Freie Universität Berlin

Department of Mathematics and Computer Science

Institut für Informatik

Bachelor in Computer Science

Software Engineering

Academic Year 2024/2025

Author:
Miriam Gaetano

Professor:
Prof. Lutz Prechelt

Contents

1 Introduction

Software is a set of programs or data, and accompanying documentation necessary or useful for their use. More technically (IEEE 1983), they are computer programs, procedures, rules, and documentation related to the operation of an information system. **It is not just code.**

Software engineering is the discipline that deals with systematically providing and using methods and tools for the production and application of software. It uses science and technology to obtain measurable results. **The task of software engineering is to analyze costs and benefits, understand what the system must do (requirements), how it should be structured (architecture), actually build the software, validate it by verifying that it works, and manage the entire project by coordinating people, time, and activities.** A software development is generally considered successful if (it's impossible to achieve them all):

- costs are low,
- the time to completion is short,
- the software quality is high,
- the software is large and has many features.

If the software is poor, it harms everyone, the client loses money, the user stops using it, and the team loses motivation.

Software engineering teaches us that knowledge is built over time and everything we have learned will serve us to reuse or at least be more aware of all the risks and problems we may encounter.

The two worlds

- **Problem world** → product, requirements, design, and process
- **Solution world** → automation techniques, reuse, and methodological aspects of requirements, design, quality, and management

1.1 Tacoma Narrows

A very important case analyzed is that of the **Tacoma Narrows**, a suspension bridge that had to cover a very large distance between two points. The chief engineer and the external expert of this project had already handled other very important projects like the Golden Gate. Although the design parameters such as pylons, suspension cables, materials, etc. were standard, in this project exceptions were not properly considered.

The financiers wanted a light bridge, so Moisseiff designed the longest bridge ever built at the time, which was extremely slender (very narrow and low). External engineers criticized the project because they considered the bridge too light and flexible, but Moisseiff, through his mathematical models, argued that the structure he designed would withstand winds of over 140 km/h. Already during construction, it oscillated greatly and in 1940 it collapsed, with a wind of just 75 km/h.

The problem was not the wind itself, but the longitudinal vibrations that **were not foreseen**, because since in other projects the bridges were much heavier and stiffer, this unanticipated factor did not cause issues. The cause was therefore **lack of experience in that type of situation.** They pushed too far based on previous experience. This type of procedure is called a **Radical procedure**, that is, when you can't base your project on previous experiences but **you have to invent something from scratch.**

Although one might think that building from scratch is always the best solution, it is not, because by testing new ground you risk wasting resources and falling into errors that may have already been spotted in very similar projects (similar requirements...).

1.2 eGK

The eGK is the electronic health card. It is a system for digitally managing patient health data and is an excellent example for understanding the various errors that can arise in a software engineering project.

The project doesn't just involve the electronic prescription but many other useful functions like medical letters, electronic health records, emergency data, organ donor cards... We will focus, however, on the **Prescriptions** where the doctor can create the prescription, the pharmacist can redeem it, and the patient can delete or hide the prescription.

The system may seem simple: read and write a prescription. But in reality, it is much more complex because we must take into account already existing environments such as healthcare, legal regulations, etc., in addition to designing it so that it can be continuously updated and adapted to an international scenario. It must therefore comply with medical classification standards, terminology, format, security... It cannot be arbitrary. **So the eGK software does not start from scratch but must be able to integrate all those already existing standards.**

One might think of writing prescriptions like this:

```
class ePrescription {
  String doctor;
  String patient;
  String drug;
  int dose;
  String instructions;
}
```

But we forget that:

- each object (doctor, patient, drug...) has **many attributes** (e.g. code, certification, address, ID, language, permissions...)
- interactions **are not direct** (e.g. authentication, security, digital signature...)
- there are **dozens of classes** connected together
- there are specific rules for each **subject, role, situation**

1.3 Basic Concepts

The following terms belong to the basic vocabulary of (object-oriented) programming and should already be familiar to you. They will be used regularly so it's important to have a clear understanding.

A **class** is a general abstract entity related to the problem to be solved such as Student, Exam, etc. It defines the common characteristics and behaviors of all instances of that class through **attributes** (which describe the entity's characteristics like for Student: name, surname, ID, etc.) and **methods** (are the actions the entity can perform, e.g. a student can enroll in lessons so will have a method like "enrollInLesson", etc.).

The **instance** or **object**, instead, is nothing but a concrete exemplar of a class. Once a class is defined, you can create multiple instances of that object that will have their own specific values for each attribute. For example, "Mario" can be an instance of the Student class and will have attributes like name=Mario, surname=Rossi, etc.

Inheritance is another key concept related to object-oriented programming involving multiple classes. We will have a "parent" class and a "child" subclass that inherits the (non-private) methods and attributes of the parent class. These methods can also be rewritten using @override keeping the same signature (same name and parameters), or using overload where the name remains the same but the parameters change.

Fortunately, in every language, there are **libraries**, i.e. a set of classes, functions, or methods already prepared that we can use by simply implementing the library into our code instead of writing it all from scratch.

Methods and functions start from a **specification**, i.e. the objective of that function, the purpose, and are then **implemented** (the code is written). To **verify** that these functions and the rest of the program work correctly, we can insert various checks, tests, etc., into the code.

2 Requirements

A requirement is a condition or capability needed by a user or system to solve a problem or achieve a goal. **It is something the system must do** and it must be expressed clearly, comprehensibly, and verifiably.

If the requirements are wrong, the system will be useless, even if it works perfectly.

If you can't describe it clearly, then the requirement is not clear.

2.1 Functional vs Non-functional requirements

- **Functional requirements** → describe **What the system does**
  → use cases + tables + logic
  "The doctor can create a prescription"
- **Non-functional requirements** → describe **How the system should behave**
  → qualitative scenarios
  "The system must be operational 99.99% of the time"

2.2 How requirements are gathered

Phase	What is done	Main techniques
1. **Elicitation**	**Needs are gathered** from stakeholders and users	Interviews, observations, scenarios, workshops, etc.
2. **Analysis**	**Understand, structure, clarify** what has been gathered	Card sorting, models, goal hierarchies, conflicts, priorities
3. **Specification**	**Write requirements clearly** and precisely	Textual use cases, formal requirements, UML diagrams
4. **Validation**	**Check with stakeholders** if everything is correct and complete	Prototypes, walkthroughs, simulations, checklists, acceptance tests

Every requirement analysis must answer 4 fundamental questions in the documentation: **Who** is interested in the system, **Why** do they want it? **What** must it do? Which **constraints** must we respect?

Requirements are NOT just a list of available functions. There are many issues surrounding requirements gathering, because **sometimes they can be ambiguous, hidden, taken for granted, or we may meet clients who don't know what they want or keep changing their minds.** They must be clear, verifiable, relevant, and traceable.

To avoid any kind of problems, **it is necessary to bring clarity**, through interviews, questionnaires, prototypes, usage scenarios (narratively telling how the user interacts with the system by observing it), and documentation. **Multiple techniques are used together.**

Example: "The system must be intuitive and easy to use" is NOT a good requirement, too vague.
"85% of users must be able to complete the operation in less than 1 minute" is better.

2.3 Techniques to discover requirements

- **Introspection** The software engineer **reflects alone on what might be needed**. Although it is a very simple practice and often used unconsciously, **it risks projecting our thoughts onto the users.**
- **Participant observation** The observer actively participates in the reality to be studied. **They spend real time with users learning from within how their work operates**. It captures invisible details, but requires a lot of time and a focused objective.
- **Interviews** Semi-structured interviews with **open-ended questions** can gather much qualitative information, opinions, and motivations, but are hard to analyze and do not capture automatic behaviors or habits.
- **Questionnaires Sending written questions to many people.** It's cheap but risks losing context, and often results in ambiguous and too simple answers. It doesn't explain the reasoning behind answers.
- **Group elicitation techniques** Like **focus groups**. You get greater interaction between people and consequently more viewpoints emerge, but an expert moderator is needed.
- **Using user feedback** Through forums, interviews, real usage data. Represents the real world but there is the risk of misinterpreting requests, and possibly encountering inexperienced clients who give useless suggestions.

- **Iterative development (Agile)** A simple first version is built, tested with the user, learned from and adapted. Then repeated.
- **Card sorting** Concepts are written on cards, then the expert groups them and explains. Great for understanding the expert's mental structures and extracting knowledge. It does not show how a system actually works though.
- **Goal hierarchies Represent why the system is being built.** Starting from the main goals and drilling down. Can become complex, vague, and superficial, but ideal for projects with many diverse interests.
- **Scenarios** A sequence of actions representing a **single interaction between one or more actors and the system.** It is a specific and linear example of behavior, without variants or error handling, focusing on what happens **from outside the system**, without describing internal logic. They can be positive (desired behavior) or negative (what must not happen). Useful because users understand them easily and they form the basis of UML use cases.

2.4 How to represent discovered functional requirements

After collecting the requirements with various techniques, we can use just as many to represent them, chosen based on our needs.
Each method has a different role. We have detailed scenarios, UML diagrams, tables, glossaries, informal and formal notations, as well as use cases.

