) – facilitator
  - Dalia (UI/UX), Daoud (Backend), Douaa (Mobile)

Phase 2 – Sprint 1

- Goal: register user + browse recipes + create meal plan.
- Tasks: UI designs, DB schemas, APIs, integration.
- Daily Scrums to discuss progress & blockers.
- Sprint Review → demo working features & gather feedback.
- Sprint Retrospective → decide improvements (pair programming, refinement, design reviews).
- Sprint 2 starts with new goals (auto grocery list, meal categories, nutrition totals).

Takeaway: Scrum delivers real value every 2 weeks through continuous feedback and team improvement.

💻 14. eXtreme Programming (XP)

Definition:
An Agile method focused on **code quality** and **developer discipline**.
Key features:

- user stories for requirements
- **test-driven development (TDD)**
- **pair programming**
- **continuous refactoring & integration**
- constant customer presence

XP Principles (12 Total)

#	Principle	Simple Meaning
1	**Code Standard**	Follow consistent naming, indentation, exception use → readable code.
2	**Refactoring**	Clean & restructure code without changing behavior → remove duplication and complexity.
3	**Pair Programming**	Two devs work together → one codes, one reviews → better quality.
4	**Test-Driven Development (TDD)**	Write unit tests before the code → code must pass tests.
5	**Collective Code Ownership**	Everyone can edit any code → shared responsibility.

#	Principle	Simple Meaning
6	**Continuous Integration (CI)**	Merge and test code frequently (automated builds).
7	**Small Releases**	Deliver frequent minor versions → early feedback.
8	**Onsite Customer**	Customer works with team daily → clarify requirements.
9	**Sustainable Pace**	No overtime → consistent performance.
10	**System Metaphor**	Common story or naming scheme everyone understands.
11	**Simple Design**	Start simple, evolve through iterations (MVP first).
12	**Planning Game**	Collaborative planning each iteration to set features and tasks.

Benefits of XP

✅ Stable, clean code
✅ Fast feature delivery
✅ High customer satisfaction
✅ Good team collaboration
✅ No overtime

Drawbacks of XP

❌ Hard to estimate time/cost
❌ Documentation often lacking
❌ Pair programming is time-consuming
❌ Short-term focus over long-term design

⚠️ 15. Agile Challenges

1. **Legal/contracts:** Agile's flexibility ≠ traditional fixed contracts.
2. **Maintenance:** Agile fits new projects better than legacy system maintenance.
3. **No single formula:** each team must adapt methods to context.

🧭 16. Choosing a Process Model

> There is **no perfect model.**
> The best process depends on:
> – project type
> – team size and skills
> – organization culture
> – time and budget constraints.

You can even combine parts of multiple methods (e.g., Scrum + XP = Scrumban).

🧾 Summary Table

Concept	Essence
Agile	Flexible, iterative dev with short cycles
Manifesto	4 values favor people, working software, collaboration, adaptability
Principles	Deliver often, stay simple, involve users, embrace change
RUP	4 phases (Inception→Transition) — use-case, architecture based
Scrum	Team framework with roles, artifacts, events (sprints)
XP	Code-focused Agile method with pair programming, TDD, CI

Concept	Essence
Agile Challenges	Contracts & maintenance don't fit well
Process Choice	Adapt model to project context

Would you like me to turn this into a **formatted, exam-ready PDF summary (with diagrams for Scrum & XP)**?

🗀 Fichier chargé

Please explain each concept covered in these slides do not miss details and be simple and efficient please

Lecture des documents interrompue

Perfect 👍 You uploaded **"Software Engineering – Lecture 4: Requirement Engineering"** by **Okba Tibermacine (Oct 22, 2025)**.
Here's a **clear, detailed, and simple explanation** of every concept from start to finish — nothing skipped 👇

🧩 1. What is Requirement Engineering (RE)?

Definition of a Requirement

- A *requirement* is **something the user or system needs** — a condition, function, or quality that must be satisfied.
  - **User point of view:** a need or problem to solve.
  - **System point of view:** a capability or behavior the system must exhibit.

