⋆˚₀⋆✧˚ Chapter 5: Behavioral Modeling: Use Case Diagrams ˚✧⋆₀˚⋆

In every professional field, modeling serves as a vital bridge between a concept and its creation. While we have previously explored the static "blueprints" of a system's structure, we now turn our focus to the dynamic forces that drive it. Just as an aeronautical engineer must model not only the shape of a wing but also how it reacts to changing wind speeds, software developers use **Behavioral Modeling** to capture how a system functions in the real world.

Today, behavioral modeling—starting with the **Use Case Diagram**—is a cornerstone of successful IT projects. It addresses the reality that a system's internal logic is meaningless if it does not align with the user's needs. By creating a Use Case model, we achieve four critical goals: we **visualize** the interactions between the system and its environment, **specify** the functional requirements from an external perspective, **provide a template** for testing and validation, and **establish a clear scope** that manages stakeholder expectations.

Ultimately, Use Case modeling is the art of defining the system's purpose, ensuring that the final product remains useful, intuitive, and perfectly aligned with the goals of those who will use it.

1. Use case Definition

The **Use Case Diagram** is a unique graphical model used to capture how external entities expect to interact with a system. Unlike other diagrams that focus on internal logic, this diagram describes the users involved, the services they require, and the services they must provide to the system.

The use case diagram is a cornerstone of the **Object-Oriented Software Engineering (OOSE)** method, originally known as **Objectory** (Object Factory). Published in 1992 by **Ivar Jacobson**, the technique proved highly effective for large-scale projects at companies like Ericsson and HP.

Its success caught the attention of other leading methodologists, such as **Grady Booch** and **James Rumbaugh**, whose methods lacked a dedicated way to document requirements from a user-centric perspective. Consequently, the use case diagram was formally integrated into UML 0.9 in June 1996.

A diagram alone is rarely sufficient. In professional practice, the diagram is part of a comprehensive three-tiered approach:

- **The Use Case Diagram:** Provides a high-level visual map of system boundaries and stakeholders.
- **The Use Case Narrative:** A detailed textual description addressing specific functional requirements.
- **The Use Case Scenarios:** Exploration of different execution paths (options) used to test requirements and provide a high-level test plan for future development.

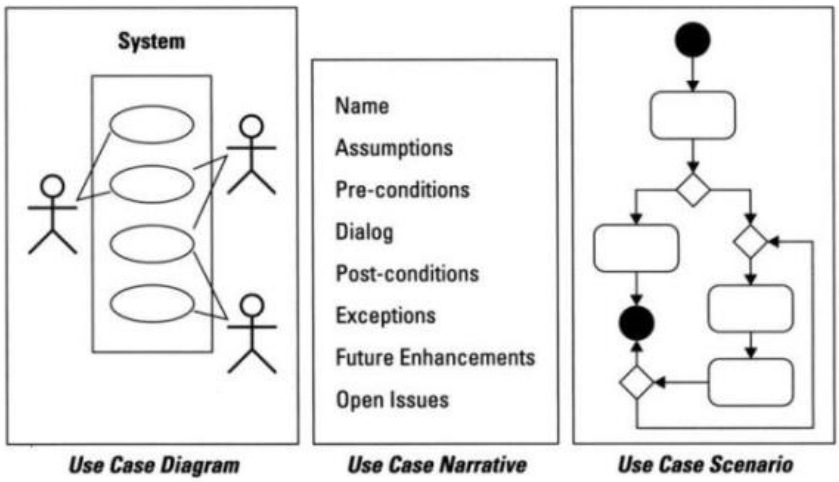

Figure 29. The use case diagram as a part of a comprehensive three-tiered modeling

The Use Case Diagram is composed of six essential elements that define the system's external behavior:

- **Actor:** Represents a role played by a person, system, device, or organization that interacts with the system.
- **Use Case:** Identifies a key system behavior or goal. Without this specific behavior, the system fails to meet the actor's requirements.

- **Association:** A line identifying the interaction between an actor and a use case, representing a "dialogue" that is later detailed in the narrative.
- **The «include» Relationship:** Represents a reusable use case that is **unconditionally** incorporated into the execution of a "base" use case. The base use case controls when this happens.
- **The «extend» Relationship:** Identifies a reusable use case that **conditionally** interrupts a base use case to add functionality. The decision logic resides within the extension itself.
- **Generalization:** An inheritance relationship where one actor or use case inherits the properties and behaviors of a more general one.

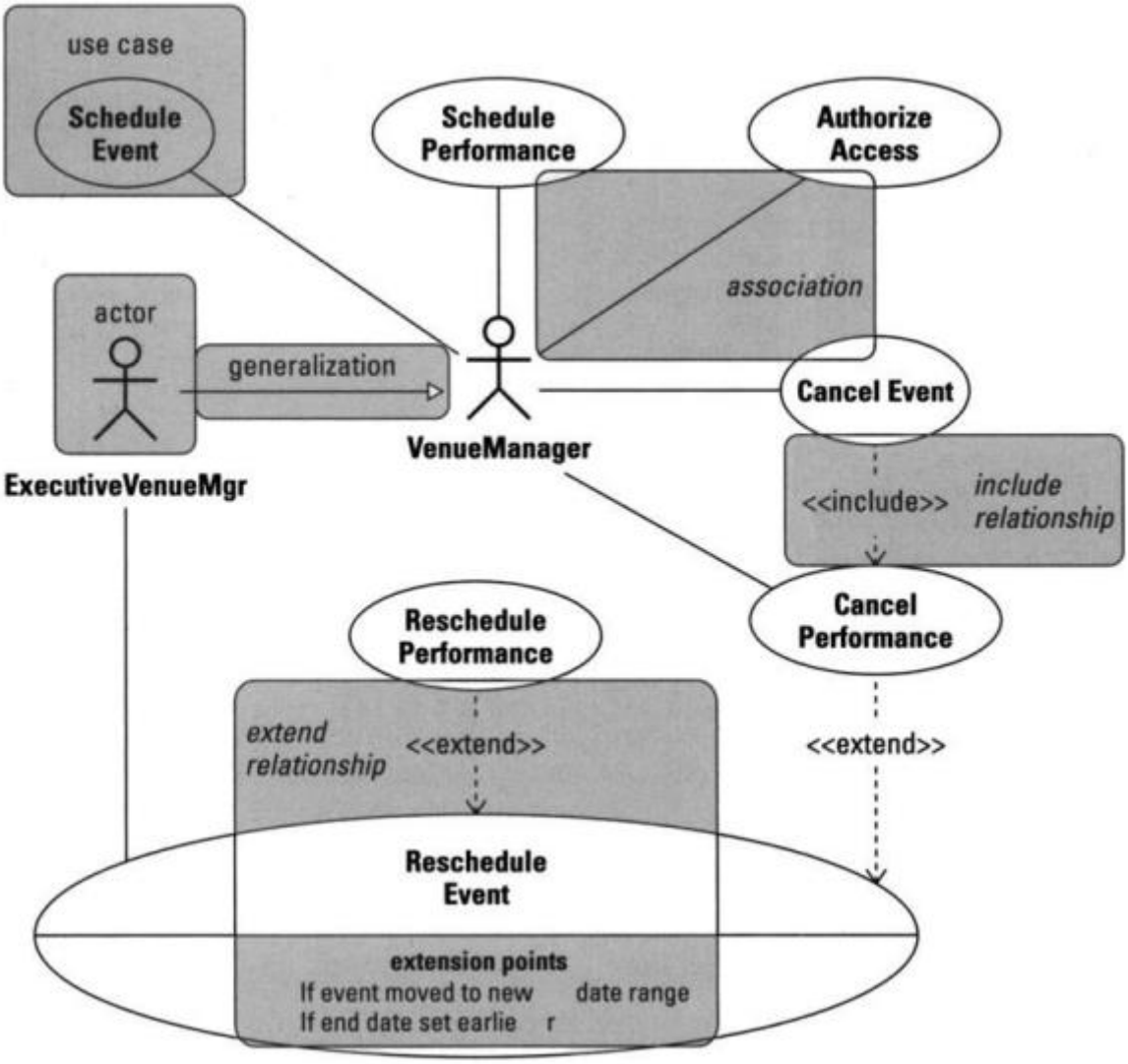

Figure 30. Use case diagram example of a venue management

2. Steps for Creating a Use Case Diagram

Developing a well-structured use case diagram is an iterative process that moves from identifying broad interactions to refining specific relationships.

Step 1: Define the System Context

The first goal is to establish the boundaries of your system and identify its primary participants.

- **Identify the Actors:** Determine who or what (people, external systems, or devices) interacts with the system and define their specific responsibilities.
- **Identify the Use Cases:** List the core system behaviors required to meet the actors' goals. Each use case should represent a discrete result or value produced by the system.

Step 2: Refine Actors and Use Cases

Evaluate your initial list to ensure clarity and appropriate granularity.

- **Merge or Split:** If two use cases perform nearly identical tasks, merge them. Conversely, if a single use case is too complex, split it into smaller, more manageable units.

Step 3: Identify «include» Relationships

Look for common sequences of behavior shared across multiple use cases.

- **Factor Out Commonality:** If several use cases require the same step (e.g., "User Authentication"), extract that behavior into its own use case and use an «include» relationship to avoid redundancy.

Step 4: Identify «extend» Relationships

Search for optional or exceptional behaviors that only occur under specific conditions.

- **Capture Conditional Logic:** Identify points where a "base" use case might be interrupted to perform an additional task (e.g., "Calculate Discount" during a "Checkout" process) using an «extend» relationship.

Step 5: Detect Generalizations

Analyze both actors and use cases to find shared properties or behaviors.

- **Apply Inheritance:** If multiple actors share the same basic rights but have specialized roles (e.g., a "Manager" who inherits from "Employee"), use **Generalization** to simplify the model and show the hierarchy.

3. Elements of the use case diagram

To build an effective Use Case diagram, you must understand the specific graphical symbols and the logic governing their interactions.

- **The Actor**

In UML, the term **Actor** refers to anything external that interacts with the system, including people, other software systems, hardware devices, or even external organizations. While often depicted as a "stick figure," the icon can vary depending on the actor's nature (e.g., a box for a system). Actors should represent **roles**, not specific individuals. For instance, "Store Manager" is a role, while "John Doe" is an instance.

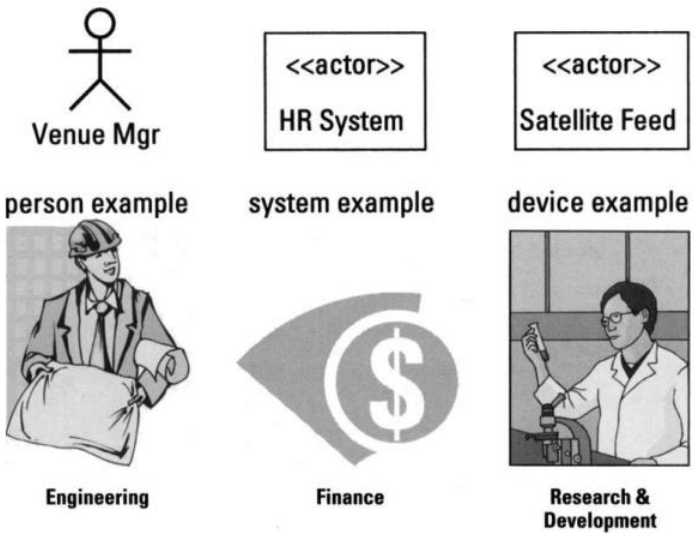

Figure 31. Actor representation in Use case diagram

The representation of an actor could use generalization, for example, specific roles like PrivateCustomer and CorporateCustomer inherit behaviors and properties from a more general Customer actor.

