Chapter 3: Unified Modeling Language (UML)

1. Introduction

In any system design project, clear requirements, structured thinking, and effective communication among stakeholders are essential. The Unified Modeling Language (UML) has become the de facto standard to meet these needs. It provides a powerful graphical notation that is both rigorous and intuitive, allowing for the modeling of both the static and dynamic aspects of software or organizational systems.

This chapter offers a general introduction to the UML language without delving into all its elements. The goal is to provide students with the fundamental skills needed to understand, interpret, and create simple yet meaningful models. We will focus on three key diagrams that play a central role in the functional and technical specification of a system:

-**The use case diagram**, which represents the interactions between actors and the system;

-**The sequence diagram**, which describes the flow of messages in a given scenario;

-**The class diagram**, which models the core entities of the system and their relationships.

“UML is not just a modeling notation; it is a language for thinking about and communicating designs.”[5]

2. Definitions of UML:

Definition 1: The Unified Modeling Language (UML) is a graphical modeling language based on pictograms, designed as a standardized method for visualizing systems, primarily in the fields of software development and object-oriented design.

UML is the result of a synthesis of earlier object-oriented modeling languages, namely Booch, OMT, and OOSE. It originates primarily from the work of Grady Booch, James Rumbaugh, and Ivar Jacobson.Today, UML is a standard officially adopted by the Object Management Group (OMG).

Historical Overview: UML 1.0 was standardized in **January 1997**, and UML 2.0 was adopted by the **Object Management Group (OMG)** in **July 2005**.The latest version of the specification officially approved by the OMG is **UML 2.5.1**, released in **2017**.

Definition 2: The Unified Modeling Language (UML) is a graphical modeling language used to specify, visualize, construct, and document the artifacts of a software system, particularly in the context of object-oriented development. It provides a standardized set of notations to represent both the structural and behavioral aspects of systems.

UML is a standard language for specifying, visualizing, constructing, and documenting the artifacts of software systems, as well as for business modeling and other non-software systems [6].

3. UML Diagrams:

UML diagrams are divided into two main categories:

-**Structural diagrams**, which represent the static view of the system;

-**Behavioral diagrams**, which illustrate the dynamic view of the system.

There are a total of **13 UML diagram types (Figure 10)**, but in this course, we will focus on **three key diagrams**:

-The **use case diagram**, which models the system's behavior and functionalities from the user's perspective;

-The **class diagram**, which describes the static structure of the system, including object types and their relationships;

-The **sequence diagram**, which depicts the chronological interactions between objects and actors.

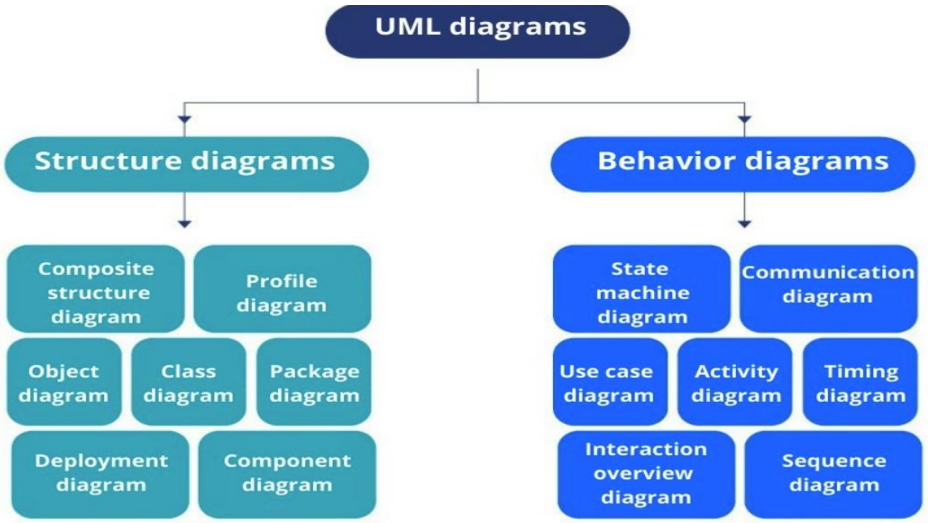

Figure 10 UML diagrams structural and behavioral

4. Use Case Diagram

The use case diagram is one of the most commonly used diagrams in UML, especially during the early stages of functional system modeling. It is used to represent **the services or functionalities provided by a system to its users**, known as **actors.** These actors can be individuals, external systems, or any other entities interacting with the system.

The primary goal of this diagram is to **describe the system from an external viewpoint**, focusing on **functional requirements** rather than technical implementation. It serves as an effective communication tool among stakeholders (clients, end-users, analysts, developers), as it is intuitive and easy to understand even for non-technical audiences.

A use case diagram typically consists of:

-**Actors**: depicted as stick figures, representing entities that interact with the system.

-**Use cases**: shown as ellipses, representing the system's functionalities or usage scenarios.

-**Relationships**: arrows or lines between actors and use cases (and sometimes between use cases), indicating interactions, inclusions, extensions, or generalizations.

For example, in a library management system, common use cases might include Borrow Book, View Catalog, and Return Book, with actors such as Reader, Librarian, or even an External Payment System.

The use case diagram answers the question: **What does the system do?** or more precisely: **What interactions are possible between the users and the system?**

4.1 Main Elements of a Use Case Diagram

A use case diagram is mainly composed of two key elements: **actors** and **use cases (Figure 11).**

Actor→An actor represents an **external entity** that interacts with the system. This may be: a **person** (user, an administrator), **another software system** (a payment service), or a **physical object** (a sensor or card reader).

The actor defines a **role**, not a specific individual: the same entity can play multiple roles.It is identified by **the name of the role**, not by the actor's personal name.

Use Case→ A use case refers to a **functionality or service provided by the system and visible from the outside**.It is an **action initiated by an actor**, to which the system responds. It is typically named using an **infinitive verb**, such as "Check balance", "Manage account", or "Confirm order".

These elements are connected through **interaction relationships**, showing who does what with the system.

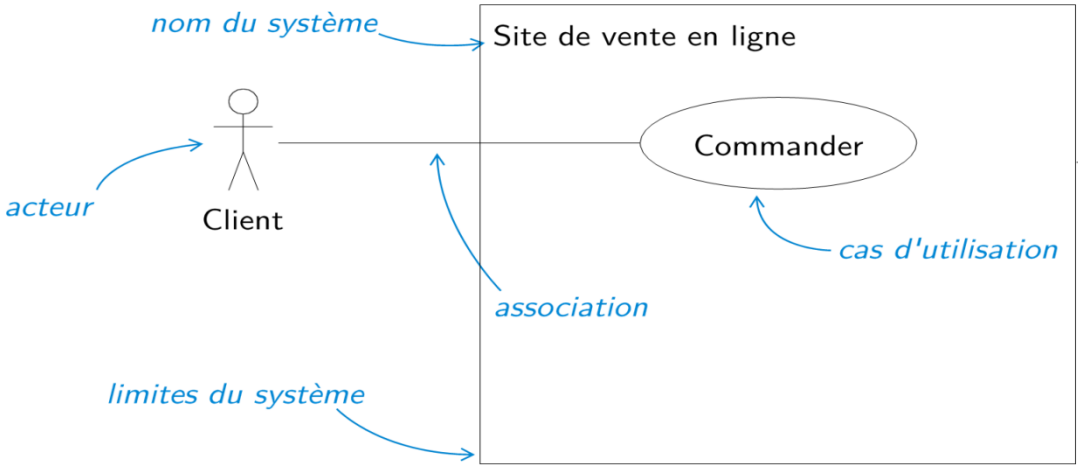

Figure 11 Main components of a use case diagram

Association in a Use Case Diagram →An **association** is a **fundamental relationship** in a use case diagram. It connects an **actor** to a **use case**, indicating that the actor **interacts with the system** by initiating or participating in that use case.

-It is shown as a **simple line** between the actor and the use case.

-It expresses that the actor **can trigger or benefit from** the functionality described.

-One **actor** may be associated with **multiple use cases**, and a single **use case** may involve **multiple actors**. **(Figure 12)**

The association does not specify the detailed behavior but clearly indicates the **potential for an actor to make use of a system function**.

