



**University of Jijel-Mohamed Seddik Benyahia**  
**Faculty of Exact Sciences and computer science**  
**Department of Computer Science**

**Field : Computer Science**  
**Level : License 3**

**Course Material**

**Module : Software Engineering**

**Written by:**

**Dr. Aissam Belghiat**

**School year: 2023 /2024**

# *Table of contents*

<b>General introduction.....</b>	<b>1</b>
<b>1. Introduction to Software Engineering.....</b>	<b>4</b>
1.1 Introduction.....	5
1.2 Software introduction.....	5
1.2.1. Software definition.....	5
1.2.2. Ubiquitous software.....	5
1.2.3. Software Positive Impacts.....	5
1.2.4. Software Negative Impacts.....	6
1.2.5. Different types of software.....	6
1.3 Software development.....	6
1.3.1. Development difficulties.....	7
1.3.2. Statistics on Development Projects.....	7
1.3.3. What is good software? .....	8
1.4 Software crisis.....	9
1.5 Software Engineering .....	9
1.5.1. Definition of Software Engineering.....	9
1.5.2. The origins of software engineering.....	9
1.5.3. Software Engineering Principles.....	10
1.5.4. Software Quality Criteria .....	10
1.6 Software Life Cycle.....	12
1.6.1. Main steps.....	12
1.6.2. Life Cycle Models.....	13
1.6.3. Software Development Process Maturity.....	20
1.7 Software Development Tools and Professions.....	21
1.8 Review.....	22

<b>2. Modeling with UML.....</b>	<b>23</b>
2.1 Introduction.....	24
2.2 System modeling.....	24
2.3 Modeling levels.....	25
2.4 Software Modeling Approaches.....	26
2.4.1. Function-oriented modeling.....	26
2.4.2. Object-oriented modeling.....	29
2.5 UML (Unified Modeling Language).....	29
2.6 History.....	29
2.7 Using UML.....	30
2.8 Advantages and disadvantages of UML.....	31
2.8.1. The strengths of UML.....	31
2.8.2. The weaknesses of UML.....	31
2.9 Modeling with UML.....	31
2.9.1. Notion of view.....	31
2.9.2. UML views.....	32
2.10 UML diagrams.....	32
2.11 Review.....	34
<b>3. Use case diagram: functional view.....</b>	<b>35</b>
3.1 Introduction.....	36
3.2 Importance of the functional view in modeling.....	36
3.3 Objective.....	36
3.4 Use case diagram elements.....	36
3.4.1. Use cases.....	36
3.4.2. Actor.....	36
3.4.2. System.....	36
3.5 Relationships in use case diagrams.....	37
3.5.1. Relationships between use cases and actors.....	37
3.5.2. Relationships between use cases.....	37
3.5.3. Relationships between players.....	38
3.6 Identifying and classifying players.....	39
3.6.1. Identifying the players.....	39

3.6.2. Main and secondary actors.....	39
3.7 Identification and description of use cases.....	39
3.7.1. Identification of use cases .....	39
3.7.2. Description of use cases.....	40
3.8 Example of a use case diagram.....	41
3.9 Review.....	42
<b>4. UML class and object diagrams: static view.....</b>	<b>44</b>
4.1 Introduction.....	45
4.2 Importance of the static view in modeling.....	46
4.3 UML Class Diagrams.....	46
4.3.1. Objective.....	46
4.3.2. Basic notions.....	46
4.3.3. Class properties: Attributes and Operations .....	46
4.3.4. Class relationships.....	48
4.3.5. Other notions.....	52
4.3.6. Building a class diagram.....	55
4.4 UML Object Diagrams.....	56
4.4.1. Objective.....	56
4.4.2. Object representation.....	56
4.4.3. Class and object diagrams.....	56
4.4.4. Links.....	57
4.4.5. Instantiation dependency relationship.....	57
4.5 Review.....	59
<b>5. UML diagrams: dynamic view.....</b>	<b>61</b>
5.1 Introduction.....	62
5.2 Importance of the dynamic view in modeling.....	62
5.3 Sequence diagram.....	62
5.3.1. Objective.....	62
5.3.2. The chronological aspect.....	62
5.3.3. Lifeline.....	63
5.3.4. Messages.....	64
5.3.5. Combined segment.....	67

5.4 State-Transition diagrams.....	68
5.3.1. Objective.....	68
5.4.2. Initial and final states.....	68
5.4.3. State and Transition.....	68
5.4.4. External and internal transition .....	69
5.4.5. Junction and decision point.....	70
5.4.6. Concurrence.....	71
5.4.7. Other concepts.....	72
5.5 Activity Diagram.....	74
5.5.1. Objective.....	74
5.5.2. Initial and final conditions.....	74
5.5.3. Activities and Transitions.....	74
5.5.4. Condition and Merge.....	75
5.5.5. Synchronisation and Concurrence.....	76
5.5.6. Swimlanes.....	77
5.5.7. The objects.....	77
5.5.8. Use cases documenting.....	77
5.5.9. Example of an activity diagram.....	79
5.6 Review.....	80
<b>Bibliography and Webography.....</b>	<b>83</b>

# *General introduction*

## **Course Description**

Software engineering is all about creating, running, and maintaining software in a clear, efficient and organized way. These days, software is everywhere, so it is very important to know how to build it properly. This course will help you learn how to analyze and design software. We will focus on object-oriented modeling and use a well-known tool called UML (Unified Modeling Language).

## **Course Objectives**

The goal of this course is to teach you the basics of software engineering and how to use different methods and techniques to design software systems. We will concentrate mainly on object-oriented methods and spend a lot of time learning how to model software systems with different UML diagrams.

## **Prerequisites**

To benefit largely of this course, you will need some basic knowledge in these areas:

- Algorithms: You should understand how algorithms work and some of their main ideas.
- Information Systems: It is nice to have a general idea of how information systems operate.
- Object-Oriented Programming: You will need to know how to program in an object-oriented language since we will build on that knowledge.

## **Course Outline**

This course is divided into five main chapters, each covers an important topic in software engineering:

### **1. Introduction to Software Engineering**

We will start with the basics, like what software engineering is, why it is important, what makes software “good,” and the different steps and models involved in developing software.

### **2. Modeling with UML**

You will learn about modeling in general and how UML makes it easier to design software. We will explain the philosophy of UML and give an overview of its diagrams.

### **3. Use Case Diagrams – Functional View**

This section shows how a system works from a user perspective. You will learn how to use UML use case diagrams to model interactions between users and the system.

### **4. Class and Object Diagrams – Static View**

Here, we will look at how to show the structure of a system using class and object diagrams. This means that we need to model the entities and their relationships in the system.

### **5. Dynamic UML Diagrams**

Finally, we will cover diagrams that show how a system behaves during its execution, like sequence diagrams, activity diagrams, and state diagrams. These help in understanding how things work and evolve step by step.

## **Evaluation**

Your grade will be based on two parts: continuous assessment, such as assignments and projects, (40%), and the final exam (60%). This way, you will get the chance to practice what you are learning before the exam.

## **Expected Results**

By the end of this course, the student is expected to know the basics of software engineering and be able to design software systems using UML. You will also have a good understanding of different methods and techniques existing in the literature. These skills will help you tackle real-world software problems and provide you with a strong foundation for a career in software engineering.

## **Related Course**

Since software engineering is a broad field, many topics that complement what you will learn here have been covered in another course called "Advanced Software Engineering" written for Master's students. Examples of subjects include Design Patterns, Advanced Modeling, and Model Driven Engineering. Thus, it is preferable to have both courses in your collection.

# *Chapter 1*

## *Introduction to Software Engineering*



## 1.1. Introduction

Software engineering is important for the design, development, and maintenance of high-quality software systems, which become parts of our daily lives. In fact, software plays a dominant role in numerous fields from business applications to critical systems, which makes it decisive to understand the principles and methodologies that guide its creation and evolution.

In this chapter, we explain the fundamental concepts of software and software engineering—an essential area within computer science. Readers will find a comprehensive introduction to software engineering, that covers its definitions, historical issues, and its types and principles. The chapter also outlines the challenges and complexities involved in software development, and how they impact the project success. Additionally, it expounds the software development life cycles from requirements analysis to maintenance, and their models processes including classical paradigms and new agile methods.

## 1.2. Software introduction

### 1.2.1. Software definition

Software is a sequence of instructions (programs) and associated data (files, images, databases, etc.) that a computer processes automatically to perform specific tasks.

It also includes a number of documents relating to these programs and necessary for their installation, use, development and maintenance including specifications, conceptual diagrams, test sets, operating mode, etc.

Example: Windows, Microsoft Office, Facebook, Google Chrome, etc.

### 1.2.2. Ubiquitous software

These days, software is everywhere. Everyday life is unimaginable without it: office applications, leisure, internet, management, administration, industry, etc. Computer systems are made up of 80% software, which is often the source of problems, and 20% hardware, which is relatively reliable and standardized. In fact, IT-related problems are essentially software problems, and give that our lives have become very attached to software, the quality of which has a direct impact.

### 1.2.3. Software Positive Impacts

Software has many positive impacts on our daily lives:

1. Improved data processing (text, video, office automation)
2. Faster storage, restoration and processing of information (DB management).
3. Introduction of new leisure activities (games, etc.).
4. Increased productivity (car production).
5. Bridging distances (network and Internet applications).
6. Software never gets tired (regulator).
7. Replacing human beings in tasks:
  - Fast and sensitive (exact): industrial processes.

- Dangerous: military operations.
- real time: production line.

#### 1.2.4. Software Negative Impacts

Several incidents of varying degrees of danger have been reported due to poorly designed software:

1. The Y2K bug: When the year 2000 was about to arrive, no one could really predict what was going to happen because dates are coded in 2 digits.
2. The Mariner-1 bug on July 22, 1962: A space rocket (to explore the planet Venus) was thrown off course by a mathematical formula that had been incorrectly transcribed into source code. The range safety officer ordered it destroyed 294 seconds after launch.
3. Therac-25 medical gas pedal (1985): The machine was designed to treat patients. Due to a bug in the radiation trigger, at least five people died.
4. Incident during the Gulf War (1991): An American missile kills 22 American soldiers instead of intercepting an enemy missile, because of a rounding function error.
5. Ariane 5 rocket explosion - Flight 501 (1996): The rocket broke up and exploded in flight just 36 seconds after lift-off. This was due to an integer overrun in the memory registers of the electronic calculators used by the autopilot.

#### 1.2.5. Different types of software

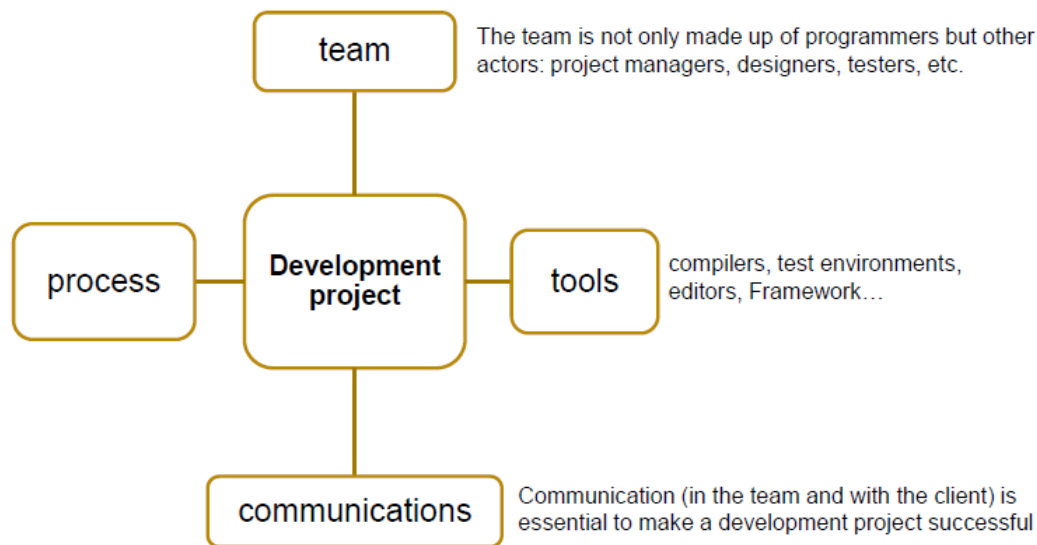
Several types of software exist:

1. System software (utilities for other systems).
2. Real-time software
3. Management software (classic applications).
4. Scientific software.
5. Critical software.
6. Combinatorial software (AI).

### 1.3. Software development

Development is the transformation of an idea or a need into functional software. The idea is generated by a customer (user) and developed by a supplier (an IT company). The development task is not an easy one. It involves a series of activities. Programming (coding) is not development, but **one of** the activities involved in development. Also, there isn't just one way to develop a given piece of software, but several.

Development projects are often long and costly (50% of maintenance costs). They often involve several people with different skills.



**Fig.1.1** Stakeholders in software development

### 1.3.1. Development difficulties

Software development is characterized by a number of difficulties:

- Customers find it difficult to describe their needs clearly to suppliers.
- Needs and the environment are constantly evolving.
- The software is not palpable (intangible).
- Language differences between technical and non-technical people
- Difficulty discovering errors before product delivery.
- Software piracy causes enormous damage to suppliers.

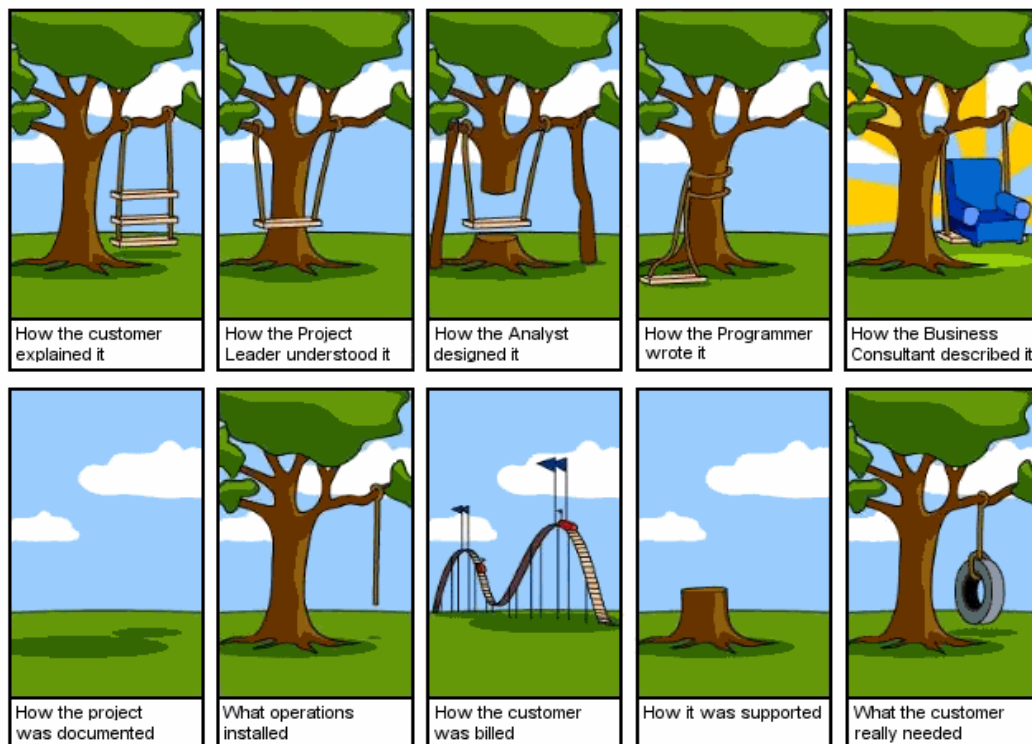
### 1.3.2. Statistics on Development Projects

According to a study by STANDISH GROUP<sup>1</sup> on 8,380 projects:

- 16% of applications are successfully built.
- 53% of projects succeed, but cause problems such as reduced functionality, failure to meet deadlines or increased costs.
- 31% of projects are simply abandoned.

What's the problem? In fact, there are several interconnected problems. This famous satirical cartoon sums it up.

<sup>1</sup> An international IT consulting firm founded in 1985



**Fig.1.2.** Software development complexity.

### 1.3.3. What is good software?

#### For the customer :

- Does what it's told
- Meets quality criteria
- Delivered on time
- Inexpensive

#### For the supplier:

- Does what it's told
- Meets quality criteria
- Minimize lead times
- Minimize costs
- Maximizing profits

So you always have to think in terms of **Cost, Time, and Quality**, to ensure the success of an IT project.



Fig.1.3. Triangle of success.

## 1.4. Software crisis

At the end of the 60s, the software crisis began. Software construction remained in the realm of **handicraft** and **folklore**, with everyone followed their own recipe that didn't fit in with the big systems. This includes:

- Software that doesn't meet customer expectations.
- Teams of programmers are overwhelmed by communication problems.
- Failure to meet deadlines and costs. Some projects have been abandoned (London Stock Exchange's TAURUS project).
- Maintenance too expensive because too difficult.
- Response time too long!
- Non-evolving applications.
- No conviviality.
- ... etc

In fact, there was an absolute necessity to bring together efforts to face the problems encountered.

## 1.5. Software Engineering

### 1.5.1. Definition of Software Engineering

Software engineering is an engineering field that enables the design, implementation and maintenance of high-quality software systems.

Software engineering is the art of producing good software. It is based on methodologies, techniques and tools that enable the production of **large-scale** software, i.e. requiring significant human resources and taking a relatively long time to complete, while satisfying both the customer and the supplier in:

- a timely and cost-effective manner
- quality

### 1.5.2. The origins of software engineering

Two famous lectures by NATO's Scientific Committee were at the origin of the definition of Software Engineering.

- in Garmisch, Germany, from October 7 to 11, 1968,
- in Rome, Italy, from October 27 to 31, 1969.

### 1.5.3. Software Engineering Principles

Software engineering uses some principles to produce quality software:

- **Generalization:** grouping of a set of similar functions into one configurable function (genericity, inheritance).
- **Structuring:** a way of breaking down software (using a bottom-up or top-down method).
- **Abstraction:** a mechanism for presenting a context by expressing relevant elements and omitting irrelevant ones.
- **Modularity:** breaking down software into discrete components.
- **Documentation:** management of document including their identification, acquisition, production, storage and distribution.
- **Verification:** determination of compliance with the specifications established on the basis of needs identified in the previous phase of the life cycle.

### 1.5.4. Software Quality Criteria

- **Reliability:** this is divided into two criteria:
  - **Validity (conformity or usefulness):** software conforms to its specifications, the results are those expected.
  - **Robustness:** the software works reasonably in all circumstances, even in conditions not initially anticipated.

Reliability can be measured by MTBF (Mean Time Between Failures). It concerns availability (percentage of time during which the system is usable) and Error Rate (number of errors per KLOC).

- **Usability:** is the ability to communicate effectively with the user. Normally the user should be able to:
  - Know and understand all the functions offered by the software (ease of learning).
  - Learn how to use them for a given purpose (ergonomics and ease of use).
- **Extensibility:** Ability of software to be enhanced and updated without discarding the software or starting from scratch.
- **Portability:** The same software must be able to run on several machines (platforms), i.e. make the software independent of its execution environment.
- **Performance:** Software must satisfy the constraints of **execution time** and **memory space** occupation (pay attention to software complexity).
- **Reusability:** The ability of software (or parts of it) to be used several times in different contexts. Considerable gains can be expected, since most software is based on the 80-20 rule <sup>2</sup>.

---

<sup>2</sup> Pareto's Law

- 80% of the code is “all-comers” that can be found almost everywhere.
- 20% of the code is specific.

- **Interoperability:** Ability of a program to work correctly in collaboration with other software that may be developed using other technologies.
- **Measurability:** Software's ability to be evaluated (cost, time, complexity, etc.).
- **Ease of maintenance:** Maintenance absorbs a very large part of the development effort (see Table 1.1). There are several types of maintenance:

	Dev effort distribution.	Origin of errors	Cost of maintenance
Definition of needs	6%	56%	82%
Design	5%	27%	13%
Coding	7%	7%	1%
Integration Tests	15%	10%	4%
Maintenance	67%		

(Zeltovitz, De Marco)

**Tab.1.1.** Maintenance cost ratio, life cycle stage.

**1- Corrective maintenance (mandatory):** Correct reliability and usability errors.

- Identify failure, operation.
- Locate the part of the code responsible.
- Estimate impacts and make modifications.