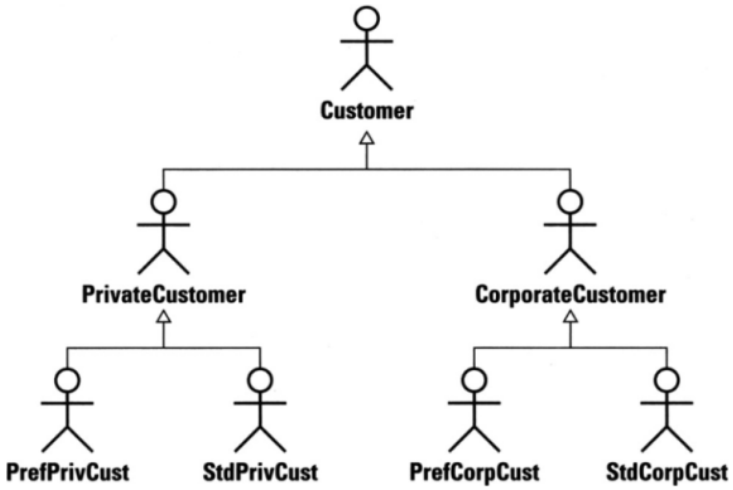

Figure 32. Actor generalization representation in use case diagram

- **The Use Case**

A Use Case represents a high-level system goal or a discrete service provided to an actor. Use cases are always named with a **verb phrase** (e.g., Create Agent Contract, Reschedule Program). The focus remains strictly on **what** the system achieves for the user, rather than the internal technical process of how it happens.

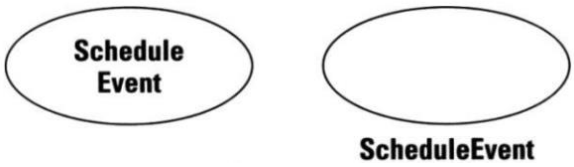

Figure 33. Use case representation

- **The Association**

The association is the simplest form of relationship, represented by a **solid line** connecting an actor to a use case. It indicates that a "dialogue" or communication occurs between the two.

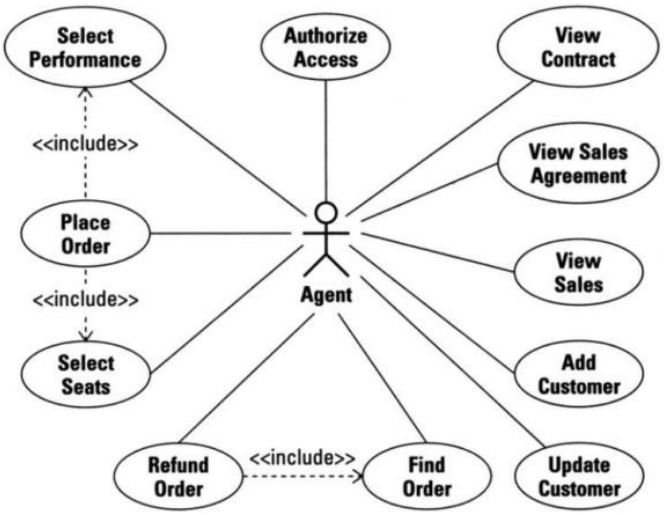

Figure 34. Association representation in use case diagram

- **The «include» Relationship**

This relationship is used to extract common behavior into a separate, reusable use case. It is governed by two strict constraints:

- **Dependency:** The calling (base) use case depends only on the *result* of the included behavior.
- **Unconditionality:** The calling use case **always** requires the execution of the included behavior to be complete.

The **«include» Relationship** is represented by a dashed arrow points **from** the base use case **to** the included use case.

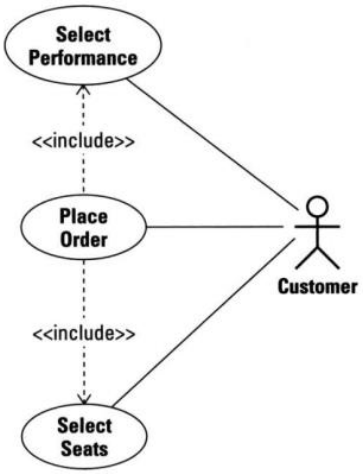

Figure 35. Representation of the «include» Relationship in use case diagram

- **The «extend» Relationship**

The «extend» relationship allows for optional or exceptional behavior to be added to a base use case without modifying its core logic. It requires four distinct parts:

- **Base Use Case:** The primary process that can be extended (e.g., Reschedule Event).
- **Extended Use Case:** The additional behavior that executes only if needed (e.g., Cancel Performance).
- **Extension Relationship:** A dashed arrow pointing **from** the extended use case back **to** the base use case.
- **Extension Points:** Specific locations within the base use case where a **condition** is checked. If the condition is met, the extension interrupts and executes.

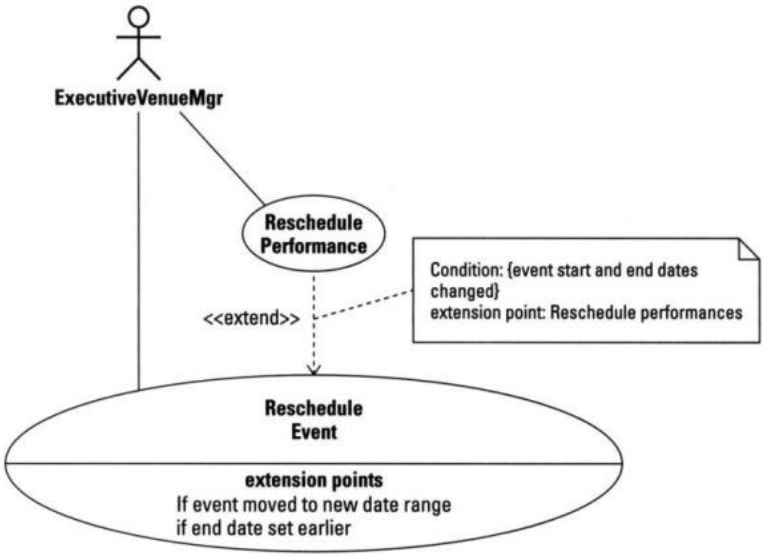

Figure 36. Representation of the «extend» Relationship in use case diagram

⋆˚₀⋆✧˚ Chapter 6: Structural Modeling: Class Diagrams ˚✧⋆₀˚⋆

If Use Case diagrams are the "voice" of the user's needs, **Class Diagrams** are the "skeleton" upon which the entire system is built. In structural modeling, we shift our perspective from the external behavior of the system to its internal composition. By defining the static structure, we establish the fundamental blueprint that developers use to translate abstract requirements into concrete, object-oriented code.

Today, class modeling is the primary tool for managing the internal complexity of IT projects. It addresses the challenge of organizing data and logic into a cohesive, manageable whole. By utilizing Class Diagrams, we achieve four critical objectives: we **visualize** the core entities and their properties, **specify** the exact relationships and multiplicities between them, **provide a direct template** for physical database and class construction, and **establish the technical documentation** necessary for long-term maintenance.

Ultimately, structural modeling ensures that the foundation of our software is robust and logical. It allows us to refine the system's "vocabulary"—the classes and their associations—ensuring that the final architecture is scalable, modular, and perfectly capable of supporting the behaviors defined in earlier stages of design.

1. Definition

The **Class Diagram** is the most widely used modeling tool in UML. Its importance is unique because it serves as the primary diagram for **code generation**, allowing developers to translate visual models directly into executable software. While other diagrams focus on behavior, the class diagram defines the system's "vocabulary"—the essential building blocks and relationships that remain constant regardless of the system's state. Class diagrams are versatile enough to model everything from aircraft navigation and medical equipment to factory automation and billing systems. Despite the variety of these applications, they all rely on a core set of elements: **classes** and **relationships**.

2. Modeling classes

A class is a blueprint or definition of an entity. It encapsulates the **characteristics** (states) and **operations** (behaviors) of that entity. It is crucial to distinguish between **class**, which is the general definition (e.g., Car), and an **object** which is a specific, unique instance of that class (e.g., myRedSedan). In UML, a class is represented as a rectangle divided into three compartments: the **Name**, the **Attributes**, and the **Operations**.

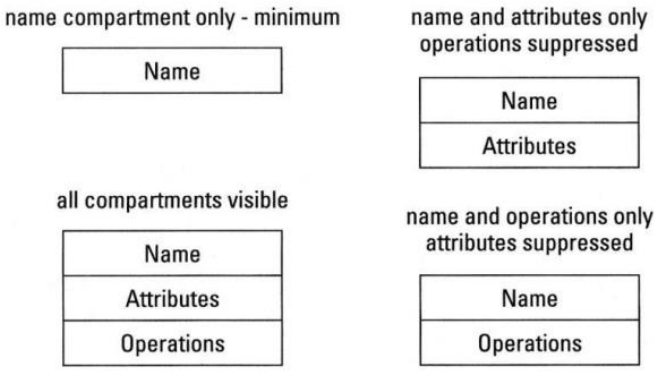

Figure 37. Different representations of a class

- **The Class Name**

The name identifies the class and must be a precise, concise descriptor. It must avoid including attributes in the name. For example, use Employee instead of ExemptEmployee, as "Exempt" is a state/attribute, not a distinct type of entity. If a class belongs to a specific package, it can be identified using the format Package_Name::Class_Name.

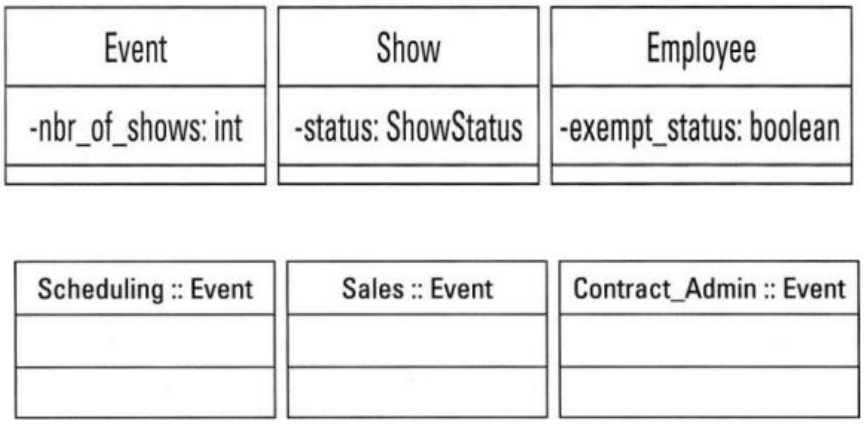

Figure 38. Example of representation of a class name

UML allows also the use of stereotypes to define specialized types of classes:

- **«entity»:** Describes objects that form the core subject of the system (e.g., Venue, Seat).

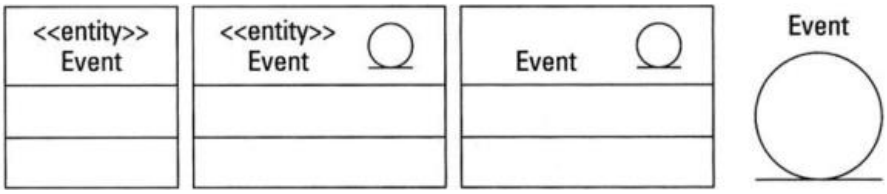

Figure 39. Example of representation of entity class

- **«control»:** Describes objects that manage application logic (e.g., Scheduling).