IEEE definition:

> A software requirement is a condition or capability needed by a user or required by a system to satisfy a standard, contract, or specification.

Definition of Requirement Engineering

> A **systematic process** of discovering, documenting, and managing requirements for a software system, including its constraints.

- RE is **iterative**: gather → analyze → refine → validate.
- It's a **complete subfield** of Software Engineering (taught as its own module).

Objectives of Requirement Engineering

1. **Decide precisely what to build** — define the system's goals clearly.
2. **Document** these requirements formally.
3. Produce requirements that are:
    - **Complete** (no missing info)
    - **Consistent** (no contradictions)
    - **Relevant**
    - **Accurate** (reflect customer needs)

Why RE is Extremely Important

1. **Comprehension:** Users often don't know exactly what they want.
2. **Communication:** Informal talk leads to misunderstandings.
3. **Protection:** Clear requirements prevent costly scope changes.
4. **Experience:** 40–60% of all software defects come from poor requirements.

💡 *Bad RE = failed projects.*

🧠 2. Types of Requirements

Functional Requirements (FR)

- Define **what** the system does — behaviors or functions.
- Usually start with "The system shall..."

📱 *Example (Quick Cup App):*

- Browse coffee menu & prices
- Add/remove drinks from cart
- Place order and receive confirmation
- Barista updates order status
- Future: apply "Happy Hour" discounts

Non-Functional Requirements (NFR)

- Define **how** the system performs or operates.
- They describe **quality attributes** and **constraints** (speed, security, usability...).

📱 *Example (Quick Cup App):*

- **Usability:** users can order within 3 clicks.
- **Performance:** load under 2 seconds.
- **Reliability:** no lost/duplicate orders.
- **Maintainability:** easy to add new drinks.
- **Security:** data protected from unauthorized access.
- **Availability:** app up during shop hours.
- **Scalability:** handle more users during rush hours.

Constraints (subset of NFRs)

Define restrictions like:

- "Must run on Windows, Linux, and iOS."
- "Use PostgreSQL as database."

Why NFRs Are Difficult

- Hard to **measure or verify**.
- Example: "System must be easy to use" → vague!

  ✅ Instead, make it **verifiable**:

  > "Medical staff shall use all functions after 4 hours of training, with ⩽2 errors/hour."

Common NFR Categories & Measures

Category	Example Metric
Speed	Response < 2 sec
Ease of Use	⩽ 4h training

Category	Example Metric
Portability	Runs on 3 OSs
Reliability	99.9% uptime
Scalability	1000 users concurrently
Robustness	Recover from crash < 1 min
Security	Encryption, access control

System vs Software vs Hardware Requirements

- **Software reqs:** functions for software parts.
- **System reqs:** include both software + hardware.
- **Hardware reqs:** physical components needed.

💡 For embedded systems → RE covers both hardware & software.

Abstract vs Detailed Requirements

- **Abstract:** high-level, for bids (e.g., "system shall track sales").
- **Detailed:** written after the contract, describing exactly what and how.

⚙️ 3. Requirements Engineering Process (REP)

A structured set of **activities** for developing good requirements.

The 4 Main Activities:

1. **Elicitation** → gather needs.
2. **Analysis** → detect and resolve problems.
3. **Documentation** → formally record them (SRS).
4. **Validation** → check correctness and completeness.

🪄 Activity 1: Requirements Elicitation

Definition

Discovering and understanding what stakeholders really need.

Stakeholders

Anyone with an interest in the system:

- **Internal:** developers, testers, project managers
- **External:** clients, users, suppliers, government

Sources of Requirements

- Goals/problems
- Domain knowledge
- Stakeholders
- Organizational context
- Operational environment

How to Elicit Requirements (Properly)

Don't just ask random questions — plan it.

Steps:

1. Understand the **domain** and **problem** first.
2. Choose the right **elicitation techniques**.
3. Ask structured, prepared questions.