**Attention!!!**

- Most corrections introduce new errors.
- Correction costs increase exponentially with the detection time.
- Corrective maintenance gives rise to new deliveries (i.e. *releases*).

**2- Adaptive maintenance (updates required):** Adjust the software so that it continues to fulfill its role in the face of changing market conditions:

- Runtime environments.
- Functions to be satisfied.
- Terms of use.

Example: change of DBMS, machine, VAT rate, year 2000, euro...

**3- Perfective maintenance (optional):** Enhances the software's capabilities. Gives rise to new versions.

Example: services offered, user interface, performance, etc.

## 1.6. Software Life Cycle

The quality of the manufacturing process guarantees the quality of the product. To obtain quality software, you need to master the development process. The succession of these stages forms the software life cycle.

The software life cycle refers to **all the stages** in the development of a software product, from the definition of requirements to its end-of-life.

Errors are more costly the **later** they are detected in the development process (generally \*10). Lifecycle management enables errors to be detected as early as possible, thereby controlling software **quality**, **deadline** and associated **cost**.

Every software development project is made up of several activities. Each activity is managed and carried out by several **Stakeholders**. An activity has inputs and outputs. **Deliverables** are the outputs of activities. **Deliverables** are artifacts produced by an activity and used by the activities that depend on it, such as **documents**, **schedules** and **source code**.

### 1.6.1. Main steps

All development projects have **common activities**:

1. Needs analysis
2. Design
3. Coding
4. Tests
5. Product delivery
6. Maintenance

**1. Needs analysis:** Consists in expressing, gathering and formalizing the requirements for a project (whether new or altered). This involves studying the various constraints and estimating the project's feasibility. This phase is both essential and difficult, because the customer and the developers don't speak the same language. The deliverable of this phase is a **specification** document (i.e. functionalities, cost analysis, planning, task allocation...).

**2. Design:** Determining how the software will deliver the various functions (needs) expected. It is divided into:

- **Architectural design (general):** determine the overall structure (architecture) of the system (i.e. the blocks or modules and the interfaces between them).
- **Detailed design:** this involves precisely defining each subset of the software with diagrams and algorithms.

Recommendation: **maximize cohesion** within components and minimize **coupling** between them.

**3. Coding (implementation):** is the translation into a programming language of the functionalities defined during the design phases.

Coding techniques are intrinsically dependent on the programming language and paradigm used.



Example: **Pair Programming**, cross-reading.



**Fig.1.4.** Peer programming.

**4. Tests:** Test the software on example data to make sure it works properly. There are several types of tests:

- **Unit tests:** These are used to check individually that each subset of the software has been implemented in accordance with the specifications.
- **Integration testing:** The aim is to ensure that the different elements (modules) of the software interface with each other.
- **Acceptance tests:** verification that the software conforms to the initial specifications (validation by the customer).

**5. Product delivery:** Provide the customer with a properly functioning software solution. This stage includes the following tasks:

- **Installation:** make the software operational on the customer's site.
- **Training:** teaching users how to use the software.
- **Support:** answering users' questions.

**6. Maintenance:** Its purpose is:

- **Correction:** correct any errors.
- **Upgradeable:** Update and improve the software to ensure its longevity.

To keep maintenance time and costs to a minimum, focus on the earlier stages.

### 1.6.2. Life Cycle Models

The sequence and presence of each of these activities in the lifecycle depends on the choice of a lifecycle model between the **customer** and **the development team**.

A **software process** is a set of activities leading to the production of software.

### Classic and agile methodologies

There is no standard categorization for development processes, but from a structural point of view they can be divided into two categories:

**Classic methodologies**

- Strict models
- Clearly defined stages
- Extensive documentation
- Works well with large and government projects
- Example: Cascade, V, Incremental, Spiral, etc.

**Agile methods**

- Incremental and iterative models
- Small, frequent deliveries
- Emphasis on code and less on documentation
- Suitable for small and medium-sized projects
- Example: Scrum, Extreme Programming (XP), Rapid Application Development (RAD), etc.

**Classic methodologies****1. The cascade model (waterfall)***Description*

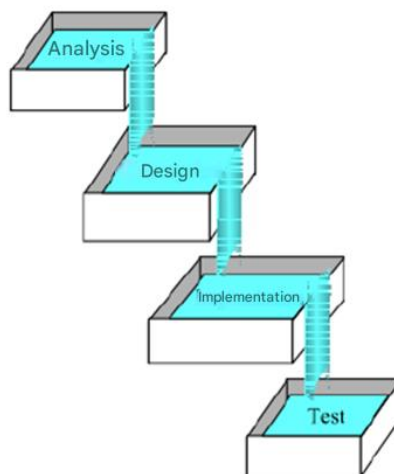
One of the first models proposed (1970) inspired by the building industry. Each phase ends on a specific date with the production of certain documents or software. A new phase can only be started if the previous one has been completed. The results are subjected to a thorough review, and only if they are judged to be satisfactory we move on to the next phase. There is no going back in the process.

*Use*

This model is well suited to small-scale projects, whose domain is well mastered (stable needs, new versions of a project...).

*Inconvenient:*

The testing phase comes late (discovering errors or changing specifications means redoing all the work).



**Fig.1.5.** The cascade model.

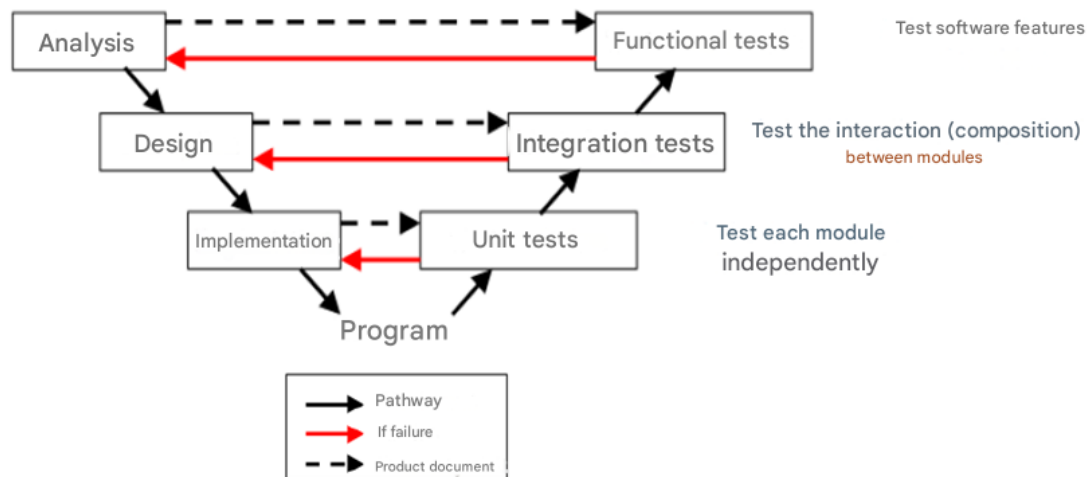
## 2. The V model:

### *Description*

The best known and certainly the most widely used. Test and software development are carried out synchronously (product testing takes place in parallel with other activities). Every deliverable (from every stage) **must be testable** (every description of a component is accompanied by tests to ensure that it corresponds to its description).

### *Use*

When the product to be developed has very high quality requirements, and when needs are known in advance.



**Fig.1.6.** The V-shaped model.

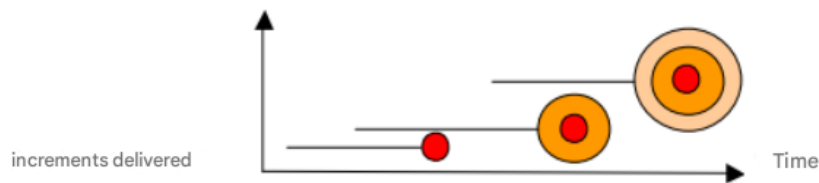
## 3. The incremental model:

### *Description*

Design and deliver to the customer a **minimal, functional subset of the system**. Start by developing the most stable parts, the most important ones, etc. Proceed by adding minimal increments until the end of the development process.

*Advantages:*

Better integration of the customer in the development process. Product fit in customer expectations.



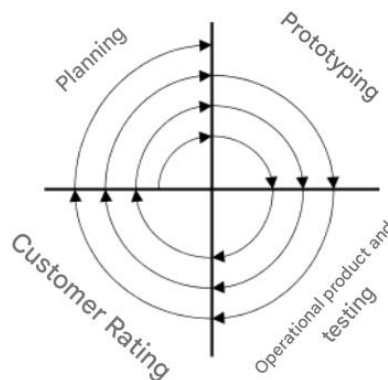
**Fig.1.7.** The incremental model.

#### 4. The spiral model (iterative):

Development follows the various stages of the V (evolutionary) cycle. By implementing successive versions, the cycle starts again, offering an increasingly complete and robust product.

The spiral cycle, however, places greater emphasis on **risk management**. In fact, the start of each iteration includes a **risk analysis** phase. This is made necessary by the fact that, during cyclical development, there is a greater risk of undoing.

This process requires extensive experience.



**Fig.1.8.** The spiral model.

#### Agile methodologies

Agile methodologies follow what are known as the twelve (12) agile principles:

1. Continuous customer satisfaction through fast, uninterrupted deliveries
2. Welcome all changes, even late ones
3. Frequent delivery of a functional system
4. Customers and developers must work together
5. Manage the project with motivated teams
6. The best way to circulate information is through direct contact between employees.
7. The first measure of progress is functional software
8. Development must be sustainable and at a constant pace
9. Good design and technical excellence increase agility

10. Simplify as much as possible
11. The best architectures, requirements and designs come from self-organizing teams
12. The team improves autonomously and regularly

## 1. XP (eXtreme Programming) methodology

### *Description*

XP is a lightweight, efficient, low-risk, flexible, scientific and fun way to develop software. For medium-sized teams with incomplete and/or vague specifications

-Coding is the **core of XP**

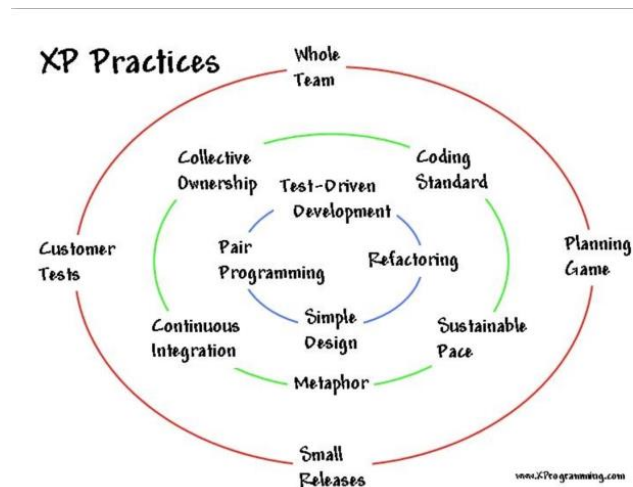
### *XP - Fundamentals:*

- Massive customer involvement
- Continuous unit testing (TDD)
- Pair programming
- Short iterations and frequent deliveries

### *XP methodology - Main activities*

- Coding (XP core)
- Tests (during coding)
- Listening (to the customer or partner)
- Design (still based on coding)

### *XP practices*



**Fig.1.9.** XP practices.

### *XP methodology - Disadvantages*

- Requires a certain maturity on the part of developers
- Pair programming still not applicable
- Difficulty in planning and budgeting a project
- Stress due to continuous integration and frequent deliveries
- Poor documentation can be detrimental if developers leave

## Scrum methodology

### *Simple*

- Can be combined with other methods
- Compatible with best practices

### *Empirical*

- Short iterations (**sprints**)
- Continuous feedback

### *Simple techniques*

- 2 to 4 week sprints
- Needs captured as **user stories**

### *Teams*

- Self-organization
- Multi-skills

### *Optimization*

- Rapid detection of anomalies
- Simple organization
- Requires openness and visibility



**Fig.1.10.** Origin of Scrum.

## Principles and concepts

### *Process*

- Sprint

### *Backlog*

- Product Backlog
- Sprint Backlog

### *Team*

- Scrum Master
- Product Owner
- Team
- Users and stakeholders

### *Meetings*

- Daily Scrum
- Planning
- magazine
- Retrospective

*Follow-up*

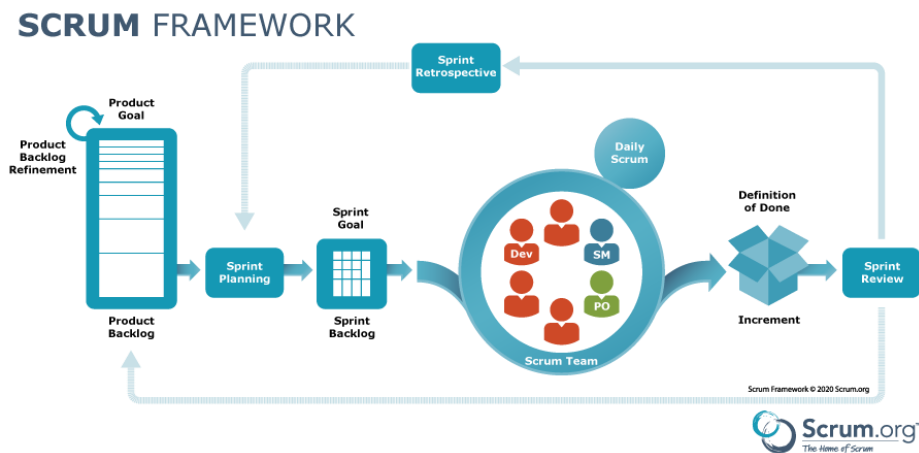
- Reports
- Velocity

*Advantages:*

- Very simple and effective method
- Adopted by market giants **Microsoft, Google, Nokia...**
- Project-oriented, unlike XP, which is development-oriented
- Can include other activities from other methodologies (especially XP)

*Disadvantages*

- Not 100% GL-specific
- Difficulty in budgeting a project



**Fig.1.11.** Overall Scrum diagram.

**Other processes**

Numerous other types of processes:

**Unified Process (UP):**

- Rational Unified Process (RUP)
- Agile Unified Process (AUP)
- Basic Unified Process (BUP)
- Enterprise Unified Process (EUP)
- Essential Unified Process (EssUP)
- Open Unified Process (OpenUP)
- Oracle Unified Method (OUM)

**Ambler****Merise****SADT****HERMES...****Another classification: three large families:**

- ascending
- descending
- Agile

**N.B.** Sometimes: **Methodology = Language + Process**, Example: UML & Y

**When to use an X process?**

The choice of process depends on a number of factors:

- Type of project
- Project size
- Customer type
- Contract requirements
- Team skills

**1.6.3. Software Development Process Maturity**

Several models have been proposed for measuring process maturity. Here, we focus on the CMM.

**Capability Maturity Model (CMM)**

It is a system defined by the **Software Engineering Institute (SEI) at Carnegie Mellon University in Pittsburgh, Pennsylvania**, for the **U.S. Department of Defense**. Its aim is to improve the software development process. It enables an organization to measure (assess) its level of maturity and evolve its software development capability.

The CMM **has five levels of maturity**, each representing a different stage on the road to mature processes, i.e., processes that comply with a set of best practices observed worldwide in companies with a reputation for managing their processes well.

In particular, compliance with the CMM model is required for contracts with the US Department of Defense.

This model offers an organization seeking to improve its software development capabilities :

- A structure for assessing its level of maturity (evaluation).
- A set of documented procedures to improve its level of maturity.
- A set of control processes to validate the stages of this progression.

The CMM describes 5 levels of process maturity: **initial, reproducible, defined, controlled and optimized**.

- Level 1 (initial/chaotic) is the lowest level (no improvement has yet begun).
- Level 5 (optimized) is the highest.
- Most organizations are currently at level 1.

Improving an organization's maturity means :

- shorter development times.
- cost reduction.
- more efficient use of resources.



### Characteristics of the Maturity levels

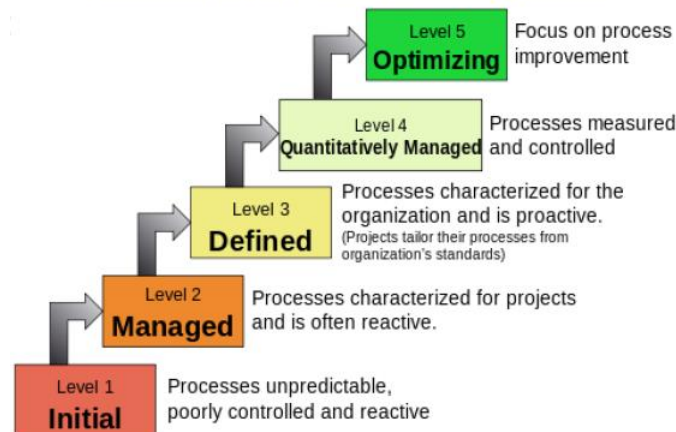


Fig.1.12. The CMM model.

To progress to a level higher than CMM, you must:

- Improves quality.
- Reduces development and maintenance costs.
- Improves lead time.

Maturity level	Deadline in months	Faults found	Delivered defects	Total cost
1	29.8	1348	61	\$5,440,000
2	18.5	328	12	\$1,311,000
3	15.2	182	7	\$728,000
4	12.5	97	5	\$392,000
5	9.0	37	1	\$146,000

Estimated impact on a 200,000 lines of code software development project

Tab.1.3. Example of a project at different CMM levels.

## 1.7. Software Development Tools and Professions

### CASE (Computer-aided software engineering) tools

CASE is the name given to the software used in GL's various activities (requirements, design, etc.).

Examples: compilers, editors, debuggers, etc.

The aim of CASE tools is to automate tasks and/or manage the development project.

### Main professions

The main software development professions are:

- Project Manager
- Analyst
- Software architect
- Developer
- Tester

## 1.8. Review

### Exercise1:

#### Statement:

1. What is software engineering?
2. Indicate the phase of the software life cycle where each of the following documents is produced: end user manual, architectural design, source code, specifications, detailed design, test report, documentation, specification.
3. What are the difficulties that characterize software development?
4. After delivering a software, the development team reports these problems:
  - The team could not evaluate the cost and deadline of this software.
  - The team could not use part of the software in another context.
  - The team found that it is difficult to correct some problems that emerge.
 What are the missing qualities in this software.
5. What are the three decisive criteria for a successful software project?
6. Lack of visibility in a software project is a common problem, why?
7. Why do we use a software process?
8. What is the major difference between classical methods and agile methodologies? Give an example for each of the two methods?

#### Solution:

1. GL is a field of engineering that allows the design, realization and maintenance of quality software systems.
2. End user manual (Implementation), architectural design (Design), source code (Coding>>Implementation), specifications (Analysis), detailed design (Design), test report (Test >>Implementation), documentation (Delivery>>Implementation), specification (Specification>>Analysis).
3.
  - Customers have difficulty describing their needs clearly enough for suppliers.
  - Needs are constantly changing as well as the environment.
  - Software is not tangible (intangible).
  - Difference in language between technical and non-technical people.
  - Difficulty in discovering errors before the product is delivered.
  - Software piracy causes enormous harm to suppliers.
4. - Measurability, - reusability, - Ease of maintenance
5. Quality, time, cost.
6. Software development is intellectual work.
7. A software process is used to structure and drive the production of good software.
8. -Strict classic methods with very clearly defined steps, as opposed to agile methodologies
  - Classic Cascade methods >> Cascade, V, Incremental, Spiral
  - Agile methodologies >> Scrum, Extreme Programming (XP), Rapid Application Development (RAD)

# *Chapter 2*

## *Modeling with UML*

## 2.1. Introduction

In this chapter, we are tackling modeling with UML. We start by illustrating the act of modeling, and the different levels of abstraction it involves. Then, we move to explain the philosophy of modeling with UML, which consists of using several views when addressing a system. After that, we present the fundamental concepts of the UML language and its different characteristics, as well as its diagrams.