2.4.1 Use Cases

It is not a functional requirement but a method to describe them. A use case is a **textual description of action sequences (multiple scenarios)** that the system performs, i.e., the expected **behavior** of a system following interaction with a user or another external actor.
We are interested in **WHAT it does, not how.** It's a functional, not technical vision.

It is used to describe and **verify the understanding of the requirements expected by the user** for initial analysis, which is why it must be simple and understandable.
It then validates the architecture and checks the system.

It includes actors, conditions, objectives and multiple scenarios with possible extensions (alternatives and errors that represent other paths within the same use case).

2.4.2 Principles for writing good use cases

1. **"product vision":** What makes a use case excellent at **the end of its realization**
   - clarify the actor's goals and **do not mention how the system works internally** ("Make a payment" not "Update transaction database")
   - verify that the sequence of scenarios is correct, inserting only what happens directly between system and actor without adding technical details
   - verify that the flow is smooth and manage alternatives well
2. **"process vision":** How to actually write a good use case?
   - first define a main actor and their goal using simple, non-technical language and only then proceed to the main scenario etc.
   - avoid technical structures or pseudo-code because **phrases must also be understood by non-technical people like the client.**
   - consider possible variations and errors
   - structure everything from the user's point of view on how they interact with the system to avoid unnecessary steps for communication

2.4.3 How to write a use case

1. **Main actor + goal** → Clarifies who does what, and why
2. **Main scenario** → What happens if all goes well
3. **Alternative scenarios** → Variants, problems, exceptions
4. **Expansions** → How those variants are handled

2.4.4 UML Use Case Diagram

A use case diagram is a **graphical representation of the various relationships between actors and use cases**, without describing interaction details.
Whereas the use case is the text that describes in detail the interaction between **actor and system**.

Advantages: being a graphic, it is easier to understand various functionalities and the actors involved.
Disadvantages: showing only actors, cases and relationships, it does not describe the system's behavior. Thus, if not accompanied by textual use cases, it is insufficient for requirements analysis.

2.4.5 Elements of a Use Case Diagram

- **Actor** An external entity to the system that interacts with it to achieve a goal. It is not necessarily a person but a **role that someone or something plays in relation to the system**. In diagrams it is represented with a stick figure.

  So an actor must have an active role and interest in the goal. The actor's name **identifies a functional role, not a specific person**. For example, if a customer only receives an automated email, does nothing with the system, they are not an actor.
  Example:
  Use Case: "Purchase product" → *Main actor:* buyer (goal: to purchase) *Use Case:* "Register data in a management system" → *Main actor:* employee

- **System Boundary** → a rectangle enclosing all the system's use cases to distinguish them from what is external to the system.

- `<<include>> relationship` → indicates that one use case **always contains** another use case as part of its main flow. It is similar to a function call in code.
  Example: "place order" includes → "select payment method".

- `<<extend>> relationship` → shows variants and scenarios outside the main flow. Like optional or conditional features followed **only in certain cases**.
  Example: `View order history <<extend>> View order details`

  (careful: the arrow is reversed on purpose) Here "view order history" is only accessed if the actor wants to see the history.

- **Generalization** → represents inheritance between actors or use cases, used to specialize roles or functionalities.

3 Dynamic Modeling in UML

If with use cases we had an external view of the system, **with dynamic modeling we move to an internal view, not just from the user's point of view.** It helps us describe and specify the expected behavior. It's not enough to know what the system does, but we must also know **when it does it, and in what order.**

1. Start by understanding the external behavior (**use cases**),
2. then understand the structure of the system through **object modeling** with class diagrams,
3. and finally understand its behavior through **dynamic modeling** by creating sequence or state diagrams that identify senders and receivers and allow us to show the sequence of messages exchanged between objects.

3.1 Sequence diagram

Communication between objects. It shows who sends messages to whom and in what order. **It is used to describe a specific interaction between objects in a specific use case.**
We already know it so we won't go over it again. There's a stick figure, vertical columns that represent the interaction duration, and arrows connecting the various classes back and forth.

3.2 Other main types of diagrams

They are all used but for different purposes.
"State-based" approach according to which:
Each entity (object) can have:

- **Internal states** → such as "on", "off", "active", "error"
  A **stable condition**, on pause until an event occurs that moves it, such as a user pressing something, or a timer expiring...
- **Transitions** between states, caused by **events**
- **Actions** are activities performed **during a transition** or when entering/leaving a state.

Example – Washing machine:
States: off, power-on, washing, spinning, finished
Events: button pressed, timer expired, water reached, door opened
Transitions: link states → "from A to B if X happens"

This logic is modeled with UML state diagrams.

3.3 State diagram

Derived from "state", it represents the life cycle of an object. For example "Awaiting payment".
It shows **how a single object changes over time** (its life cycle) based on states, events, transitions, and actions.
It is applied to objects with complex life cycles (e.g. orders, devices, sessions).

The state is represented by rectangles with rounded corners.
Inside, various actions are written.
The initial state is shown by a solid black dot, while the final state is a black dot with a border.
The diamond shape represents a decision or merge.
Transitions are shown by arrows labeled like this:

```
event [condition] / action
```

Example: `payment received [total >= 0] / update status → [Paid]`
Where "payment received" is the event, the total is the (optional) condition, and "update status" is the action.

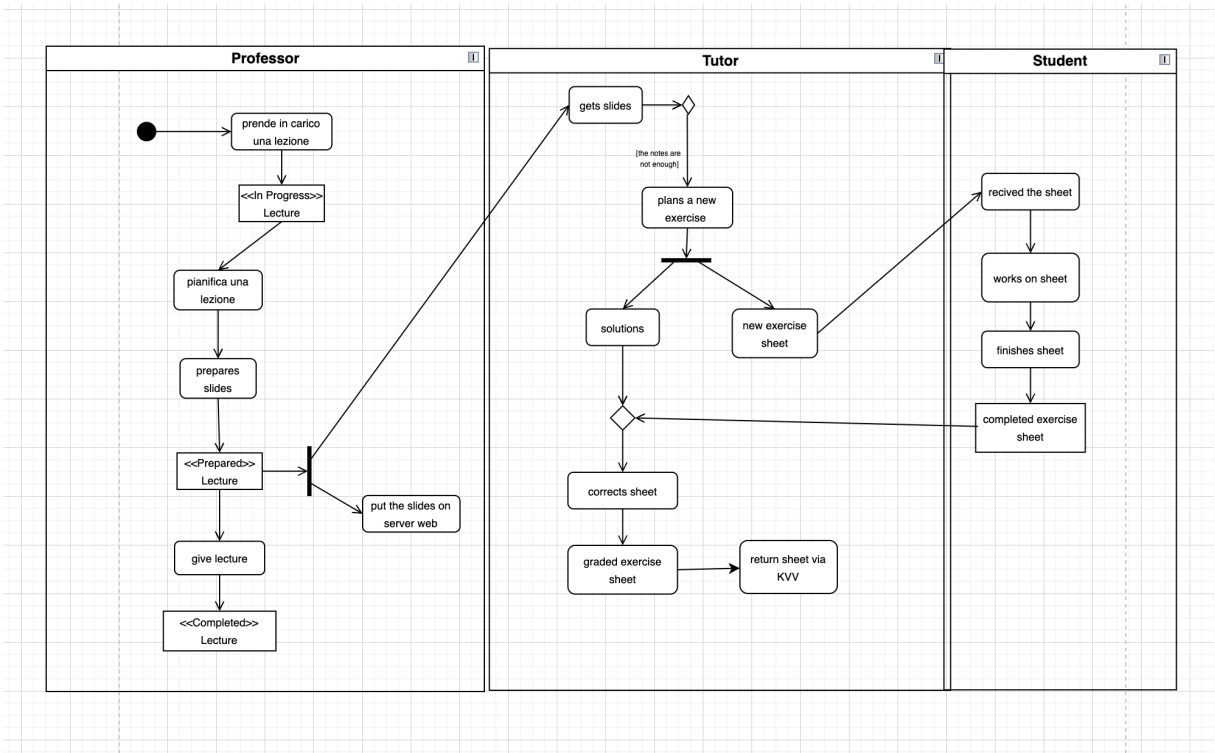
3.3.1 User Interface Modeling

State diagrams are reused to design user interfaces, especially to **represent the navigation path or screen flow in the system.**
Each state corresponds to a screen, and transitions represent user actions like clicking a button, choosing from a menu, or moving the cursor.
The name of the state represents the name of the screen and the items listed under the state are the available actions.

3.4 Activity diagram

Represents a logical sequence of actions. For example "check availability", "calculate price"...
They use the same style as state diagrams, but instead of representing internal states of an object, **they show operational control, that is, the sequence of actions/activities.**
Here, there is no need to explicitly represent join/split symbols because they are implicit in the flow.