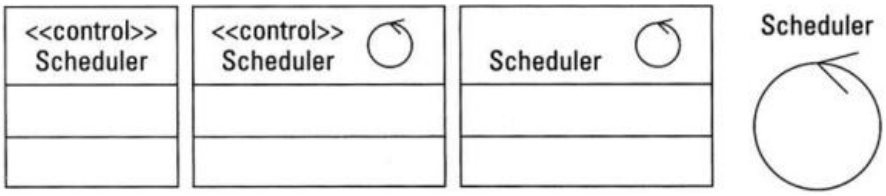

Figure 40. Example of representation of control class

- **«abstract»:** Represents a class that cannot be instantiated on its own (often written in *italics*).

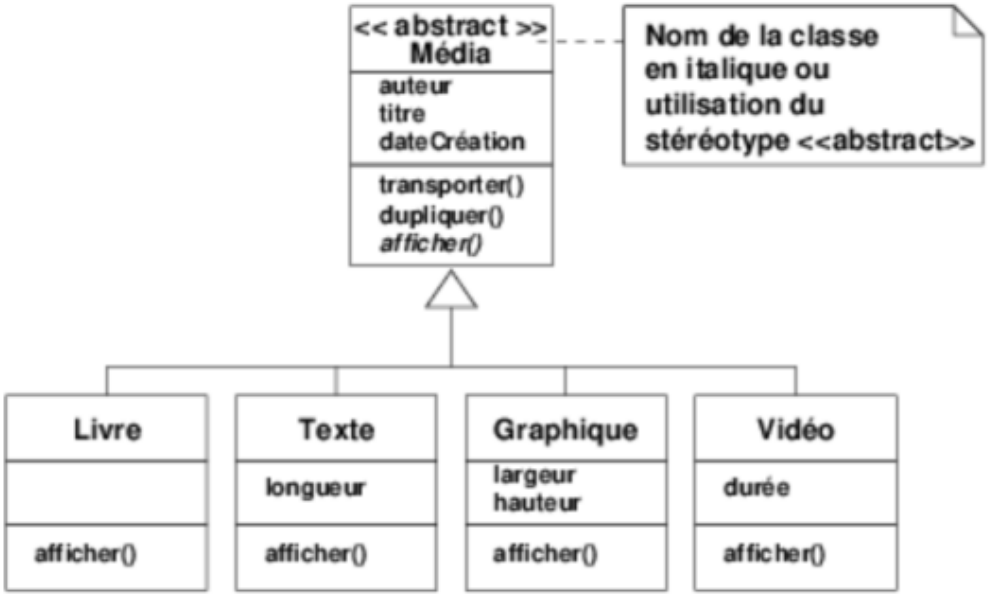

Figure 41. Example of representation of abstract class

- **«enumeration»:** A user-defined data type representing constant values (e.g., AccountType).

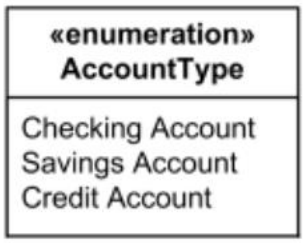

Figure 42. Example of representation of abstract class

Above the class name, we can also add **properties** using **tagged values**.

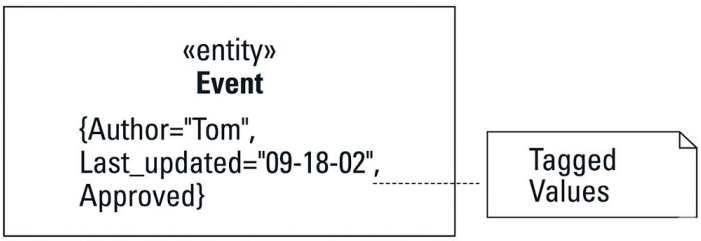

Figure 43. Example of representation of class using tagged values

- **Attributes**

Attributes represent the information an object possesses. For example, in a theater system, different objects provide specific information about the overall operation. The **Venue** object contains details such as its seating capacity and current status, which may be open, closed for holidays, closed for repairs, or reserved for a private event. The **Event** object provides information about its start and end dates, the number of shows already scheduled, the maximum number of shows that can be scheduled, and its current status, which can be tentative, contracted but not yet scheduled, scheduled, completed, or canceled. Meanwhile, the **Seat** object describes its location within the venue as well as its current condition, indicating whether it is disabled or available for use.

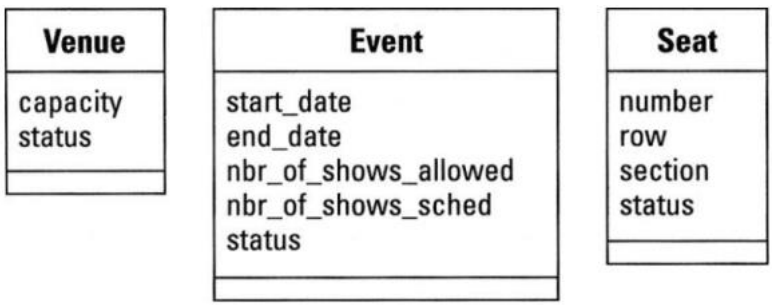

Figure 44. Example of representation of attributes in class diagram

In UML, they follow a specific syntax: **[visibility] [/] name [: type] [multiplicity] [=default] [{property-string}]**

- **Visibility:** Defines who can access the data: **Private (-)**, **Package (~)**, **Public (+)**, or **Protected (#)**.

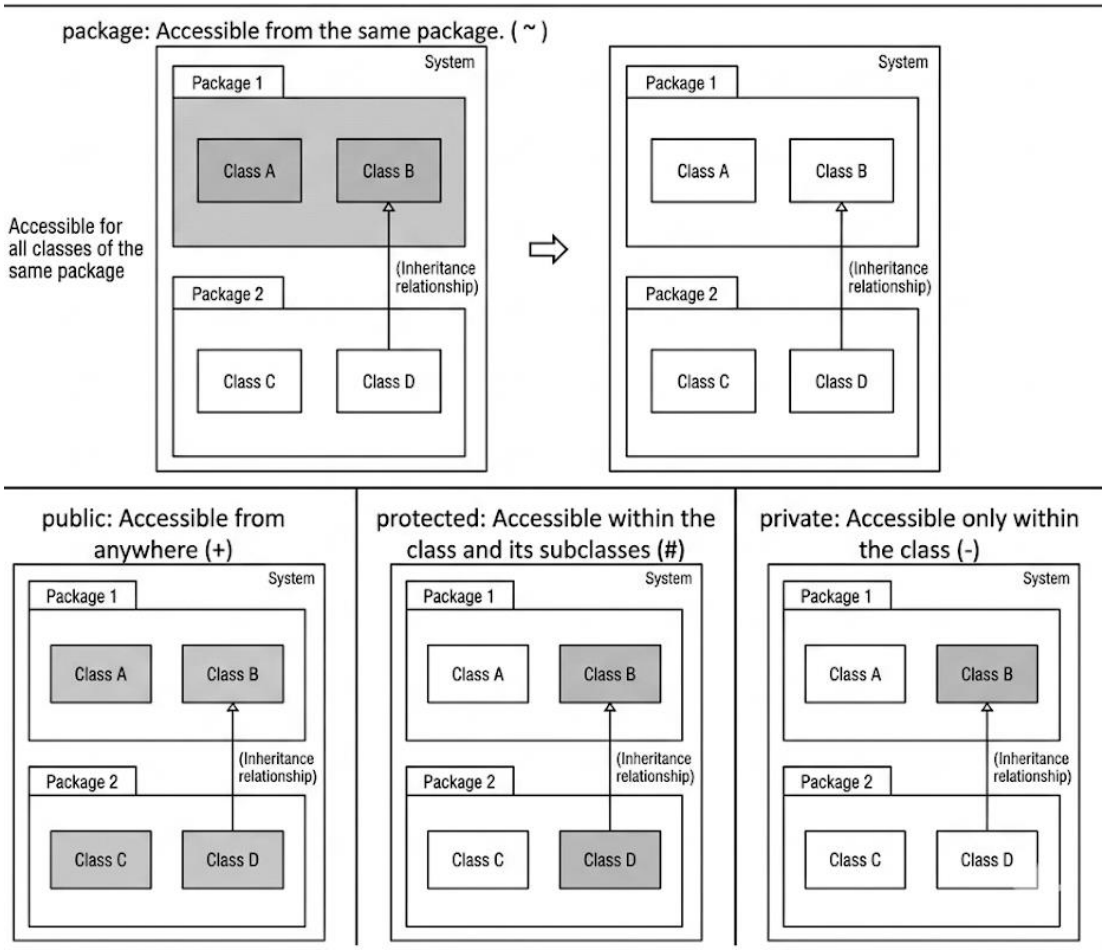

Figure 45. Different representations of attributes visibility

- **Derived Values (/):** Indicated by a forward slash, these are values calculated from other data (e.g., /age derived from birthDate).

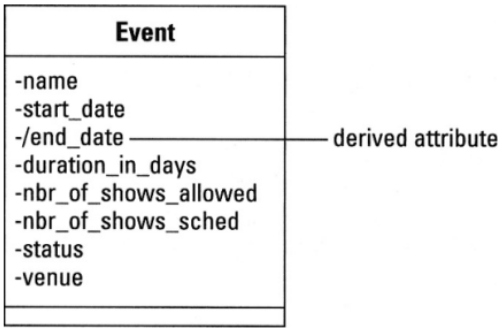

Figure 46. Example of derived attributes in a class

- **Name:** It must be unique within the class.
- **Data type:** It represents the nature of the information that can be stored in the attribute. It can refer to a data type (e.g., *integer*, *float*, *long*, *short*, *string*, etc.), or to another class in the system, or to an enumeration class.

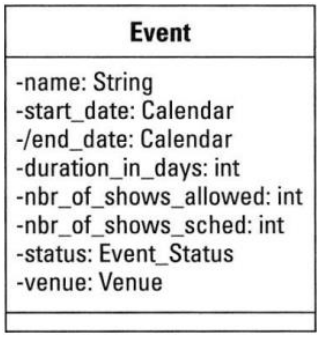

Figure 47. Example of type attributes in a class

- **Multiplicity:** Indicated by [], this defines how many values an attribute can hold (default is 1).

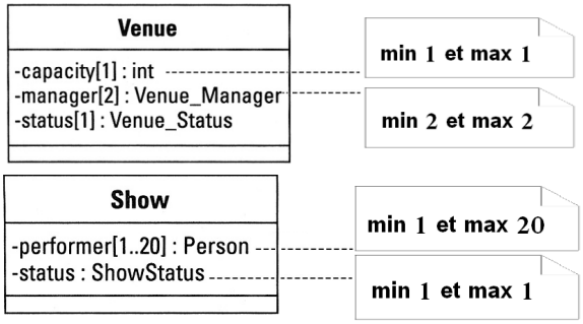

Figure 48. Example of attribute multiplicity in a class

- **Default values**: They help protect the system from being corrupted due to missing or invalid values and provide significant and easy-to-use improvements for the client.

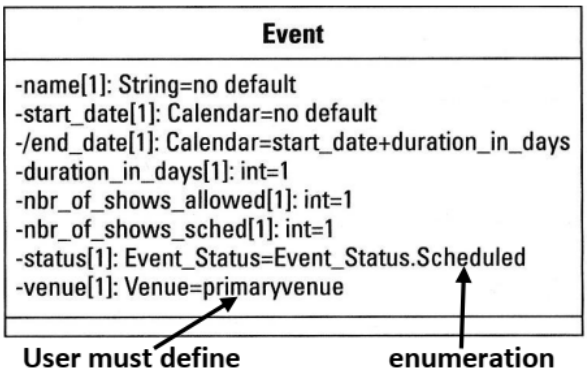

Figure 49. Example of default values of attribute in a class

- **String properties**: A text including any other information regarding the attribute. It is written between { }.