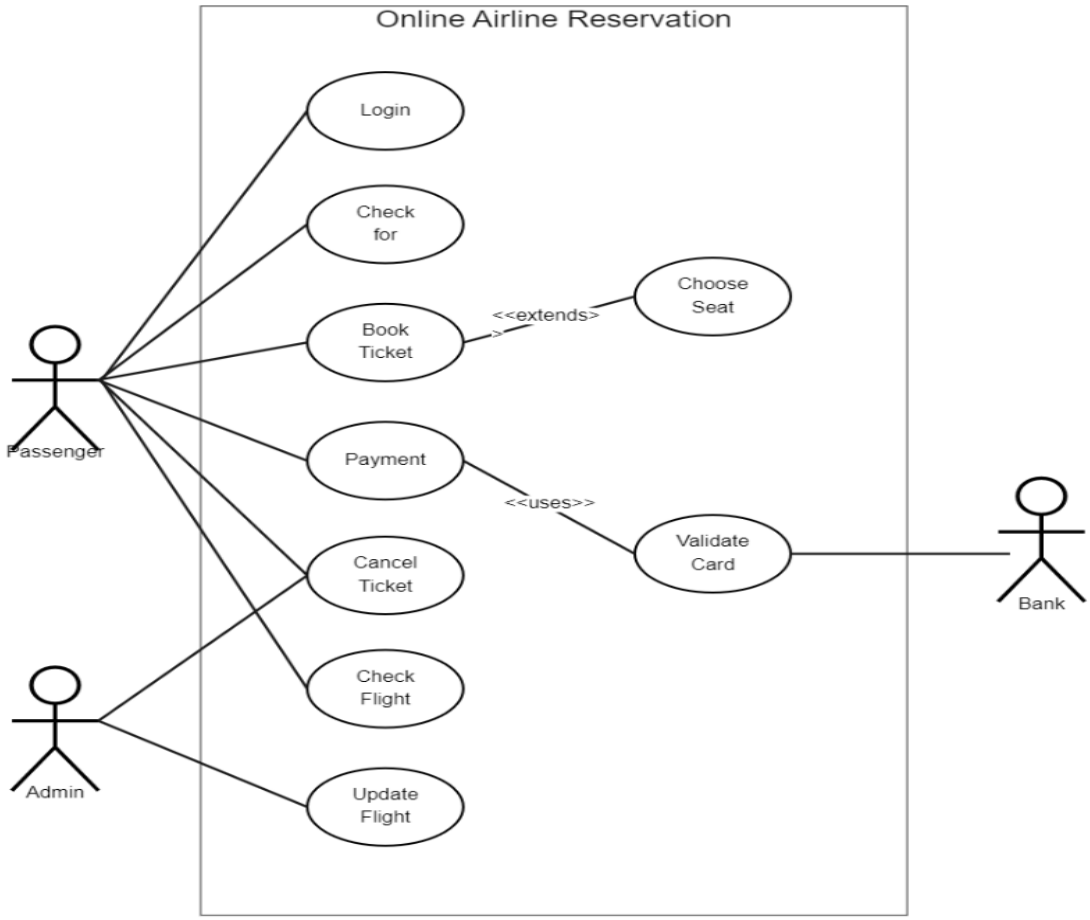

Figure 12 Example of a use case diagram

In UML use case diagrams, **role generalization** is used when one actor (the child) inherits the behavior and relationships of another actor (the parent). This is useful when multiple roles share common interactions with the system but also have specific behaviors.

As shown in the figure "**Figure 13"**, both Supervisor and Student are considered users above all.

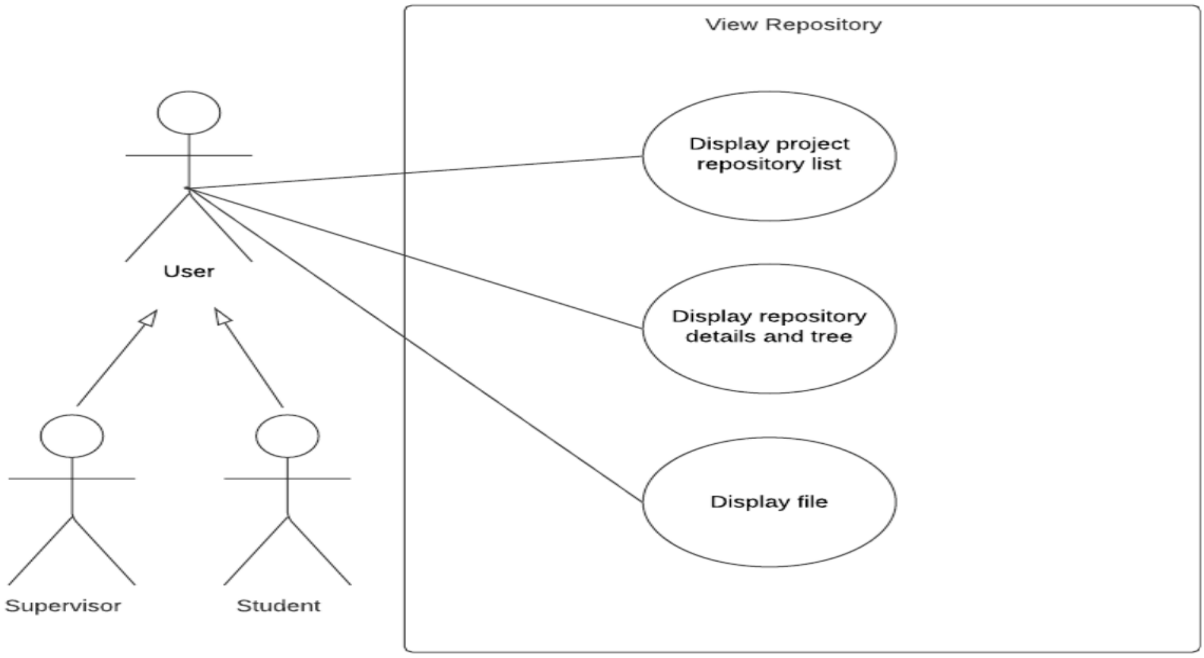

Figure 13 Role generalization

In a use case diagram, two common relationships can exist between use cases: «include» and «extend». The «include» relationship represents a **mandatory inclusion** of common behavior (**Figure 14)**: it allows one use case to incorporate the functionality of another use case that is shared across multiple scenarios. This helps to reduce redundancy and improve modularity the included use case is always executed when the base use case runs. For example, both Place Order and Cancel Order might include Authenticate User. On the other hand, the «extend»relationship represents an **optional or conditional extension** of behavior (**Figure 15)**: it allows a use case to insert additional steps into another use case under specific conditions. The extended use case defines a base scenario, while the extending use case adds alternative or exceptional behavior. For instance, Make Payment might be extended by Apply Discount only if the user is eligible. These two relationships help model reusable and adaptable functionality in complex systems.

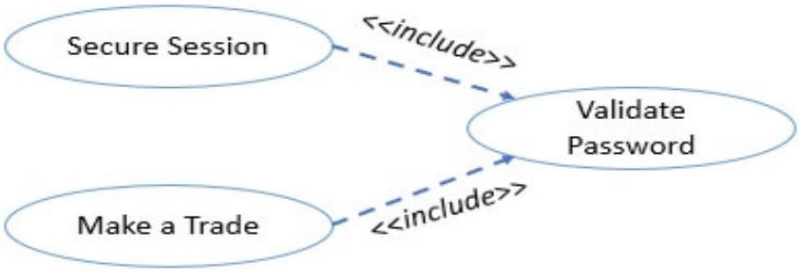

Figure 14 Include use case Relationship

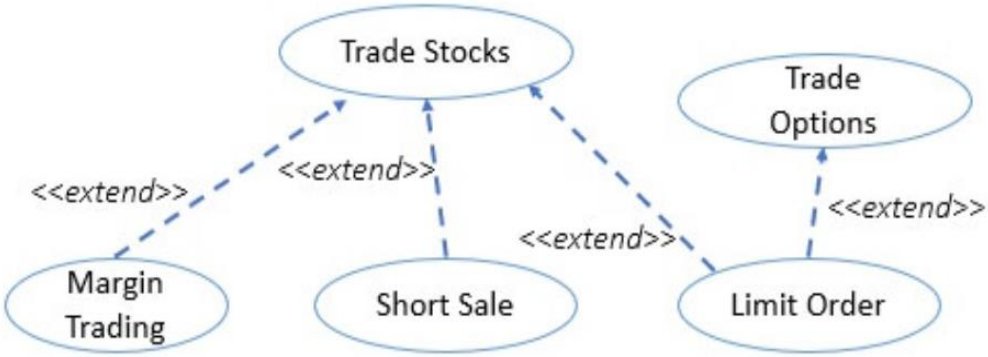

Figure 15 Extend use case Relationship

Guidelines for Designing a Use Case Diagram

To ensure clarity and usefulness, the following best practices are recommended when designing a use case diagram:

-**Limit the number of use cases** to around **6 to 8 per diagram**. More than that can reduce readability and make the diagram harder to interpret.

-**Break down the system** into subsystems or logical functionalities if needed, and use **multiple diagrams** to cover different areas.

-**Avoid excessive use of relationships** between use cases (such as include or extend): use them only when they add real value, to avoid cluttering the diagram.

-For **detailed behavior**, prefer **textual descriptions** such as structured use case narratives or detailed scenarios.

-**Dynamic scenarios** involving interactions between actors and system elements can be illustrated using **sequence diagrams**, which show the temporal flow of messages.

5. Class Diagram

Object-Oriented Design: The system is represented as a set of interacting objects.

The class diagram→ Describes the internal structure of the software; defines classes, their attributes, methods, and relationships.

Object Diagram→Represents the state of the software at a specific moment (objects and their relationships); evolves during the execution of the software:

-Creation and deletion of objects.

-Changes in object state (attribute values).

-Changes in relationships between objects.

The **class diagram** is one of the core elements of UML modeling. Its primary goal is to **show the static structure of the system**, by representing the different **classes**, along with their **attributes** (data) and **methods** (functions).

This diagram is **mainly intended for developers**, as it provides a comprehensive view of the software architecture, object types, and the relationships between them (associations, compositions, generalizations, etc.).It often serves as a **starting point for implementation** in an object-oriented programming language.

In summary, a class diagram allows the representation of:

-the **classes** in the system;

-their **attributes** (properties or characteristics);

-their **methods** (behaviors or operations);

-and the **relationships** between classes (such as association, inheritance, or dependency).

Definition of an Object: In object-oriented modeling, an **object** is a **concrete or abstract entity** from the **application domain** being modeled. **(Figure 16)**

It can represent a real-world element (e.g., a customer, a product, an order) or a conceptual one (e.g., a transaction, an event, a business rule).

An object is defined by three fundamental characteristics:

Identity: what distinguishes it from other objects, even if they have the same attribute values (technically, its memory address).

State: the set of **attribute values** it holds at a given moment.

Behavior: the set of **operations** or **methods** it can perform.

Thus, an object is both a **data unit** and a **processing unit**, capable of interacting with other objects within the system.