- When **two activities** start from the **same arrow**, it means they are parallel (split)
- When **two arrows merge into one**, it means they must **both complete before proceeding** (join)



4 OCL - Object Constraint Language

Used to **express constraints** (logical rules) **within UML models**, defining preconditions, postconditions, and invariants. Utilized during the design phase.
For example, in UML we can always express rules between elements only via arrows, whereas thanks to OCL we can also represent conditions like: "*A tournament cannot have overlapping matches*" or "*A player must be registered before participating*"...

Useful not only to gain a clearer idea of our system, but also to find errors already present in UML models. In Java, it's possible to simulate OCL behavior with Javadoc (with tags like @pre, @post) or with standard assert statements.

4.1 Rules

Invariant An invariant is a condition that **must always be true** for an object of a class, **after each public operation**.

Example: every tournament must have a maximum number of players greater than zero

```
context Tournament inv:
maxNumPlayers > 0
```

Precondition A precondition is a **rule that must be true BEFORE executing a method**.

Example: before accepting a player, check that he has not already been accepted

```
context Tournament::acceptPlayer(p) pre:
not isPlayerAccepted(p)
```

Postcondition A postcondition is a rule that **must be true AFTER** a method has been executed.

Example: After accepting the player, they must **appear as accepted** in the system.

```
context Tournament::acceptPlayer(p) post:
isPlayerAccepted(p)
```

4.2 Operators and Quantifiers

To refer to the value of an attribute BEFORE a method is executed, we use the **@pre operator**, same for others. For example, the number of players before accepting a new one is referred to with `getNumPlayers@pre`. **Used only to compare the value of something BEFORE and AFTER execution.**

The quantifier **forALL** means that **all elements** must meet a condition.
Example: *all matches must start after the tournament starts and finish before the end*, represented as:

```
context Tournament inv::
matches->forAll ( m |
    m.start.after(self.start) and
    m.end.before(self.end) )
```

The quantifier **exists** means that **at least one element** must meet a condition. Example: *at least one match must start on the same day as the tournament*

```
context Tournament inv:
matches->exists ( m | m.start.equals(self.start))
```

Exercises

a) Rewrite the provided constraints in natural language

- `context task inv c1: score>0`

  Every task must always have a score greater than 0

- `context Student inv c2: solution->size() = exam.tasks->size()`

  Every student must have submitted as many solutions as there are tasks in the exam

- ```
  context exam inv c3:
  participant->forAll (t |
      t.passed implies t.solutions->exists (l |
          l.points > 0 and l.task.exam = self
      )
  )
  ```

  For each exam participant, in order to pass the exam, the number of solutions submitted must be ¿0 and score ¿0 and refer to tasks of the same exam, not another one

b) Write correct OCL

- In every exam there is at least one task worth 1 point:

  ```
  context Exam inv:
      self.task->exists ( t | t.point = 1)
  ```

- A post-exam cannot have another post-exam:

  ```
  context Exam inv:
      self.postExam->isEmpty() or
      self.postExam.postExam->isEmpty()
  ```

- After an exam is completed, the status must change from ”active” to ”completed”:

  ```
  context Exam post:
      status = "completed" and status@pre = "active"
  ```
  ```

5 Software Architecture

Designing the architecture means deciding **how to divide the system (into modules) and how to ensure that all its desired qualities emerge** from its overall functioning.
Architecture **means understanding what EACH PART of the system does, how it connects to the others, and what behaviors are expected.**
It serves to satisfy the requirements, especially the non-functional ones.

The principles for good design are good architecture, modularization, reuse, and documentation.

Once requirements are obtained, both functional (what the system must do) and non-functional (which qualities), **it is time to figure out how to build it.**
So we must decide:

1. How to divide the system into modules;
2. How to satisfy the non-functional requirements with an appropriate overall structure.

We must remember that **a system's global properties derive from how the modules interact** with each other. For example, robustness or security are not functions of a single piece of code, but of the entire system.
To address this, we often refer to "standard architectures", meaning solutions that have already been tested for similar problems in the past.
A simple architectural choice example is building a house. If the goal is mobility, we choose a trailer; if it's defense, a castle. If we want both, **we need a compromise.**

5.1 Architectural styles

Relying on standard architectures reduces risks and improves quality. A good architecture is the foundation for building robust, scalable, and maintainable software.

- **Client/Server:** a **central server stores the data, clients access the network to use services.** Suitable for distributed systems, with lots of data and high interactivity. Clients must have network access and networks must be sufficiently fast and available.
- **3-tier/layer architecture: Presentation (GUI)** encapsulates user/system interaction, **Logic (business logic)**, **Data (database)**. Each layer uses only lower layers. This style promotes modularity and organization. **Advantages:** reduces coupling, clear structure, easy to replace entire layers. **Disadvantages:** can be inefficient, unnatural.
- **Event-based:** modules register with a center to receive event notifications. **Disadvantage:** behavior can be hard to see and change. Used in GUIs, IoT, etc.
- **Pipe-and-filter:** data passes through a sequence of transformations (filters). Simple but inflexible. Common in bioinformatics, signal processing.
- **Web architecture:** includes security, internationalization, scalability, distributed development support. Often combines different architectures.

Another important point is that **architectural diagrams are not the architecture**, but only partial representations. Many diagrams (e.g., UML) and textual descriptions (OCL) are needed to properly document an architecture.

5.1.1 Standard Architectures

In general, standard architecture is chosen for a specific application area. These provide principles, technologies, and guidelines for building professional systems.

Examples:

- QUASAR
- Jakarta EE
- RM-ODP

5.2 Modularization

Once the overall architecture is decided and the functional and non-functional requirements are understood, the next step is to decide **how to divide the system into modules**, meaning independent but interconnected components. This division helps to **reduce complexity**, increase **maintainability**, facilitate **parallel development**, and improve **system comprehensibility**.

Each module is a **part of the system** with a clear responsibility. It may contain classes, functions, data, algorithms – but the key is to have **clearly defined boundaries** with other modules.

Each module should be designed with three fundamental concepts:

1. **Interface:** what the module offers to others – functions, methods, or externally visible data.
2. **Contract:** the **rules and conditions** that the user of the module must respect, and that the module guarantees in return.
3. **Secret:** everything that happens **inside the module** and should not concern others (implementation, internal data structures, etc.).

This approach is known as **information hiding**: each module exposes only what's needed and keeps the rest hidden. This allows us to **modify a module** without having to change the others, as long as the interface remains the same.

5.2.1 Criteria for good modularization

The professor lists **four main criteria** for deciding how to divide a system into modules:

1. **Separation of concerns:** each module should **handle only one "concept"** or responsibility.
2. **Locality:** if a change concerns a behavior, it should only affect one module.
3. **Change isolation:** if something changes in the system (e.g., data format), only one module should need to change.
4. **Comprehensibility:** modules should have clear names and responsibilities so that anyone reading the code immediately understands their purpose.

A module is well-designed when:

- it has a precise and limited responsibility;
- a simple and well-documented interface;
- is isolated from unnecessary dependencies;
- can be reused in other contexts.

5.2.2 Practical examples of good modularization

A good example is a module for user management:

- **Interface:** `createUser()`, `authenticate()`, `editProfile()`.
- **Contract:** `createUser()` fails if the email is already in use; `authenticate()` returns a token only if the password is correct.
- **Secret:** how the data is stored (file? database? hashed passwords?) is irrelevant to the outside.

If tomorrow we switch from a relational database to a NoSQL one, the rest of the system does not need to change: we only update the module's secret part.

Modularization and team development

Good modularization is crucial for teamwork. If each group works on an independent module, conflicts are avoided. It's important that interfaces are discussed and fixed at the beginning: **once we decide "what" each module does, teams can work in parallel even without knowing "how" the others will implement their modules.**

Lastly, modularization is not a one-time task. During the entire project life cycle, we must **re-evaluate and improve** the module division, adapting it to changing requirements or emerging complexity.

6 Quality Assurance (QS)

Quality assurance (QS) is the set of all activities to ensure that the software is of high quality, which does not just mean that it "works" but that it is:
reliable, usable, modifiable, robust, and secure

Types of quality

- **External quality** → seen by the user, based on ease of use, reliability, performance, etc. Ensured by functional tests and user experience.
- **Internal quality** → seen by the developer, based on testability, maintainability, comprehensibility, efficiency, etc. Ensured by static and structural tests.

6.1 Verification and Validation

These are two key activities in **Quality Assurance**, they help ensure that the software is of high quality but focus on different aspects:

- **Verification:** Are we building the product in the **right way**? This is done through test cases like **code review (static)** and **dynamic tests** to check that the software works as specified during *development*.
- **Validation:** Are we building the **right product**? This checks, *at the end of development*, if the software **meets the needs and requirements of the final customer or real environment**, through **user acceptance testing**.

6.2 What is a test case?

A test case represents a single situation that is verified to ensure the software behaves as expected. Each test case consists of:

- **Input**: data provided to execute the test
- **Preconditions**: conditions that must be true before execution
- **Expected result/behavior**: what the system is supposed to do
- **Execution context**: the environment or conditions under which the test is run
- **Expectations**: regarding the result

A test case is considered "successful" if the actual behavior matches the expected behavior.

6.3 Approaches to ensure quality

- **Constructive quality testing** focuses on **prevention**, through good documentation, team training, etc.
- **Analytical quality**, which focuses on **control**, aiming to find **existing errors** through tests, inspections, and analytical analysis.

7 Analytical Quality

7.1 Types of tests in the analytical approach

Analytical quality uses two types of tests:

7.1.1 Static – The code is not executed but read manually or by tools.

- **Review/inspection:** A report is compiled, there are authors, a moderator, a reader, and reviewers. The author presents the code, colleagues ask questions, identify issues, and corrections are made.
- **Static analysis:** Tools analyze the source code directly without executing it. Unlike review (which requires human input), this is fully automated using tools like `FindBugs`, `Lint`, etc. These detect syntax errors, empty catches, missing finally blocks, malloc/free errors...

7.1.2 Dynamic – The code is executed.

Its goal is to verify that the software behaves as expected using test cases.

Black box testing – functional test Black-box testing is a technique where **the code is not seen** but we rely only on what the function should do, checking if the system does what the specification says. We test by covering error cases, boundary cases, and **equivalence classes**.

Equivalence classes Equivalence classes **aim to reduce the number of necessary tests by creating sets** of inputs that represent similar behaviors, so we test just one value per group.
It belongs to black-box testing because we test "blindly". Example: if we know values from 1 to 19 are considered F, we just test one number in that range [1,19].

White box testing – structural test White-box testing is the opposite of black-box. Here we **look into the code**, and after understanding how it works, we **test all logical decisions (if, else)** to verify correctness. Useful for catching hidden bugs.

C0: statement coverage → Every statement must be executed at least once **C1: branch coverage** → Every if must be tested both as true and false **Data flow coverage** → Ensures every variable used has been initialized

Other types of dynamic tests

- **Regression test** → After modifications, these verify that nothing broke.
- **Usability test** → Focus on user experience: how easy and pleasant is the software?
- **Acceptance test** → Verifies if the program **meets the user's needs**.
- **Stress test** → Pushes the program **beyond** its capabilities.
- **Load and performance test** → Simulates high usage.
- **Top-down and bottom-up tests** → Order of testing modules: from top/main or from bottom/base.

In any case, both static and dynamic tests are necessary. Used together, they are more effective: **functional tests clarify expected behavior**, while **structural tests find bugs in the code**. **Static tests can detect errors that are not caught by execution.**

7.2 How to write a dynamic test

Start by developing a good **test plan**, a document, checklist or schema that describes everything about the tests: What we're testing (performance, security, etc.), how (test type), and what parts of the system are being tested.

7.2.1 Modular and reusable tests

A good test is not a long block of code, but a small, well-structured, reusable piece.
Avoid test interdependency so that each test can run independently. **Organize tests into classes or files by functionality.**

Test automation

Once the test plan is developed, the goal is to automate execution.
This **saves time**, **avoids human errors**, and **quickly alerts if something breaks** after code changes.
Example:

```
public class CalculatorTest {

  @Test
  public void testAdd() {
    Calculator c = new Calculator();
    int result = c.add(2, 3);
    assertEquals(5, result);
  }
}
```

This is an automated test – it can be run as many times as needed and will fail if the result isn't 5.
But beware: **automated doesn't mean better**.

7.2.2 Frameworks

Used frequently for test automation. A **framework is a support library** that helps write, run, organize tests, etc.
It includes:

- **Assertions** → e.g., `assertEquals(a, b)` checks `a == b`
- **Test runner** → runs all tests
- **Test organizer** → groups tests by functionality
- **Report generator** → shows passed/failed tests and reasons
- **Annotations** like @test, @before, @after (e.g., JUnit in Java)

7.2.3 Test recording and playback

Used to test **graphical interfaces (GUI)** automatically. It records what the user does (clicks, typing...) and replays it later.
Tools:

- Selenium
- TestComplete
- Robot Framework

Limitations: *A button name change can break the test*, execution is slower, and they **cannot replace functional or structural tests** – just complement them for visible parts of the app.

8 Defect, Error, Failure

- **Defect or bug** → *code error*
- **Error** → *invalid program state*, can result from a defect
- **Failure** → *observable wrong behavior* of the system

`Bug → Error State → Visible Failure`

9 Constructive Quality

This is the part of quality that is built **during development**, by designing processes, rules, and work environments well, in order to **prevent** errors rather than look for them afterwards.

Constructive quality works to improve both internal and external quality, and it is divided into **process management** (according to which having a good process — a good way to produce software — leads to a good final product) and **project management** (for a single software project).

Constructive quality **is not achieved through tools alone**, but also through:

- **Training**
- **Company culture**
- **Collaboration**
- **Shared responsibility**

The process must be **monitored and continuously improved**, also thanks to metrics, risk analysis, and effective communication.
There are two approaches to managing constructive quality:

- **Traditional (Waterfall)**
  Guarantees quality through control. Roles are well-defined, there is little room for changes, and documentation is formal. The process follows this order: analysis → design → implementation → testing.
- **Agile (Scrum, XP)**
  Guarantees quality through collaboration. There is continuous communication, self-organizing teams, and accepted changes.

Retrospective:
A meeting held at the end of each sprint to reflect on what went well, what to improve, and what to do in the next cycle.

9.1 Tools for Constructive Quality

Frameworks and models are used to ensure the development process is of high quality.

CMMI (Capability Maturity Model Integration) This is a **5-level model** of software process maturity. It starts from the most "chaotic" level 1 and progresses to level 5 with "continuous optimization".
It teaches us that **quality is not only technical but also organizational**: at level 1, even a good developer will struggle, whereas at level 5, even average teams can achieve great results.

It makes work **predictable**, helps identify areas of improvement, and **improves organization**. However, it must be used wisely because **it can become a bureaucratic machine that kills creativity**.

- Level 1: everything depends on individuals, no documentation or defined process.
- Level 2: we begin setting goals, deadlines, responsibilities.
- Level 3: the first "industrial" level – all teams follow the same development model.
- Level 4: data is collected on time, defects, quality – even allowing for numeric forecasting.
- Level 5: the company continuously analyzes its data and teams collaborate with each other.

ISO 9000 This is the **international standard** for quality management. It is a set of international standards that define how an organization should be structured (not just in software). Focused on documentation, controlled processes, and continuous improvement.
It focuses on organizational rules, not on the product. It says how to organize, not how to code or test.

If a company follows this standard, a third-party certifier can assess its processes and issue an ISO 9001 certification valid for 3 years.
There are yearly audits to maintain the certification.

TQM (Total Quality Management) This is a **management philosophy**, not just an organizational standard, where **to achieve quality, responsibility must be shared by everyone** (not just in software projects, but across the entire organization). Originated in Japan.

Applied to software, quality is seen as code that is readable, well-tested, and documented, **written from the beginning with future developers in mind.** There are no certificates or formal levels, just a philosophical mindset on how to live work.

10 Software Process Models

Process models **describe how to organize and plan all the work and development phases of software**. They are not just theory but **practical strategies** that tell us which phases to follow and in what order:
- when to analyze requirements, when to design, when to code, how to test, and how to manage changes.
THEY ARE THE SKELETON ON WHICH YOU APPLY QUALITY.

10.1 What is a process?

It is a **sequence of structured activities** with defined roles, ordered phases, and clear objectives. It allows you to not skip any steps and to know exactly what you're doing.
Example: given a task, I will follow the order 1. analyze requirements, 2. design the system, 3. implement, 4. test, 5. deliver

It is made of **activities** (designing, testing...), **roles** (developer, tester...), **artifacts** (documents, code, UML, test cases...).

10.2 Models

Each model is analyzed through questions such as: When does it work well? What are its limitations? Is it suitable for small or large projects? Is it good for in-house development, contract work, or product development?

10.2.1 Traditional Models

Waterfall Model It is the most classic and rigid. **You go through the phases in order without ever going back.**
Phases are:

1. Analysis
2. Detailed design
3. Implementation
4. Integration
5. Validation (final test)
6. Release

It's not very functional because each phase communicates only through documents. If no one reads them, it fails. Moreover, personnel changes between phases, so context is lost. If it's not clear from the beginning what must be done, that's a problem. To work, **you must have everything clear from the start and never change**, which is very difficult.

V-Model A more modern version of waterfall, developed by the German government, designed to create a precise, adaptable, and controllable process, especially in public and corporate settings.

Why is it called V-Model? The left side of the V represents **system definition** (analysis, design...)
The tip of the V is the **implementation**
The right side represents **verification and validation**

Each phase on the left has a corresponding test on the right. This **links design and testing**, and it is one of the strengths of the model.
It is very large and detailed, with 32 roles, 89 activities, and 97 artifacts. It is flexible because you can select only the "project modules" suitable for your case. There are 4 mandatory ones:

1. project management
2. quality management
3. configuration management

4. problem/change management

It is product-centered, so each artifact goes through: "planned" → "in progress" → "proposed" → "completed"

So, while it's advantageous for very large and complex projects where every detail must be formalized, for small projects it can be a disadvantage because it's too heavy to manage and requires training to use.

Iterative Models (Spiral...) Designed for *complex and high-risk projects*. In each iteration, you do what is needed most to reduce the main risk at that time. **Each time, the focus is on a different specific risk** and it is addressed with tests, simulations, discussions.

It's called a spiral because each loop is an **iteration following four phases**:

1. **Plan** (determine objectives, alternatives, constraints)
2. **Risk analysis** (identify problems, estimate impacts)
3. **Develop and test** a provisional solution
4. **Evaluate results** with the client and **decide whether to continue**

Then you return to phase 1 if everything is okay with the client.

Although very flexible, since it adapts to each situation and allows risk-aware management, it is not recommended for small projects because it is too heavy and requires great experience to identify real risks. Suitable for high-risk projects.

10.2.2 Agile Models (Scrum, XP...)

An agile model is a way of organizing software development **based on frequent interaction with the client, small steps, continuous adaptation, and quick feedback**. Less formalism, less fear of change.
The goal is to respond quickly to changes rather than follow a rigid plan defined at the start.

In these models, **not all details are planned from the beginning**, only **general objectives**. Then planning is done iteratively.

This is the approach of **agile methodologies**:

- small and frequent goals
- quick feedback
- continuous face-to-face communication
- adaptability to change
- minimal documentation
- self-organizing teams

Used in small projects where requirements are unclear and the environment (time, budget) is constantly changing. To be avoided if the project is very large and needs strict documentation traceability and clearly separated team responsibilities.

Scrum An agile framework with **short cycles** (**Sprints** of **1–4 weeks**) and **clear rules**. It doesn't describe how to code, but **how to work together as a team**. **It doesn't work in passive or disengaged teams.**

- **Sprint Planning** → At the beginning of the **Sprint**, the team *takes a list of requested features* (from the product backlog) and plans the work.
- **Daily Scrum** → Daily stand-up meetings of max 15 minutes, where you say what you did yesterday, what you'll do today, and if there are any problems.

- **Sprint Review** → *Features are implemented until a working, tested increment of software is achieved.* At the end of the sprint, the product is potentially releasable and shown to the client.