## 2.2. System modeling.

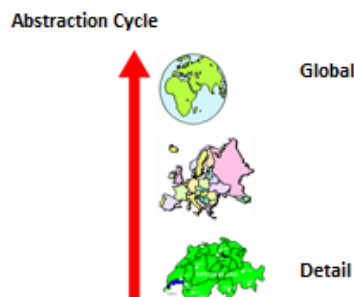
A model is a simplified representation of reality, designed to help us better understand a system under development.

Modeling is particularly useful for understanding complex systems, as it allows us to study them without having to apprehend them in their entirety.



**Fig. 2.1 - Modeling.**

The choice of models has a big impact on the way we approach a problem and find a solution. Each model can have a different level of accuracy, but the most effective ones never lose the sense of reality. Because no single model is sufficient on its own, it's best to break down a large system into a set of smaller, almost independent models in order to understand it better.



**Fig. 2.2 - Abstraction cycle example.**

## 2.3. Modeling levels

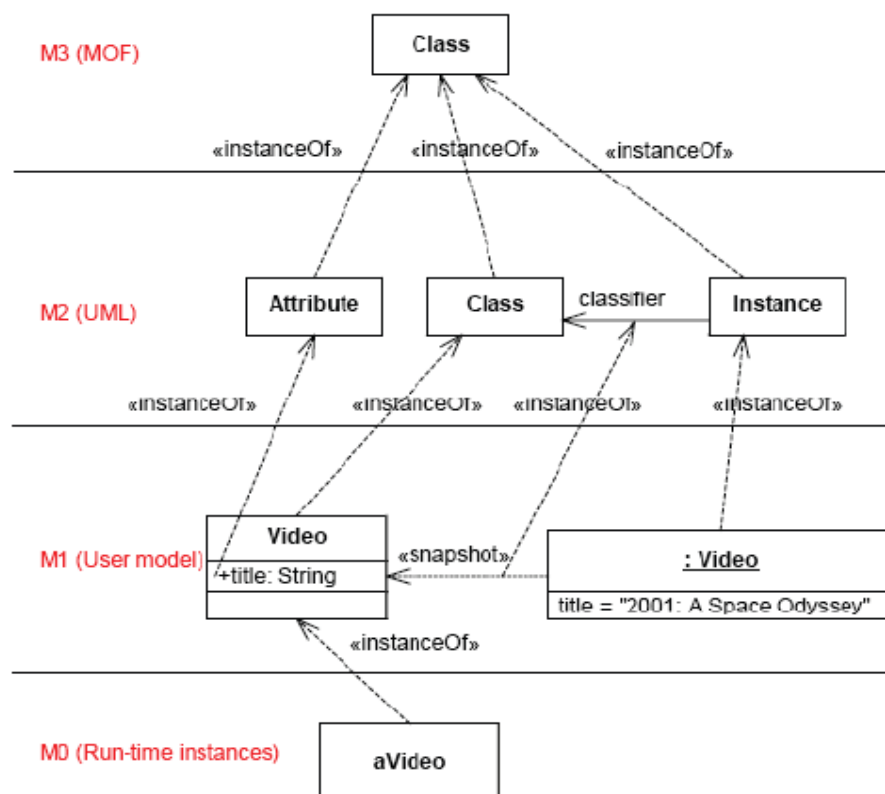
The Object Management Group (OMG) is an international non-profit organization created in 1989 at the initiative of major companies (HP, Sun, Unisys, American Airlines and Philips) to meet the growing need for standardization in object-oriented technologies.

The OMG has defined four modeling levels:

- **M0**: real system, modeled system
- **M1**: real system model defined in a certain language
- **M2**: meta-model defining this language
- **M3**: meta-meta-model defining the meta-model.

The M3 level is the MOF (Meta-Object Facility). The MOF serves as the basis for the definition of all other meta-models. It is meta-circular, meaning that it can define itself. The MOF is - for the OMG - the single meta-meta-model serving as the basis for the definition of all meta-models.

Of these levels, only the two layers *user model* (M1) and *run time instance* (M0) are intended for the user. In this course, we focus on these two layers.



**Fig. 2.3** - The OMG's four levels of abstraction.

## 2.4. Software Modeling Approaches

There are two famous and fundamentally different approaches to software design in use today:

- Function-oriented modeling
- Object-oriented modeling

### 2.4.1. Function-oriented modeling

The software is designed from a **functional** point of view, i.e. everything is a function.









- **Top-down** approach (Decomposition): In top-down decomposition, starting at a high-level view of the system, each high-level function is successively refined into more detailed functions.
- Centralized system status: this means that status information is available in a centralized shared data store.
- **DFD** is used.

#### Data flow diagram (DFD):

Semi-formal representation that describes the flow of input data through a set of functional transformations (**tasks** or **software components**) to give output results. DFDs are an intuitive and useful way of describing a system, and can be understood without special training.

#### DFD elements

DFD has two well-known notations: Gane&Sarson and DeMarco&Yourdon:

Notation	De Marco & Yourdon	Gane and Sarson
External Entity		
Process		
Data Store		
Data Flow		

Tab. 2.1 - DFD elements.

- **Processes (functions):** A process transforms an input stream into an output stream. It is also called a transformer. Each process has a unique name and number, which must be a meaningful verb (avoid vague names such as "data processing").

For example:

- Perform a calculation.
- Make decisions (i.e., determine product availability).
- Sort, filter and other group operations on data.

- **Data streams:** used to represent data flowing through the system. Data can be physical (a product) or abstract (a decision). Each data stream has a name (e.g. customer order). The name must be meaningful and unambiguous. A flow cannot start from an element and end at it. Two outgoing arrows mean: simultaneous or exclusive.
- **External elements:** Entities located outside the boundaries of the application under study (supply or retrieve data, trigger an action, etc.). They have a name: manager, customer, other system or department, etc. They can be sources or destinations (or both).
- **Data storage:** A set of data (digital or physical) in a repository (data storage location). This is the data required for system operation. Its contents can only be accessed by a process (files, database, list, etc.).

### *DFD good practices*

#### **-Correct data streams:**

- Between processes.
- From a data store to a process (data usage).
- From a process to a data store (update).
- From a source (external element) to a process.
- From a process to a destination.

#### **-Example of errors:**

- Deposits between external agents.
- Between data storage.
- Between an external agent and a data store.
- A process (or data store) that has only inputs (black hole).
- A miraculous process.
- Data flow between processes with 2 directions.
- Inputs and outputs are identical.
- Named process with a nominal phrase.
- Named data stream with a verb.
- A DFD should have no more than 7 processes (overloaded).

### *DFD Types*

There are 2 types of DFDs that offer different perspectives on a system or process: logical DFDs and physical DFDs.

- **Logical DFDs:** A logical DFD allows the modeling of a high-level view of the data flows that are needed to perform business or system processes, without more technical or implementation details. Logical DFDs can represent business activities such as order fulfillment at a warehouse, a customer making an online purchase or the intake of a patient at a healthcare facility.

Example: A simplified logical data flow diagram that shows the ordering process on an e-commerce website

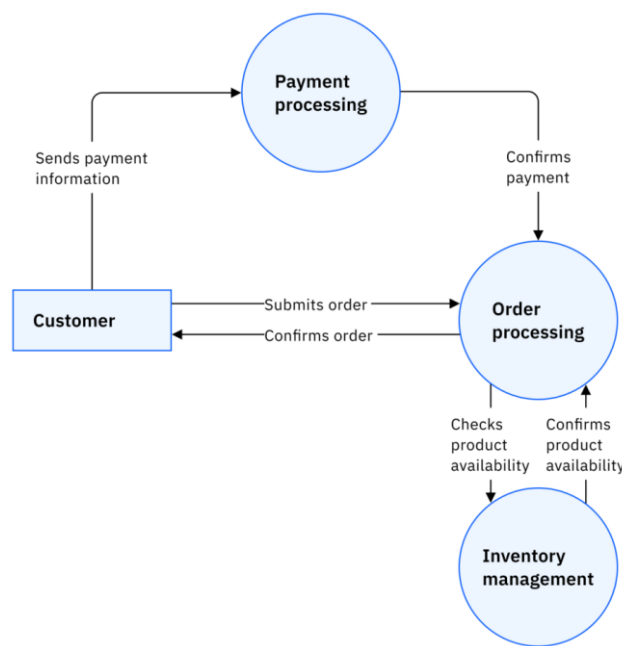


Fig. 2.4 – Example of logical DFD.

- **Physical DFDs:** A physical DFD illustrates the implementation of a system or process, including the needed software, hardware and files. Physical DFDs are often used to represent complex systems and workflows, such as how supply chain software maintains inventory at a warehouse or how electronic health records securely move through a hospital system.

Example: A simplified physical data flow diagram shows the ordering process on an e-commerce website.



Fig. 2.5 – Example of physical DFD.

### DFD levels

A DFD can be refined at several levels, where level 0 is the simplest, and level 2+ is the most complex.

#### -Level 0 DFD

Level 0 DFD also called **context diagram**. This is the most pared back form, and it shows only a single process and its external influences. It shows the entire system as a single process, and considered to be understandable to anyone.

#### -Level 1 DFD

This is still considered very top level, but it offers more detail than a level 0 DFD. It consists of a single process, which is then broken down into sub-processes. These, in turn, will be linked by different processes and data stores.



**-Level 2 DFD... and beyond**

DFDs can potentially go to level 3 and further, but level 2 is usually enough. As with level 1, level 2 is simply a step up in complexity again. This means more granular subprocesses, more data stores, and more flows.

**2.4.2. Object-oriented modeling**

Software is seen as a collection of **objects** (i.e. entities)

- **Bottom-up** approach
- **Distributed system state:** state information exists in the form of data distributed among several objects of the system.
- **UML** is used

In this course, we focus on Object-Oriented Modeling and UML.

**2.5. UML (Unified Modeling Language)**

UML (Unified Modeling *Language*) is a language for *visualizing, specifying, constructing* and *documenting* all aspects and artifacts of a software system [OMG10]. UML is the result of the merger of a number of other object-oriented graphical modeling languages (OMT, Booch and OOSE), and has rapidly become an essential standard. One of the ingredients of its success is that it is a generic language. The set of concepts and views making up UML's semantics can be adapted to all design domains and problems. UML enables modeling elements and their semantics to be defined, it is considered a powerful communication medium that facilitates the representation of object-oriented solutions, it offers a set of diagrams reflecting a particular aspect of the system, and its graphical notation enables an object solution to be expressed visually, facilitating the comparison and evolution of solutions. However, it should not be forgotten that UML notation is an object modeling language, not an object method. The evolution of the UML language is controlled by the OMG.

**2.6. History**

In the 90s, a number of methods were developed to meet the needs of software documentation and specification (OMT, OOSE, Booch, etc.). These methods were very similar; the same basic concepts could appear differently in the notation accompanying these methods, leading to confusion in the interpretation made by their users. This diversity created a problem of interoperability between tools, which slowed down the deployment of object-oriented development methods. The ambition of UML, the first version of which was published in 1997 by the OMG, is to bring together in a single notation the best features of the various object-oriented modeling languages. The UML specification defines the fundamental concepts of the object, as well as other concepts useful in software modeling (use cases, etc.). The figure below shows the chronic evolution of UML.

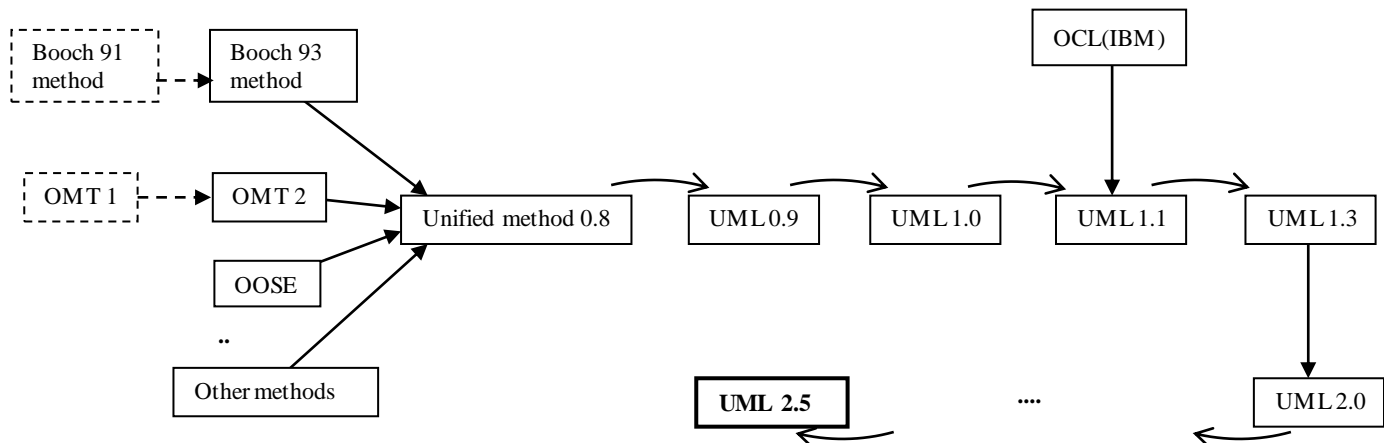


Fig. 2.6 - Historical development of UML.

## 2.7. Using UML

The UML language is a set of notations that have emerged in an attempt to standardize a multitude of modeling languages. Due to its many origins, UML has many different facets and is used in a wide variety of ways by different software development companies. It should be noted that when using UML, we must bear in mind that it is not a method or a process, so to take advantage of UML, we need to use a process that is very well mastered by the UML tool and includes certain characteristics described later. UML can be used in three different ways:

- ✓ **UML as illustration:** With this usage, UML can be used to communicate certain aspects of the system. The basis of illustration is selectivity: the aim is to use illustrations as a means of communicating ideas and alternatives about the work to be done. Illustration is a rather informal and dynamic activity. Illustrations are also useful in documentation, where communication takes precedence over completeness.
- ✓ **UML as a blueprint:** In contrast to illustrations, blueprints strive for completeness. The central idea is that they are developed by a designer whose job it is to construct a detailed specification of what the programmer will have to code. The design must be sufficiently complete that no decision is left to the programmer.
- ✓ **UML as a programming language:** use UML as a real programming language, with a much more precise and rigid syntax. This usage automatically produces executable code from a UML source.

The first is direct, where you start by writing a schema of the software in UML, then develop and implement based on this schema. This is classic direct design. The other is reverse engineering, in which UML is used as a tool of understanding rather than design.

## 2.8. Advantages and disadvantages of UML

### 2.8.1. The strengths of UML

- ✓ UML is a semi-formal, standardized language
  - improved precision
  - a guarantee of stability
  - encourages the use of tools
- ✓ UML is a powerful communication tool
  - It frames the analysis.
  - It facilitates the understanding of complex abstract representations.
  - Its versatility and flexibility make it a universal language.

### 2.8.2. The weaknesses of UML

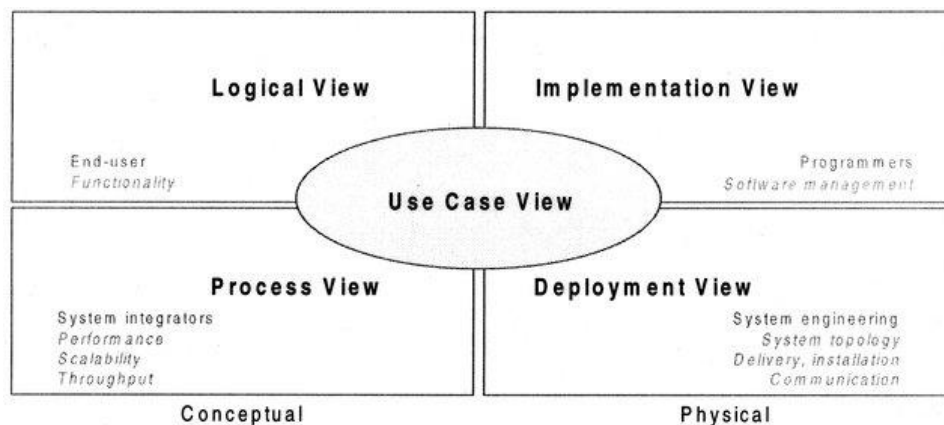
- ✓ Putting UML into practice requires a learning curve and a period of adaptation.
  - Even if Esperanto is a utopia, the need to agree on common modes of expression is important in IT.
- ✓ Process (not covered by UML) is another key to project success.
  - But integrating UML into a process is not trivial, and improving a process is a complex and time-consuming task.
    - The authors of UML are well aware of the importance of this process, but the industrial acceptability of object modeling depends first and foremost on the availability of a high-performance, standard object analysis language.

## 2.9. Modeling with UML

This section is dedicated to the presentation of modeling with the UML language. It presents the various diagrams and their uses.

### 2.9.1. Notion of view

UML is based on the notion of "view", which reflects the fact that there are several ways of looking at a system. UML follows **Kruchten's** "4+1 views" model. These views are embodied in 14 diagram types in UML.



**Fig. 2.7** - Kruchten's "4+1 views" model.

- ✓ **The logical view** focuses on the static aspects of the system in terms of objects and classes, enriched by dynamic descriptions in terms of activities, sequences and communications.
- ✓ **The process view** focuses on execution flows and their synchronization, in terms of tasks, threads and processes, with their states and interactions.
- ✓ **The implementation view** describes the organization of software components.
- ✓ **The deployment view** specifies hardware resources and the implementation of software components on these resources.
- ✓ **The use case view** gives an overview of the system's purpose in terms of actors, functionalities and use scenarios.

### 2.9.2. UML views

System modeling with UML is based on a multi-view approach, which enables different aspects of a system to be represented in a complementary way. According to the UML approach, these views are organized into two main categories: **structural views** and **behavioral views**. The representation of a view is guaranteed by the use of a set of models called diagrams, each of which is specially designed to model a specific aspect. This separation into different views enables us to better understand and control the complexity of systems, by offering specific perspectives adapted to the various aspects to be modeled.

## 2.10. UML diagrams

A UML diagram is a graphical representation that focuses on a specific aspect of the model; it is a perspective of the model. Each type of UML diagram has a specific structure and conveys specific semantics. Combined, the different UML diagram types offer a complete view of the static and dynamic aspects of a system. An important feature of UML diagrams is that they support the notion of abstraction, which helps to simplify the representation and the control of complexity in the expression and elaboration of object solutions.