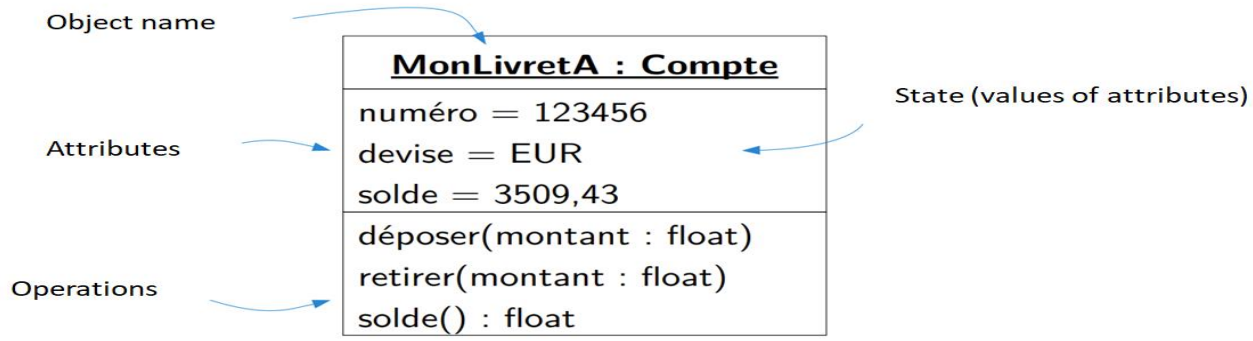

Figure 16 Declaration of an object

Class and Object: In object-oriented modeling, a **class** is a **template or blueprint** used to create objects. It represents a **group of objects with the same nature**, meaning objects that share: the **same attributes** (properties or data), and the **same operations** (methods or behaviors).

In other words, a class defines the **common structure and behavior** of all its instances. An **object** is a **concrete instance of a class**.

When an object is created from a class, we say the class is **instantiated**. Each object has its **own values for the attributes**, although it follows the structure and behavior defined by the class.

Example 1: The class Car might define attributes such as color, brand, maxSpeed, and methods like accelerate() or brake().The object myCar is an instance of this class with color = red, brand = Toyota, etc.

Example 2: Three objects MonCompteLivretA, MonCompteJoint, and MonCompteSuisse are instances of the class Compte. **(Figure 17)**

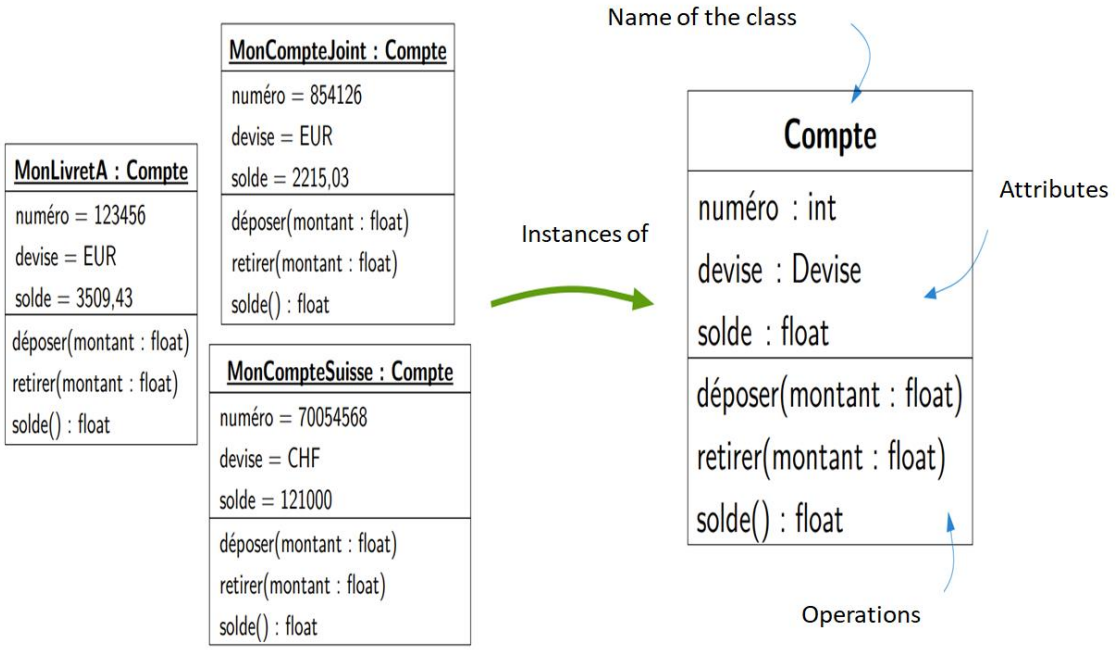

Figure 17 Declaration of three objects instances of a class

Attributes in UML Class Diagrams: In UML, **attributes** represent the **properties or characteristics** shared by all instances of a class. Each object holds a **specific value** for each attribute, but the **definition of the attribute** its name and type is specified at the class level. Attributes are always associated with a **data type**, which can be:

-a **primitive type** (e.g., int, bool, float); a **complex type** (such as a Date object); or an **enumeration**.

An **enumeration** is a special type of class that defines a fixed set of constant values. It is identified by the stereotype <<enumeration>> and consists only of named values (literals). In

class diagrams, an attribute can be declared with an enumeration as its data type, meaning that it can only take one of the predefined values. Enumerations provide a clear and constrained way to model categorical or status-type data within a system.

Object State and Attribute Values: The **state of an object** is defined by the **values of its attributes** at a given moment. While all objects of the same class share the **same structure** (the same set of attributes), each object can hold **different values** for those attributes. This is what allows objects to behave uniquely within the same class definition.

It is also possible for two distinct objects each with its own **identity** (memory address) to have **identical attribute values**, yet they remain different instances (**Figure 18)**. The identity of an object is independent of its state, meaning that object uniqueness is preserved even if two objects temporarily look the same.

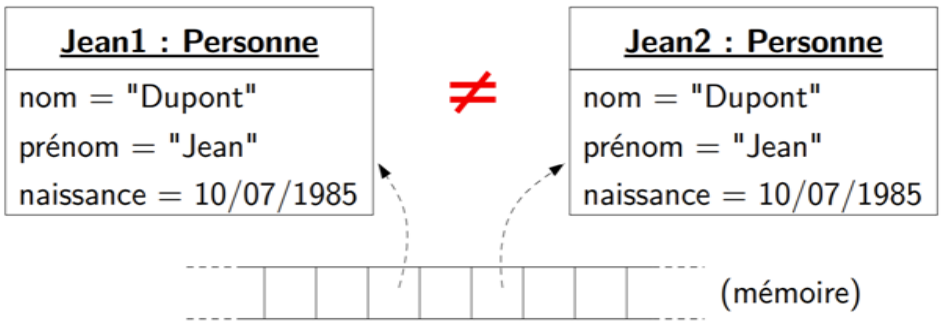

Figure 18 Different objects with the same values of attributes

In UML terminology, an **operation** refers to the **specification or declaration** of a behavior that instances of a class can perform (**Figure 19)**. It defines the **signature** that is, the method name, parameters, and return type without describing how the behavior is implemented.

The **implementation** of that operation, written in code within a specific programming language, is what is commonly called a **method**. However, this dual usage of the term "method" creates a **semantic ambiguity**:

-In modeling (UML), "method" is often used as a synonym for **operation**,

-In programming, "method" usually refers to the **code implementation** of that operation.

Therefore, when teaching or documenting object-oriented design, it's important to clarify the context in which the word “method” is used whether it refers to the **model-level declaration** (operation) or the **code-level definition** (implementation).

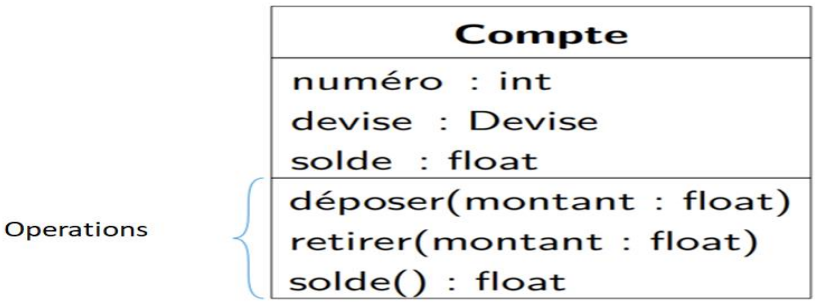

Figure 19 Operations in a class

Relationships Between Objects: In UML, a **relationship between objects** represents a **link**, typically **binary**, that connects two instances of classes (**Figure 20)**. This link indicates a form of interaction or dependency between the objects during the system’s execution. In most cases, there is **at most one link** between any given pair of objects for a specific **association**.

Such object-level relationships are instances of **class-level associations**, meaning they are concrete manifestations of the abstract relationships defined in class diagrams. These links are dynamic and reflect how objects **collaborate** or **communicate** at runtime.

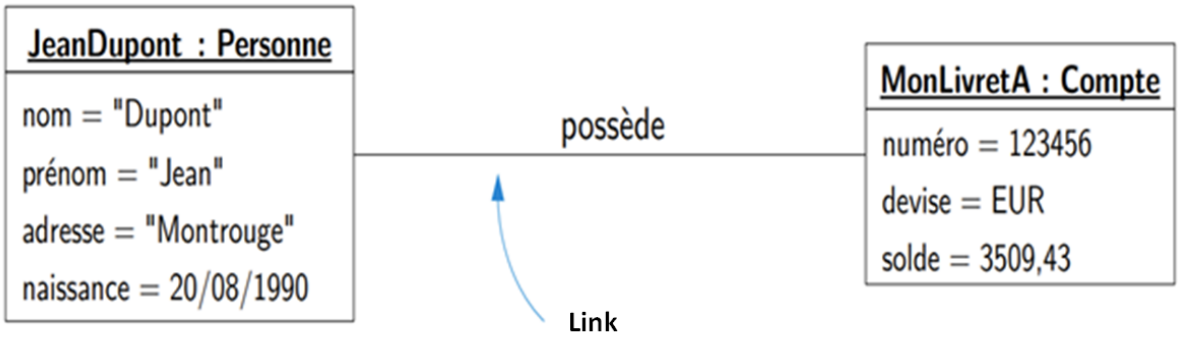

Figure 20 Relationship between objects

Relationships Between Classes: In UML class diagrams, a **relationship between classes** is typically modeled as a **binary association**, meaning it connects two classes **(Figure 21)**. This association represents a potential link between instances of these classes at runtime.