- **Retrospective** → Finally, the retrospective, where the team reflects on what went well and what to improve for the next sprint.

eXtreme Programming (XP) Unlike Scrum, here the **focus is on code development**, not project management. A release is delivered every **1–2 weeks** and **tests play a key role**. **The client** doesn't meet the team at fixed intervals (like at the end of each scrum), but **is always present.** Everything revolves around the team and the customer.

- All code is written by **two programmers together** on a single computer. One types, the other observes and suggests → then they switch.
- **Tests are written before the code.**
- A working version is released every 1–2 weeks.

Therefore, it should NOT be used when the client is not constantly available, when teams are distributed, or when automatic testing is not possible.

Agile models are always preferable because in reality, unforeseen events are inevitable. Agile models account for uncertainty, use feedback as a driver for progress, and minimize the damage when things change. Especially when unexpected events are caused by human factors — we'll later see how agile communication better suits social relationships within a project. [[**Personality and teamwork in software projects**]]

11 Project Management

A project does not develop by itself — planning and risk management are needed to prevent problems. (The most difficult problems to prevent are related to people — code can always be fixed.)

Doing project management means **managing** the project: - defining objectives, coordinating people, resolving conflicts.

We have **two styles of project management** which are the basis of different process models and how they are put into practice.

1. **Planned-leitend** (planned and directive) A **detailed and documented plan** is created, a **manager with decision-making power** is assigned. More suitable for *large, long projects, where making changes can be costly.*
2. **Agil-selbstorganisierend** (agile and self-organizing) **Minimal planning**, since the project and ideas can change easily and decisions are shared within the team. Suitable for *small teams, where the client is available for frequent meetings.*

11.1 The 9 areas of project management according to PMI

The **PMI (Project Management Institute)** is an international organization for project management standards and certifications. It defines what a project is, its phases, and areas of responsibility. The most known model is PMBOK, which defines the **9 areas (practical activities to manage so the project respects the triangle)**:

- scope → controls **functionality** (what the system does)
- time → controls **duration**
- cost → controls **costs**
- quality → ensures **standards**
- human resources, communication, risks, integration, procurement → support everything else

These are indicative areas — each project can omit or simplify some, depending on its needs.

11.2 The PM Triangle

At the base of project management there are **three conflicting factors**, where increasing one implies sacrificing another.

Time, Cost, and Quality

In software projects, quality is often divided into: - **Functionality** (what the program does) - **Quality** (how well it does it — robustness, reliability...) - **Duration (Time)**

While **Cost** is proportional to time, so **Time and Cost are essentially the same**.

You cannot improve everything simultaneously:

- If I want more quality, I necessarily need more time or resources → more cost and more work.
- If I want less time, I must give up some functionality or accept lower quality.
- Likewise, if I want lower costs, I must reduce personnel → slow down or sacrifice quality.

To improve one, I must sacrifice another. No project can maximize all three at the same time. Thus, balance is needed between the three, and the role of the project manager is to manage these conflicts and **understand what to prioritize based on the client's requests.** If you think you can satisfy everything, you end up in the death march problem.

11.3 The Todesmarschprojekte Problem

A very clear example of what can happen: **"If you mismanage a project from the beginning, there are no miracles or heroes. Only *failed projects* and *burnt-out people*."**

Todesmarschprojekte: **death march project** (from the start it's known that it will end badly, but people

keep working endlessly and no one says it). It's a project where the resources are half what's needed (time, staff, money), or the requests are double. In summary, a project with **unrealistic expectations**.

Example: a complete system must be developed in 3 months, with little staff, and no requested feature can be dropped.

This happens due to naive promises from salespeople, optimism, expectations of working nights, excessive competition, and overconfidence in technology.

These situations are rarer today thanks to greater awareness and organization — but still happen, especially among younger teams.

It's essential to understand from the beginning where the client's main focus lies, and decide in advance what to sacrifice. But how to understand that? By doing an effort estimation.

11.4 Effort Estimation

Estimating software projects is very difficult because often **architecture is unclear at the start, requirements may vary, and conditions can be unstable** (inexperienced team, new technologies). **Overly innovative or messy projects cannot be reliably estimated.** To get a more reliable estimate, you must use several techniques — which vary based on the chosen approach.

11.4.1 Estimation techniques in agile projects

1. **Analogy estimation** Compared to previously done projects, using them as a reference. Depends on experience.
2. **Expert estimation** Ask developers or project managers to estimate based on their experience. Practical and *fast but very subjective.*
3. **Combined estimation** Averages the various estimates. Reduces individual error — but if everyone is wrong, the result is still wrong.
4. **Adjustment factors** Take a similar project and apply correction percentages for each difference with the new project. Uses the **CoCoMo model**.
5. **Decomposition estimation** Split the request into two: **requirements**, by estimating each feature separately, and **design**, estimating architectural subsets. Risk: integration or invisible costs may be missed.
6. **Proxy-based estimation** If I can't estimate the type directly, but I can estimate the size of the code.

11.4.2 Estimation techniques in planned projects

WBS (Work Breakdown Structure) – what to do? A structure that breaks down the work — **it tells us what needs to be done**. It means breaking the entire project into manageable blocks (tasks) to have a clear overview, know what to estimate, and assign parts to people.

It proceeds in levels:
- Level 1: the entire project, i.e., what the product requires. Ex: "Development of a family expense management system"
- Level 2: main phases, divided by process or key functionality. Ex: "user module, expenses module, reports"
- Level 3: detail of each functionality. Ex: "for the user module you need to create an account, login, edit profile..."

Function Points (FP) – how long does each activity take? A unit of measurement of functionality provided to the user, regardless of language, system... **Based on this unit, we derive how much time is needed to develop them.** It is the most formal and objective method.

It can estimate data-based software projects. Not suitable for embedded systems (electronic devices), real-time systems, video games, robotics... **Requires certified experts** and might be overkill in small agile teams.

Each function has a weight, which can vary based on complexity. The final number is not in days of work, but an objective measure — which, combined with past projects calculated using the same method, can give a precise estimate.
Example: "If in the past a 120 FP project took 3 months, then a project with 80 FP may take 2 months."

Type	Example	Weight in FP
External Input	Form to enter a customer	3–6
External Output	Printable invoice, PDF report	4–7
Query	Search customers → show results	3–6
Internal Logical Files	Tables, database	7–15
External Interfaces	API, connection to external systems	5–10

So if I have 5 inputs, I multiply the weight 4×5 times etc.
To use function points, I must **be able to identify how many and which functions apply to my project**, meaning I need a **decomposition of requirements.**

Gantt – when to do each activity? A visual tool that **tells us when and for how long to do things.**
On the horizontal axis: days or weeks. On the vertical axis: activities from WBS.
Each activity is associated with a **planned duration**, represented by a horizontal bar from start to end — can be calculated using Function Points.

Each activity is also given a **resource** (one or more people) responsible for it, each with a **capacity**, i.e. how much time that resource is expected to dedicate. "100% = one full-time person"

There are tools (e.g. Microsoft Project) that automatically generate and update the Gantt chart as you enter: - activities (from WBS) - estimated duration (days or hours via FP) - assigned resources - dependencies - workloads - final deadline
They also calculate total costs.

Critical Path – what cannot be delayed? The critical path is the sequence of activities that **determines the overall duration of a project.** It's called critical because even if just one activity is delayed, then the whole project is delayed.

Activities outside the path have a certain **slack time** that they can use without affecting the final date. If slack = 0, then the task is critical.
Slack (or float) is automatically calculated based on the estimated duration (FP) and dependencies.

Term	Meaning
ES (Early Start)	First day the task can start
EF (Early Finish)	ES + Duration (FP)
LS (Late Start)	Last day to start without delaying anything
LF (Late Finish)	LS + Duration (FP)
Slack (or Float)	`LS { ES` or `LF { EF`

Cost Estimation As previously stated, cost does not play a primary role in the triangle since it's calculated based on time and resources. `time × resources = cost`

Important for: - budget definition with the client - comparing solutions - assessing feasibility - choosing priorities (triangle: time – cost – quality)

Various techniques:

- **Expert estimation:** ask an expert how long / how much effort something takes.