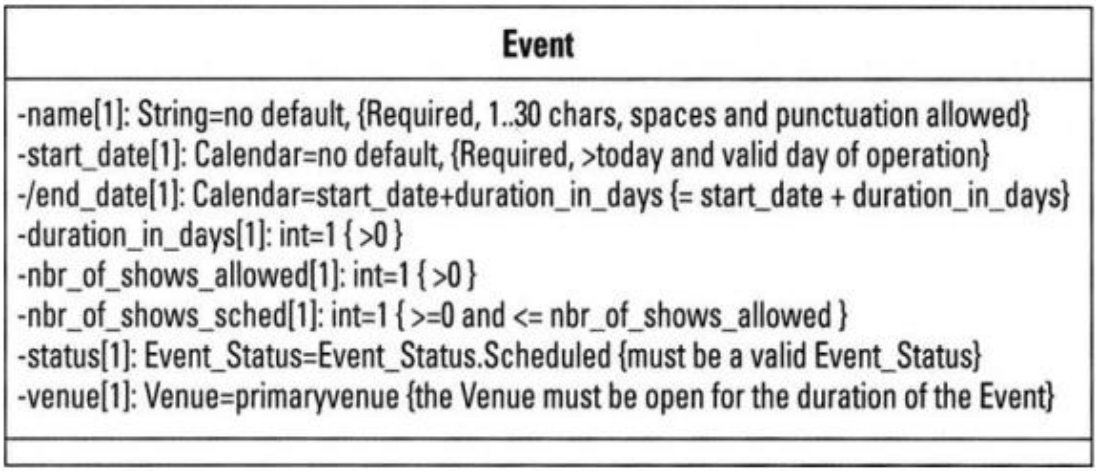

Figure 50. Example of String property of attribute in a class

- **Static Attributes:** While most attribute values belong to a specific object, an instance of a class (called instance-level attributes), **Class-level attributes** are shared by all instances of a class and are **underlined** in the diagram.

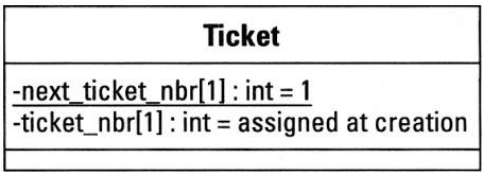

Figure 51. Example of class-level attributes

- **Operations**

Operations define the behavior of the object (the actions it can perform). They can use stereotype to label their usage.

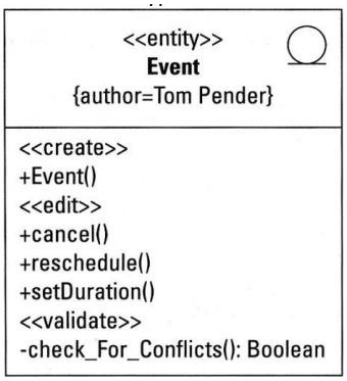

Figure 52. Example of stereotype usage in class operation

Operations follow this syntax: **[visibility] name ([parameter-list]) ':' [return-result] [{properties}]**

The UML operation syntax begins with **visibility** (+, −, #, ~) and the **name**, followed by a **parameter-list** in parentheses. Each parameter can include an optional direction (in, out, inout), a name, a data type, and a default value. This part of the notation clearly defines the operation's identity and the specific inputs required for it to function.

After the parentheses, a colon introduces the **return-result**, which specifies the data type the operation produces, or void if nothing is returned. The signature ends with **properties** inside curly braces, providing extra metadata or constraints. Common examples include {readOnly}, which ensures the object's state isn't modified, or {ordered} to indicate a sorted result.

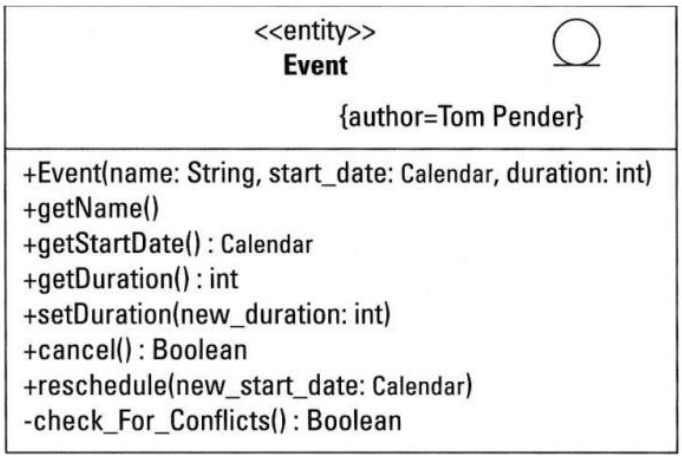

Figure 53. Example of operations in class operation

The most common operations, owned and executed by a specific object are called **instance-level** operations, while **class-level (Static)** Operations are those accessible by the class itself rather than a specific instance. Like static attributes. As static attributes, static operations are underlined in the class.

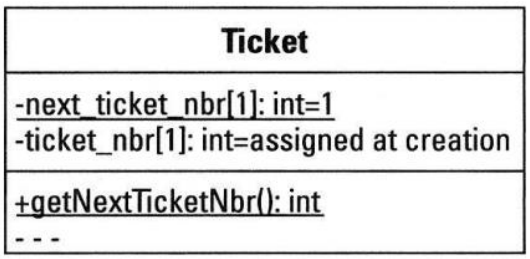

Figure 54. Example of static operations in class operation

By defining these elements clearly, structural modeling ensures that the foundation of the software is robust, manageable, and ready for construction.

3. Modeling relationships

Just as human beings communicate with each other, objects must also find a way to communicate among themselves. To do this, objects use three types of relationships: **association**, **generalization**, and **dependency**

a- Association

The purpose of an association is to establish the reason why two classes need to be aware of each other, as well as the rules that govern their relationship. For example, an event may take place at a location. The event must know where it is taking place, and the location must know what is happening within it. These are two perspectives within the same association. The modeling of an association begins with identifying the participating classes. There are different types of associations: binary associations (between two classes) and n-ary associations.

D. Binary association

It is an association that consists of a connection and two association ends. The instance of a class is called an *object*, and the instance of an association is called a *link*.

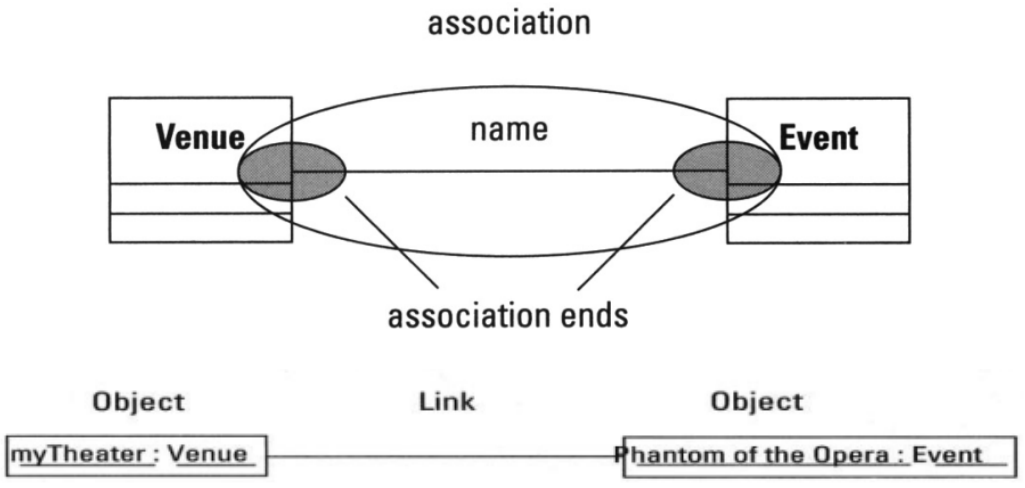

Figure 55. Example of binary association

An association is generally named using a verb or a verbal phrase. The name of the association becomes even more important when two classes have more than one reason

to collaborate. In the theater example, one venue might sponsor an event, while others host the event.

Association ends define the participation of each object in the association. Each association end can include the following characteristics: role, interface specifier, visibility, multiplicity, order, constraint, qualifier, navigability, and changeability.

- **Role**

The role explains how an object participates in the association. For example, in the theater scenario, the venue may act as the sponsor of an event and may also be the host of an event.

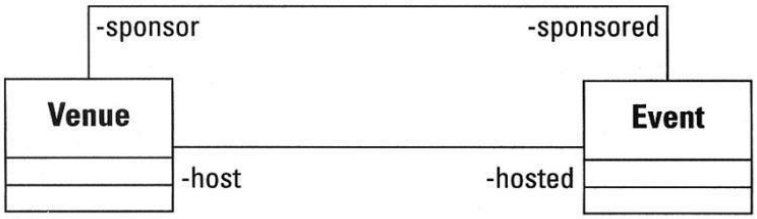

Figure 56. Example of a role name in a binary association

The role can generate code during implementation (it becomes an attribute that refers to the class it belongs to). The code for the previous example would be:

```
class Venue{

private Event hosted ;

private Event sponsored;

}

class Event{

private Venue host;

private Venue sponsor;

}
```

- **Interface specifier**

It is a property of an association end that defines which specific interface an object must use when interacting with another via that link. Instead of an object seeing the entire target class, it only sees the methods and attributes defined in a specific interface. This is a key principle for **decoupling** systems, as it ensures a class only depends on the behaviors it actually needs.

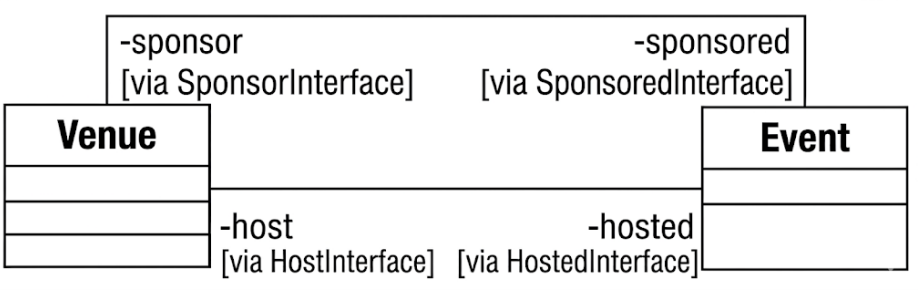

Figure 57. Example of interface specifier in a binary association

for example, when an **Event** interacts with a **Venue** over this specific link, it is restricted from accessing the **Venue's** full range of public methods. Instead, it must interact strictly through the operations defined in the SponsorInterface. This ensures the **Event** only sees the sponsorship-related functionality of the **Venue**, effectively hiding any unrelated attributes or behaviors of the **Venue** class from that specific connection.

- **Visibility**

Indicates whether the associated object can be seen by other elements outside the association. It is typically denoted by symbols like + (public), - (private), # (protected), or ~ (package).

- **Multiplicity**

Defines the number of instances of one class that can be linked to a single instance of another class (e.g., 0..1, 1..*, *). the "hosts" association specifies that one or more **Venues** can be linked to zero or more **Events**, while the "sponsors" association indicates that exactly one **Venue** acts as a sponsor for any number of **Events**.