Each end of an association can be given a **role name**, which identifies the perspective or function of the related class in the context of the association. This role name is useful for **navigating** the relationship and for **clarifying the semantics** of the connection.

Associations also include **multiplicity constraints**, which define how many instances of one class can be associated with a single instance of the other class. For example, a "1..*" multiplicity indicates that one object must be associated with one or more objects of the other class. These constraints are essential for accurately modeling real-world cardinalities and system rules.

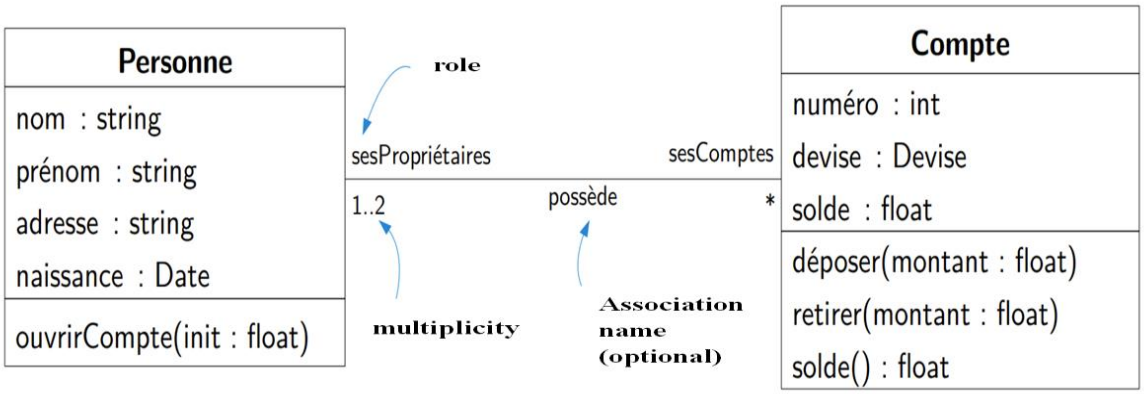

Figure 21 Relationship between classes

Attributes and Associations: Initial Modeling Guidelines

As a starting point in class diagram modeling, we will **limit attribute types** to **simple**, **primitive**, or **enumerated types** (e.g., int, bool, String, or predefined enumerations). This simplifies the design and focuses attention on the essential structure of the system. **(Figure 22)**

At this stage, we **avoid using attributes whose type is another class** from the same diagram. Relationships between classes especially when one class references another should be modeled explicitly through **associations**, rather than as class-type attributes. This distinction

helps maintain clarity between structural links (**associations**) and internal data (**attributes**), which is fundamental in object-oriented modeling.

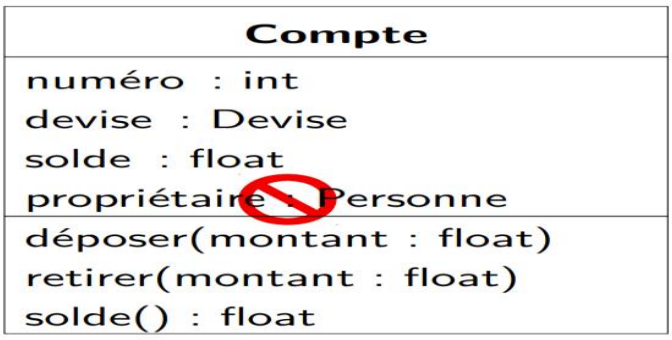

Figure 22 No instance attributes in the class in the first time

Instead of adding attributes, it is preferable to create an association to this class (Figure 23).

This approach promotes better modularity and reflects the relationship between classes more clearly in the design.

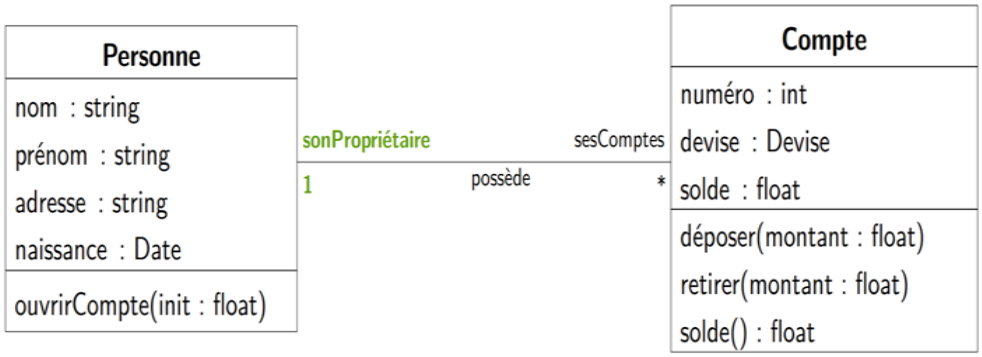

Figure 23 Association between two classes

Multiplicities in UML Associations: Multiplicity defines how many instances of one class can be associated with a single instance of another class in a UML class diagram. For example, if class **A** is associated with class **B**, the multiplicity specifies the **number of objects of class B** that can be linked to **one object of class A**.

Common multiplicity values include (**Figure 24)**:

1 → exactly one object

0..1 → zero or one object

* (or 0..*) → zero or more objects

1..* → one or more objects

These constraints are placed at the ends of the association lines and help model real-world rules and constraints clearly. They are essential for understanding the **cardinality** of relationships between classes and for guiding correct object instantiation and interaction.

Figure 24 Multiplicities

Class Hierarchy: Generalization and Specialization: A **class hierarchy** is a fundamental principle in object-oriented modeling that involves **grouping classes** that share common **attributes** and **operations (Figure 25)**, and organizing them in a **tree-like structure**. This structure promotes **reusability**, **clarity**, and **scalability** in system design.

Generalization refers to the process of identifying common characteristics among several classes and abstracting them into a **superclass**. The superclass encapsulates shared features, while subclasses inherit from it **(Figure 26).**

Specialization, on the other hand, is the process of defining a **subclass** that extends or refines a more general class by adding specific attributes or behaviors.

These relationships are typically represented in UML using a solid line with a **hollow triangle** pointing toward the more general class. Hierarchies help model real-world taxonomies and promote consistent design through **inheritance**.