UML 2 offers us 14 diagram types, representing as many distinct views for modeling particular system concepts. They are divided into two main groups:

### Structural or static diagrams

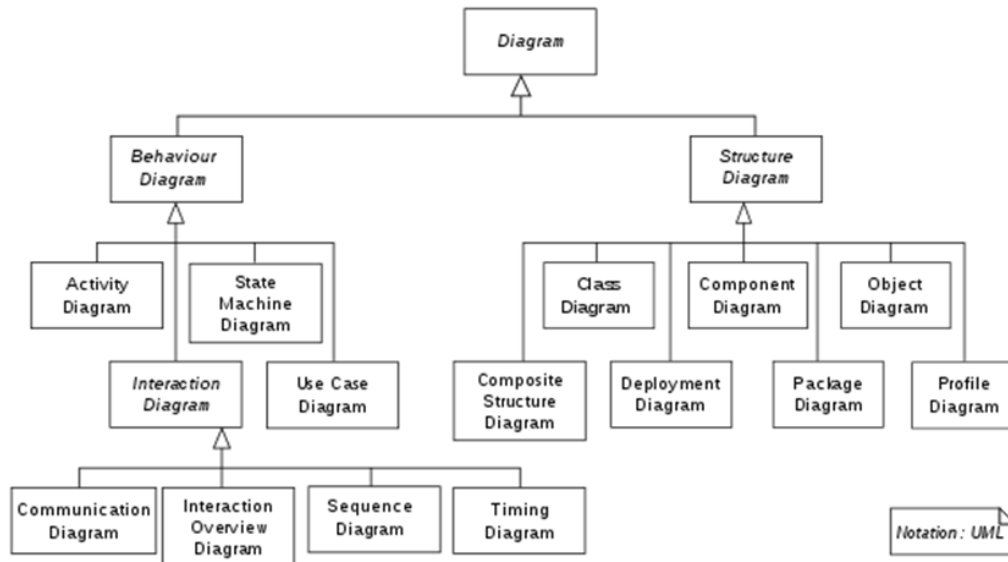
- ✓ class diagram
- ✓ object diagram
- ✓ component diagram
- ✓ deployment diagram
- ✓ package diagram
- ✓ composite structure diagram

### Behavioral or dynamic diagrams

- ✓ use case diagram
- ✓ activity diagram
- ✓ state-transition diagram
- ✓ Interaction diagrams
  - sequence diagram
  - communication diagram

- global interaction diagram
- time diagram

The figure below shows the organization and classification of UML diagrams:



**Fig. 2.8** - Classification of UML diagram types.

## 2.11. Review

### Exercise1:

#### Statement:

1. Quel est l'objectif de l'utilisation d'un DFD ?
2. What are the different UML diagrams?
3. Is UML a language or a method? Why?
4. Is UML a semi-formal or formal language? justify?
5. Is it possible to generate a complete C++ application from a UML model?
6. Expliquer l'approche Button-up, et Top-down, dans quel paradigme on trouve chacune d'elles, donner un exemple de formalisme utilisé dans chacune d'elles?

#### Solution:

1. Quel est l'objectif de l'utilisation d'un DFD ?
  - Modéliser la circulation des données dans un système.
2. What are the different UML diagrams?
  - *Class diagram, Object diagram, Component diagram, Deployment diagram, Package diagram, Composite structure diagram, Use case diagram, Activity diagram, State-transition diagram, Communication diagram, Sequence diagram, Global interaction diagram, Time diagram.*
3. Is UML a language or a method? Why?
  - *language, it has no process*
4. Is UML a semi-formal or formal language? justify?
  - *semi-formal,*
  - *well-defined syntax, imprecise semantics.*
5. Is it possible to generate a complete C++ application from a UML model?
  - *Yes, like Java code generation, but there are still limits that require developer intervention.*
6. Expliquer l'approche Button-up, et Top-down, dans quel paradigme on trouve chacune d'elles, donner un exemple de formalisme utilisé dans chacune d'elles?
  - Top-down, decomposition , fonctional, DFD
  - Bottom-up, set of entities, object, UML

# *Chapter 3*

*Use case diagram: functional  
view*

### 3.1. Introduction

This chapter introduces the most concepts of UML use case diagrams; an important tool for functional requirements modeling in software engineering.

### 3.2. Importance of the functional view in modeling

The development or enhancement of a system must meet specific needs, defined by the client who orders the software. The client intervenes throughout the project; they express the needs and validate solutions and the final product. The general contractor, such as an IT services company (SSII), is chosen for its technical skills and ability to understand the customer's needs. Understanding the client's field of activity is crucial to adapting the proposed solutions. It is therefore important to supplement the specifications with discussions with users, and to analyze relevant documents to guarantee that all the information required for system design is gathered.

### 3.3. Objective

In UML, requirements are modeled using use case diagrams. The use case diagram answers the question "WHAT" to develop. Indeed, before developing a system, it is crucial to define precisely what it is to be used for, i.e. what needs it is to meet. Modeling requirements enables designers to list and organize different functionalities expected. The use case diagram provides a simple notation, understandable by all stakeholders including the client.

### 3.4. Use case diagram basic elements

A use case diagram is mainly composed of the following elements (graphically depicted in Fig.3.1):

#### 3.4.1. Use cases

A use case is a service rendered to the user. It involves a series of more elementary actions.

#### 3.4.2. Actor

An actor is an entity outside the system that interacts directly with it.

#### 3.4.3. System

A system represents an application in the UML model. It is identified by a name and contains a set of use cases.



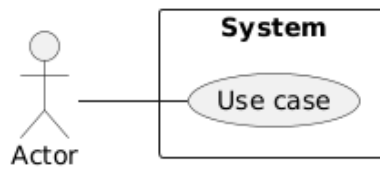


Fig. 3.1 – Main elements of a use case diagram.

## 3.5. Relationships in use case diagrams

### 3.5.1. Relationships between use cases and actors

Actors involved in a use case are linked to it by an **association**. An actor can use the same use case several times.

Example: a "Car Owner" can do a "Book Service Appointment" use case.

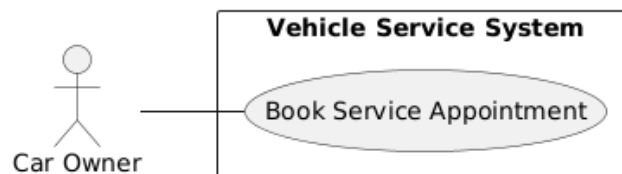


Fig. 3.2 - An association between an actor and a use case.

### 3.5.2. Relationships between use cases

**Inclusion**: This relationship is used when a use case needs another use case (obligatory)

Example: a Car Owner must first do the mandatory "Log In" use case before they can either "Book Service Appointment" or "Renew Car Registration".

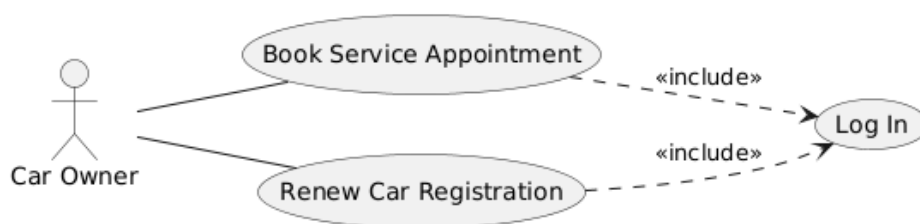
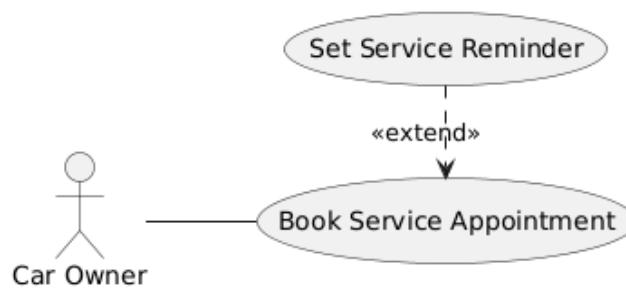


Fig. 3.3 - Example of "include".

**Extension**: This relationship is used when a use case extends another use case (optional).

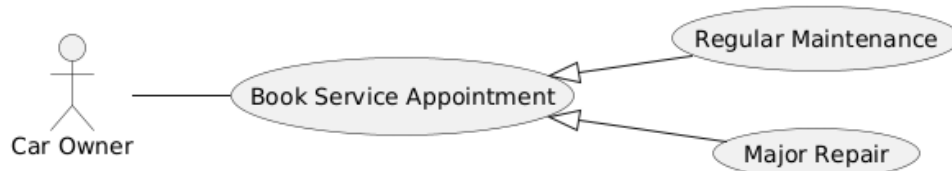
Example: a Car Owner can do the "Book Service Appointment" use case independently, while the "Set Service Reminder" functionality is entirely optional (the owner may choose this additional feature).



**Fig. 3.4** - Example of "extend".

**Generalization:** This relationship is used when a use case is a particular case of another use case (kind of).

Example: "Regular Maintenance" and "Major Repair" are specialized kinds of "Book Service Appointment" use case, where each represents a specific type of appointment a Car Owner can book.



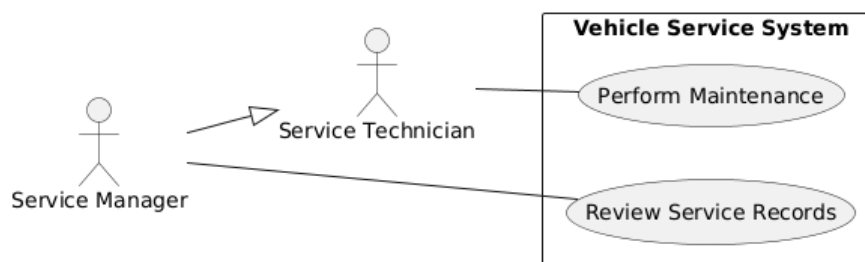
**Fig. 3.5** - Example of generalization.

All these relationships enable reusability and decomposition of use cases. Reusability is useful to not specify the same behavior several times. Decomposition is useful to avoid too complex cases by breaking them up to simpler ones.

### 3.5.3. Relationships between actors

Generalization is the only possible relationship between actors. It is an "is-a" relationship which mean an actor can be substituted by another one.

Example: the "Service Manager" is a specialized type of "Service Technician" who inherits all technician capabilities (including performing maintenance), while having additional administrative privileges to review service records.



**Fig. 3.6** - Generalization relationship between actors.

## 3.6. Identifying and classifying actors

### 3.6.1. Identifying the actors

Actor identification consists in defining the main roles involved in the system, with a clear distinction between **role** and **physical person**. One person can fill several roles, and the same role can be played by several persons, represented by a single actor.

Candidate actors typically include all direct human users of the system—so it's important to identify every possible user profile (e.g. administrator, maintenance operator, etc.). Additionally, any external systems that interact directly with the system under consideration should also be identified as actors, especially those that communicate through bidirectional protocols. In few words, to identify actors, it is useful to imagine the boundaries of the system. Anything external that interacts with the system is an actor, while anything internal is a system feature.

### 3.6.2. Main and secondary actors

An actor is considered to be the **main** actor in a use case when he initiates the exchanges necessary for its realization. On the other hand, **secondary** actors are solicited by the system and do not initiate interactions. Often, these secondary actors are other IT systems interconnected with the system under development.

Example: the "Car Owner" is the primary actor who initiates the action to file an accident report, while the "Insurance Company" is marked as a secondary actor that participates in but doesn't initiate the use case.



Fig. 3.7 - Main and secondary actors.

## 3.7. Identification and description of use cases

### 3.7.1. Identification of use cases

There is no systematic way to identify use cases. The goal is to guarantee that the complete set of use cases captures all the functional requirements of the system. Each use case should represent a separate business function from the perspective of an external actor interacting with the system. For every actor, it's important to identify their business goals when using the system and to determine the functional services the system is expected to deliver, as outlined in the specifications. We need also to choose the right level of abstraction to avoid redundancy and get a balanced number of use cases. Use cases should be named using an infinitive verb.

### 3.7.2. Description of use cases

Each use case can be documented in detail to avoid any ambiguity regarding its progress. Activity diagrams (will be discussed later) can be used for this purpose. Besides the graphical diagram, a textual description is often used, it may contains:

- Use case name
- Actors involved
- Description
- Preconditions
- Postconditions
- Nominal scenario
- Alternative scenarios

Example:

<b>Use case:</b> Issue bike	
<b>Preconditions:</b> 'Maintain bike list' must have been executed	
<b>Actors:</b> Receptionist	
<b>Goal:</b> To hire out a bike	
<b>Overview:</b> When a customer comes into the shop they choose a bike to hire. The Receptionist looks up the bike on the system and tells the customer how much it will cost to hire the bike for a specified period. The customer pays, is issued with a receipt, then leaves with the bike.	
<b>Cross-reference:</b> R3, R4, R5, R6, R7, R8, R9, R10	
<b>Typical course of events:</b>	
<b>Actor action</b>	<b>System response</b>
1 The customer chooses a bike	
2 The Receptionist keys in the bike number	3 Initiate 'Find bike'
4 Customer specifies length of hire	
5 Receptionist keys this in	6 Displays total hire cost
7 Customer agrees the price	
8 Receptionist keys in the customer identification	9 Initiate 'Maintain customer list'
10 Customer pays the total cost	
11 Receptionist records amount paid	12 Initiate 'Print receipt'
<b>Alternative courses:</b>	
Steps 7–12	The customer may not be happy with the price and may terminate the transaction

**Fig. 3.8** – Textual description of a Use case.



### 3.9. Review

#### Exercise1:

##### Statement:

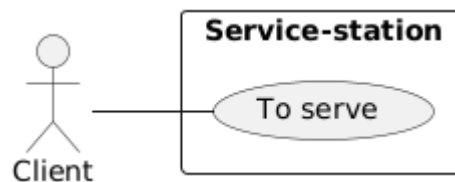
Consider the computer system that manages a gasoline service station.

We're interested in modeling how a customer fills up on gas.

1. The customer uses gasoline as follows. He picks up a pistol attached to a pump and pulls the trigger to take the petrol. Who is the actor in the system? Is it the customer, the gun or the trigger?
2. The gas station attendant can help himself to gas for his car. Is this a new actor?
3. The station has a manager who uses the computer system for management operations. Is this a new actor?
4. The service station has a small vehicle maintenance workshop run by a mechanic. The manager is replaced by a workshop manager who, in addition to running the workshop, is also a mechanic. How can we model this?

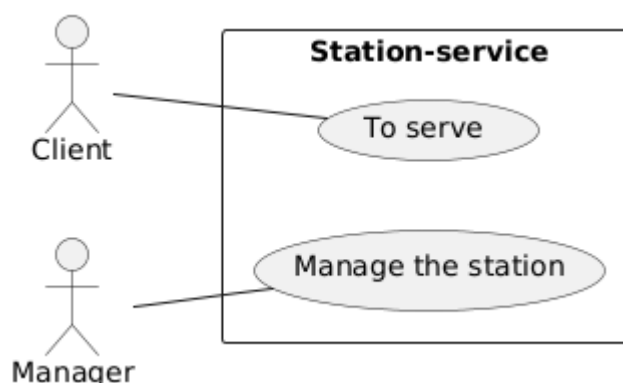
##### Solution:

1.



**Fig. 3.10** – An actor and a role.

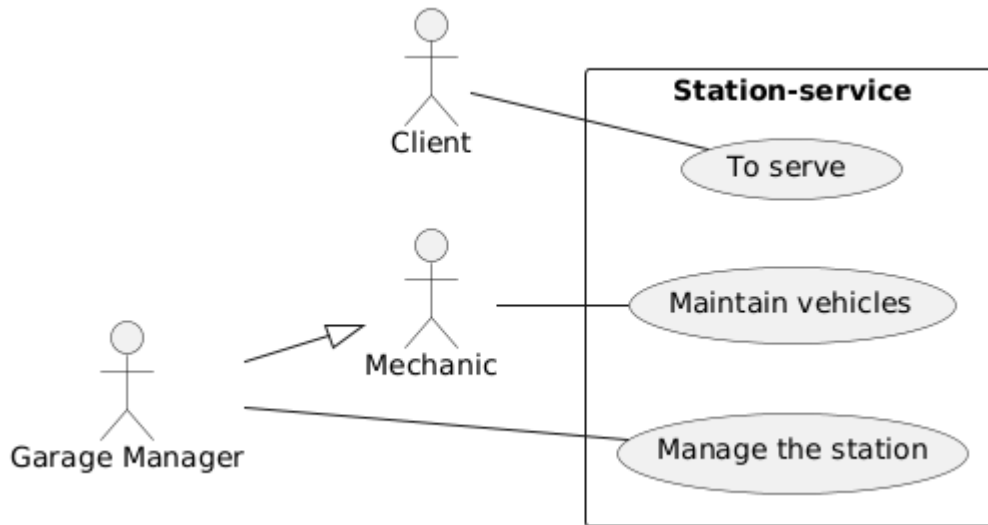
2. An actor is characterized by the role he plays in the system. The pump attendant, although a different person from the customer, plays an identical role when he helps himself to petrol. For the "Filling up" case, it is not necessary to create an additional actor representing the attendant.
3. Service station management defines a new functionality to be modeled. The manager takes on the main role; he's a new actor.



**Fig. 3.11** – Two actors for two roles.

4. The station offers a third service: vehicle maintenance. The computer system has to handle this additional functionality. A new actor appears: the mechanic. The manager is now a garage manager who is a mechanic with the ability to manage the station. There is thus a

generalization relationship between the Mechanic and Garage Manager actors, meaning that the Garage Manager can, in addition to managing the station, also maintain vehicles.



**Fig. 3.12** – Generalization relationship between actors.

# *Chapter 4*

*UML class and object diagrams:  
static view*



## 4.1. Introduction

In this chapter, we discuss modeling the static design of systems using UML class and object diagrams.

## 4.2. Importance of the static view in modeling

The static view concentrates on the system architecture, systems entities and the relationships between these entities without regard to any dynamic aspect. UML uses static diagrams such as class and object diagrams. They are beneficial for an early stage design and documentation showing the overview of the structure and connectivity between the system components.

## 4.3. UML Class Diagrams

Class diagrams in UML are used to model the internal structure of systems by abstracting the classes, the attributes, methods and the links among them.

### 4.3.1. Objective

Class diagrams show the internal structure of a system by providing a detailed description of the system's objects and the links between them. While use case diagrams illustrate what the system is for, class diagrams focus on **how** it is organized. The system itself is made up of objects that interact with each other and with actors to realize the functionalities defined in the use cases.

### 4.3.2. Basic notions

#### Concepts and instances

A **concept** is an abstract idea. An **instance** is the concrete realization of a concept.

Example:

- Concept : **Book**
- Instance: The novel "1984", written by *George Orwell*, is an instance of the book concept

A **class** is an incarnation of a concept. An **object** is an instance of a class.

Example:

- Class: **Car**
- Objects: *Toyota Corolla 2022*, *Ford Mustang 2021*

An **association** is a relationship between concepts. A **link** is an instance of association.

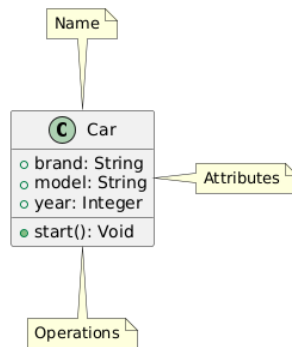
Example:

- Association: The concept of "working" links an Employee to a Company.
- Link: instance *Alice works at Google*, *Bob works at Microsoft*.

## Classes and objects

A class groups a set of objects sharing the same characteristics (i.e. properties). It defines them by a **name**, **attributes** and **operations**. As modeling progresses, some of this information may remain unknown. It is also possible to add other elements, such as **responsibilities** or **exceptions**, to refine the description.

Example: A simple class modeling



**Fig. 4.1** - An example of a class.

### Visibility

Visibility is the mechanism used in UML (and OOP in general) to implement the encapsulation principle shown the previous chapter. We use visibility (access modifiers) on properties or classes (there is no default visibility):

- **Public** "+": property or class visible everywhere
- **Protected** "#": property or class visible within the class and by all its descendants.
- **Private** "-": property or class visible only within the class
- **Package**, or "~": property or class visible only in the package

### 4.3.3. Class properties

The properties of a class are their attributes and operations.

#### Attributes

An **attribute** represents **data** linked to a class. In the model, you can specify the type of the attribute, its initialization and visibility (access modifiers). When the class is instantiated, attributes take on values, functioning as variables associated with objects. The general syntax for declaring an attribute is:

**visibility attributeName : className [multiplicity] = valueInitialisation**

Example: a `Car` class could have attributes such as `name`, `price` and `customerReviews`.

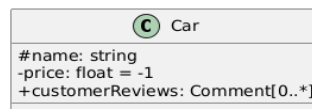


Fig. 4.2 - Attributes.

### *Class attributes*

By default, the values of class attributes vary from one object to another. However, it may be necessary to define a class attribute that retains a single value, shared by all instances. Graphically, such an attribute is **underlined** to distinguish it.

### **Operations**

An **operation** represents a treatment the class can do, i.e. actions that its objects can perform. It is defined by its name, parameter types and return type. The declaration of an operation follows this syntax:

**visibility operationName (parameters) : ReturnClass**

with the following list of parameters:

**className1 parameterName1, ..., classNameN parameterNameN.**

Example: the `start()` method in the `Car` class enables the `Car` object to start.