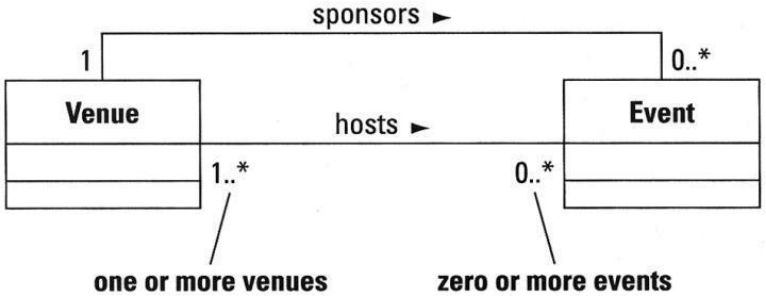

Figure 58. Example of multiplicity in a binary association

- **Order**

It specifies whether the set of related objects at that end follows a specific sequence. Common values include ordered (a specific sequence is maintained) or unordered. For example, when a venue hosts a certain number of events, the list of events must be organized in a specified sequence, whereas it is not when a venue sponsors an event.

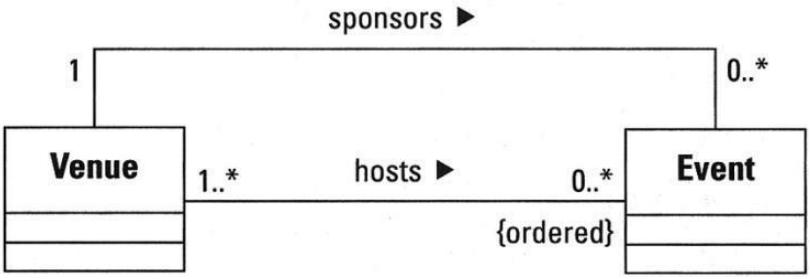

Figure 59. Example of order in a binary association

- **Constraint**

This is one of the three UML extension mechanisms used to define rules concerning an association. For example, when a venue hosts an event, it must be ensured that the cost of this event does not exceed a certain amount, that events must be sorted by start date, and that each event must be approved by the venue manager (VM).

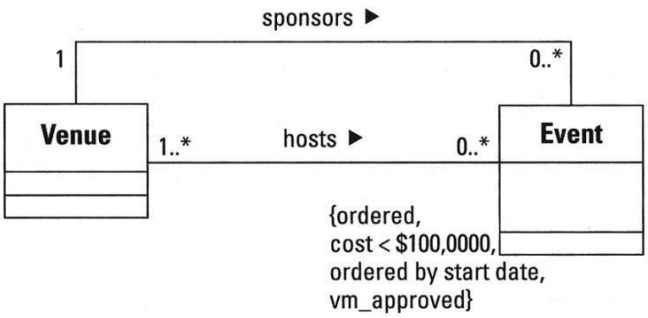

Figure 59. Example of constraints in a binary association

- **Qualifier**

It is a property or attribute used to uniquely identify or refine the related objects in an association. It acts like a key to distinguish instances in an association. For example, A **Venue** can host multiple **Events** and each **Event** at a Venue is uniquely identified by its **Eventid** (the qualifier).

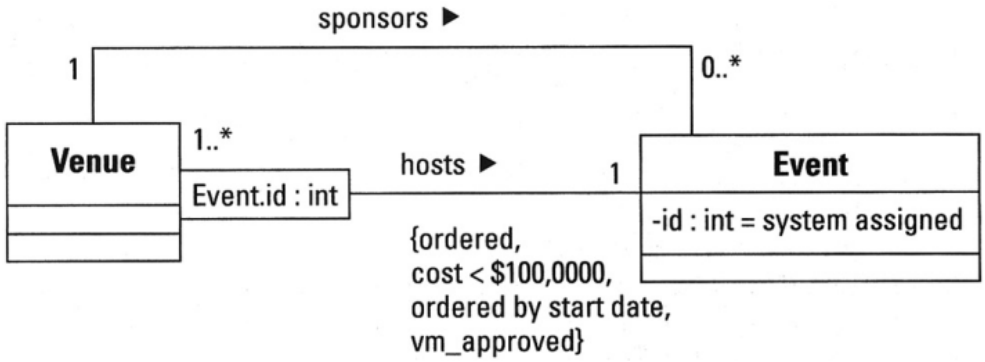

Figure 60. Example of qualifier in a binary association

- **Navigability**

Determines if an object can "see" or access the other object at the end of the link. It is shown by an arrow for one-way navigation or a line without arrows for two-way navigation.

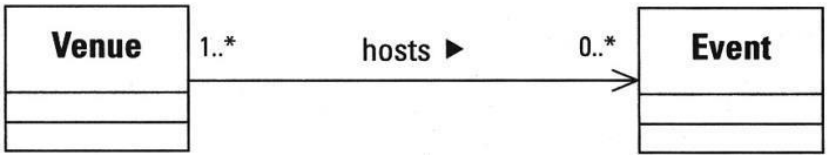

Figure 61. Example of navigability in a binary association

- **Changeability**

This property allows us to specify the operations permitted on the links defined by an association. The predefined options include {frozen}, which means that once a link has been established, it cannot be modified or moved. If the application only allows creating new links (but no deletion), use the {addOnly} property.

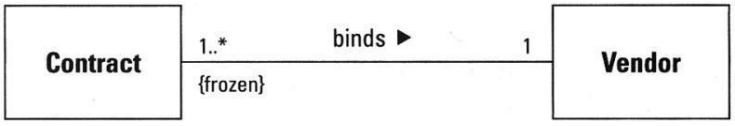

Figure 62. Example of changeability in a binary association

E. Reflexive association

A **reflexive association** occurs when a class has an association relationship with itself. This means that an instance of the class can be linked to other instances of the same class. For example, a single VenueManager act as an **auditor** oversees zero or more other VenueManagers who fulfill the **coordinator** role.

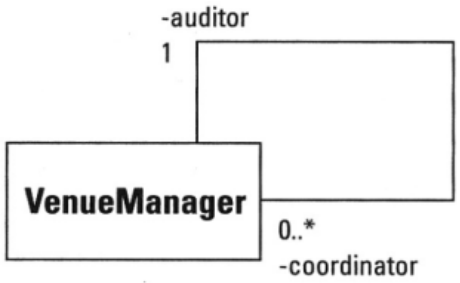

Figure 63. Example of reflexive association

F. N-ary association

This type of association is rarely used in UML models. It is an association between multiple classes. The symbol used to represent this type of association is a diamond. For example, to define a calendar, there must be an event, a location manager, and a venue for the event. As an example of an **n-ary association**, the **Venue**, **Event**, and **VenueManager** classes are connected to a central diamond to show they are part of one atomic relationship. This structure indicates that a specific booking or assignment only exists when all three participants are present simultaneously.

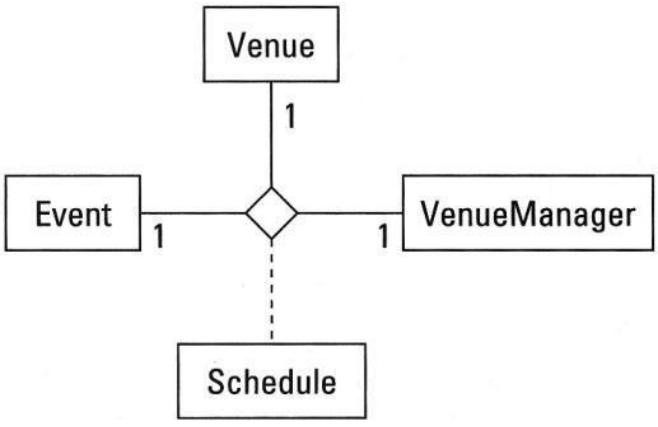

Figure 64. Example of n-ary association

G. Aggregation

In a typical association, the participating classes are peers. Each class remains independent of the other and neither is superior. Aggregation, on the other hand, refers to an assembly or configuration of elements to create a larger and more complex unit. Aggregation includes a control point (called the aggregate) that represents the master object having control over its parts (no actions on the parts are allowed without the aggregate's permission). An aggregation can represent either a physical or a logical assembly. For example, a computer is composed of several devices (keyboard, mouse, etc.), and a director manages several employees (agents, department heads, etc.). The control point can represent multiple levels — for instance, an engine controls its parts, and the car controls the engine and its parts. An aggregation is represented by an **unfilled diamond** next to the aggregate (the assembling class). For example, an **Agency** acts as the aggregate "whole" composed of at least one **Agent**, where each agent functions in the role of an employee. While an agency must have one or more members (1..*), an individual employee is restricted to belonging to no more than one agency and may currently belong to none (0..1). Furthermore, the model enforces a specific constraint where an agent is only considered an active employee within the agency if they possess a valid, current contract.

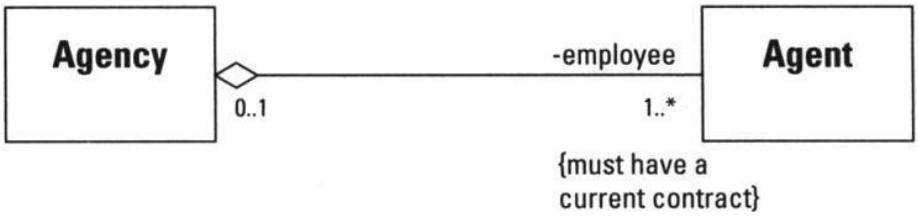

Figure 65. Example of aggregation association

H. Composition

Composition is used for aggregations where the lifetime of the member object depends on the lifetime of the aggregate. The aggregate not only controls the behavior of the member but also has control over its creation and destruction. This high level of responsibility held by the aggregate is why composition is referred to as a **strong aggregation**. Composition is represented similarly to aggregation, but with a **filled diamond** next to the aggregate (the owning class).

b- Dependency

Dependency represents a client-supplier relationship in which a change to the supplier requires a change to the client. Dependency can also represent precedence — for example, in a theater system, selecting a seat on the touchscreen precedes the payment of the ticket.

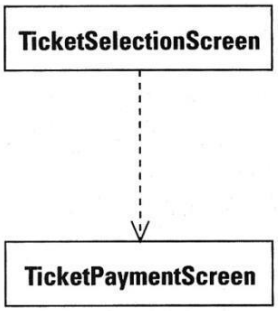

Figure 66. Example of dependency relationships

c- Generalization

Generalization is the process of organizing the properties of a set of objects that share the same purpose. Generalization is also synonymous with **inheritance**. For example, in a theater system, a play is a type of event, so a play inherits all the functionalities of an event. A play is also a **specialization** of an event because it inherits all the generalized functionalities of an event and adds unique functionalities, or replaces/redefines existing ones. **Event** is called a **superclass**, and the other classes are called **subclasses**, and the diagram is referred to as a **class hierarchy**. Another representation of generalization can also be used

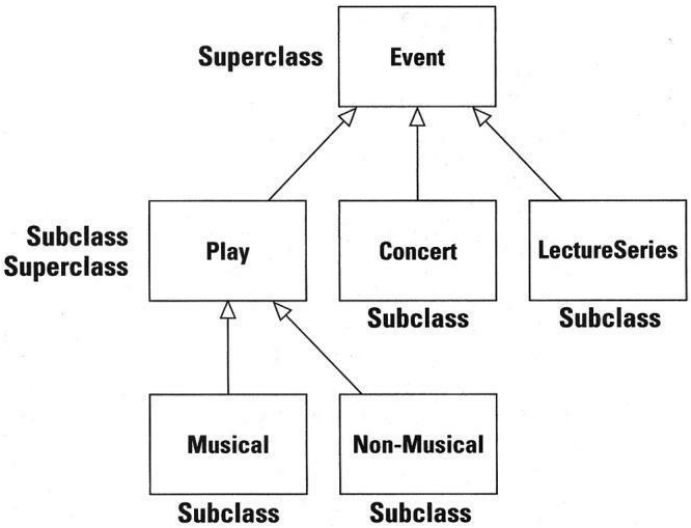

Figure 67. Example of generalization relationships

An important point to note is that a **leaf class** in the hierarchy can never be an **abstract class** (a class from which no objects can be instantiated).