Techniques for Gathering Requirements

Technique	Description
Interviews	Ask stakeholders directly (structured/unstructured).
Scenarios & Use Cases	Describe how users interact with system.

Technique	Description
Prototyping	Build mock-ups to clarify needs.
Observation (Ethnography)	Watch users in their real context.
JAD Workshops	Developers + users collaborate together.
Questionnaires	For large user groups.
Document Analysis	Study existing systems, manuals, policies.

Common Elicitation Problems

1. Stakeholders don't know what they want.
2. Stakeholders use ambiguous terms ("fast", "good").
3. Conflicting requirements (security vs simplicity).
4. Resistance to change ("Excel works fine").
5. Unnecessary preferences ("make button blue").
6. Political influences (hidden agendas).
7. Requirements change mid-project.
8. **Implicit assumptions** — people assume things are "obvious."

💡 *Lesson:* Never assume anything. Make everything explicit and verified.

Questions to Ask During Elicitation

- Why do you need this software?
- Who are the users?
- Where will it be used?
- What tasks should it perform?
- What defines success?
- Can we describe a use scenario?

🔍 Activity 2: Requirements Analysis

Goal: Detect and fix problems in raw requirements.

Common Issues Found

- Missing requirements
- Duplicates / overlapping
- Ambiguities (“user-friendly”)
- Unrealistic goals
- Conflicts between stakeholders
- Inconsistencies (contradictions in documents)

🧠 *Solution:* Review & negotiate with stakeholders to clarify and reconcile issues.

📝 Activity 3: Requirements Documentation

Formal recording of requirements into a clear, complete document.

Software Requirements Specification (SRS)

- Central document describing what the system will do.
- Used by: developers, testers, managers, clients.
- Must be: **Correct, Complete, Consistent, Verifiable, Understandable, Modifiable, Traceable.**

Standards for Writing SRS

- IEEE 830-1998 → most common format.
- IEEE 1233-1998 → system requirements guide.
- IEEE 1362-1998 → operational concept guide.
- Volere Template → modern, human-centered version.

🧾 SRS Typical Structure (IEEE 830)

1. Introduction (purpose, scope, definitions)
2. Overall description (context, constraints)
3. Functional requirements
4. Non-functional requirements
5. Interface requirements
6. Appendices, glossary, etc.

✅ Activity 4: Requirements Validation

Goal: Confirm that documented requirements are correct and accepted by stakeholders.

Check for:

- Completeness & consistency
- Conflicts or ambiguities
- Conformance to standards
- Technical feasibility

Usually done in **review meetings**, iteratively.

🧭 4. Recommended Principles of RE

#	Principle	Explanation
1. Value Orientation	Focus on solving real problems, not just listing features.	
2. Involve Stakeholders Early	Communicate and gather input from all roles and levels.	
3. Write Clearly (SMART)	Requirements must be Specific, Measurable, Achievable, Relevant, Time-bound.	
4. Context Matters	Understand environment & constraints (business, users, devices).	
5. Traceability	Link requirements to design, code, and tests to track changes.	

#	Principle	Explanation
6. Adapt to Process Model	For Waterfall → complete before dev; for Agile → evolve continuously.	
7. NFRs Matter	Don't ignore non-functional requirements; they're critical.	
8. Acceptance Criteria	Define when a requirement is considered "done."	
9. Problem–Requirement–Solution Cycle	Understand the problem before jumping to solutions.	
10. Use Visuals	Diagrams, mockups, and flowcharts aid understanding.	
11. Prioritize	Use MoSCoW (Must, Should, Could, Won't) based on value, cost, and risk.	

🧱 5. User Stories (Agile Requirements)

Definition:
Short, simple descriptions of a feature told from the user's perspective.

Format:

> As a [type of user], I want to [do something] so that [I get a benefit].

📚 *Example (Library System):*
"As a student, I want to search for books so that I can find resources quickly."