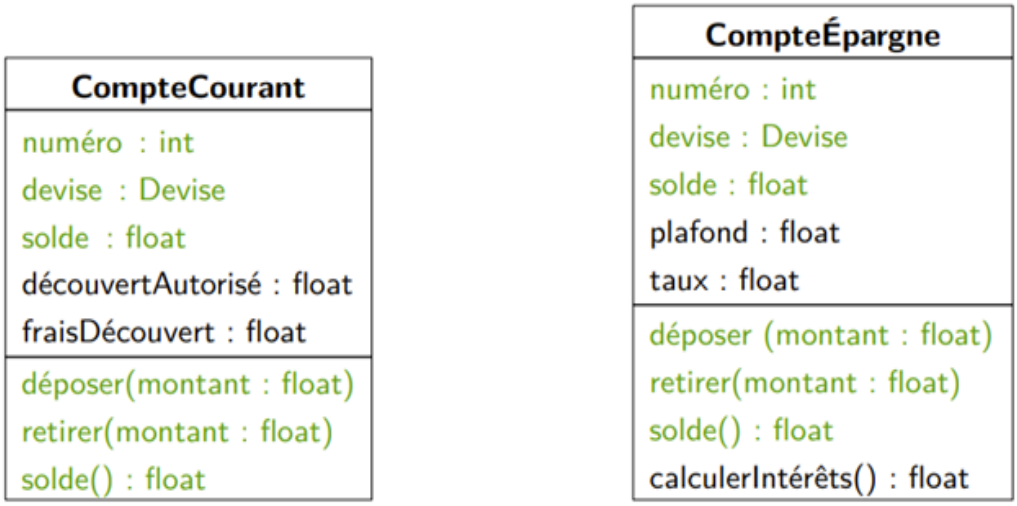

Figure 25 Classes without inheritance

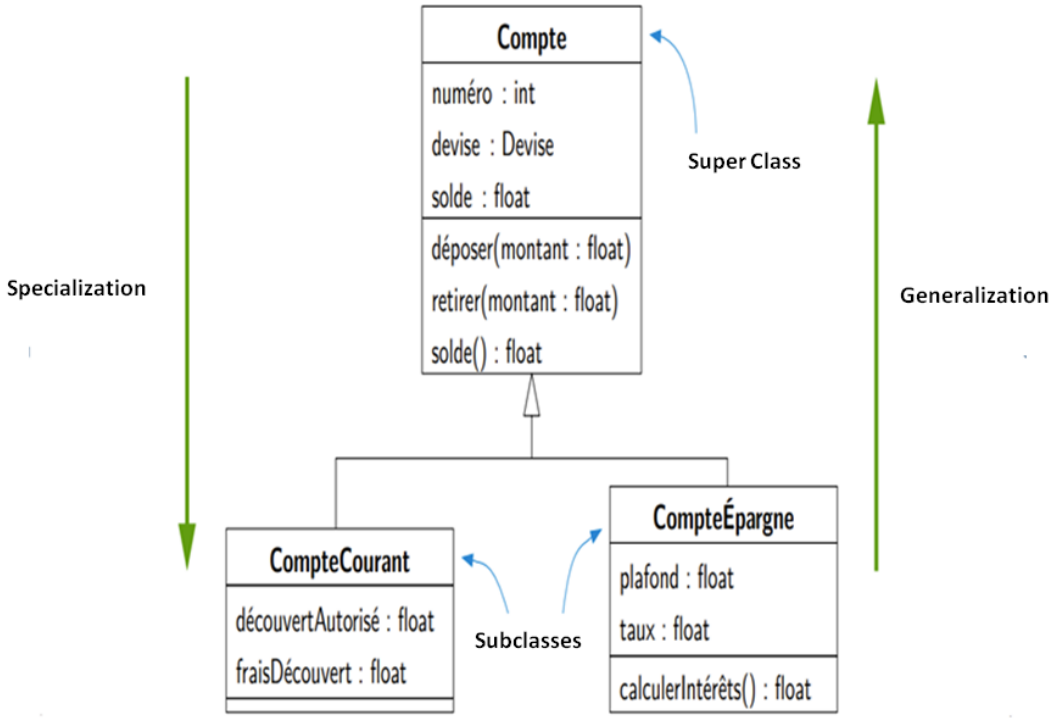

Figure 26 Classes with inheritance

Special Case: Reflexive Association in Class Diagrams: A **reflexive association** (or **self-association**) is a particular type of UML association where a **class is associated with itself**. This means that instances of the same class can be related to one another (**Figure 27**).

For example, in a class Employee, an association might indicate that **an employee can supervise other employees**. This creates a link between instances of the same class (Employee supervises Employee). In the class diagram, this is represented by a line that **loops back to the same class**, typically with role names at each end (e.g., manager and subordinate) and **multiplicities** to indicate how many instances can be related.

Reflexive associations are useful when modeling **hierarchical or networked relationships** within a single entity type.

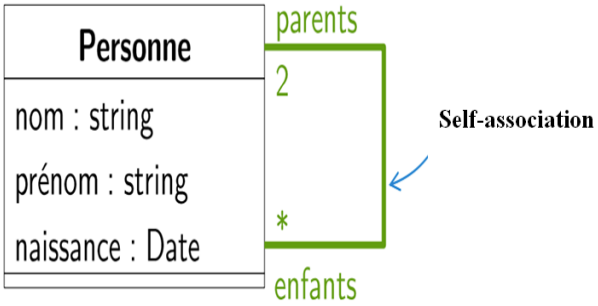

Figure 27 Self-association

Figure 28 shows an example of an object diagram associated with the previous class diagram.

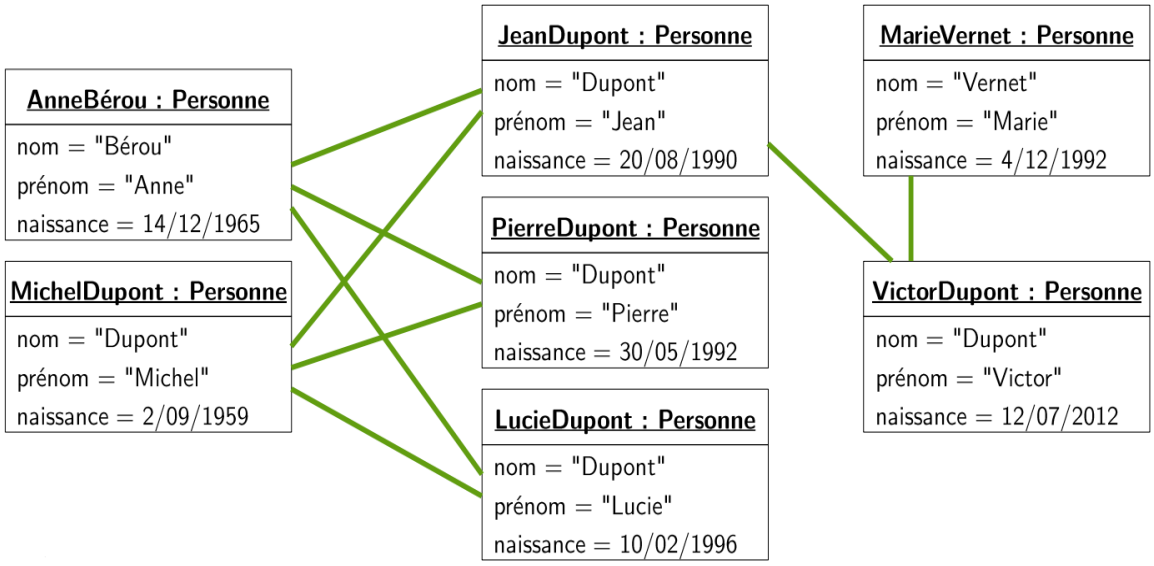

Figure 28 Example of an object diagram associated with the previous class diagram

Multiple associations: In the following class Diagram, between a ***Person (Personne)*** and an ***Apartment (Appartement)***, there may be a **tenant** relationship and an **owner** relationship **(Figure 29).**

Note: The fact that there are two associations **rents** and **offers (loue et propose)** between the two classes Person and Apartment **does not imply** that there are two relationships between every object pair *(*:Person and :Apartment) in the **object diagram**.

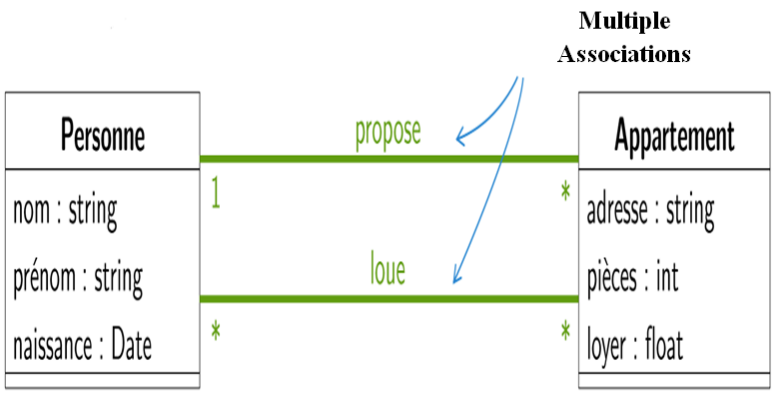

Figure 29 Example of a multiple associations

Figure 30 illustrates an example of an object diagram associated with the previous class diagram.

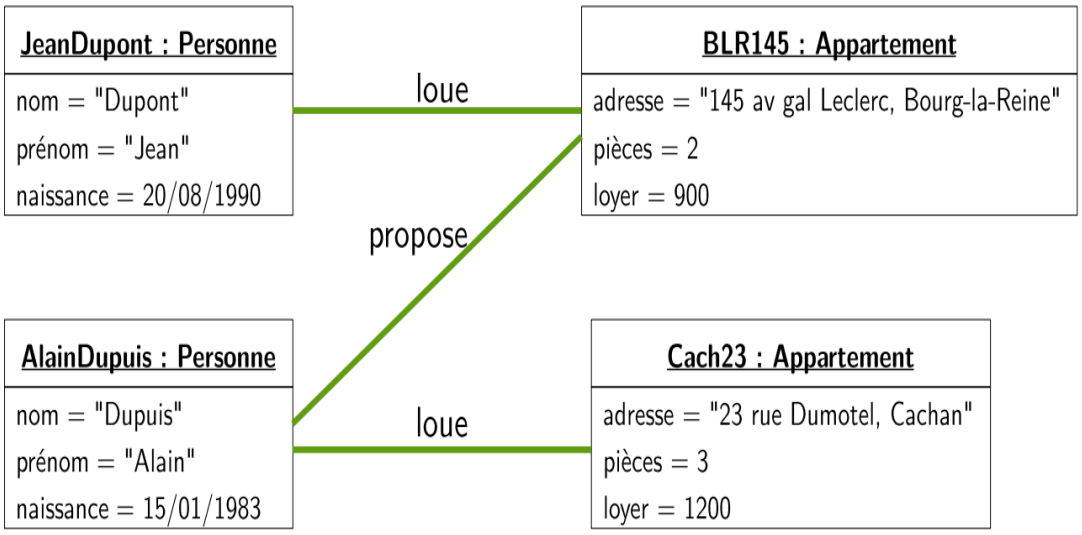

Figure 30 Example of an associate objects diagram

Association Class in UML: An **association class** is used when an association between two classes needs to carry additional information properties that do **not belong** to either of the associated classes individually but to the **relationship itself**.

For instance, consider the association Employment between the classes Company (Entreprise) and Person (Personne). This relationship may have attributes such as salary and hireDate (**Figure 31)**. These details **do not naturally belong** to the Company or the Person alone, especially when a **person can have multiple jobs** within the same company or **no job at all**. Likewise, a company may offer **one or several jobs** to the same person. This situation cannot be modeled accurately with a simple association.

The solution is to introduce an **association class** a hybrid element that behaves both like an association and a class. It is connected to the association line with a **dashed line** and can contain attributes just like any other class.

Important note on multiplicity: In this context, the multiplicities (e.g., *, 1..*) **do not refer to the number of companies or persons**, but to the **number of job instances** (Employment) between a given person and company.

-A person can have **zero or many** employment relationships (*).

-A company must offer at least **one** employment relationship to a given person (1..*).

This level of expressiveness is not achievable with basic associations, which only indicate **whether or not** a relationship exists, not **how many** instances of it occur or what properties it has.