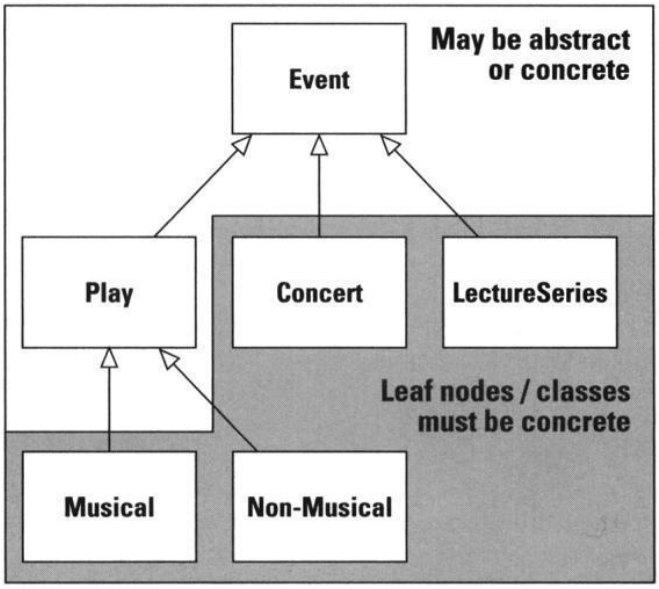

Figure 68. Example of difference between leaf and nodes in a generalization relationships

When modeling a generalization, you must describe how the subclasses of a superclass were identified. For example, concerts can be categorized based on the size of the performance (solo or group). A solo concert can be further categorized according to the size of the band, and a group concert according to the size of the group.

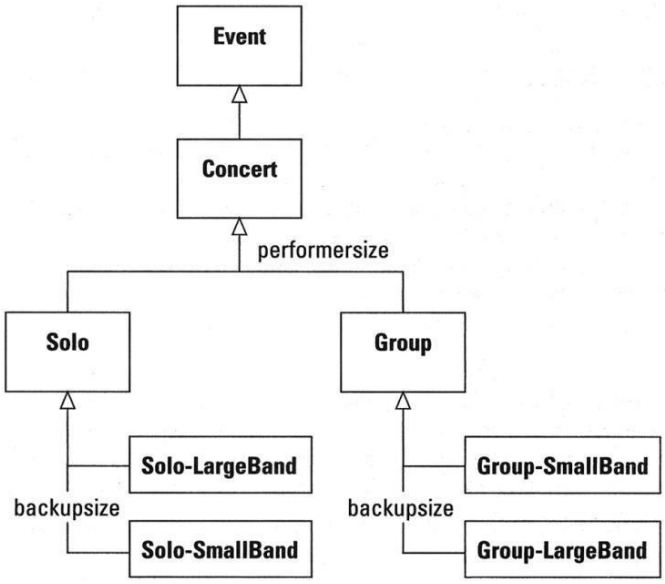

Figure 68. Example of generalization identification

d- Association class

An association between two classes often requires information not only about the participating objects but also about the association itself. For example: When did the association begin? When will it end? Are there any conditions that apply to the association?

An association class is a class that describes a relationship between classes. For instance, an association between agents and the theater company can be described in terms of start date, end date, terms and conditions, and authorization. These descriptive attributes are encapsulated in a class called Contract. The Contract class is then connected to the Authorization association it describes using a dashed line. The dashed line has no arrow.

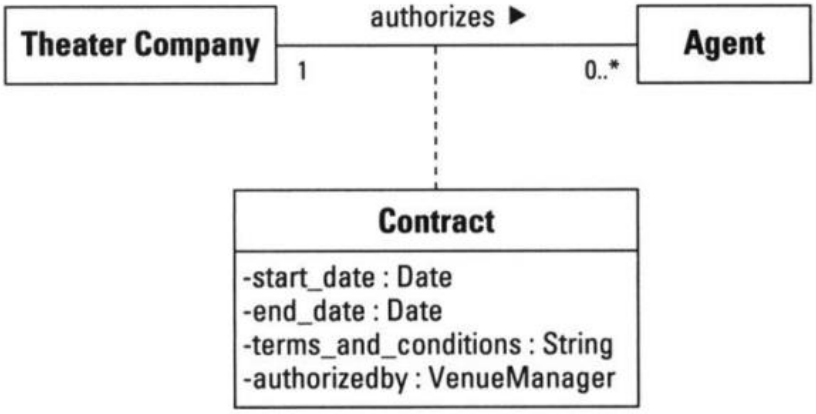

Figure 69. Example of class association

An association class is a class like any other class, and therefore, it can have associations with other classes.

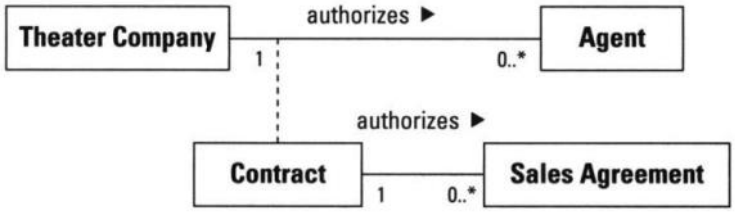

Figure 70. Example of class nature of a class association

⋆˚₀⋆✧˚ Chapter 7: Behavioral Modeling: Sequence Diagrams ˚✧⋆₀˚⋆

In the realm of software engineering, capturing the static structure of a system is only half the battle. While a map shows us the layout of a city, it tells us nothing of the traffic flowing through its streets or the rhythm of its daily life. To truly understand a system, we must look beyond its components and witness its **behavior**. Behavioral modeling serves as the pulse of the design process, shifting our focus from what a system *is* to how it *acts*.

Sequence diagrams are the definitive tool for this exploration, providing a chronological narrative of communication. By mapping the exchange of messages between objects over time, these models transform abstract logic into a clear, visual choreography. In this chapter, we delve into the mechanics of the **Sequence Diagram**, mastering the art of documenting interaction to ensure that every process—from the simplest login to the most complex transaction—is executed with precision and clarity.

1. Definition

The **Sequence Diagram** is the primary tool used to model the execution of a scenario within a system. It is widely considered the most important UML diagram after the Class Diagram because it focuses on the **sequential order** of interactions.

While developers use them to map out code execution, these diagrams are equally valuable for business stakeholders. They provide a clear visual for discussing how a business currently operates by showing how different entities—from users to databases—interact to accomplish a task. Essentially, it answers two questions: **Which messages are sent, and when?**

A sequence diagram is made up of two basic building blocks: the frame and the interaction.

2. Frame

The frame is the space in which the sequence diagram is drawn. It represents the border that surrounds the diagram and consists of a header and a contents area.

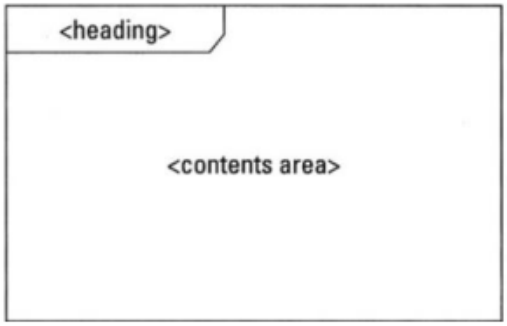

Figure 71. Frame representation in UML

3. Interaction

Simply put, an interaction is a sequence of messages exchanged between objects to accomplish a task. Objects can be created and destroyed. They can ask questions or make requests to other objects by invoking operations.

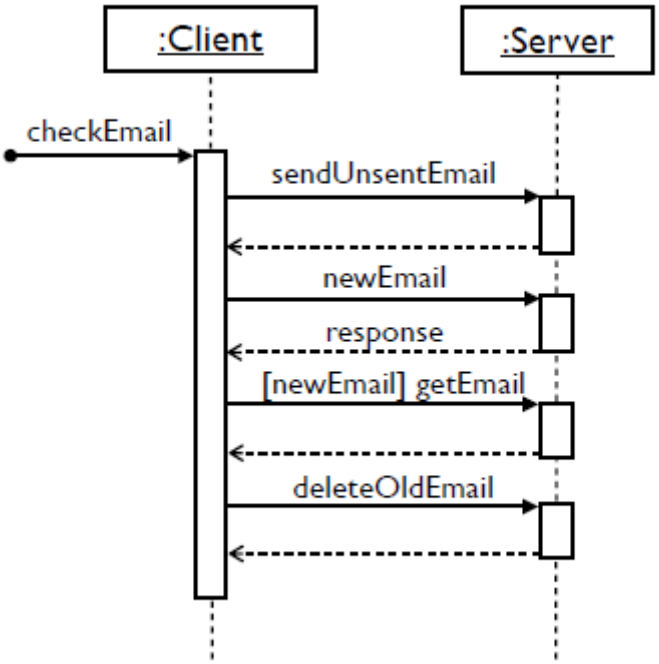

Figure 72. Example of interaction in sequence diagram

To help illustrate how interaction works in a real-world scenario, we can examine a common interaction: **Checking Email**. This process involves a series of messages between a Client (the user's device) and a Server. The interaction follows a specific chronological order to ensure data integrity and synchronization:

- The user clicks the "Check Email" button.
- First, the client sends all the emails that have not yet been sent.
- After receiving an acknowledgment of receipt, the client asks the server if there are any new emails.
- If there are, it downloads the new email.
- Then, it deletes the old overwritten emails from the server.

To draw a sequence diagram, we need to master three main elements: **who** is involved, **when** they are working, and **what** they are saying.

- **Lifeline (Who)**

The Lifeline represents an object's existence over time. It is represented by a box with the object's name at the top and a vertical dashed line stretching downward.

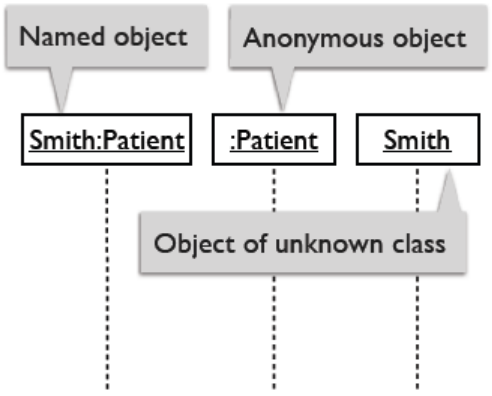

Figure 73. Example of representation of lifeline in a sequence diagram

In sequence diagrams, timing is everything. One of the most important concepts to visualize is the **creation of an object** during the system's execution. There is a critical difference between **object creation** and a **standard message call**. When the message arrow points directly to the **object box**, it indicates a **Constructor Message** (for example creation of the `:Performance`)**.** When the message arrow points to the **dashed lifeline**, it signify that the object already exists and is simply being invoked to execute a method.

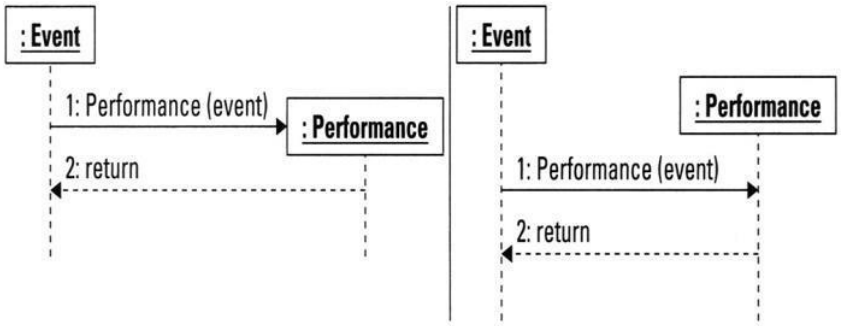

Figure 74. Example of difference between constructor and method calling in sequence diagram

- **Control focus (When)**

The control focus describes the period during which an object performs an action, either directly or through a subordinate procedure. The activation of an object is represented by the start and end of the control focus.