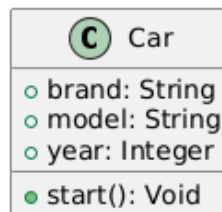


Fig. 4.3 - Operations

**N.B.** there is a small difference between the term method (already used in OOP) and operation in UML. An operation defines the method signature<sup>1</sup>, regardless of its implementation, and operations can be specified in any language.

### *Class operations*

Like class attributes, a class operation belongs to the class itself, not to its instances. It cannot therefore access the attributes of individual objects in the class. It is also **underlined** to distinguish it.

<sup>1</sup> the name of the method and the description (i.e., type, number, and position) of its parameters

#### 4.3.4. Class relationships

There are several types of relationship:

- Association for general links
- Aggregation for "has-a" relations with independent parts
- Inheritance when one class generalizes another
- Dependency for temporary or behavioral reliance

#### Association

An association is a structural relationship between objects, used to represent possible links between objects of different classes. The associations are often **binary**, i.e. relate two classes.

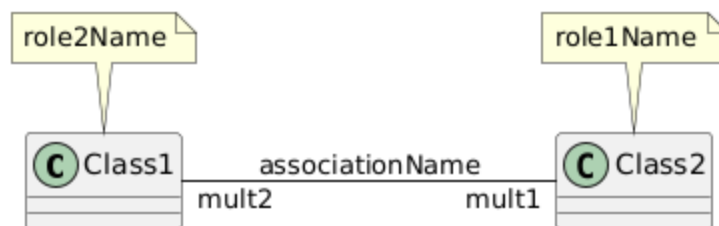


Fig. 4.4 - The association relationship.

#### • The roles and multiplicities

**Role** in an association represents the function that one class plays vis-à-vis another in the relationship. It is indicated at each end of the association, close to the class concerned, and helps clarify the nature and purpose of the relationship from the point of view of each class involved. It is optional.

**Multiplicity** specifies how many objects of a class can be linked to an object of another class. Its syntax is **Min.. Max**:

"\*" in place of **Max** indicates several without specifying a number.

"n..n" can be abbreviated to "n", and "0..\*" is simplified to "\*".

A multiplicity must be mentioned, otherwise it is considered "1" by default.

Example: a Person can own several Car(s), but a Car belongs to only one Person.



Fig. 4.5 - Example of association with role and multiplicity.

#### • Association navigability

Navigability defines the direction in which an association can be browsed. When the navigability is in one direction the association is called **Unidirectional**; it is depicted by an

arrow (c.f. Fig. 4.6). When the navigability is in both directions the association is called **Bidirectional**; it is represented without arrows (c.f. Fig. 4.5).

Example: Given a Car, you can find all its ServiceRecords, but given a ServiceRecord alone, you cannot navigate back to determine which Car it belongs to.



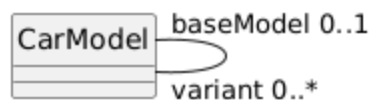
**Fig. 4.6** - Example of a navigable association.

**N.B.** navigability is different from reading sense shown in Fig. 4.5 on association "owns". This latter has no semantics; it is just used for helping reading the association.

### Reflective associations

An association is **reflective** when it goes from a class and returns to the same class.

Example: The relationship represents base models and their variants. It creates a hierarchy where car models can have variant models (e.g. a Toyota Corolla base model has variants such as Corolla LE, Corolla XSE, etc.)

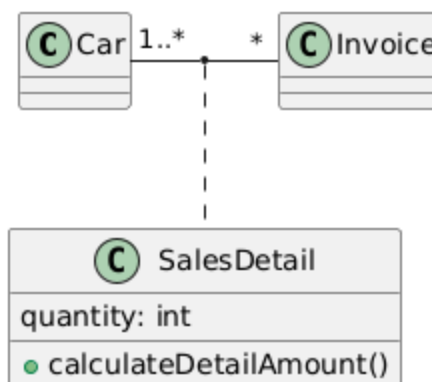


**Fig. 4.7** - Example of reflexive association.

### Association class

An association can have its own attributes which cannot be put on any of the classes it links. This is what we call it a "**class-association**".

Example: This diagram shows cars and invoices with an association class (SalesDetail) that illustrates additional information about each car-invoice relationship.



**Fig. 4.8** - Example of an association class.

## Qualified association

Qualifying an association limits its impact by restricting the relationship between two classes to a specific aspect.

Example: to model the relationship between a person and a bank, we introduce an Account class to indicate that the association is limited to account ownership, without including other banking activities.

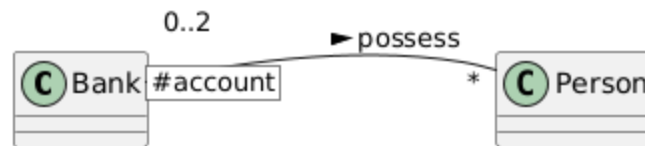


Fig. 4.9 - Example of a qualified association.

## N-ary associations

An n-ary association links more than two classes. It is represented by a central rhombus, which may contain a class-association. n-ary associations are rare and are mainly used when multiplicities are all equal to "\*". Instead, it is often more practical to use several binary relations or class-associations.

Example: This diagram shows an N-ary association between Cars, Drivers, and RaceTracks, representing how these three entities relate in race entries.

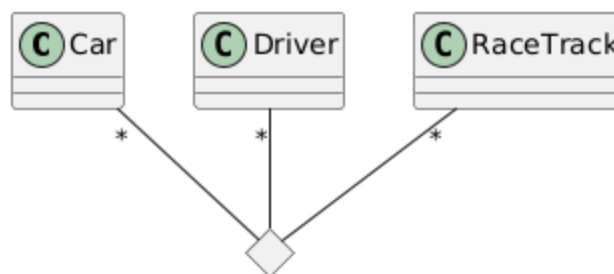


Fig. 4.10 - Example of n-ary association.

## Aggregation and Composition

**Aggregation** is a particular type of association that illustrates the inclusion relationship of an element in a set. It is represented by an empty diamond on the side of the aggregate. This relationship shows a link between a set (the aggregate) and its parts (the aggregates).

The **composition** relationship, represented by a solid diamond, describes a strong structural relationship between instances. Destroying or copying a composite object leads to the destruction or copying of its components. Furthermore, an instance of a component can only belong to one composite object at a time.

Example: A team contains several players, but the players can exist without the team. A house contains rooms, and if the house is destroyed, so are the rooms.

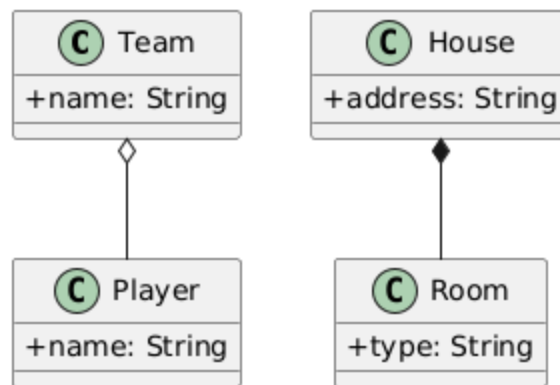


Fig. 4.11 - Example of aggregation and composition.

### Inheritance relationship

Inheritance is a specialization/generalization relationship in which specialized elements inherit the attributes and operations of more general elements.

Example: an `ElectricCar` class automatically inherits the `make`, `model` and `start()` properties from `Car`.

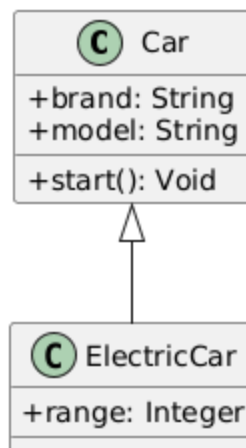


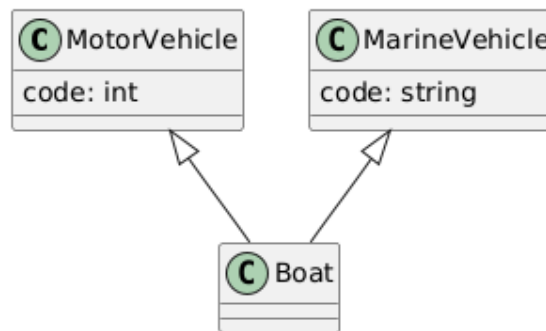
Fig. 4.12 - Example of inheritance.

The child class (i.e. the specialized class, here `ElectricCar`) has access to all the properties (attributes and operations) of its parent classes (i.e. the general class, here `Car`), except for private ones.

The inheritance philosophy in UML is the same existing in the OOP paradigm, i.e. regarding redefinition, overloading and substitution.

A class can also inherit from several parent classes, which is called **multiple inheritance**. C++ supports the implementation of multiple inheritance, whereas Java does not.

Example:



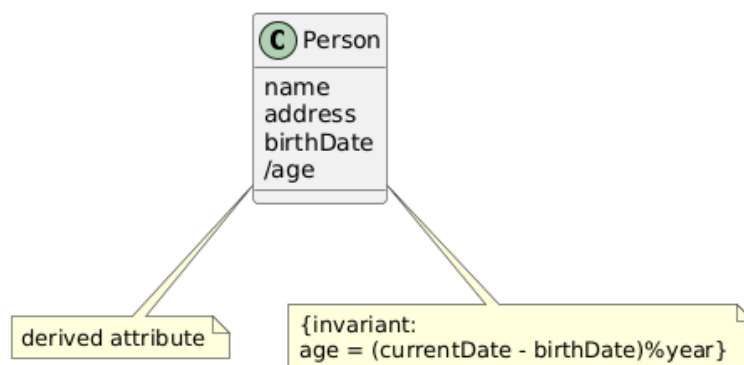
**Fig. 4.13** - Example of multiple inheritance.

#### 4.3.5. Other elements

##### Derivative elements

**Derived attributes** are calculated from other attributes and calculation formulas, and are represented by a "/" in front of their name. During the design phase, they can be used as reference points until the calculation rules are established.

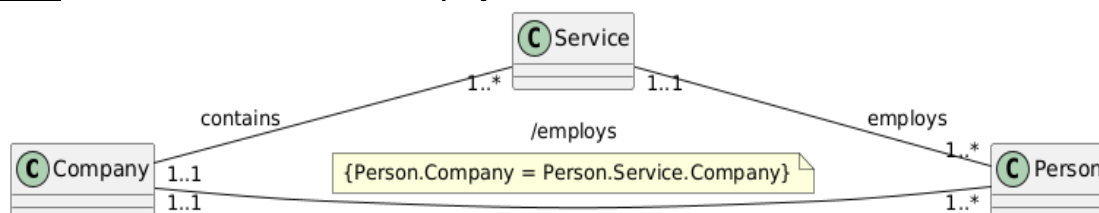
Example:



**Fig. 4.14** - Example of a derived attribute.

A **derived association**, conditioned or deduced from other associations, is also represented by the "/" symbol.

Example: the derived association "/employs".



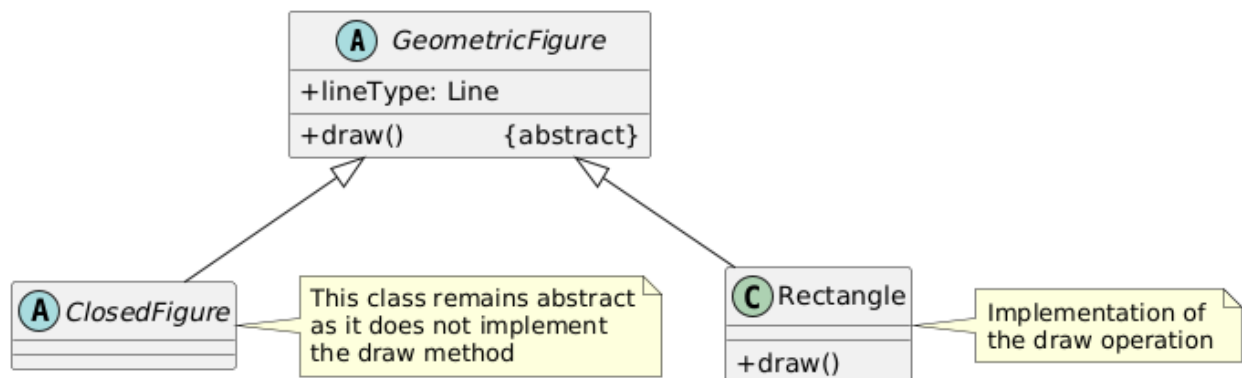
**Fig. 4.15** - Example of a derived association.



## Abstract classes

A method is said to be "**abstract**" when we know its signature, but not its implementation. It is the child classes that must define these abstract methods. A class is said to be abstract if it contains at least one abstract method, or if it inherits from an as yet unimplemented abstract method.

Example:

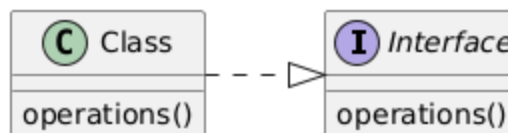


**Fig. 4.16** - Example of abstract classes.

## Interface

An interface defines sets of operations (providing a coherent service) which other model elements, such as classes, or components must implement. An implementing class realizes an interface by overriding each of the operations that the interface declares. The **realization** relationship is used between an interface and the class that implements it. The interface has the form of a class, but is preceded by the "**interface**" stereotype.

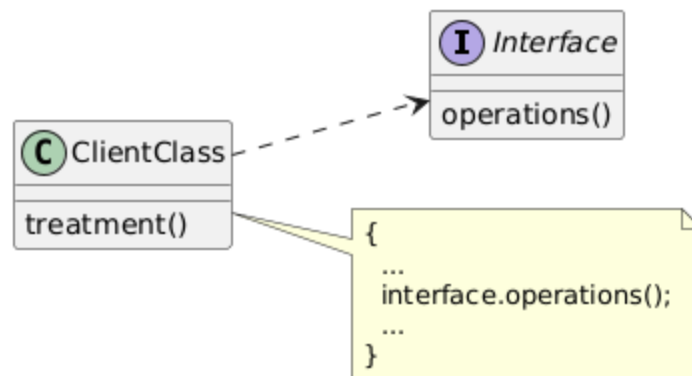
Example:



**Fig. 4.17** - Example of interface.

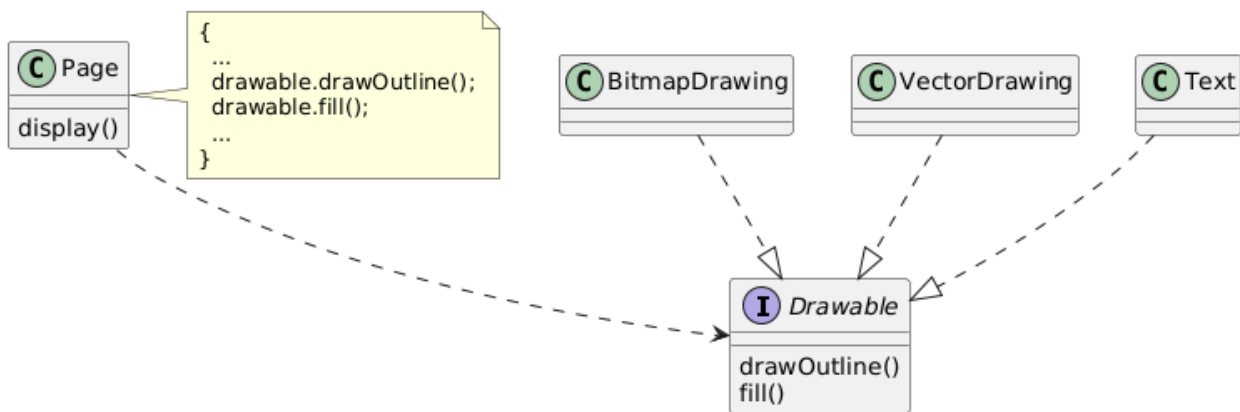
A class is called an "**interface client class**" when it depends on an interface (required interface) to perform its operations. In this case, a dependency relationship is established between the client class and the interface. Any class that implements this interface can then be used by the client class.

Example:



**Fig. 4.18** - Interface client class.

And this is a full example of interface modeling:



**Fig. 4.19** - Example of interface modeling.

## Package

A package is used to organize modeling elements into groups. A package can contain classes, use cases, interfaces, other packages, etc. Model elements use the encapsulation mechanism to regulate their access to each other.

Example: Package1 uses package Package2. Class Class1 defines four attributes with different visibility. The private attribute attribute1 is only accessible in class Class1. Class2 can access to attribute4, 3 and 2. Class3 can access to attribute4 and 2. Class4 can only access to attribute4

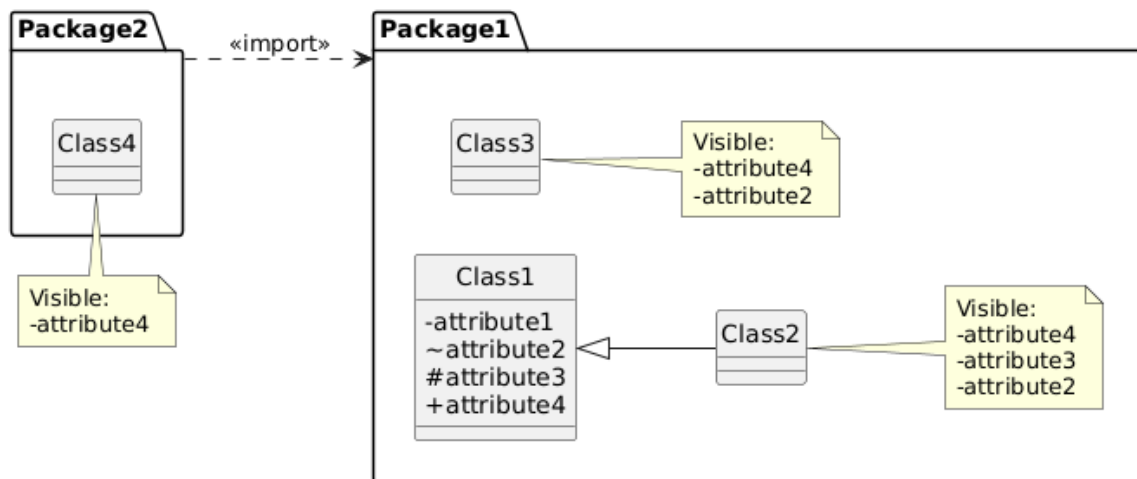


Fig. 4.20 - Example of packages (and encapsulation).

#### 4.3.6. Building a class diagram

**Modeling Purpose:** Before starting the construction of a class diagram, you need to fix the purpose of your diagram. There are three main perspectives (usage):

- *Specification:* Focus on class interfaces.
- *Conceptual:* Capture domain concepts and relationships.
- *Implementation:* Detail class contents and structure.

**Steps to build a class diagram:** To build a class diagram, these steps are generally followed:

- Define the Purpose and Scope.
- Identify and Label Classes.
- Add Attributes.
- Add Operations.
- Show Relationships.
- Add Multiplicity.
- Organize the model
- Review, Refine, and Iterate.

## 4.4. UML Object Diagrams

### 4.4.1. Objective

**Object diagrams** represent specific instances of a system at a given point in time. They illustrate a snapshot of the system, providing a concrete example based on the class diagram.



**Fig. 4.21** - Example of an object diagram.

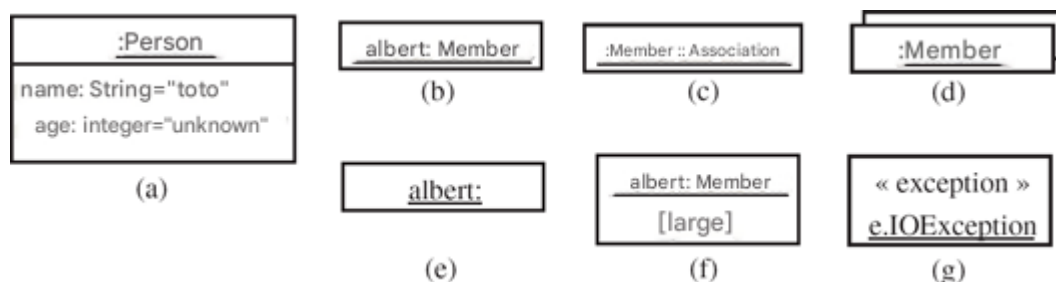
Using an object diagram allows, depending on the situation, to illustrate the class model (by showing an example that explains it), clarify certain aspects of the system (highlighting details not visible in the class diagram), and express an exception (by modeling special cases, non-generalizable knowledge, etc.).