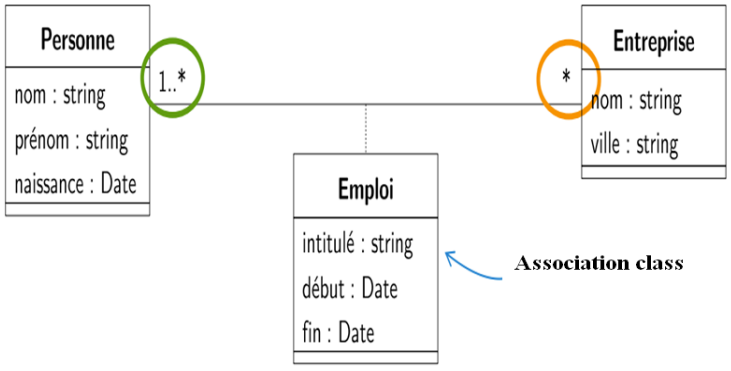

Figure 31 Association class

Figure 32 shows the equivalence in terms of classes and associations.

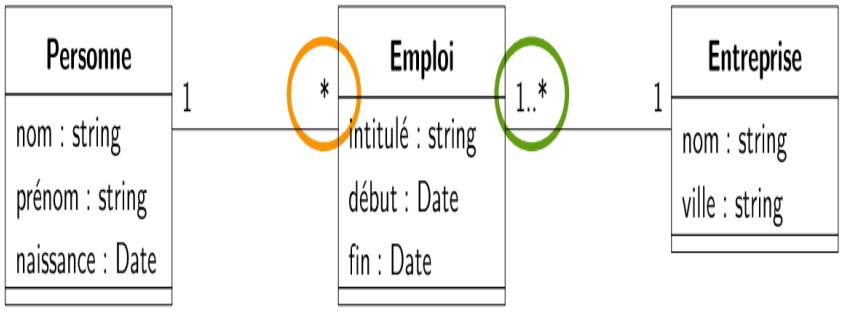

Figure 32 Associate simple classes diagram

Figure 33 illustrates an example of an object diagram (an instance of the previous class diagram).

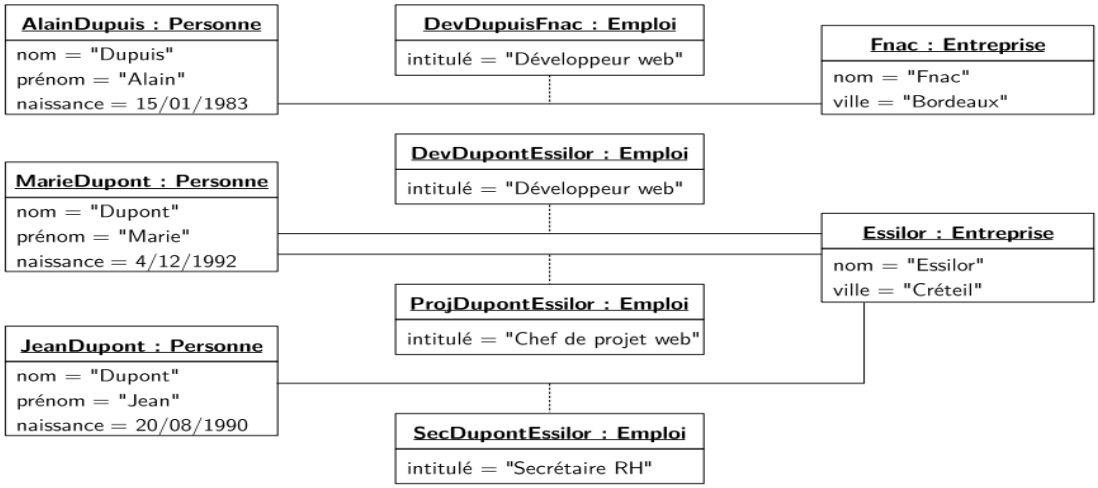

Figure 33 The associate objects diagram

Figure 34 illustrates the object diagram corresponding to the class diagram (without association classes)

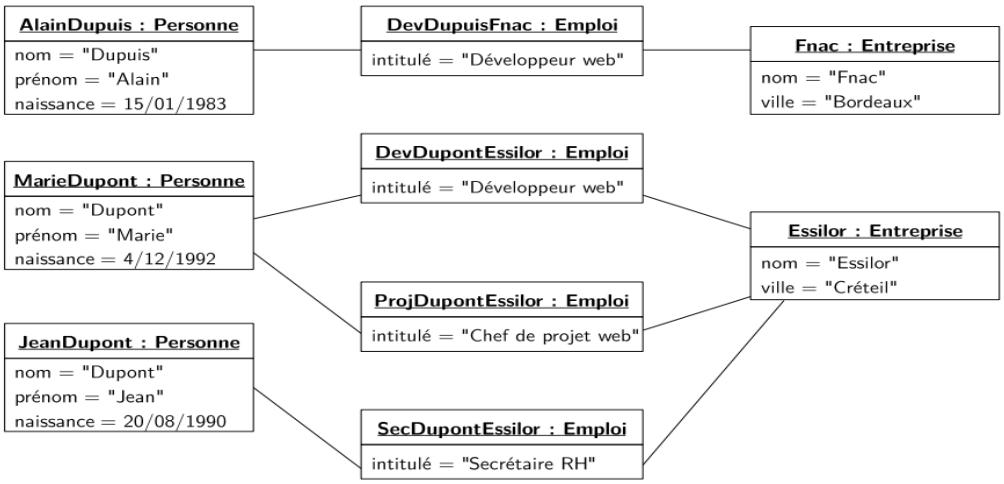

Figure 34 The associate class diagram (without association classes)

N-ary Association in UML: An **n-ary association** is a type of UML association that connects **more than two classes**. Unlike binary associations, which relate exactly two classes, an n-ary association involves **three or more participants** in a single, unified relationship.

A classic example is: "This student has this teacher for this course."This statement involves three entities Student, Teacher, and Course which are **simultaneously** related (**Figure 35)**. Representing this as three separate binary associations would not capture the precise relationship among all three roles.

In UML, an n-ary association is represented as a **diamond shape** connected to the involved classes by lines. Each line is labeled with a role name and may carry **multiplicities**.

Important note: Although n-ary associations are powerful, they are generally **discouraged unless truly necessary**, because they can be **difficult to understand, implement, and maintain**. In many cases, it is better to refactor an n-ary association into a **class with multiple binary associations** for example, by introducing a class like Assignment or Enrollment to model the interaction between student, teacher, and course.

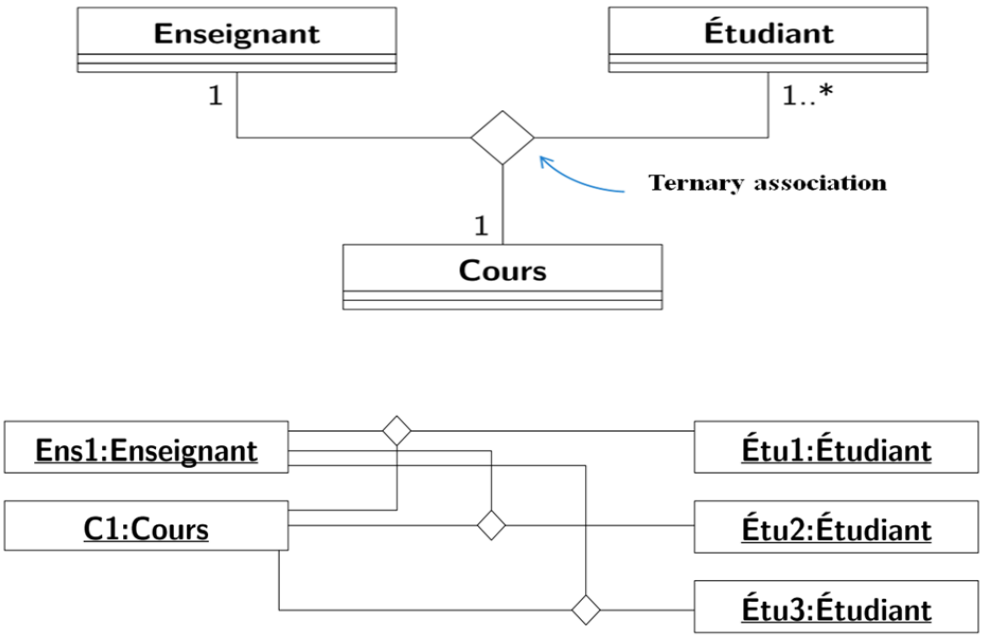

Figure 35 Example of ternary association

Aggregation in UML: Is a **special type of association** that represents a whole-part relationship between two classes. It is **asymmetric**, meaning that one class the **composite** is conceptually dominant over the other the **component**. Aggregation expresses that one object **contains** or is **composed of** others, but with varying degrees of dependency.

UML distinguishes **two types of aggregation**:

Shared Aggregation (Weak Aggregation)

This is a **loose** whole-part relationship. The component can exist **independently** of the composite. It is represented by a **hollow diamond** on the composite side. **Example*:*** A Team aggregates Players. A player may belong to multiple teams or none at all and still exist independently.

Composition (Strong Aggregation)

This is a **strong** form of aggregation where the component's **lifecycle depends** on the composite. It is represented by a **filled (black) diamond** at the composite end. ***Example*:** A House is composed of Rooms. If the house is destroyed, its rooms cease to exist as well.

6. Sequence Diagram

The **sequence diagram** is one of the **behavioral (dynamic)** diagrams in UML. It is used to represent the **interactions between system elements and external actors** over **time**.

More specifically, a sequence diagram shows **how objects and actors communicate** through **message exchanges**, organized chronologically. It is particularly effective for modeling **scenarios**, such as how a user performs a task or how different components of the system collaborate to complete an operation.