- **Historical comparison:** use a historical project repository.
- **Top-down decomposition:** split the project into tasks via WBS and estimate each part.
- **Function Point Analysis:** use function point weights and convert to time/cost. Ex: 100 FP, team delivers 4 FP/day/person, and 1 day/person = \$400 → 25×400 = \$10,000

Even costs are automatically calculated by Microsoft Project. You can define fixed and variable costs for resources, servers, licenses, etc.

11.5 Risk Management

How a good project manager thinks to not be surprised by problems. **Risk is something that might happen — a problem is something that already has.**

In the Project Management process, risk management does not come later — it accompanies all phases.

Phase	Example	Linked Risk
Planning (WBS, Gantt)	Defining activities and duration	Overload or rigid dependency → risk
Cost Estimation	Time, cost, resource assessment	Tight budget or uncertain productivity → risk
Execution	Work begins	Operational risks (absence, bugs, new requests)
Monitoring and control	Time tracking	Delays or unforeseen events

So risk helps prevent problems, not just react to them. *"If you don't attack the risks, the risks will attack you."*

Risk	Probability	Damage	Total Risk
"Client changes requirements"	High	Medium	High

Risk = Probability × Severity of damage

Three risk management styles

- **Optimistic** (the worst) → assumes nothing will go wrong.
- **Reactive** → deals with problems only after they happen.
- **Preventive** → foresees problems and plans how to avoid them. Basis of professional Risk Management.

11.5.1 How to prevent problems

- Identify risks
- Evaluate risks
- Decide which risks to address first
- Regularly monitor all risks (known and new) **Document them**

You can write automated tests to avoid bugs, or have backup plans if a supplier delays.
A possible risk list includes: insufficient or incompetent staff, unrealistic time or budget, wrong features, constantly changing requirements...
No plan ever goes exactly as imagined.
Also pay attention to the people in the project! `[[Personality and teamwork in software projects]]`

12 Personality and Work in Software Projects

Problems don't only arise from technical errors, but also from conflicts between people involved in projects, caused by different personalities — that is, **ways of thinking and reacting that remain similar across many situations.**
This includes communication styles, decision-making methods, and reactions to stress or change. Some people are naturally more organized, others more spontaneous, some are analytical, others empathetic, etc. All these factors influence how we work and collaborate.

Therefore, understanding different personality types helps improve collaboration, avoid unnecessary conflicts, and build more balanced and productive teams.

12.1 Personality Models Used in Software Projects

MBTI – Myers-Briggs Type Indicator A model based on the studies of Carl Jung. **Divides people into 16 types by combining 4 dimensions**, **NOT with the goal of labeling people** (there are no right or wrong types), but **to better understand ourselves and others.**

Dimension	Choices	Meaning
E / I	Extroverted / Introverted	Where do you get energy?
S / N	Sensing / Intuitive	How do you perceive the world?
T / F	Thinking / Feeling	How do you make decisions?
J / P	Judging / Perceiving	How do you organize your time?

Keirsey Temperaments A simplification of this model is called **Keirsey**, which **groups the 16 types into 4 "temperaments"**, each with its strengths and limitations.

Type	Main Focus and Contribution	Typical Problems	Work Tendencies
SJ (Guardians)	Order, rules, structure	Too rigid, bureaucratic	Plan, love procedures
SP (Artisans)	Action, practicality, improvisation	Lack of discipline	Act fast, solve on the spot
NF (Idealists)	Empathy, relationships, meaning	Trouble handling constraints	Communicate, motivate others
NT (Rationals)	Logic, systems, perfection	Perfectionism, lack of empathy	Visionaries, build architectures

According to this study, each temperament has natural tendencies toward the following activities:

12.1.1 In the requirements analysis phase:

- SJ → seek **clear and detailed specifications**
- SP → prefer a **flexible and agile approach**
- NF → **care about how the user will experience the product**
- NT → want **logical, consistent, and verifiable requirements**

Example: An SJ would say: "Give me the complete requirement document with all use cases." An SP would respond: "Let's get started and see what happens."

12.1.2 In planning:

- SJ → want **precise and well-documented plans**
- SP → hate planning in advance: "It all changes anyway."
- NF → fear the plans are unrealistic: "We won't make it in time."
- NT → plan only what they can **control**

The key to a well-balanced team is to have a good mix of types, so they can **compensate each other**.

12.1.3 In design:

- SJ → love documenting everything, with **lots of UML diagrams**
- SP → want to **start coding right away**
- NF → focus on **user interfaces and accessibility**
- NT → design the **logical architecture**, but may get lost in detail

12.1.4 In quality:

- SJ → plan tests, document, measure, all done **rigorously**
- SP → do only the tests that **seem necessary** (risk: too few)
- NF → want to **validate** the product: is it useful and pleasant to use?
- NT → get **frustrated** knowing **you can never test everything**

12.1.5 In process improvement:

- SJ → want many **rules, controls, audits, procedures**
- SP → hate rules, see them as **limitations**
- NF → focus on **communication within the team**
- NT → ask "But **do these practices actually work?**" — want **empirical evidence**

12.2 Other general-purpose personality models

Classical Temperaments **Not a scientific method** but a simple way to reflect. Based on Ancient Greece, it **associates personalities with the 4 elements:**

- **Sanguine** (air): optimistic, sociable, enthusiastic
- **Melancholic** (earth): reflective, precise, introverted
- **Phlegmatic** (water): calm, reliable, passive
- **Choleric** (fire): determined, dominant, impatient

Enneagram A system with **9 personality types**, each driven by a deep need. Useful in **private life** to understand oneself and conflicts, **less suited to software engineering**.

1. Wants to be right
2. Wants to help
3. Wants success
4. Wants to be unique
5. Wants knowledge
6. Wants to feel safe
7. Wants to enjoy life
8. Wants power
9. Wants peace and harmony

Big Five (OCEAN) The **most scientific and reliable** in psychology, **less used in software projects because it's more clinical**. Analyzes 5 major psychological dimensions, measured on a scale:

Acronym	Name	What it measures
O	Openness	Creativity, imagination
C	Conscientiousness	Precision, self-discipline
E	Extraversion	Social energy
A	Agreeableness	Kindness, empathy
N	Neuroticism	Tendency to stress and anxiety

12.3 How to collaborate between people

Essential for good Project Management. Because despite having precise tools (WBS, FP...), **in the end it's always the human mind making decisions.**

There are 4 key elements for a well-functioning team:

- **Technical skills** A minimum level of competence is required. Not everyone needs to know everything, but key skills like requirement analysis and architecture must be covered.
- **Motivation** If people don't believe in the project, feel uninvolved or stressed, progress stalls.
- **Effective communication** A team that doesn't communicate wastes time and argues.
- **Smooth collaboration** It's not enough to do your job — you must also interact and help others. This is where personality, mutual trust, and role rules come into play.

12.3.1 Collaboration approaches and roles

Neither of the following approaches is "best".

Self-organizing approach Each person chooses their role, based on skills and experience. The role is unique, not part of a formal list. Sometimes curious roles emerge like "fun coordinator" or "mediator"...

The team naturally adapts, so there's no hostility and everyone feels useful. But it takes time to find balance. **Everyone talks to everyone and decisions are made together.** Agile method for dynamic environments with frequent change. **Requires maturity.**

Uses Scrum:

- **Product Owner** → understands **what the client wants**, writes and orders features
- **Scrum Master** → team coach, not boss. Helps the team self-organize and follow agile rules
- **Development Team** → the actual group work. Team self-organizes using everyone's skills

Predefined roles approach Fixed roles with specific tasks. Clear from the start — but can trap people in roles that don't suit them.

More appropriate for very large and critical projects. **Communication is top-down, and decisions are made only from the top.**

Uses V-Modell XT — 32+ professional roles. German method used in public administration.

Some roles: - **Project Leader** - **Developer** - **Quality Responsible** - **Privacy** - **Test Manager** - **Change Manager**

Brook's Law

In software projects, **effects are not proportional to causes** — they feed back and create cycles: A delay → add people → communication slows → more delays...

The clearest example of a nonlinear dynamic is **Brook's Law:** *"Adding people to a late project makes it even later."*

This happens because of:

- **Increased communication load** Too many people = more interactions Everyone has to talk to more colleagues → productivity drops

- **Onboarding costs** New people need training, but veterans are busy — so both are slowed down

  Solution: Don't divide tasks further, let new members take over existing tasks and work independently.

Not everything can be done "in parallel."

12.4 Types of communication

Communication matters. Agile is the most effective — here's why.

There are two types of communication:

1- Synchronous vs Asynchronous

- **Synchronous** → real-time: meetings, calls Good for clarifying doubts quickly and discussing complex decisions.
- **Asynchronous** → delayed: emails, documentation Useful across time zones and for documentation — but slower.

2 - Formal vs Informal

- **Formal** → recorded: meeting minutes, contracts Can feel cold and slow.
- **Informal** → spontaneous: chats, "hallway conversations" More honest, no official record.

Both are necessary. That's why it's better to use agile models like Scrum and XP that **place communication at the center.**