### 4.4.2. Object representation

Like classes, we use compartmentalized frames. However, object names are underlined, and you can add your identifier before the name of your class.

Example:

- The values (a) or state (f) of an object can be specified.
- Instances can be anonymous (a,c,d), named (b,f), orphaned (e), multiple (d) or stereotyped (g).



**Fig. 4.22** - Object representations.

### 4.4.3. Class and object diagrams

The class diagram shows the structure of classes and the relationships between them, while object diagram depicts the real objects and their links.

Example: consider this class diagram:



Fig. 4.23 - Example of a class diagram.

- Object diagram consistent with this class diagram :

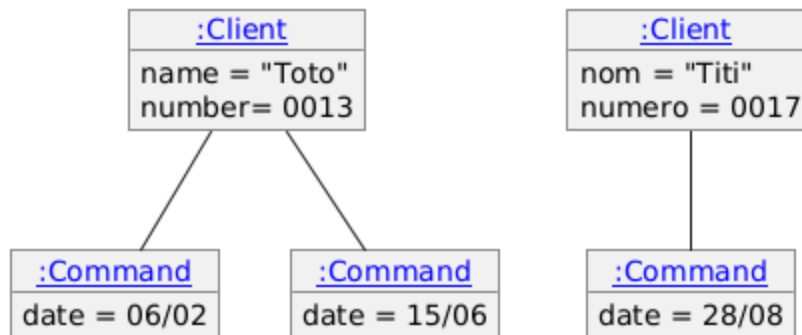


Fig. 4.24 - Coherent object diagrams.

- Object diagram inconsistent with class diagram :

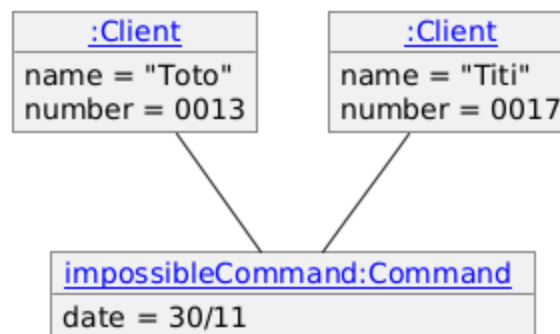


Fig. 4.25 - Inconsistent object diagram.

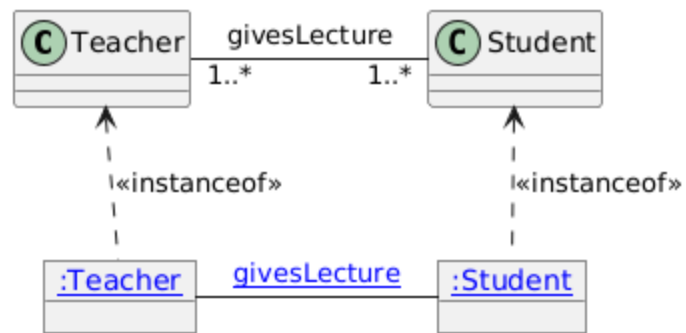
#### 4.4.4. Links

A link is an instance of an association. A link is represented as an association, but if it has a name, it is underlined.

There are no multiplicities on links in the object level, because they are always equal 1 (one object related to one object).

#### 4.4.5. Instantiation dependency relationship

The instantiation dependency relationship (stereotyped) describes the relationship between a classifier and its instances. In particular, it links associations to links and classes to objects.



**Fig. 4.26** - The instantiation dependency relationship.

## 4.5. Review

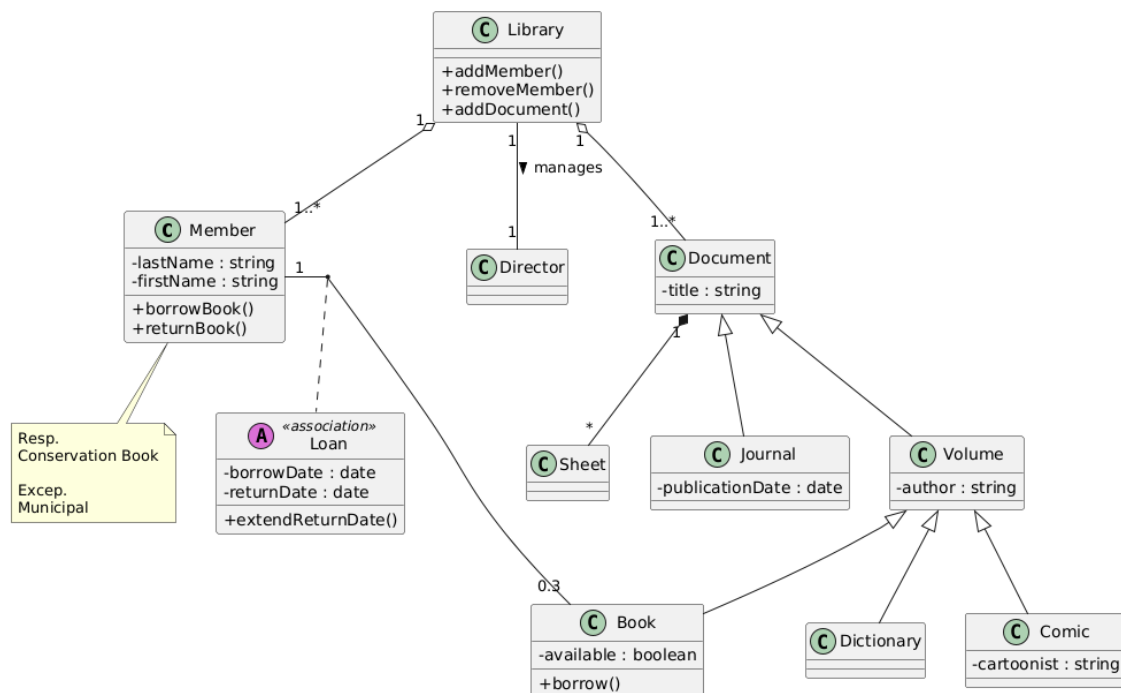
### Exercise1: (Class diagram)

#### **Statement:** (Management of a municipal library)

We want to automate the management of a small municipal library described as follows: The library, managed by a director, comprises a set of documents and a set of members. Members have a first and last name. Members can join or leave the library on request. New documents are added to the library regularly. A document is made up of several sheets. These documents are either journals or volumes. Volumes are either dictionaries, books or comics. Documents are characterized by a title. Volumes also have an author. Comics also have a cartoonist's name. Journals have, in addition to the characteristics of the documents, a publication date. Only books can be borrowed. A member may borrow or return a book. Members can borrow books (and only books), and we must be able to know at all times which books a member has borrowed. A member may borrow a maximum of 3 books. The return date of a borrowed book is fixed at the time of loan. This date can be extended on request. A member who is a municipal civil servant will receive preferential treatment (this exceptional treatment is not specified at present). It is the member's responsibility to ensure that the documents are kept in good condition, in accordance with regulations (regulations not yet defined). All attributes are private, and all operations are public.

1. Create a class diagram to automate the municipal library.
2. Define the attributes and operations of each class in this diagram, as well as the type and cardinalities of associations between classes.

#### **Solution:**



**Fig. 4.27** – The municipal library class diagram





# *Chapter 5*

*UML diagrams: dynamic view*

## 5.1. Introduction

UML provides dynamic diagrams to model the behavior of objects within a system. Unlike static diagrams (such as class diagrams), dynamic diagrams focus on how objects interact with each other across time, based on events, messages and state transitions.

Among the main dynamic diagrams are interaction diagrams (sequences and communication), state-transition diagrams and activity diagrams. This chapter presents these diagrams, their uses and practical examples.

## 5.2. Importance of the dynamic view in modeling

The modeling of dynamic view in software system is essential, because it allows the representation of the system behavior over time. This perspective offers a good understanding of the interactions between components, the different states of the system, and the transitions that occur in response to various events. Integrating sequence diagrams, state-transition diagrams, or activity models permit to engineers to better anticipate the reactions of the system to external events and internal changes. This not only facilitates the detection of potential, but also optimizes the performance, which guarantees that the system effectively meets functional and non-functional requirements. In short, the dynamic view is very important to guarantee the reliability and robustness of complex software systems.

## 5.3. Sequence diagram

### 5.3.1. Objective

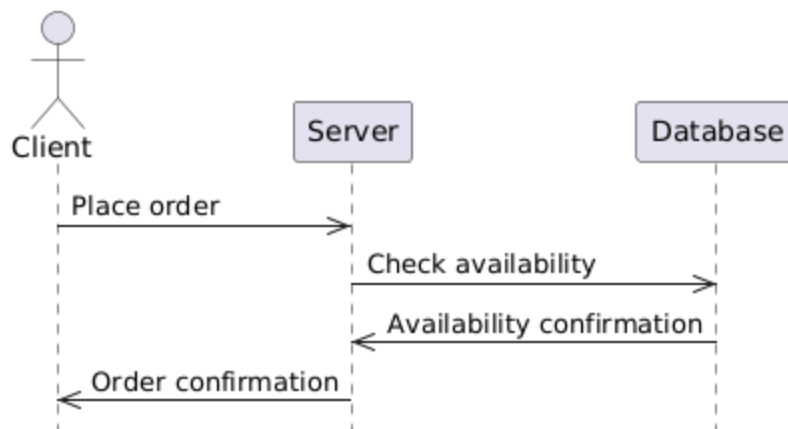
Use case diagrams describe **what** the system is used for, organizing the possible interactions with the actors. Class diagrams define the structure of the system and the relationships between objects, specifying **who** will be implicated in implementing the identified functionalities. Interaction diagrams show **how** these elements interact with each other and with stakeholders. Objects exchange messages to collaborate, while actors interact with the system via Human Machine Interfaces (HMI).

There are multiple types of interaction diagram. This chapter introduces the well-known among them, i.e. the sequence diagram.

### 5.3.2. The chronological aspect

The sequence diagram focuses on the **chronological** order of interactions between objects. Each object is represented by a **lifeline**, and the messages exchanged are represented by arrows between these lines. The sense of time is from up to down.

Example: Here is a simplified example of an online product ordering scenario modeled by a sequence diagram:



**Fig. 5.1** - An example of a sequence diagram.

In this example, the customer sends a request to the Server to place an order, which then checks the availability of the items in the Database before sending a confirmation to the customer

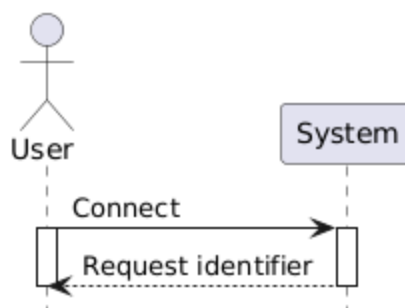
### 5.3.3. Lifeline

A lifeline represents an object or actor implicated in an interaction. It is depicted by a vertical line and a reference to the object at the top in the form of (cf. Fig. 5.2):

**nameLifeline [ selector ]: nameClassOrActor**

If several participants are present, a selector can be used to specify a particular object **n** in the collection, e.g. objects[2].

Example: lifelines of a management system :



**Fig. 5.2** – Example of a lifeline.

### 5.3.4. Messages

Sequence diagrams mainly present messages exchanged between lifelines, in chronological order. Each message represents a specific communication between participants.

The most common messages are:

- ✓ Sending a signal,
- ✓ Calling a method (invoking an operation),
- ✓ Creating or destroying an object.

When a message is received, it triggers a **period of activity**, symbolized by a vertical rectangle on the lifeline, indicating that the message is being processed (cf. Fig. 5.2).

#### *Type of messages*

Messages exchanged between lifelines can be of several types:

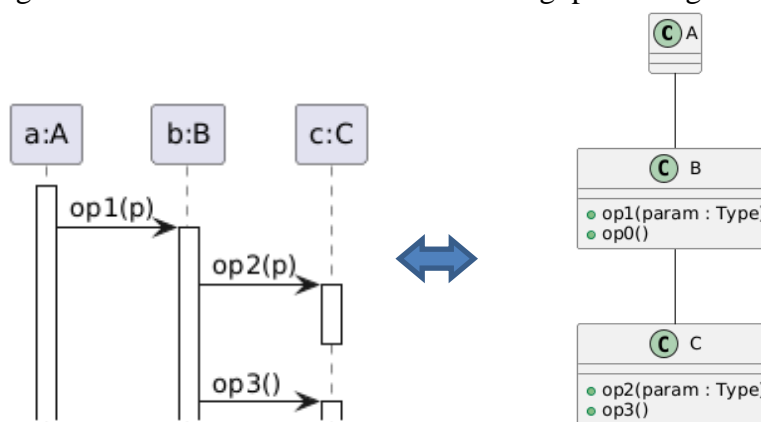
- ✓ **Synchronous** message: the sender sends a request and waits for a response before continuing. A typical example is a method call. This type of message can include a return.
- ✓ **Asynchronous** message: the sender continues execution without waiting for a response. The receiver can process the message at any time, or ignore it. A typical example is the sending of a signal.



**Fig. 5.3** – Message synchronone (left) and asynchrone (right).

#### *Correspondence messages / operations*

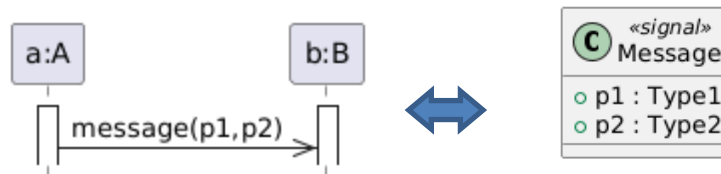
Synchronous messages are linked to operations defined in class diagrams. Sending a synchronous message and waiting for a response before continuing is equivalent to invoking a method and waiting for it to return a result before continuing processing.



**Fig. 5.4** – Correspondence messages / operations.

**Correspondence messages / signals**

Asynchronous messages correspond to signals in class diagrams. These signals are objects whose class is marked with the “signal” stereotype, and their attributes, which contain information, correspond to the parameters of the message sent.



**Fig. 5.5** – Correspondence messages / signals.

**Complete, lost and found messages**

A complete message is one whose sending and receiving events are clearly identified. It is represented by an arrow linking one lifeline to another, symbolizing the transmission of the message between two participants in the interaction.

A lost message is one for which the sending event is known, but not the receiving event. It is represented by an arrow starting from a lifeline and ending on an isolated circle, indicating that the recipient is unknown. A classic example is the sending of a broadcast message.

A found message is one whose reception event is known, but whose transmission event remains unknown. This type of message indicates that the origin of the transmission is not identifiable, but that the reception has been received.



**Fig. 5.6** – lost and found messages

**Syntax of messages**

The message syntax is :

**nameSignalOrOperation ( parameters )**

The argument syntax is as follows:

**nameParameter = valueParameter**

For a modifiable argument:

**nameParameter : valueParameter**

**Examples :**

- f
- f( x , y )
- f( y = 50 )
- f(x:12)

### Return messages

The receiver of a synchronous message returns control to the sender by sending a return message. These return messages are optional, as the end of the activity period can also indicate the end of method execution. They are mainly used to indicate the result of the invoked method.

Asynchronous messages are returned by sending new asynchronous messages.

Return messages are represented by dotted arrows.

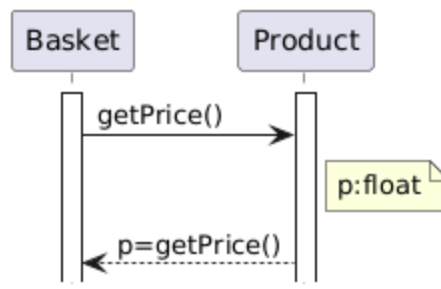


Fig. 5.7 – Return messages.

### Syntax of return messages

The return message syntax is:

**attributTarget = nameOperation (parameters): valueReturn**

The parameter syntax is :

**nameParameter = valueParameter**

or

**nameParameter : valueParameter**

### Creation and Destruction of Objects

The creation of an object is symbolized by an arrow pointing to the top of the lifeline. It is also possible to use a standard asynchronous message with the label «create» to indicate the creation of an object.

The destruction of an object which means ending its behavior is marked by a cross at the end of the lifeline.

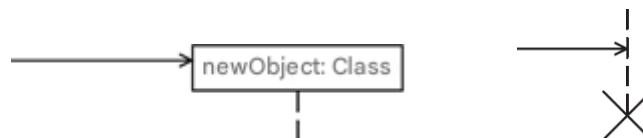


Fig. 5.8 – Creation and Destruction of Objects.

### 5.3.5. Combined fragment

A combined fragment divides a complex interaction into several simpler parts, making it easier to understand. When reassembled, these fragments restore the original complexity. UML 2 has provided a comprehensive syntax for clearly representing different behaviors. A combined fragment is visually similar to an interaction, represented by a rectangle with a pentagon in the upper left corner, where the type of combination is indicated, called the interaction operator.

#### *Type of interaction operators*

Interaction operators in UML fall into several categories:

- **Branching operators:** used to manage choices and loops. Example: alternative, option, break and loop.
- **Operators for sending messages in parallel:** used to send messages simultaneously. Example: parallel and critical region.
- **Operators for controlling message sending:** define specific conditions for messages. Example: ignore, consider, assertion, and negative.
- **Operators for ordering messages:** govern the order in which messages are sent in an interaction. Example: weak sequencing and strict sequencing.

Example: Combined Fragment with *alt* conditional operator. This sequence diagram shows an alternative flow based on the price of the article. If the price is less than 100, the Basket adds the price to the Checkout. If the price is greater than or equal to 100, the Basket applies a discount using the Discount object, then adds the discounted price to the Checkout.

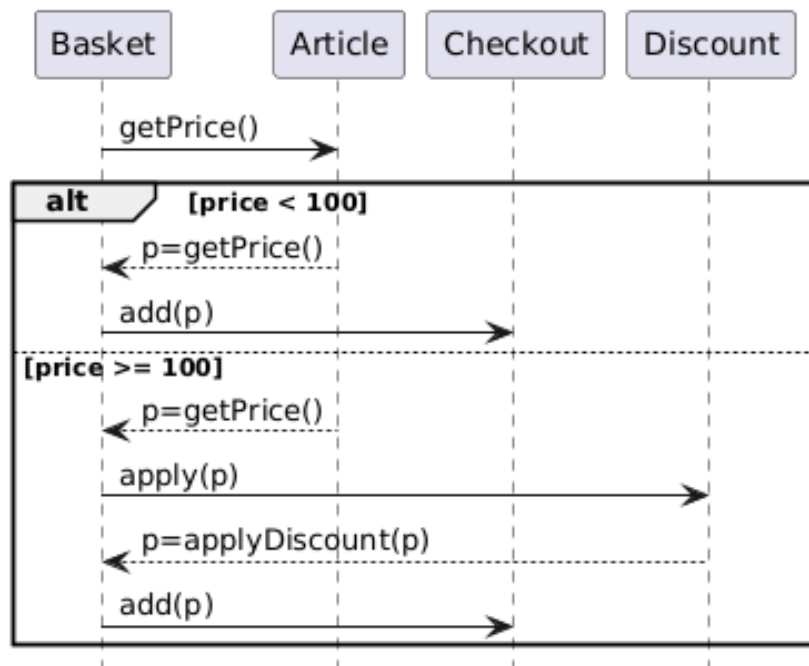


Fig. 5.9 – Example with *alt*.

## 5.4. State-Transition diagrams

### 5.4.1. Objective

A state-transition diagram represents the internal behavior of an object through a finite-state automaton. During its life, an object may go through multiple states. These state changes occur in response to external events, which trigger transitions from one state to another. As a general rule, each object can be found in only one state at a point of time, except in specific cases of concurrency.