6.1 Core Elements:

Actors: External entities interacting with the system (e.g., users, external systems).

Objects: Internal elements of the system that collaborate to fulfill tasks.

Messages: Arrows representing communications (method calls, responses) between actors and objects, or between objects (**Figure 36).**

The sequence diagram provides a **graphical timeline** of how messages are exchanged either **within the system** or **between the system and its environment**. It highlights the **order of events** and the **temporal coordination** of system behavior, making it a valuable tool for understanding and specifying **system dynamics**.

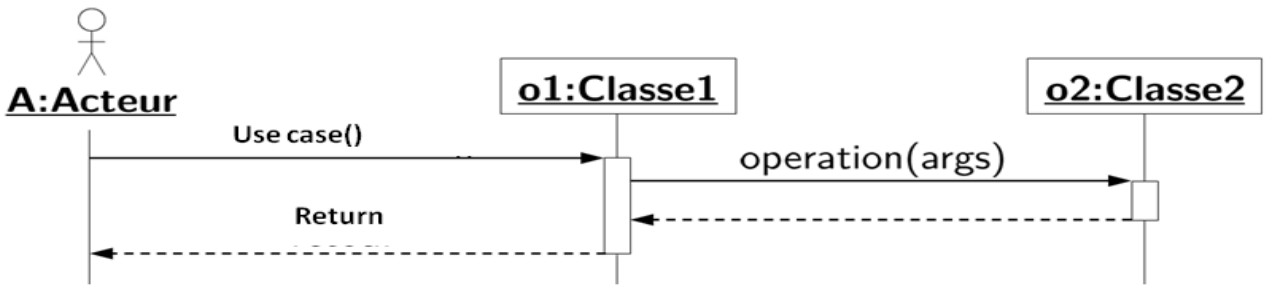

Figure 36 Sequence Diagram

Lifelines and Message Exchanges in Sequence Diagrams: In a **sequence diagram**, the **lifeline** of each entity (actor or object) is represented as a **vertical dashed line**. This line

illustrates the **existence of the entity over time**, starting from the moment it is created or starts interacting in the scenario.

The **horizontal arrows** between lifelines represent **message exchanges**. These messages can be: **Synchronous** (a response is expected) (**Figure 37)**, **Asynchronous** (no immediate response) (**Figure 38)**, or **return messages** (dashed lines indicating responses or results).

Vertical axis: Represents **time**, flowing **top to bottom** (the further down, the later the event).

Horizontal axis: Represents **communication** between different **lifelines** through **message passing**.

Figure 37 Synchronous message
Source: BookMyEssay. Synchronous Message in Sequence Diagram. Accessed June 3, 2025

Figure 38 Asynchronous message
Source: BookMyEssay. Asynchronous Message in Sequence Diagram. Accessed June 3, 2025

Objective during the Design Phase: To describe **how a use case is implemented** within the system, based on the structure defined in the **class diagram (Figure 39).**

The goal is to **map the dynamic behavior** (as described in the use case) onto the **static architecture** of the system, identifying which classes are responsible for which parts of the functionality.

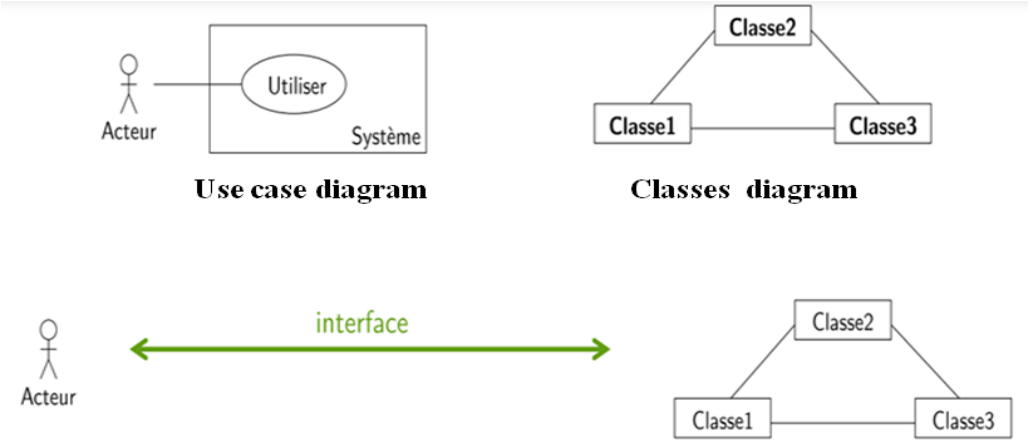

Figure 39 Role of sequence diagram

Object Creation and Destruction in Sequence Diagrams: In UML **sequence diagrams**, it is possible to represent the **creation and destruction of objects** as part of the system's dynamic behavior. An **object is created** when a specific message is sent to instantiate it. This is visually represented by a **dashed lifeline** that begins below the top of the diagram, indicating that the object did not exist before the creation message. The **message that creates the object** is typically a constructor or initialization method, drawn as a solid horizontal arrow pointing to the **object's lifeline at its start**.

On the other hand, the **destruction of an object** is represented by placing a large **"X"** at the end of the object's lifeline. This indicates the moment when the object ceases to exist, usually after receiving a message that triggers its termination or deletion. After this point, the object cannot participate in any further interactions (**Figure 40).**

Representing creation and destruction is particularly useful for understanding **object lifecycles**, **resource management**, and the **scope of interactions** in a system scenario. It helps designers visualize **when objects are active**, and **how long they live** during the execution of a use case.

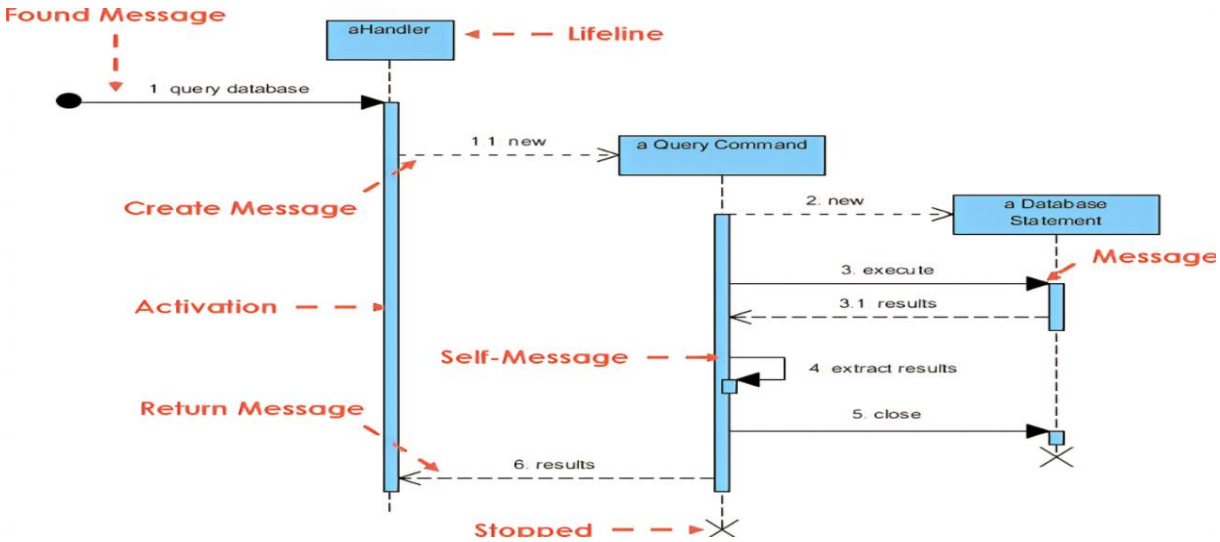

Figure 40 Creation and destruction of an object

Alternatives in Sequence Diagrams: In UML sequence diagrams, the **alternative (alt)** construct is used to model **conditional behavior**, meaning that **only one of several possible message flows** will occur, depending on a specified condition. This construct is useful for representing **"if-else" logic** within the interaction between objects and actors.

The **alt block** is represented as a **frame** (a rectangular container) labeled alt, which is divided into **two or more compartments**, each corresponding to a possible **execution path**. Each compartment is annotated with a **guard condition**, written between square brackets (e.g., [condition1] or [else]), which determines whether the messages inside that compartment will be executed **(Figure 41).**

Only the messages in the **first compartment whose guard evaluates to true** will be executed during a scenario. If none of the conditions hold, and there is no else branch, then no action is taken.

The use of alternatives allows designers to **capture decision logic explicitly**, making system behavior **clearer and more traceable** in scenarios where **multiple possible outcomes** exist.

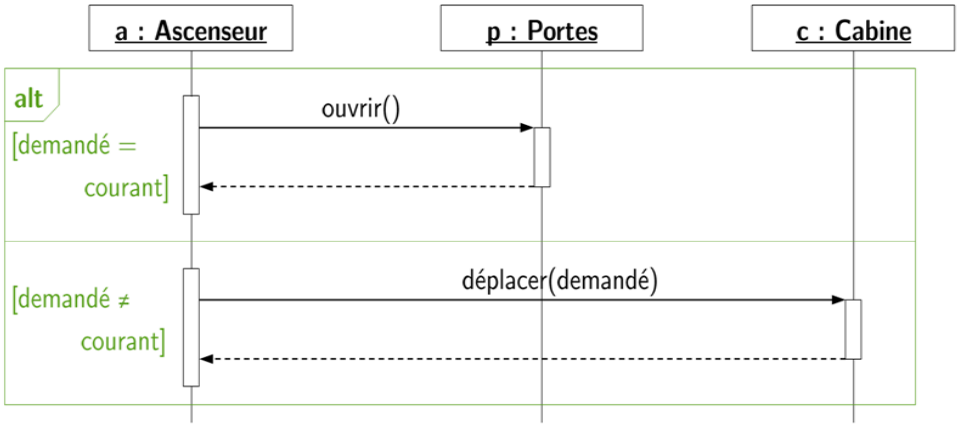

Figure 41 Alternative bloc in a sequence diagram

Loops in Sequence Diagrams: In UML sequence diagrams, a **loop** is used to represent the **repetition of a sequence of messages** under a specified condition. This is particularly useful for modeling **iterative behaviors**, such as traversing a list, retrying an operation, or repeating a request until a certain condition is met (**Figure 42).**

Loops are depicted using a **frame** labeled loop, which surrounds the sequence of messages to be repeated. A **guard condition** is placed inside the loop frame, written in square brackets (e.g., [i < 5] or [while moreItems]), and it determines how long the sequence should be repeated. The loop executes **as long as the condition evaluates to true**.