INVEST Principles

- **I**ndependent
- **N**egotiable
- **V**aluable
- **E**stimable
- **S**mall
- **T**estable

User Stories vs Functional Requirements

- User stories describe **user goals** in simple language.
- FRs are **formal** and detailed specifications.

🧪 Acceptance Criteria (AC)

Define when a specific user story is complete and acceptable.

📘 *Example:*
Story: “As a customer, I want to reset my password.”
AC:

- Receive reset email.
- Link expires after 24h.
- Password must have ⩾8 chars and 1 symbol.
- Process completes under 3 sec.

🧩 Definition of Done (DoD)

Defines quality standards for *any* backlog item.

🧠 *Example:*

- Code reviewed & merged.
- Unit tests (⩾80% coverage).
- Deployed to staging.
- Docs updated.

✅ *If all DoD items met → ready for release.*

⚙️ Acceptance Tests (AT)

Executable tests verifying that AC are met.
Often automated using BDD syntax (*Given / When / Then*).

🧮 6. Use Cases

- More detailed than user stories.
- Describe a **sequence of interactions** between an actor and the system.
- Include **main flow** and **extensions (exceptions).**

📍 *Example (POS System)*

1. Cashier starts a new sale.
2. Enters item → system shows price.
3. Customer pays.
4. System logs sale and prints receipt.
   Extensions: paying by cash, by card, system errors, etc.

✅ Summary

Concept	Key Idea
Requirement	A needed function or property
Requirement Engineering	Process to gather, analyze, and manage requirements
Functional Requirement	What the system does
Non-Functional Requirement	How the system behaves
RE Process	Elicitation → Analysis → Documentation → Validation
SRS	Formal document of all requirements

Concept	Key Idea
RE Principles	Clarity, traceability, prioritization, stakeholder focus
User Story	Short user-centered feature statement
AC & DoD	Define when a story is acceptable or done
Use Case	Detailed step-by-step user-system interaction

Would you like me to create a **well-formatted summary PDF** (with tables, visuals, and diagrams for the RE process, SRS, and User Story flow)? It'll be perfect for exam revision.

🗀 Fichier chargé

Please explain each concept covered in these slides do not miss details and be simple and efficient please

Lecture des documents interrompue

Perfect 👍 You uploaded **Software Engineering – Lecture 5: UML Part 1 (Use Cases, Class, and Sequence Diagrams)** by *Okba Tibermacine*.
Here's a **complete, clear, and simplified explanation** of *every concept in these slides* — nothing skipped 👇

🧩 1. Motivation

There is a **gap between requirements and code**:

- **Requirements** → define *what the system should do*.
- **Code** → defines *how it will do it*.

🧠 **Software design** bridges this gap.
It transforms user needs into **visual models** and **blueprints** for implementation.

🏗️ 2. Software Design

Definition:

Software Design is the **creative and iterative process** of translating requirements into an implementable structure.

Goals:

- Transform "**what**" (requirements) → "**how**" (design).
- Produce a solution that can be implemented in code.
- Ensure each design element is **traceable** to the SRS (Software Requirement Specification).

Types of Design Artifacts:

Type	Focus	Target Audience
Conceptual Design (What)	Describes what system does	Customers
Technical Design (How)	Describes structure and behavior	Developers

Examples:

- **Conceptual:** UI mockups, Use Case Diagrams
- **Technical:** Class Diagrams, Sequence Diagrams, Architectural Models

---

🧱 3. Why Software Modeling?

Even with clear requirements, the software is too complex to imagine mentally.

Models help to:

- Visualize and clarify the system.
- Specify its structure and behavior.
- Document design decisions.

- Improve communication among developers and stakeholders.

📈 **Model = Simplified representation** of a real or planned system.

📘 4. Software Models

Goal: **Bridge the gap** between requirements and code.

Types:

1. **Formal Models:**
    - Mathematical or logical notation.
    - Used in safety-critical systems (rarely in this course).
2. **Graphical Models:**
    - Diagrams to visualize the system.
    - UML is the most popular.