12.5 Psychological Effects in Software Projects

Cognitive psychology (individual) Concerns what happens in a person's mind. Logical/deductive errors, underestimation, bias — **can influence code writing, design...**

Cognitive Biases Systematic thinking errors. Not distractions, but flawed mental shortcuts that **seem logical.**

Examples:

- **False induction** → generalizing from a few cases
- **Representativeness heuristic** → instinct to categorize too quickly "This looks like a big project, so it'll go like that other big one."
- **Availability heuristic** → judge likelihood based on recent/frequent events "Last year the database failed 3 times → must be the weakest part."
- **Confirmation bias** → only look for info that confirms what you believe, ignore what contradicts it

Social psychology What happens **between team members**. Motivation, group behavior, trust, mutual influence — **impact meetings, communication, collaboration.**

*"Everyone has their own **subjective perception** of the project."* But the team must build a **shared reality**.

That's why **continuous, transparent communication is essential.** People must feel free to express their opinions without judgment.

Motivation is also crucial — what drives someone to commit. It's a combination of: `Motivation = Value × Expectation` (Do I care? Do I believe I can do it? Do I have time?)

Everyone takes on attitudes (optimistic, pessimistic, cynical...) The project manager must recognize them, understand the cause, and know how to deal with them. Otherwise, the team may fall apart.

Examples of harmful team dynamics:

- **Blaming** → Creates a climate of fear
- **Overcontrolling (micromanagement)** → People shut down, lose motivation
- Ignoring **relational problems** → The team breaks apart over time

"Pretending everything is fine is not enough."

THERE IS NO SUCH THING AS A PERFECT TEAM STRUCTURE
You must be able to read people and communicate.

13 Reuse Techniques

We must learn to understand the forms of reuse, their advantages and disadvantages, and especially why **"patterns" are perfect examples of reuse.**

Reusing something known can increase quality, improve productivity, and reduce risk. Everything can be reused, but balance is key: using a standard module saves time, but if I need something specific, I risk losing flexibility.

Reuse doesn't just mean avoiding repetition. It means **learning from past experience**, **building on previous work**, and **standing on the shoulders of giants.** Every form of reuse saves time and reduces the risk of errors — but it **requires organization, awareness, and adaptation.**

Reuse is not just about code, but a strategy applicable to any software artifact.

- reuse of requirements
- reuse of architectural solutions
- reuse of design solutions
- reuse of code
- reuse of processes/methods

13.1 Reuse Risks

- **Quality risk** I may think of reusing a library or tool used in another project, but it may contain hidden bugs, be undocumented, or too optimized compared to my minimal project, making it heavier.
- **Inadequacy risk** What I reuse may not meet all requirements. Something might be missing, and adapting it could distort the original project.
- **Vendor risk** If you rely on third-party software, you may be in trouble if the vendor changes direction (adds useless features) or stops updating it.
- **Loss of flexibility** Using external components ties you to others' decisions. If something goes wrong, you can't fix or adapt it yourself.

Faced with these risks, one might think reuse isn't worth it — but in reality, **the choice is personal and based on many factors**, like time available, budget, need for control...

- If I choose to **Buy and reuse** an existing component: I save time on development and debugging (assuming the component is correct), but I **risk quality, have less control...**
  Writing reusable code costs much more (but saves time later), because it needs to be cleaner, well-tested, documented, with shared repositories and versioning standards.
  Many avoid reuse because of this, and because they feel that writing it themselves helps them understand it better.
- If I choose to **be inspired but build it myself**: I can adapt it better to my requirements and ensure quality, but I **do much more work, spend more time, and risk making mistakes myself.**

13.2 Learning to Reuse

Over time, **software engineering hasn't evolved from zero, but by accumulating reusable solutions**, to the point of reducing software development time by over 25% annually.

It's not necessary to "invent" new solutions every time — it's better to build on existing work and established experiences. In theory, this lets us **reduce risk** by reusing something proven.

Until the 1990s, reuse meant just code reuse.
Then, with the rise of **Design Patterns**, reuse extended to requirements, design, and architecture.

A **Pattern** is a pair/problem that can be reused in many contexts. It doesn't tell you what to do, but gives you a mindset.

13.3 Recurring Patterns

We use recurring patterns every day without realizing it — simple but powerful concepts at the root of understanding design:

1. **Abstraction** → ignoring unnecessary details Essential for reducing complexity and improving readability.
2. **Structuring** → making something complex more understandable by assigning explicit roles to its parts. If I divide a system into blocks, it's easier to understand, modify, extend.
3. **Modularization** → splitting software into independent parts, each with a well-defined responsibility. Lets you modify one module without touching the others, and test it in isolation.
4. **Locality** → changes should remain localized. That is, changing one part of the code shouldn't cause ripple effects elsewhere.
5. **Consistency** → similar parts should be treated similarly.
6. **Adequacy** → everything must be fit for purpose. *"You don't build a rocket to deliver a letter."*

Reusability is the principle that unifies all the others. If I apply all these principles, I am building software that anyone can reuse.

13.4 Anti-patterns

Anti-patterns are bad solutions that show up repeatedly in practice, because they're common and may look like good ideas at first.

- **Analysis Paralysis** → endless analysis without ever starting
- **Death by Planning** → months spent planning without knowing if the plans work
- **Reinvent the Wheel** → ignoring existing solutions and doing everything from scratch — often worse
- **Golden Hammer** → using the same solution for everything, even when it doesn't fit
- **Dead End** → modifying an external component so much you can't update it anymore
- **Bleeding Edge** → using cutting-edge tech that's not stable yet
- **Dumm Halten** → isolating developers from users, so they don't understand what's important
- **Design by Committee** → too many people want to decide architecture — different opinions, no direction

14 Design Patterns

By **Pattern** we mean a **recurring schema**, a structure that helps understand something very complex and manages to gather different concepts under a single idea. It's useful because by knowing the pattern, **the idea can be reused even if the code changes**, without having to reinvent new methods each time.
An example is certainly the Tic-Tac-Toe game and Get-15. Although they seem like completely different games, through in-depth analysis we understand that in both cases the goal is to reach the number 15 first (in tic-tac-toe every row and column adds up to 15). So the idea can be reused even for different code.

Patterns do not exist in isolation, but work in groups (Mustersprachen).

14.1 Concepts

a) Not to get confused

Concept	What it represents	Example
Component	A modular and reusable part of a system, often with a well-defined interface. It is an **element of the real system that can use patterns**.	`PaymentProcessor`
Design pattern	Abstract solution to a recurring problem in software design. It is a **reusable conceptual solution**.	Singleton, Adapter
Architecture/Architectural style	General model of overall system organization, focused on the macro structure. It's a **macro view** of the entire system and its connections.	MVC, Microservices

b) Definitions

- **Signature:** description of a method including name, parameters, and return type. Example: `int add(int x, int y)`
- **Interface:** a set of abstract method signatures that a class can implement. Defines **what** a class does, not **how**.
- **Class:** defines structure and behavior of objects.
- **Component:** an **independent and reusable** software module, with a well-defined interface, often composed of several classes. It's a distributed/reusable entity at the architectural level. Example:
  - Component `LoginService` handles login only.
  - Component `DBConnection` manages database access.
- **Module:** a set of components working together for a larger function. Example: "authentication" module containing login, logout, and permissions components.
- **Cohesion:** measures how well parts of a component work together to do a single task. High cohesion = **component does one thing well**, so the code is readable, testable, and clean.
- **Coupling:** measures **dependency** between modules. Goal: low coupling → changes in one module don't force changes in others.

14.2 Types of Design Patterns – Families

Structural Patterns	**Creational Patterns**	**Behavioral Patterns**
Organize objects and classes to build flexible, reusable structures	Create objects flexibly. *Separate object creation from usage*	Focus on how objects **behave and communicate**
- Adapter - Bridge - Facade - Proxy - Composite	- Abstract Factory - Builder - Singleton	- Observer - Command - Strategy - Iterator

14.3 Creational Patterns

14.3.1 Singleton

Sometimes an application needs **only one object** to manage something central — like a config file or logging system. That's where Singleton comes in.

Singleton ensures that a class has **a single instance across the system**, and provides a **global access point** to it, by keeping a private static reference to the single instance. *Avoids object duplication.*

Nothing e

xcept the Singleton class itself can replace the stored instance. The singleton object is initialized only when requested the first time.
So: make the constructor private and allow access only through a method like `getInstance()`.