### 5.4.2. Initial and final states

The initial state is a pseudo-state that marks the default starting point of the automaton or a substate. When an object is created, it starts in this initial state.

The final state is a pseudo-state indicating the end of execution of the automaton or substate.



Fig. 5.10 – Initial and final states.

### 5.4.3. State and Transition

A **state** is a situation during which an object satisfies a certain condition or performs a particular activity.

A **transition** connects two states and is triggered by a specific event.

States are represented by rectangles with rounded corners, or sometimes as compartments. Transitions are illustrated by oriented arcs. Some states, called composite states, may include sub-diagrams to represent more complex behavior.

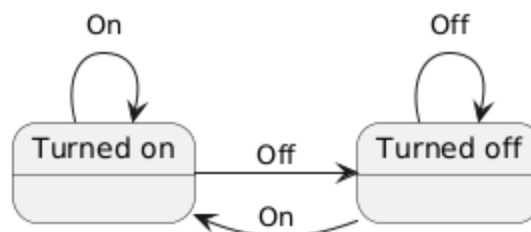
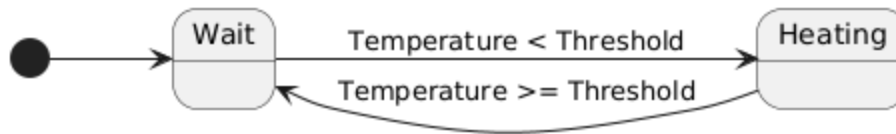


Fig. 5.11 –State and Transition.



Example: State-transition diagram for a thermostat :



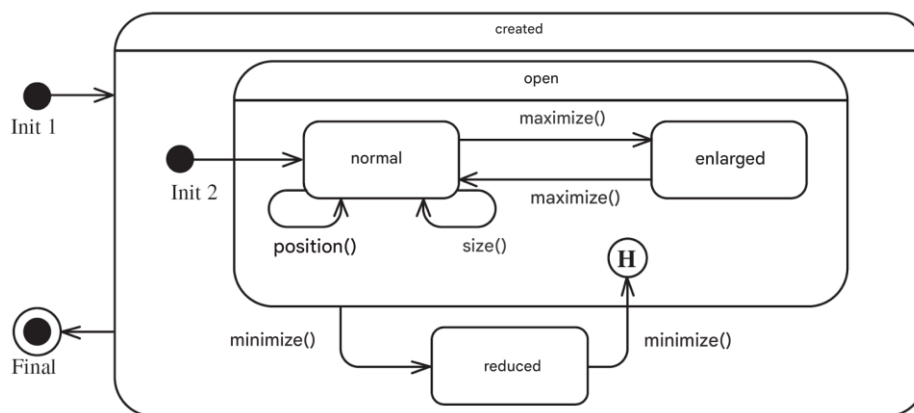
**Fig. 5.12** –Simple State-transition diagram.

In this example, the thermostat transitions from Standby to Heat when the temperature is below the threshold, then back to Standby when the temperature reaches or exceeds the threshold.

The organization of states and transitions is illustrated in a state-transition diagram. A dynamic model can include several state-transition diagrams to represent the different behaviors of a system.

**N.B.** each state diagram is used to for a single class. Each finite-state machine runs concurrently and can change state independently of the other machines in the system.

Example: State-transition diagram that models the behavior of an application window with three main states: reduced (minimized), normal, and enlarged (maximized)



**Fig. 5.13** – Example of State-transition diagram.

#### 5.4.4. External and internal transition

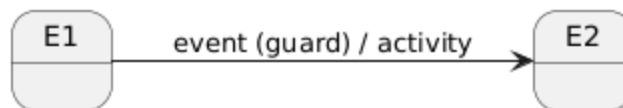
##### External Transition (simple)

State-transition diagrams describe system reactions to events. An **event** is something notable that occurs during the execution of a system and needs to be modeled. An event occurs at a specific time and has no duration. When received, it can trigger a transition, moving the object into a new state. Events can be of different types: **signal**, **call**, **change** or **temporal**.

A transition between two states is symbolized by an arc connecting the two. It shows that an instance can move from one state to another and perform actions, provided that a triggering event occurs and the guard conditions are met. The syntax of a transition is as follows:

**nameEvent ( params ) [ guard ] / activity**

The guard is a condition that must be satisfied before the transition can be triggered. The activity corresponds to the actions to be performed at the time of the transition.



**Fig. 5.14** – External Transition.

### Internal Transition

An object remains in a state for a certain period of time, and internal transitions can occur during this time. These internal transitions do not change the current state, but follow the rules of a simple transition. Three specific actions are used for these internal transitions:

- **entry/** (action to be executed on state entry),
- **do/** (continuous action during state),
- **exit/** (action to be executed on exit from state).



**Fig. 5.15** – Internal Transition.

#### 5.4.5. Junction and decision point

A **junction point** is used to group together several transition segments to make the representation of alternative paths more compact and legible. The aim is to improve the clarity of the diagram. All guards on a given path must evaluate to true as soon as the first segment of that path is crossed.

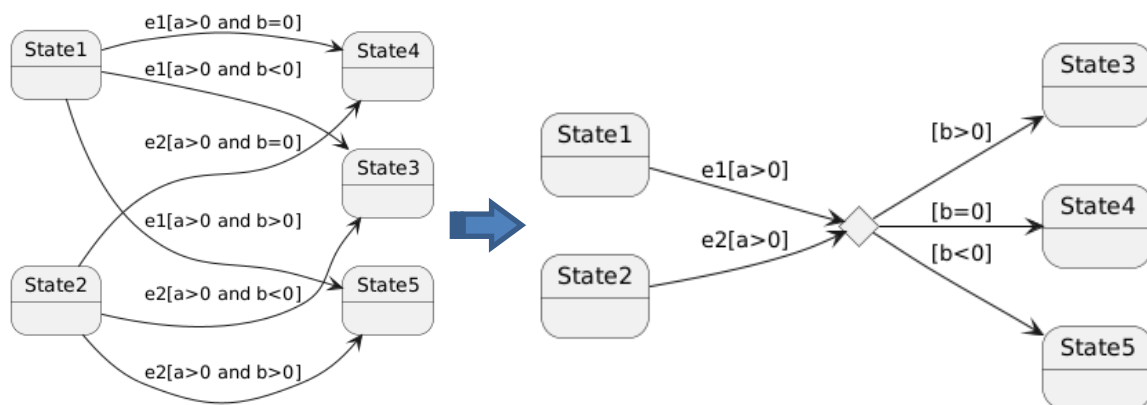


Fig. 5.16 – Junction Point.

### Point de décision

A **decision point** has one input and at least two outputs. The guards associated with the different exits are evaluated when this point is reached. As soon as the decision point is reached, at least one of the paths must be passable.

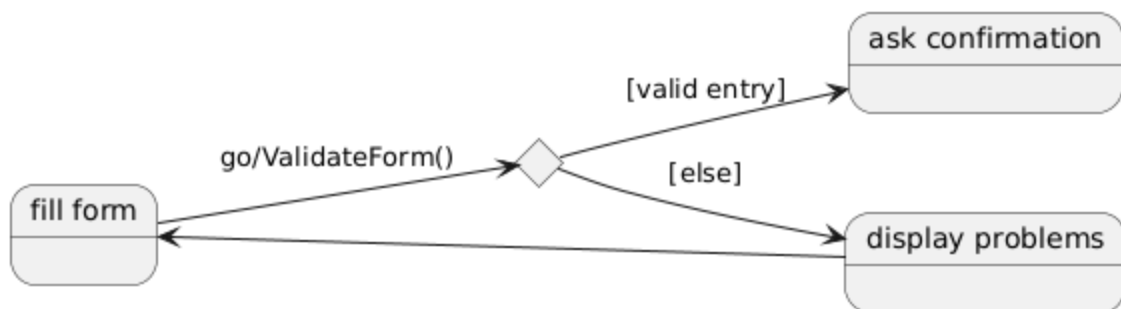


Fig. 5.17 – Decision Point.

### 5.4.6. Concurrency

#### Composite state

A composite state is a state divided into one or more **regions**, each containing one or more sub-states. If there is only one region, the state is **non-orthogonal**. Each state-transition diagram is included in a **global composite state** that envelops the entire system.

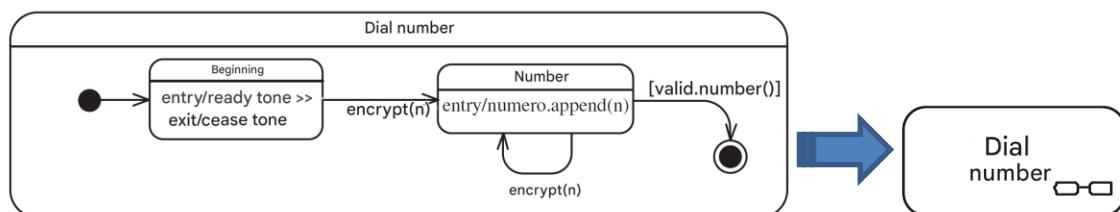


Fig. 5.18 –Composite state.

## Concurrent state

When there are several regions, the state is said to be **orthogonal**, and the regions operate in parallel. We use a dotted separator to represent different regions. This allows an object to be in several concurrent states simultaneously.

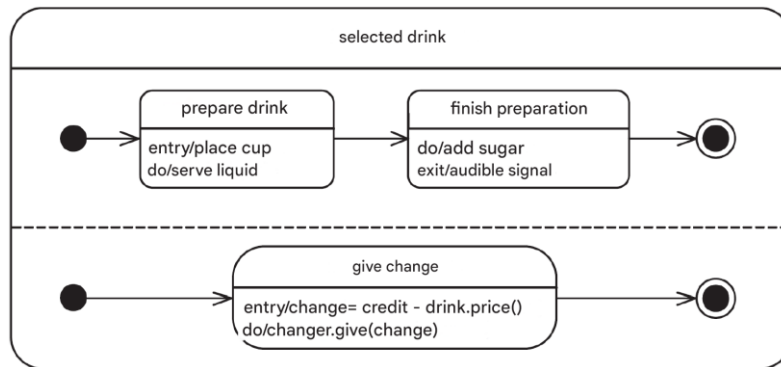


Fig. 5.19 – Concurrent state.

## Concurrent transition

A **Fork** transition creates two states that run in parallel, while a **Join** transition acts as a synchronization barrier, eliminating concurrency. Before continuing their execution, all concurrent tasks must be ready to cross this barrier.

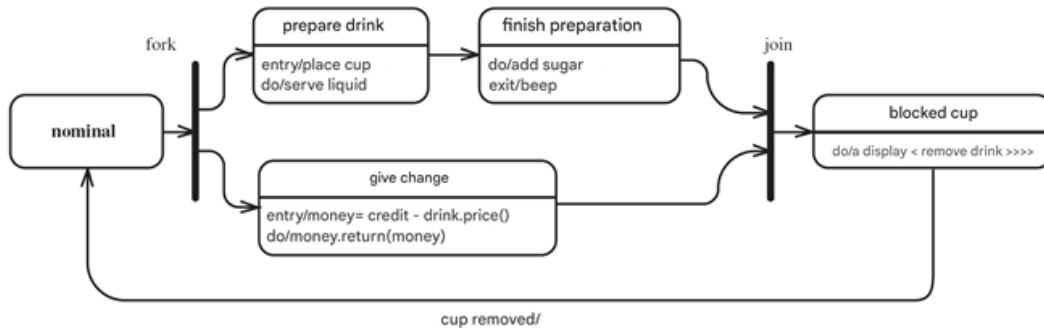


Fig. 5.20 –Concurrent transition.

### 5.4.7. Other concepts

#### Historical

A **flat historical state** is represented by a circled H. A transition that targets this state returns to the last visited state in the region containing this H. In contrast, a **deep historical state**, denoted H\*, retains a history for all nesting levels, enabling a return to the last visited sub-state, regardless of its level in the hierarchy.

**Example:** The use of a **deep history** ( $H^*$ ) allows you to retrieve, after an interruption, the exact previous sub-state where the object was. Conversely, a **surface history** ( $H$ ) would only retrieve the main state, such as *Treatment 1* or *Treatment 2*, without returning to the nested sub-states (e.g. *E11*, *E12*, *E21*, *E22*) that were active before the interruption.

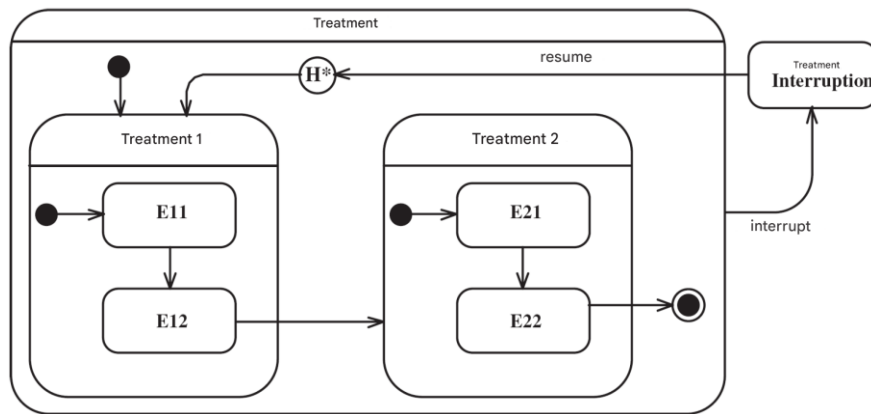


Fig. 5.21 – Historical state.

### Composite states interface

**Connection points** are used to represent a sub-state independently of a macro-state. Output points are indicated by an “X”, while input points are left empty. These interfaces facilitate the abstraction of sub-states, making macro-states more reusable in different parts of the model.

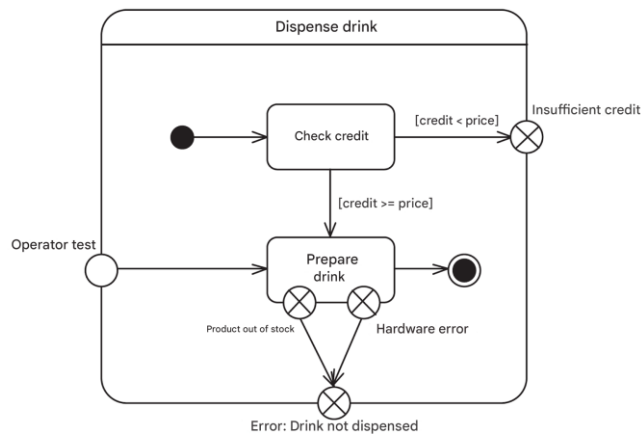


Fig. 5.22 –Connection points.

## 5.5. Activity Diagram

### 5.5.1. Objective

Activity diagrams offer an intuitive approach to modeling processes, similar to state-transition diagrams but with a different interpretation. Unlike state-transition diagrams, which are specific to each classifier and do not link several classifiers, activity diagrams allow a more smooth description, less tied to classifier structuring, and are organized according to actions or activities. They describe processes carried out without external interruption, with few or no events on transitions, and offer a general representation with activity flows. They are particularly recommended for modeling the procedural nature of operations and parallel tasks. They are designed to represent the internal behavior of an operation or use case.

### 5.5.2. Initial and final conditions

**Initial state:** Represents the starting point of a process or control flow. It is the state where the sequence of activities begins.

**Final state:** Represents the termination of a process or control flow. It is the state where the process ends and no further transitions are planned.



Fig. 5.23 – Initial and final conditions.

### 5.5.3. Activities and Transitions

**Action:** Achieves a specific objective. It is atomic, meaning it cannot be interrupted.

**Activity:** Groups together several actions to achieve a specific goal. It can be broken down into sub-actions or sub-activities and can be interrupted.

**Transition:** Indicates the passage of the flow of control from one activity to the next once an activity has been completed. Transitions can also handle exceptions (with events).

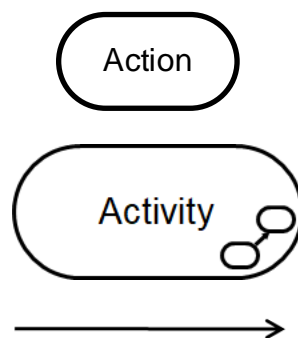
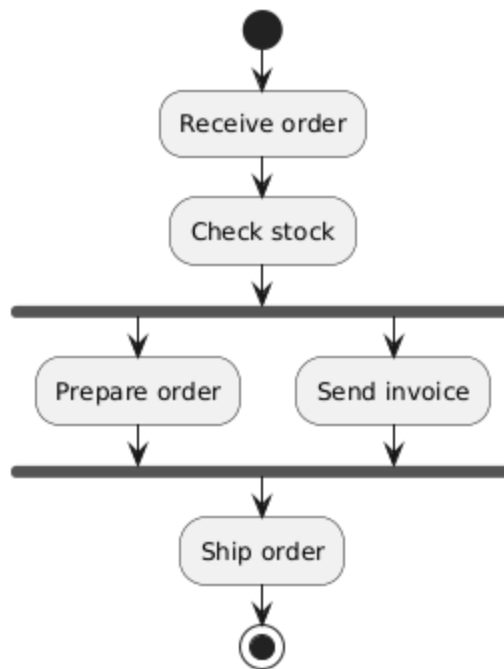


Fig. 5.24 – Action, Activity and Transition.

Example: An order management flow :



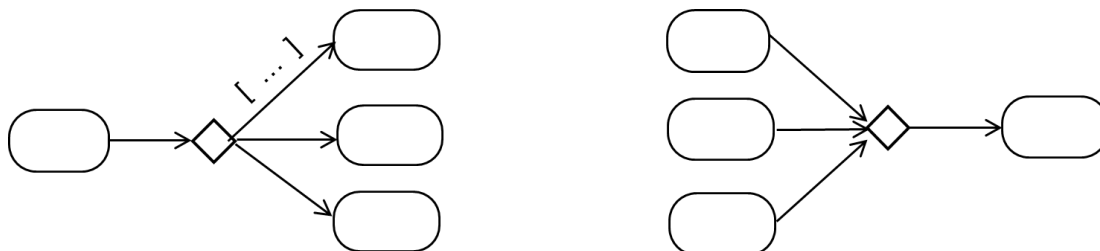
**Fig. 5.25** – Example of a simple activity diagram.

#### 5.5.4. Condition and Merge

**Decision Node:** Point where the control flow breaks into multiple branches according to established conditions. Each branch represents an alternative path based on the defined criteria.

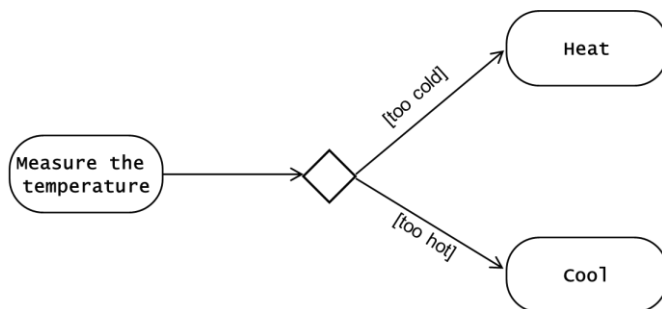
**Merge node:** Point where several branches of the control flow join to continue in a single flow. It combines divergent paths into a single flow.

**Guard:** Condition or expression that determines which path to take at a decision node. It specifies the criteria that must be met for a particular branch to be followed.