🧭 5. UML (Unified Modeling Language)

- Developed in **1995** to unify various modeling techniques.
- A **standard visual language** for designing and documenting software.

UML allows you to:

- **Visualize** the system.
- **Specify** structure and behavior.
- **Construct** the system (as a guide for coding).
- **Document** it clearly.

It can be used for both:

- **Software systems**, and
- **Business processes.**

📋 6. Main Uses of UML

Use	Description
As a Sketch	Informal, quick design to discuss ideas (Agile-friendly).

Use	Description
As a Blueprint	Complete, detailed plan for implementation (traditional development).

UML as a Sketch

- Used in Agile to visualize parts of design or code.
- Focus: **Understanding**, not perfection.
- Helps both in:
  - **Forward engineering** → from design to code.
  - **Reverse engineering** → from existing code to diagrams.

🧩 7. UML Diagrams Overview

UML diagrams are of two kinds:

Type	Focus	Examples
Static Diagrams	System structure (components & relations)	Class, Object, Component
Dynamic Diagrams	System behavior at runtime	Use Case, Sequence, Activity

📘 This lecture covers:

- **Use Case Diagrams**
- **Class Diagrams**
- (and later) **Sequence Diagrams**

🎭 8. Use Case Diagrams

Definition

A **Use Case** describes a **set of actions** the system performs to deliver something of value to an **actor** (user or external system).

Actor: anything that interacts with the system — person, device, process, or another system.

> 📖 "A use case specifies system behavior and its sequence of actions that yield a result of value to an actor." — *Booch, 1999*

Purpose

- Capture the **intended behavior** (the *what*) of the system.
- Early analysis tool — defines customer requirements.
- Does **not** describe implementation (the *how*).

Elements of a Use Case Diagram

Element	Description
System	Represented as a rectangle; defines system boundary.
Actors	Stick figures — entities interacting with the system.
Use Cases	Ovals — functions or actions provided by the system.
Relationships	Lines/arrows showing how actors and use cases relate.

1️⃣ System Boundary

- Shown as a **rectangle** with the system's name.
- Everything inside → system behavior.
- Everything outside → environment.

2️⃣ Actors

- Stick figures representing users or external entities.

Two types:

Type	Role	Placement
Primary Actor	Initiates use case	Left
Secondary Actor	Reacts to system (e.g., notification, API)	Right

---

3️⃣ Use Case

- **Oval shape** with a descriptive name starting with a verb (e.g., *Login, Place Order, Pay Bill*).
- Each represents a high-level system function.

4️⃣ Relationships Between Use Cases

(a) Association

- Solid line connecting actor ↔ use case.
- Means “actor uses or communicates with system.”

(b) Generalization

- Shows inheritance among actors or use cases.
- Arrow points from **child → parent** (specialized → general).
  - Example: “Administrator” generalizes “User.”

(c) Include

- Dotted arrow labeled `<<include>>`.
- Reused functionality shared by multiple use cases.
  - Example: “Login” included in “Place Order” and “View Profile”.

(d) Extend

- Dotted arrow labeled `<<extend>>`.
- Represents *optional or conditional* behavior.
  - Example: “Apply Discount” extends “Checkout”.

✅ Use Case Design Tips

- Each use case = meaningful system usage.
- Use clear nouns & verbs (helps derive classes later).
- Factor out repeated logic:
  - `<<include>>` for mandatory shared parts.
  - `<<extend>>` for optional additions.
- Keep abstraction levels consistent.
- Organize large diagrams into **packages**.

🧱 9. Class Diagrams

Purpose

Show the **static structure** of a system:

- Classes (types of objects)
- Their attributes and operations
- Relationships among them

📘 Used for **technical design** and **object-oriented modeling**.

Elements

Element	Meaning
Class Box	Represents a type of object. Contains name, attributes, and operations.
Relationships	Links between classes — association, inheritance, composition, etc.