```
public class Logger {
    private static Logger instance;

    private Logger() {}

    public static Logger getInstance() {
        if (instance == null) {
            instance = new Logger();
        }
        return instance;
    }
}
```

Examples:

- **Global logger:** without Singleton, each class would log to its own file — no way to merge logs. Singleton lets all classes use `Logger.getInstance().log(...)`
- **Database connection:** `DBConnection.getInstance()` ensures a single active or thread-safe shared connection.

Some experts don't even consider Singleton a true pattern, but a *technique* to limit resource access. Use it carefully — overuse can introduce hidden dependencies and make testing hard (since it's global).

14.3.2 Factory Method

Creates **one object type at a time**.
The idea is to delegate object creation to a subclass instead of using `new` directly.

Example: if you want to create a document (PDF or Word), instead of writing:

```
if (type.equals("pdf")) doc = new PDFDocument();
else doc = new WordDocument();
```

...and then needing to edit this every time you add a type (like HTML)...

We use Factory. **So it reduces class dependency, making code more flexible and testable, but it does require one class per type.**

```
abstract class Editor {
    public abstract Document createDocument();

    public void openDocument() {
        Document d = createDocument();
        d.display();
    }
}

class PDFEditor extends Editor {
    public Document createDocument() {
        return new PDFDocument();
    }
}

class WordEditor extends Editor {
    public Document createDocument() {
        return new WordDocument();
    }
}
```

14.3.3 Abstract Factory

Creates **an entire family of related objects**.
The key idea is to create families of objects without specifying concrete classes.

Example:
create a GUI for different OS like Windows and macOS. Avoid mixing classes with `if system == "win"` code, etc.

Instead: **create a GUIFactory interface with methods to create components**. Each family (Windows, macOS) has its own concrete factory implementation.

```
interface GUIFactory {
    Button createButton();
    Checkbox createCheckbox();
}

class WinFactory implements GUIFactory {
    public Button createButton() { return new WindowsButton(); }
    public Checkbox createCheckbox() { return new WindowsCheckbox(); }
}

class MacFactory implements GUIFactory {
    public Button createButton() { return new MacButton(); }
    public Checkbox createCheckbox() { return new MacCheckbox(); }
}
```

14.4 Structural Patterns

14.4.1 Adapter

Used when you want to **reuse an existing component** and adapt it to a new system.
Example: using an old XML API in a program that expects JSON — adapter converts XML to JSON behind the scenes.

It makes two incompatible interfaces compatible. Can be implemented via multiple inheritance in Java.

Bridge

Goal: **separate abstraction from implementation**.
Example: GUI for multiple OS.
With a complex hierarchy — on one side functionality (Window, Dialog...), on the other platforms (Windows, Linux, Mac). If you use direct inheritance, you lose combinations like WinDialog, MacDialog.
Solution: **split into two hierarchies**: - Abstraction: Window, Dialog... - Implementation: WinImpl, LinuxImpl...

14.4.2 Facade

Provides a **simplified interface** to a complex subsystem.
Used to **hide internal details** of a system and make it easier to use. Example: behind the scenes it's chaos — but all you see is a single ON button.

The Facade class wraps the complexity and exposes just 1–2 simple methods. Example: in a home theater system, the `Home` class manages DVD, projector, lights, screen, etc. User just calls `startMovie("Matrix")`

Ideal for building public interfaces.

14.4.3 Proxy

Replaces a real object with another that behaves the same, but adds something.
Used when you want to **control, protect, or delay** access to an object — maybe it's heavy to load, sensitive, or remote.

Example: - represent the object on another server - load only when needed - limit access by permissions - cache to avoid recalculation

Like lazy-loading images in a browser.

14.4.4 Composite

Lets you **treat individual objects and groups of objects uniformly**.
Used to represent hierarchical structures (like trees), where nodes can be simple (leaf) or containers of others (composite node).

Just create a common `Component` superclass — subclasses are Leaf and Composite. **Important:** This is not inheritance — you're not just "specializing" behavior, you're building a **recursive structure**.

Example: I want to call `print()` on both a single file and a folder that contains multiple files. `print()` must work identically in both.

14.5 Behavioral Patterns

Behavioral patterns describe **how objects interact**, who does what, who makes decisions, and how system behavior changes over time. Their goal is to organize responsibilities and ensure flexibility and extensibility.

14.5.1 Observer

Perfect when you want **automatic reactions to state changes** without hardcoded dependencies. Used in GUIs (Swing, JavaFX), events (mouse listeners), plugins...

You split the system into two roles: - the subject (data holder) - the observer (listens to the subject and reacts to changes)

The observer subscribes to the subject and gets notified every time it changes.

Risk: hard to control notification order, and observers might cause cyclic updates.

Example: Changing a cell in a table — you want the table, charts, and all other views to update.
Observer has an `update()` method that gets called when the subject changes. Multiple observers can subscribe to one object.

14.5.2 Command

Encapsulates a request as an object — so it can be delayed, undone, saved, passed, etc.

Example: building a GUI with a menu: Each menu item executes an action (home, open, report...) — but we want a reusable, reconfigurable GUI.

A naive approach: call `homepage()` when the button is clicked — but then you can't undo it, change behavior at runtime, or save user history.

Also, each button is tied to its own handler via `@FXML`:

```
@FXML
public void changeFamily(ActionEvent e) {
    showDialog("/ui/editFamily.fxml", changeFamily, "Edit family");
    initialize();
}
```

Instead, the correct solution:

- Create a `Command` interface - Implement actions inside classes that implement it - Buttons hold a command object and call `command.execute()`

14.5.3 Strategy

Strategy Pattern lets you **dynamically change an object's behavior at runtime, without if/else**, keeping code clean, modular, and reusable.

Example: In a database app where behaviors vary — like payment method or sorting method.

A payment system supports PayPal, credit card, wire transfer... But the UI remains the same.

How? Define a common interface (`Payment`), and create classes like `PayPalPayment`, `CreditCardPayment`, each with its logic.

14.5.4 Iterator

When you have a collection (List, Set, Tree, Array...) and want to access elements one at a time without exposing the internal structure — use Iterator.

Iterator introduces a separate object with a standard interface to loop over elements.

Advantage: structure can change without breaking client code. Difficult to implement if the structure is not linear (e.g., graphs).

15 Documentation

In software engineering, **documentation is essential** because a product is not only made of code — it must be understandable, usable, and modifiable over time.
The purpose of documentation is to communicate clearly, precisely, and completely to those who will use, maintain, or verify the software.

15.1 Types of Documentation

There are several types, depending on the recipient:

- **Requirements documentation** Explains what the software must do, user needs, use cases. It's the base for the entire development — if it's wrong or unclear, the product will be too.
- **Design documentation** Describes the architecture, components, modules, patterns used. Useful to understand how the system works and how to change it later.
- **Code documentation** Comments, docstrings, interface specs, examples. Must be short and to the point — the code should speak for itself.
- **Testing documentation** Describes test plans, test cases, expected results. Serves to verify the system and explain what has been tested.
- **User documentation** Manuals, tutorials, FAQs. Needed for end users, must use clear and non-technical language.

15.2 How to Write Good Documentation

Not all documentation is equal — a lot of it is useless or confusing. To write good documentation:

- **Be clear and concise** Avoid long, vague texts. Better short and direct phrases.
- **Adapt to the reader** The language of a developer guide is not the same as a user manual.
- **Use examples** They clarify better than long explanations.
- **Update regularly** Outdated documentation is worse than none.
- **Use diagrams and tables** Visual representations help a lot.
- **Don't duplicate information** If something is written in the code or another document, refer to it.
- **Be consistent** Use the same terms, structure, formatting.

15.3 Tools

Documentation can be written manually or generated automatically.

- **Javadoc / Doxygen / Sphinx** → generate documentation from code comments
- **Markdown / AsciiDoc / LaTeX** → for writing guides, reports, manuals
- **UML** → for diagrams of classes, use cases, sequences...
- **ReadTheDocs, MkDocs** → for publishing online documentation

Examples of Good and Bad Documentation

BAD

```
This function modifies the data in a meaningful way.
```

→ too vague. What data? What is "meaningful"?

GOOD

```
/**
 * Updates the user's age based on the new birth date.
 * Precondition: user must be registered.
 * Postcondition: age updated in user profile.
 */
```

Final Advice

Documentation is not just a final task, but a fundamental part of software development. **Writing it well takes time, but saves much more later.**

It helps:

- onboard new team members
- understand and modify code after months or years
- reduce bugs
- increase user satisfaction

Even if you're the only developer, good documentation is a gift to your future self.