In some cases, a **note** may be added outside the frame to provide further explanation, such as the number of iterations or a reference to a specific algorithm. Loops in sequence diagrams help clarify **repetitive processes** and ensure that the logic of repeated interactions is explicitly documented in the design.

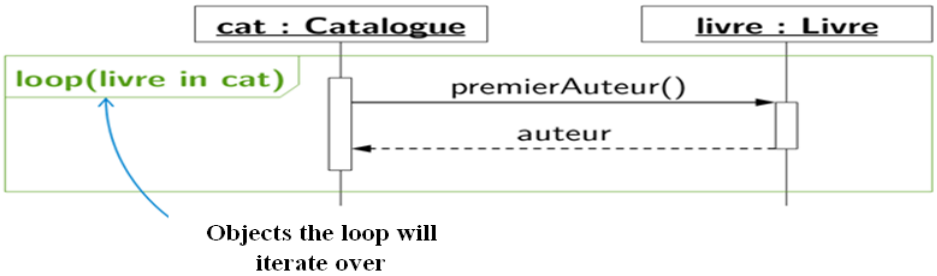

Figure 42 Loop in a sequence diagram

Referencing another Diagram in a Sequence Diagram: In UML sequence diagrams, it is possible to **refer to another interaction or diagram** using a construct called an **interaction use**. This feature allows designers to **reuse predefined interactions** in different contexts, which helps simplify complex diagrams and avoid redundancy.

The reference is represented using a **"ref" frame**, labeled ref, which contains the **name of the referenced interaction** along with any **parameters or conditions** required. This frame acts as a **placeholder** for the detailed sequence of messages described elsewhere (**Figure 43).**

Using interaction references promotes **modularity** and **clarity** by breaking down a large scenario into smaller, more manageable components. It also supports **consistency** across multiple diagrams and makes maintenance easier when updating system behavior.

This technique is particularly useful in large systems where several use cases share common sub-interactions, such as authentication steps, logging, or data validation.

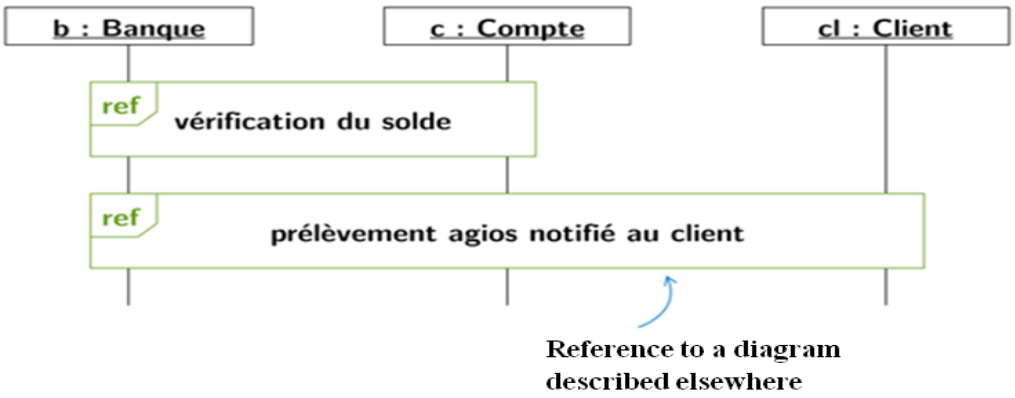

Figure 43 Referencing the Diagram

Exercises

Exercise 1

In a store, a shopkeeper uses a stock management system for articles, with the following functionalities:

-Editing a supplier's profile.

-Possibility to add a new article, which requires first editing the supplier's profile. If the supplier does not exist, it can then be created.

-Editing the inventory. From this screen, the user can choose to print the inventory, delete an article, or edit an article's profile.

Determine the appropriate use case diagram.

Exercise 2

The fourth-year Industrial Engineering students at ESSAT have decided to create a new club within the school. To organize properly, they established a clear hierarchy, with specific positions assigned tasks and privileges:

- The club president is elected by the club's general assembly from among the candidates running for the position.
- The general assembly is composed of the club's founding members.
- When a member leaves the general assembly, their replacement is elected by the assembly from the candidates running for the position, with conditional approval from the president.
- The club treasurer is appointed by the president with conditional approval from the general assembly and is responsible for managing the club's treasury.
- The general assembly has the sole right to monitor and review the treasurer's work by approving or rejecting the annual financial report presented by the treasurer. The financial report is signed by the president once approved by the assembly.
- The event manager is appointed by the president. They are responsible for booking venues for club events, sending invitations, coordinating with caterers for meals (buffets, table service, coffee breaks, etc.). However, no expenses (e.g., signing a purchase order) can be made without the conditional approval of the treasurer.
- The communications officer manages the club's various communication campaigns, finds partners, attracts sponsors, gathers feedback from different audiences, conducts surveys, etc. However, contract signing with partners or sponsors requires approval from the general assembly.
- The meeting secretary is solely responsible for drafting the minutes of each general assembly meeting. This is a rotating position based on seniority among the assembly members.

The following figure illustrates a UML use case diagram that represents the club's operations. It contains 7 errors. Identify these errors and correct them, explaining why each correction is necessary.

Exercise3: A company wants to improve its ERP by introducing new functionalities. This concerns two departments:

Human Resources: The company wants to manage its databases of absences, employees, and job descriptions. The company has:A personnel file containing key employee data: Employee ID, Last Name, First Name, Date of Birth, Hiring Date, Job Position.

A job description file containing: Job Code, Job Title, Base Salary, Main Tasks. Upon arrival, employees clock in using a time clock before starting their shift, which automatically records each punch-in in the database. To achieve this, developers want to modify the ERP to display the following interfaces:

Procurement: The company wants to manage its databases of purchase orders, products, and suppliers. The company has: A products file containing key product data: Product Code, Product Type, Weighted Average Price, Stock Quantity.

A suppliers file containing: Supplier Name, Business Name, Address.

Please create UML class diagrams for each of the two modules (Human Resources and Procurement), indicating class names, their attributes, the relationships between them, and their multiplicities.

Note: The two modules (and thus their diagrams) are completely independent from each other.

Exercise 4:

Draw the sequence diagram for the use case **"Perform a Personal Transfer"**, based on the class diagram provided below (**Figure 44)**. Then, complete the class diagram if necessary after creating the sequence diagram.

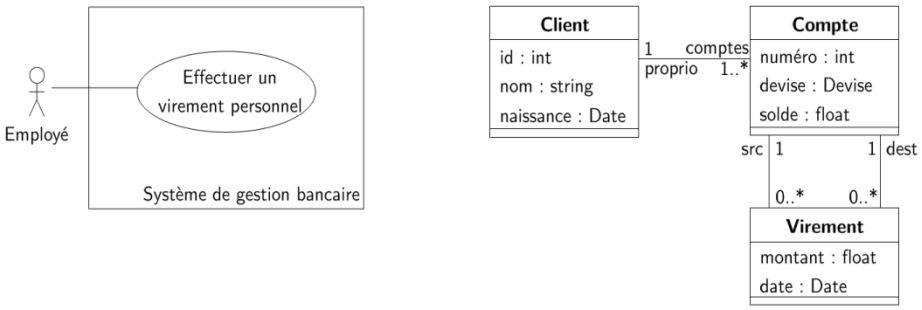

Figure 44 Use case and class diagrams

References:

[1]: Logistiikan Maailma. (2024, December 30). *Information, money and material flow*. Retrieved from https://www.logistiikanmaailma.fi/en/logistics/logistics-and-supply-chain/information-money-and-material-flow/

[2]: Laudon, K. C., & Laudon, J. P. (2020). *Management Information Systems: Managing the Digital Firm* (16th Edition). Pearson.

[3]: SOLIDitech. (2021, July 13). *An Introduction to Centralised Databases*. The SOLID Blog. https://blog.soliditech.com/blog/an-introduction-to-a-centralised-databases

[4]: Monk, E., & Wagner, B. (2012). *Concepts in Enterprise Resource Planning* (4th Edition). Cengage Learning.

[5]: Booch, G., Rumbaugh, J., & Jacobson, I. (2005). *The Unified Modeling Language User Guide* (2nd ed.). Addison-Wesley.

[6]: Booch, G., Rumbaugh, J., & Jacobson, I. (1999). *The Unified Modeling Language User Guide*. Addison-Wesley.

[7]: **BookMyEssay.** (2021, January). Synchronous message in sequence diagram [Image]. BookMyEssay. https://www.bookmyessay.co.uk/wp-content/uploads/2021/01/Synchronous-Message-768x513.jpg

[8]: BookMyEssay. (2021, January). [Asynchronous message in sequence diagram] [Image]. BookMyEssay. [https://www.bookmyessay.co.uk/blog/visualizing-system-design-using-uml-based-sequence-diagrams/]