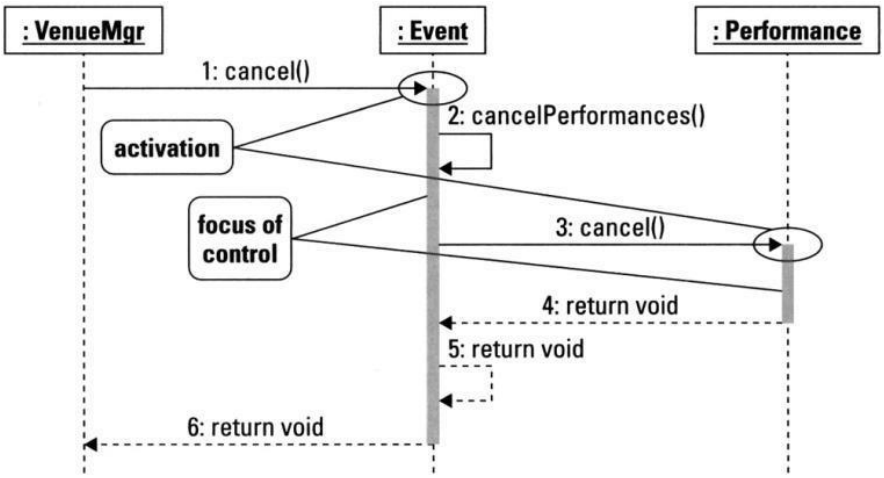

Figure 75. Example of control focus in sequence diagram

- **Message**

A message is the basic form of communication in a sequence diagram. It defines a specific type of communication. The communication may invoke an operation, and create or destroy an instance. The message specifies not only the type of communication but also the sender and the receiver. The message is represented by horizontal arrows. We can distinguish several types of message:

➢ Synchronous message

A synchronous message calls a method of an object and waits for a response. The call parameters must match the parameters of the invoked method's prototype.

➢ Asynchronous message

An asynchronous message calls a method of an object and does not wait for a response. The call parameters must match the parameters of the invoked method's prototype.

➢ Constructor message

A message that creates an object is called a constructor. It is represented by a dashed line with an open arrowhead.

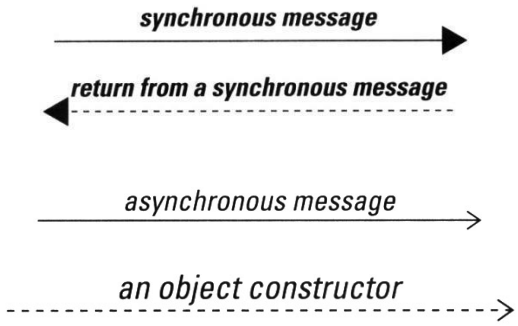

Figure 76. Principle of synchronous, asynchronous and constructor message

➢ Reflexive message

A reflexive message represents a call to an internal method of an object.

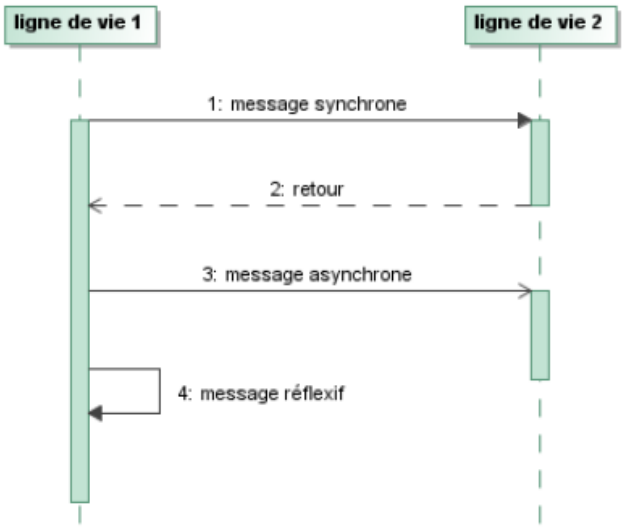

Figure 77. Example of reflexive message in sequence diagram

4. Example of sequence diagrams

In this section, we give a set of examples that resume most frequent blocs found in sequence diagrams.

- **Example of synchronous message**

In this example, labeled **sd Rechercher livre** (Search book), a **Client** initiates a synchronous interaction by invoking the chercher("Tintin") method on the **:Médiathèque** object. This is represented by a solid line with a filled arrowhead, indicating that the client must wait for a result before proceeding. Upon receiving the request, a **control focus** appears on the library's lifeline to show it is actively processing the search. Finally, the library returns the result, nombreLivres, via a dashed response arrow, which signals the end of the operation and allows the client to resume its activity.

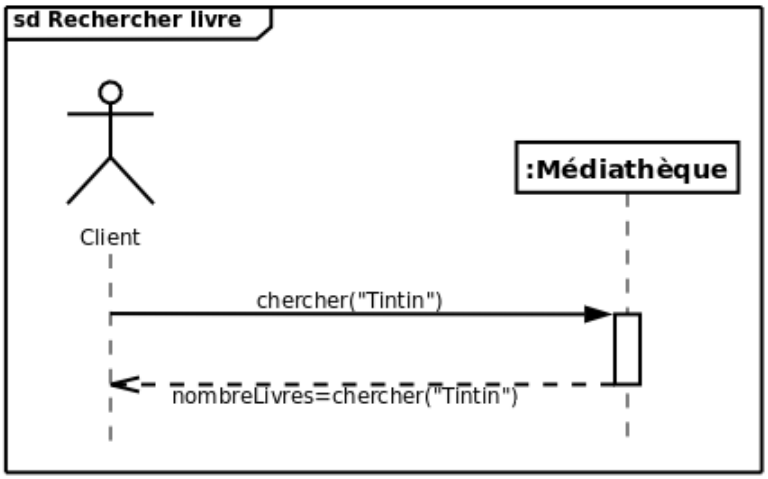

Figure 78. Example of synchronous message in sequence diagram

- **Example of asynchronous message**

In this **sd Retrait argent** (Cash withdrawal) example, a **Client** initiates an asynchronous interaction by sending the introduireCarte method to the **:Distributeur** object. This is represented by a solid line with an **open arrowhead**, signifying that the client triggers the action and does not wait for a response before continuing. The diagram highlights the message lifecycle through four key events: the **sending event** at the client's lifeline, the **reception event** at the dispenser, and the subsequent **start and end of execution** events. The control focus on the **:Distributeur** lifeline illustrates the period during which the

machine processes the card independently, allowing the system to handle multiple parallel actions without blocking the sender.

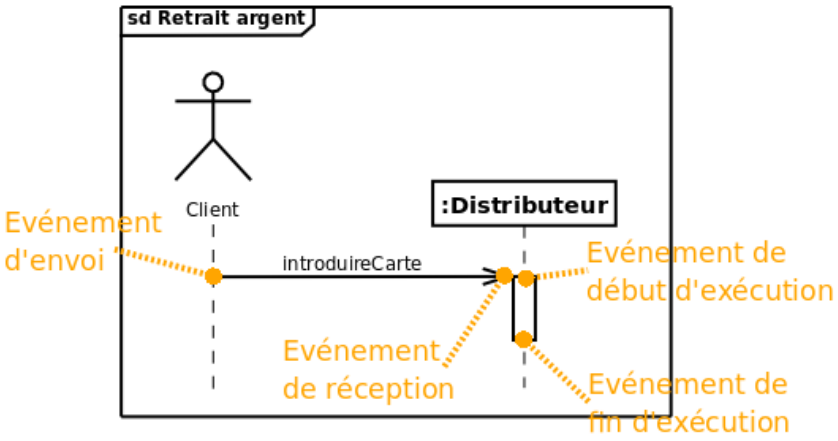

Figure 79. Example of asynchronous message in Cash withdrawal sequence diagram

Another example is the common interpersonal interaction, where the person **P1** sends the message Envoyer SMS (Send SMS) to the person **P2**. This is depicted as an asynchronous interaction because the sender, **P1**, does not wait for a response or a "read receipt" before moving on with their own lifeline. Once the message reaches **P2**, a brief control focus appears on their lifeline, representing the moment the message is received and processed, while **P1** remains free to initiate other tasks or complete their own execution independently.

Figure 80. Example of synchronous message in SMS delivery sequence diagram

- **Example of constructor and destructor messages**

In these examples, the system manages the lifecycle of an account within an online library environment. The object **:LibrairieEnLigne** creates a new instance **c: Compte** by invoking the Compte() constructor. This is represented by a dashed line with an open arrowhead pointing directly to the object's box, which is vertically positioned to mark the exact moment of its creation. The same system terminates the object's existence by sending a <<destroy>> message. This is visually confirmed by the large **red "X"** at the end

of the **c: Compte** lifeline, indicating that the object has been removed from memory and its lifecycle has officially ended.

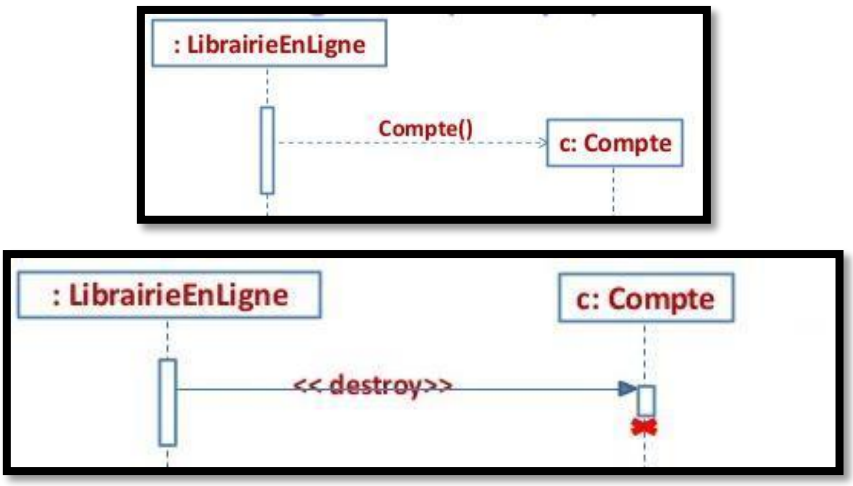

Figure 81. Example of object creation and destruction in a sequence diagram

- **Example of lost message and unknown source**

In this example, the diagram illustrates how a system handles communication failures and external inputs. The interaction begins with a small circle, representing a **found message** from an **unknown source**, where the sender exists outside the scope of the current diagram. The message introduireCarte (insertCard) is dispatched toward the **:Distributeur** object but is intercepted by a **red "X"**, which is the standard notation for a **lost message**. This signifies that the transmission failed and never reached its destination. As a result, the expected **start of execution event** on the dispenser's lifeline is never triggered, visually demonstrating a scenario where the receiver remains inactive because the signal was lost in transit.