Class Structure

A **class** is drawn as a **rectangle with up to 3 compartments**:

```pgsql
```

```
+---------------------+
| ClassName           |
+---------------------+
| attributeName: Type   |
+---------------------+
| operationName() : Type|
+---------------------+
```

Attributes (Properties)

- Represent data stored in the class.
- In programming → *instance variables.*

Syntax:

`visibility name : type = defaultValue`

Visibility symbols:

Symbol	Meaning	Access
+	Public	Accessible from anywhere
-	Private	Only within class
#	Protected	Within class & subclasses
~	Package	Classes in same package

Special notations:

- Underlined → static (shared among all instances).
- `/attribute` → derived or computed value (not stored).
  - Example: `/age` computed from birthDate.

Operations (Methods)

Describe actions the class can perform.

Syntax:

`visibility name(parameterList) : returnType`

Example:

`+calculateTotal(price, tax): float`

🧠 UML excludes simple "getters/setters" to keep diagrams concise.

Interface

- Represents a **contract** — defines *what methods* a class must implement.
- Labeled with `<<interface>>`.
- A class connects to its interface with a *realization arrow* (dashed line + hollow triangle).

Finding Classes from Requirements

Steps:

1. Read the problem statement carefully.
2. Identify **nouns → candidate classes**.
3. Eliminate irrelevant nouns (UI elements, verbs, vague words).
4. Refine attributes vs classes.

📘 Example:

> "A customer places an order that contains items. The system generates an invoice."

✅ Classes: `Customer`, `Order`, `Item`, `Invoice`.

Relationships in Class Diagrams

1️⃣ **Association**

- Basic relationship ("connected to").
- Shown as a solid line.
- May have **multiplicity** (how many objects link).

Multiplicity Examples:

Symbol	Meaning
1	exactly one
0..1	optional
*	many
1..*	one or more

Example:

- `Customer 1..* ---- 1 Order`

2️⃣ Association Class

- Used when a relationship itself has **attributes**.

  Example:

  A `Student` and `School` have an association `Enrollment` (with attributes like `year`, `status`).

3️⃣ Aggregation

- "Has-a" relationship (logical part).
- Represented with an **empty diamond**.
- Whole can exist without parts.

Example: `Library ◇– Book`

4️⃣ Composition

- Strong "part-of" relationship (physical containment).
- **Filled diamond**.
- When the whole is deleted, parts are deleted too.

Example: `House ◆– Room`

5️⃣ Inheritance (Generalization)

- Shows that one class inherits attributes/methods from another.

- Arrow: solid line with **hollow triangle** from subclass → superclass.

Example:
`SavingsAccount —▷ BankAccount`

6️⃣ **Dependency**

- A weak link showing one class **depends** on another.
- Dashed arrow.
- Example: `BankAccount → Database`

🧠 10. Interpreting UML in Code

UML Symbol	Code Meaning
`+`	`public`
`-`	`private`
`#`	`protected`
Association	Reference variable
Multiplicity	Arrays or Lists
Inheritance	`extends` or `implements`
Dependency	Local variable or method call

🧩 11. Example (Summary Exercise)

Example 1: Bank System

- **Customer** ↔ **BankAccount** (1..* relationship)
- **BankAccount** subclasses:
  - `SavingsAccount`
  - `SalaryAccount`

- `BankAccount` depends on `Database`.

Example 2: Order System

- **Order** ↔ **OrderItem** (1 ↔ many)
- **Student** has attributes `name, ID, course`
- Methods: `getCourse()`, `cancelEnrollment()`

✅ Summary

Concept	Key Idea
Software Design	Transforms "what" into "how"
UML	Universal modeling language for design
Use Case Diagram	Describes system behavior (actors + actions)
Include/Extend	Reuse or optional relationships
Class Diagram	Defines structure (classes, attributes, methods)
Associations	Links between classes
Aggregation/Composition	"Has-a" / "part-of" relationships
Inheritance	"Is-a" hierarchy
Dependency	Weak relationship (uses another class)

Would you like me to continue with **Lecture 6 (UML part 2: Sequence + Activity Diagrams)** in the same simplified and detailed style next?