**Fig. 5.26** – Decision, Guard, and Merge.

Example:

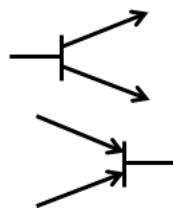


**Fig. 5.27** – Example of a decision.

### 5.5.5. Synchronisation and Concurrency

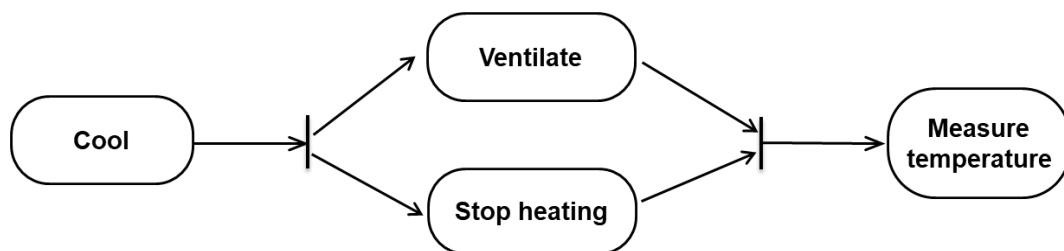
**Synchronization bars:** Structure used to coordinate the start or end of concurrent activities. It is used to synchronize several control flows and ensure that they progress together.

The **fork** breaks the flow into several parallel branches, and the **join** synchronizes them again.



**Fig. 5.28** – Synchronization bars, Fork, and Join.

Example:



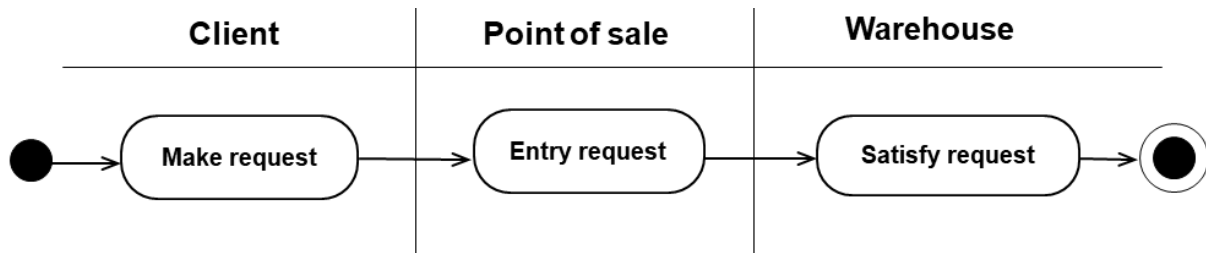
**Fig. 5.29** – Example of Fork and Join.



### 5.5.6. Swimlanes

Water lines, also known as Swimlanes, are used to assign activities to particular elements of the model, facilitating the distribution of tasks between the departments or people responsible for each action. They also help define lines of visibility, clarifying interactions and the scope of responsibilities within the model.

Example:



**Fig. 5.30** – Example of Swimlanes.

### 5.5.7. The objects

So far, we've shown that the behavior of the control flow is visible, but that of the data flow is not. Object flows, on the other hand, represent the passage of data into and out of activities, facilitating the transit of the necessary information.



**Fig. 5.31** – The objects.

Example:

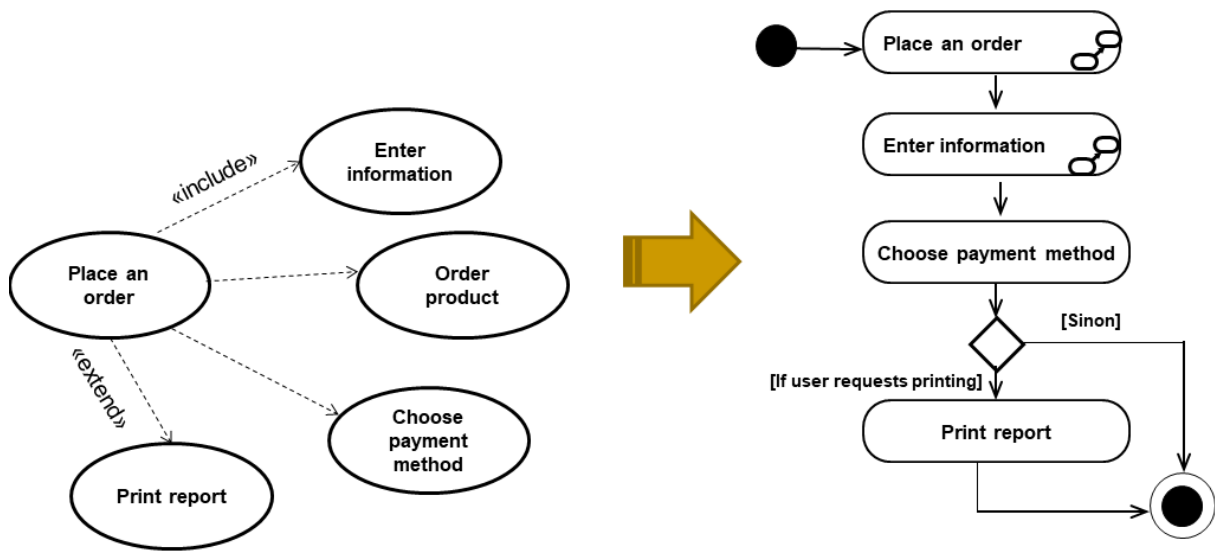


**Fig. 5.32** – Example of linking an object.

### 5.5.8. Use cases documenting

To document a use case, it is important to clearly specify the use case in question. It is also important to represent this use case as an activity, integrating the included and extended use cases that complete it.

Example:



**Fig. 5.33** – documenting a use case by an activity diagram

## 5.5.9. Example of an activity diagram

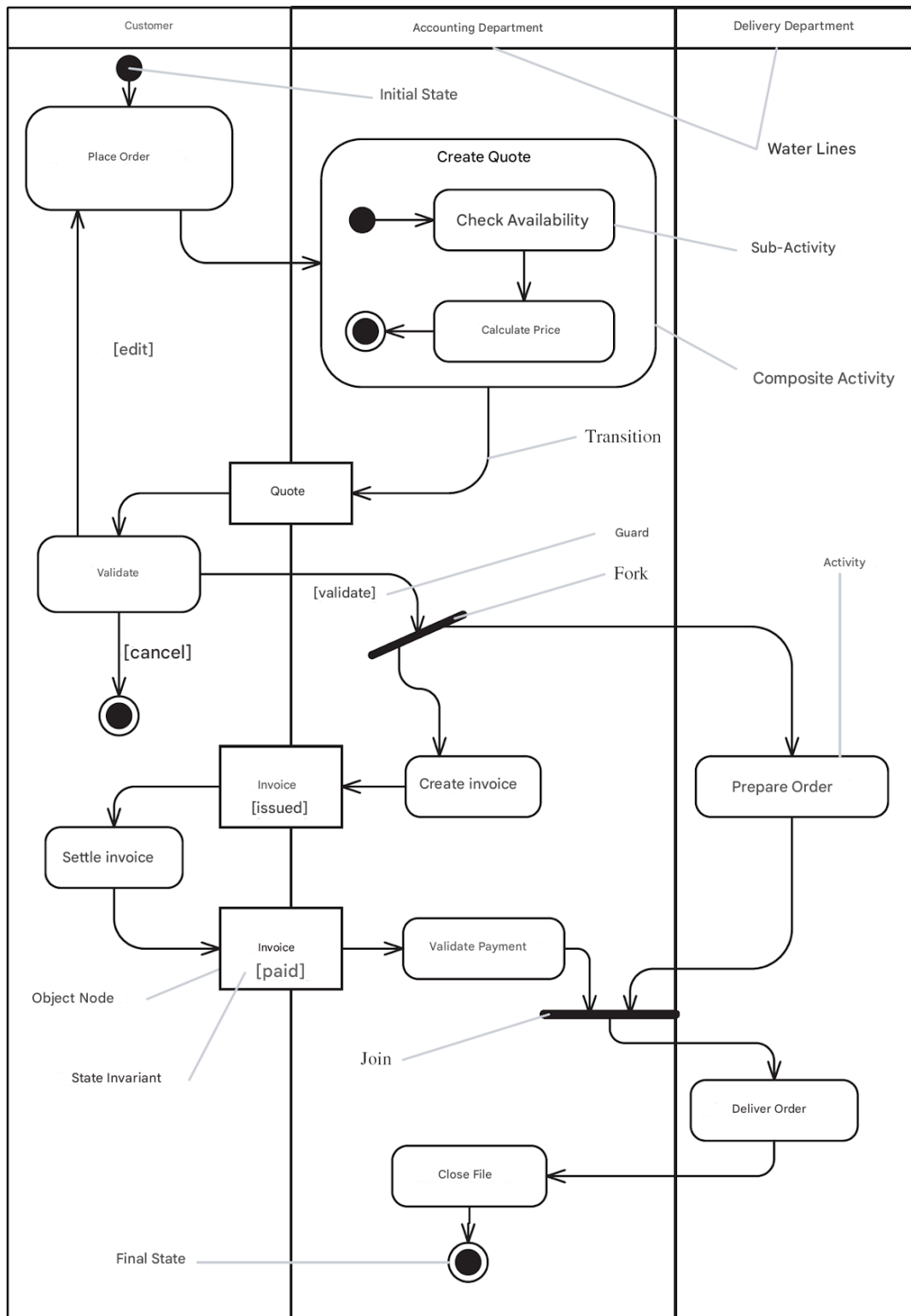


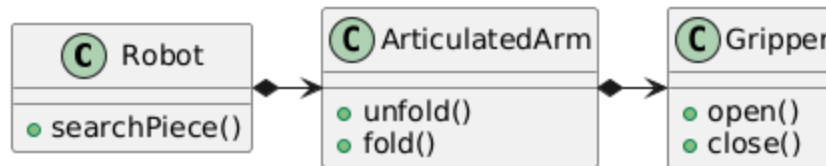
Fig. 5.34 –Example of an activity diagram

## 5.6. Review

**Exercise1:** (Sequence diagram)

**Statement:**

The class diagram below models a robot with an articulated arm ending in a gripper. The robot operates as follows: it unfolds its arm, grabs the piece with its gripper, folds back its arm and releases the part.

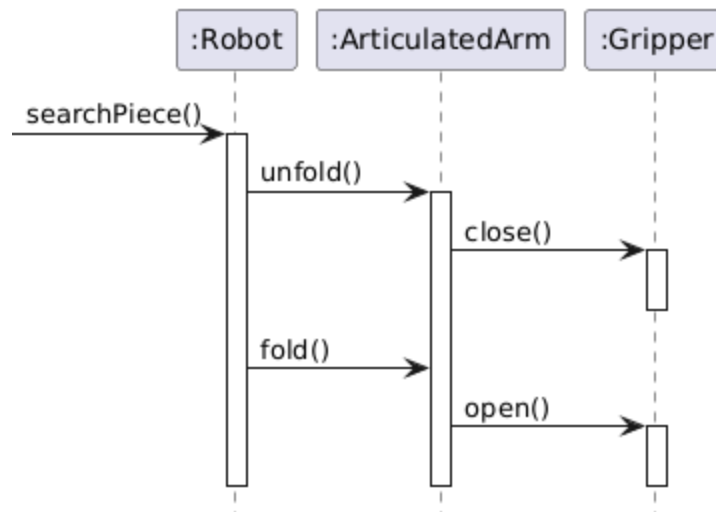


**Fig. 5.35** –the class diagram

1. Use a sequence diagram to illustrate the exchange of messages between the robot, articulatedarm and gripper objects.
2. Modify this diagram to take account of the following case. When the gripper is in the process of releasing the part, two scenarios are possible: Either all goes well, and the robot receives a success message from the gripper. Or the gripper cannot release the part because of an obstacle, in which case the robot receives a failure message from the gripper, and sends an alert signal to the control system.

**Solution:**

1. Sequence diagram.



**Fig. 5.36** –The sequence diagram.

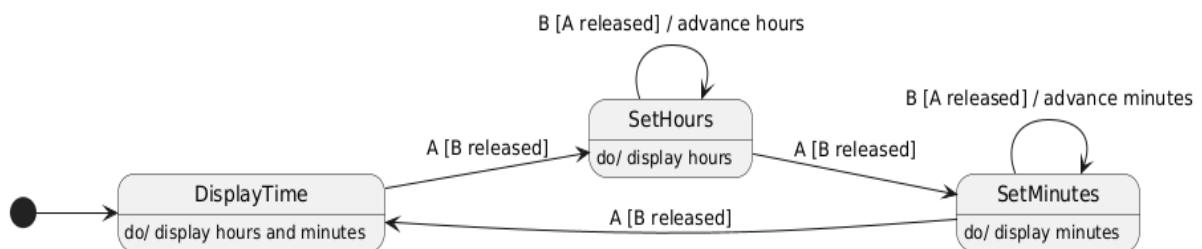
2. To think about ....

**Exercise2:** (State-transition diagram)**Statement:**

A simple digital watch has a dial and two buttons, called A and B, to set the time. The watch has two modes of operation, time display and time setting. In display mode, hours and minutes are displayed, separated by an intermittent colon.

The time-setting mode has two sub-modes, hours and minutes. Button A is used for both modes. Each time it is pressed, the mode changes in the following sequence: display, set hours, set minutes, display, and so on. In a submode, button B is used to advance the hours or minutes each time it is pressed. The buttons must be released before another event can be produced.

Give the watch a state/transition diagram.

**Solution:**

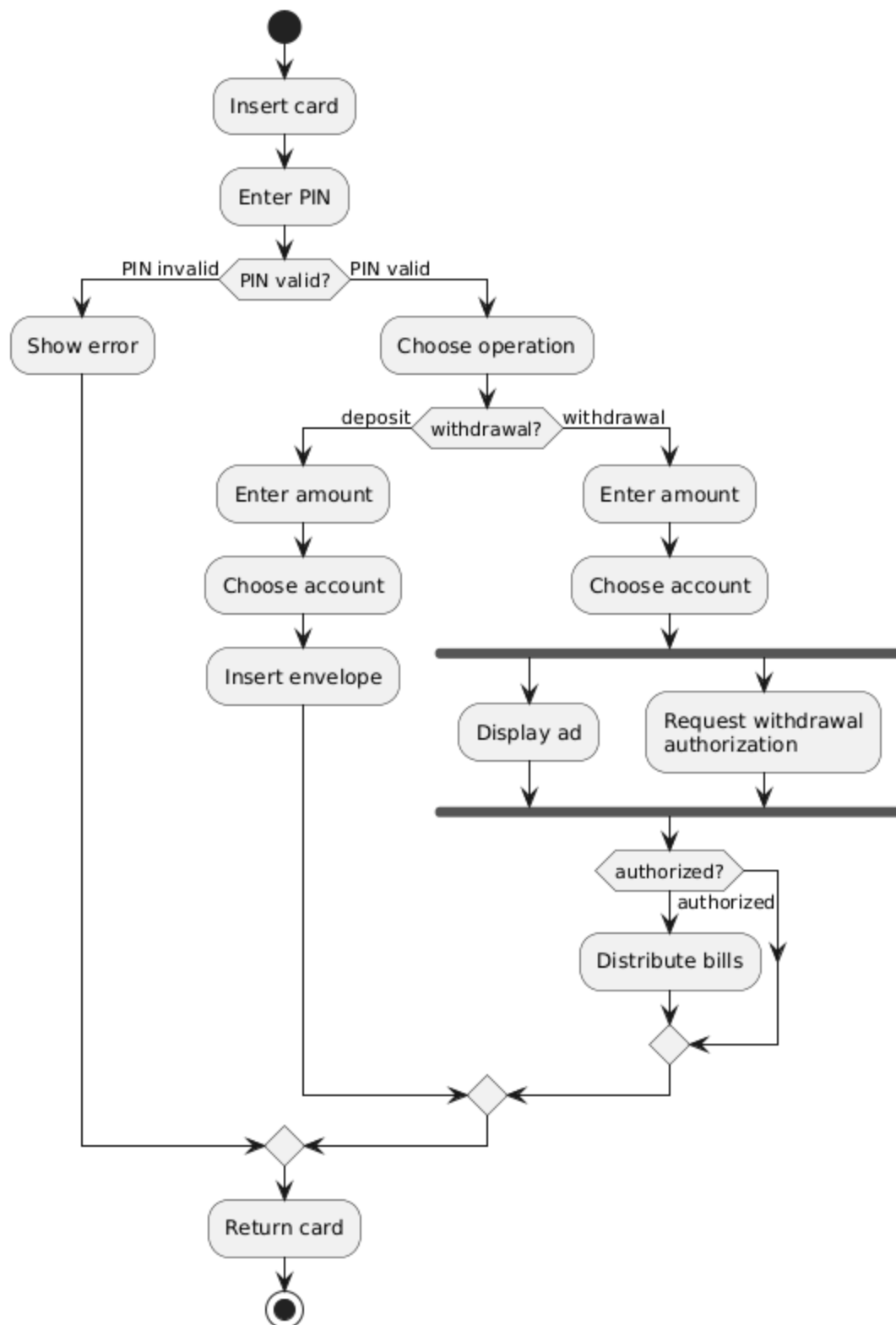
**Fig. 5.37** –The state-transition diagram.

Event A corresponds to pressing button A. In this diagram, releasing the button is not important and is not indicated (although it is obviously necessary to release the button before pressing it again). The constraint that a new “button” event cannot occur while one of the buttons is pressed, would be better expressed as a constraint on the input events themselves. It is not necessary (though not incorrect) to include it in the state diagram.

**Exercise3:** (Activity diagram)**Statement:**

In other words, a system representing the operation of a bank terminal. After inserting the card and entering the code, two activities are triggered: we choose the desired operation if the code is valid, or the card is returned if we cancel the operation. After choosing the operation, we find two alternatives: choosing an amount to deposit or an amount to withdraw. In both cases, we specify the account afterwards. If we have chosen to deposit banknotes, we must insert an envelope, which entitles us to the return of our card at the end of the operation. If we have chosen to withdraw banknotes, we initiate two simultaneous actions: displaying an advert and requesting a withdrawal authorization at the same time. If the latter is obtained, we are dispensed tickets. In all cases, the bankcard is returned and we reach the end of the process.

What is the activity diagram for this system?

**Solution:****Fig. 5.38** –The activity diagram.

# Bibliography and Webography



[1]	Martin Fowler, “ <i>UML Distilled - Third Edition - A Brief Guide to the Standard Object Modeling Language</i> ”. Number ISBN: 0-321-19368-7 in The Addison-Wesley Object Technology Series, 2003.
[2]	Laurent Audibert, “UML 2”, <a href="http://www.lipn.univ-paris13.fr/audibert/pages/enseignement/cours.htm">http://www.lipn.univ-paris13.fr/audibert/pages/enseignement/cours.htm</a> , 2007.
[3]	Roques, P., & Vallée, F. (2007). UML 2 en action. éditions Eyrolles.
[4]	Blanc, X., & Mounier, I. (2011). UML 2 pour les développeurs: cours avec exercices corrigés. Editions Eyrolles.
[5]	Benoît Charroux, Aomar Osmani, Yann Thierry. “ <i>UML 2 Pratique de la modélisation</i> ” - 2e édition, 2009.
[6]	OMG, “ <i>Unified Modeling Language (OMG UML) Superstructure</i> ”, version 2.3, <a href="http://www.omg.org/spec/UML/2.3/Superstructure">http://www.omg.org/spec/UML/2.3/Superstructure</a> . May 2010.
[7]	Object Management Group (OMG), “ <i>Model Driven Architecture (MDA)</i> ”, <a href="http://www.omg.org/mda">http://www.omg.org/mda</a> . 2004.
[8]	A. Belghiat, “ <i>Transformation des modèles UML vers des ontologies OWL</i> ”, Mémoire de Magister, Université de M'sila, Algérie, 2012.
[9]	A. Belghiat, “ <i>Une approche de spécification et de vérification des systèmes logiciels à base d'agents mobiles en utilisant UML mobile et <math>\pi</math>-calcul</i> ”, Thèse de doctorat, Université de Constantine2, Algérie, 2017.
[10]	Laurent Piechocki , “ <i>UML, le langage de modélisation objet unifié</i> ”, Date de publication : 22/10/07, Date de mise à jour : 14/09/09.
[11]	Pierre Gérard. “Introduction à UML 2 Modélisation Orientée Objet de Systèmes Logiciels”. Université de Paris 13, IUT Villetaneuse, 2009.
[12]	IBM, <a href="https://www.ibm.com/think/topics/data-flow-diagram">https://www.ibm.com/think/topics/data-flow-diagram</a> , 2023