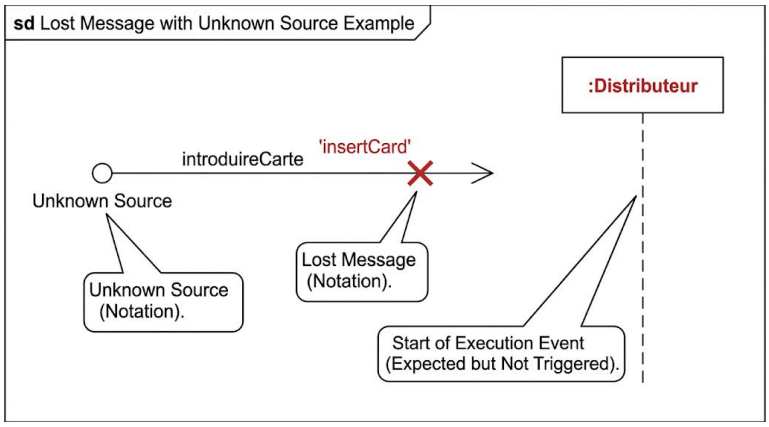

Figure 82. Example of lost message and unknown source in a sequence diagram

5. Advanced interaction fragments

In addition to standard message flows, UML sequence diagrams use **Interaction Fragments** (combined fragments) to model complex logic like conditions, loops, and error states. These are represented by frames with a specific operator in the top-left corner.

- **Option fragment (OPT)**

The **Opt** fragment models an "if-then" scenario. It contains a single interaction that only occurs if a specific **guard condition** is met. If [condition] is **True**, the Action() is performed. If **False**, the entire block is ignored.

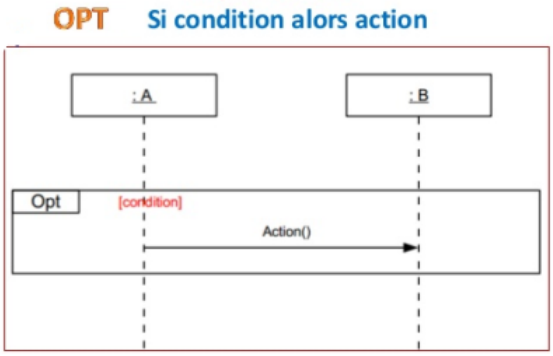

Figure 83. Representation of option fragment in a sequence diagram

- **Iterative fragment (LOOP)**

The Loop fragment represents repetitive behavior. The interaction inside the frame is executed multiple times based on a condition or a specific range. The bloc continue its execution as long as the [condition] remains True. It can also be noted as LOOP (min, max) to define exactly how many times the sequence repeats.

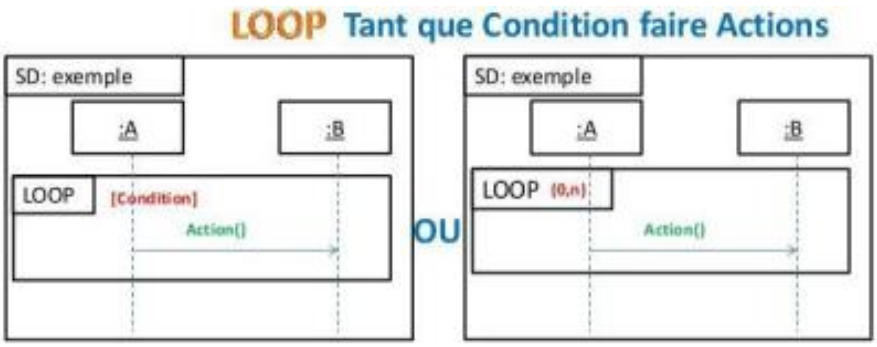

Figure 84. Representation of iterative fragment in a sequence diagram

- **Alternative fragment**

The Alt fragment models an "if-then-else" structure. The frame is split into two or more horizontal operands separated by a dashed line. If the first [condition] is met, ActionA() is executed. If that condition is false, the [else] path is taken, and ActionB() is performed instead. Only one of the paths can ever be executed.

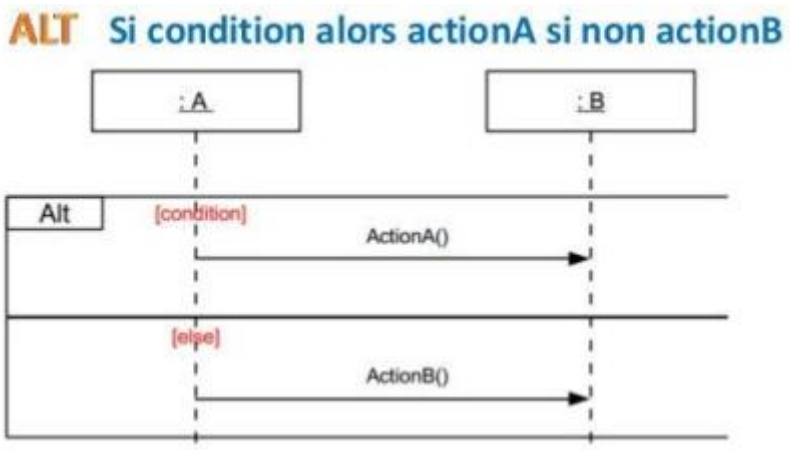

Figure 85. Representation of alternative fragment in a sequence diagram

- **Interruption fragment**

The Break fragment is a specialized scenario used to model an "interruption" or an early exit. If the [condition] is verified, the interactions inside the Break frame are performed, and then the rest of the enclosing fragment is skipped. This is often used to model exception handling or "cancel" operations.

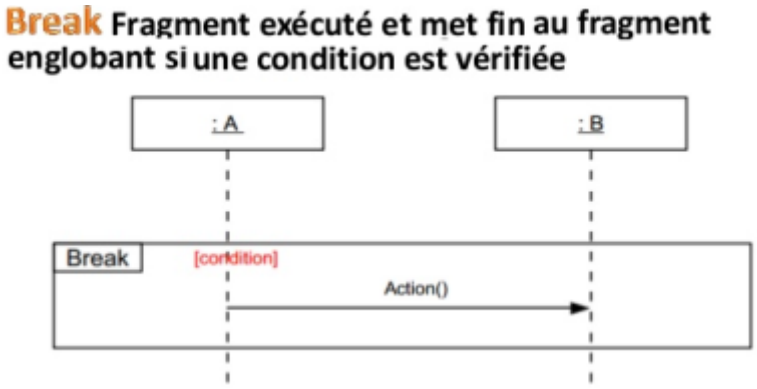

Figure 86. Representation of interruption fragment in a sequence diagram

An example of application of some of this fragment is the **sd Online_Bookshop** example. Here the the system uses a **loop** fragment to allow the **:Web Customer** to repeat searches within a single session. After receiving search results, two independent **opt** fragments

provide the flexibility to either view book description or add to shopping cart based on user preference. This demonstrates how combined fragments model iterative behavior and conditional logic without forcing a strictly linear execution.

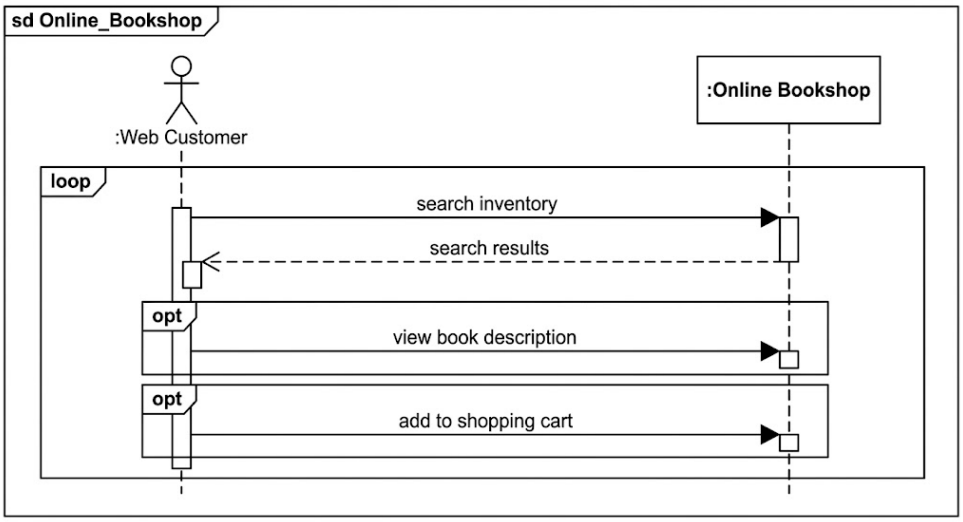

Figure 87. Example of fragment application in a sequence diagram

⋆˚₀⋆✧˚ Conclusion ˚✧⋆₀˚⋆

As students conclude this journey through the fundamentals of Java and the strategic blueprints of the Unified Modeling Language, it becomes clear that professional software development is a dual discipline of logic and design. By mastering the syntax of Java alongside the architectural clarity of UML, students have moved beyond simply writing code to engineering resilient, scalable systems.

The chapters within this handbook have equipped students with the tools to translate complex real-world requirements into structured class diagrams and dynamic sequence models. Whether students are defining the functional scope of a project or optimizing internal mechanics, the principles of encapsulation, inheritance, and formal modeling remain their most reliable guides. As students step forward into their own development projects, it is vital to remember that the most effective software is not just functional—it is well-documented, thoughtfully modeled, and built upon a foundation of structural integrity.

Bibliography

[1] Cay S. Horstmann. *Core Java: Fundamentals (Vol. 1, 12th ed.)*. Pearson, 2023.

[2] Herbert Schildt. *Java: The Complete Reference (12th ed.)*. McGraw Hill, 2021.

[3] Kathy Sierra, Bert Bates, and Trisha Gee. *Head First Java (3rd ed.)*. O'Reilly Media, 2022.

[4] Harold Abelson and Gerald Jay Sussman. *Structure and Interpretation of Computer Programs (2nd ed.)*. MIT Press, 1996.

[5] Grady Booch, Robert A. Maksimchuk, Michael W. Engle, Bobbi J. Young, Jim Conallen, and Kelli A. Houston. *Object-Oriented Analysis and Design with Applications (3rd ed.)*. Addison-Wesley Professional, 2007.

[6] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[7] James Rumbaugh, Ivar Jacobson, and Grady Booch. *The Unified Modeling Language Reference Manual (2nd ed.)*. Addison-Wesley Professional, 2004.

[8] Martin Fowler. *UML Distilled: A Brief Guide to the Standard Object Modeling Language (3rd ed.)*. Addison-Wesley Professional, 2004.

[9] Russ Miles and Kim Hamilton. *Learning UML 2.0*. O'Reilly Media, 2006.

[10] Tom Pender. *UML Bible*. Wiley Publishing, Inc., 2003.

[11] Martina Seidl, Marion Scholz, Christian Huemer, and Gerti Kappel. *UML @ Classroom: An Introduction to Object-Oriented Modeling*. Springer, 2015.

[12] Alistair Cockburn. *Writing Effective Use Cases*. Addison-Wesley Professional, 2001.

Webography

[13] Oracle Corporation. *The Java Tutorials: About the Java Technology*. Available at: https://docs.oracle.com/javase/tutorial/getStarted/intro/definition.html (n.d.).

[14] Baeldung. *The JVM Architecture Explained*. Available at: https://www.baeldung.com/jvm-architecture, 2024.

[15] OpenJDK Project. *OpenJDK: The Place to Collaborate on an Open-source Implementation of the Java SE Platform*. Available at: https://openjdk.org/ (n.d.).

[16] Computerphile. *Object Oriented Programming*. Available at: https://www.youtube.com/watch?v=pTB0EiLXUC8, 2017.

[17] Ada Computer Science. *Object-oriented programming*. Available at: https://adacomputerscience.org/topics/object_oriented_programming (n.d.).

[18] Java Language Specification (JLS). *Types, Values, and Variables*. Available at: https://docs.oracle.com/javase/specs/jls/se21/html/jls-4.html, 2024.

[19] Exercism. *The Java Track: Basics*. Available at: https://exercism.org/tracks/java (n.d.).

[20] W3Schools. *Java Tutorial*. Available at: https://www.w3schools.com/java/ (n.d.).

[21] Educative. *OOP Design Confidential*. Available at: https://www.educative.io/blog/object-oriented-programming (n.d.).