⋆˚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**.

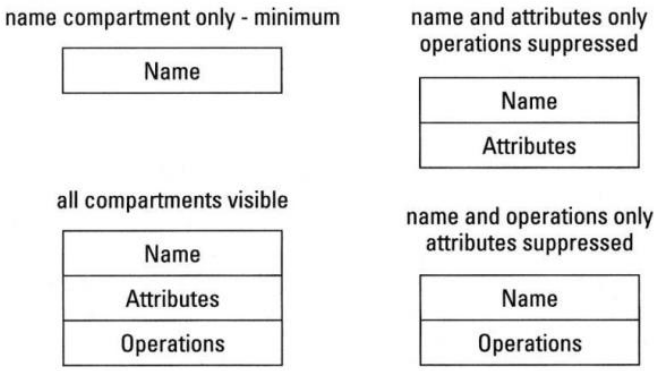

Figure 37. Different representations of a class

- **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.

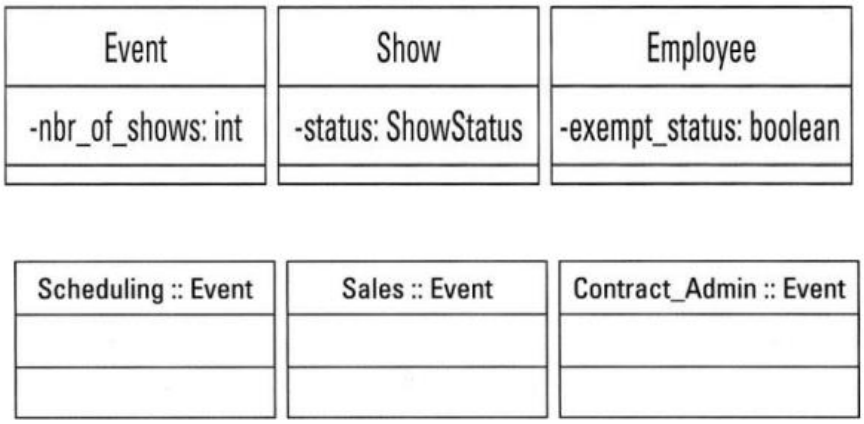

Figure 38. Example of representation of a class name

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).

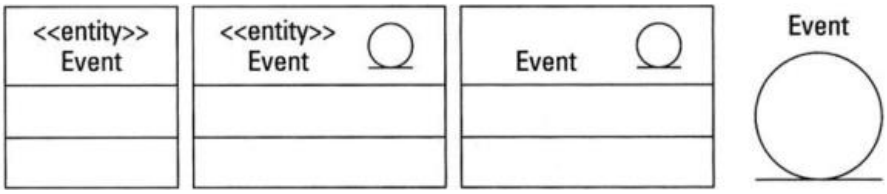

Figure 39. Example of representation of entity class

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

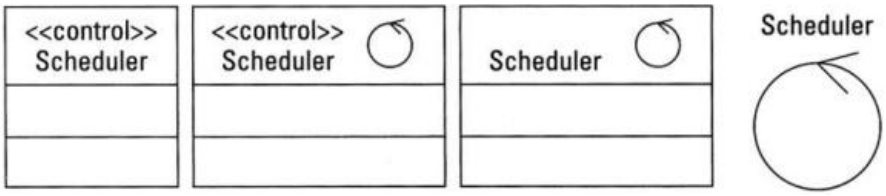

Figure 40. Example of representation of control class

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

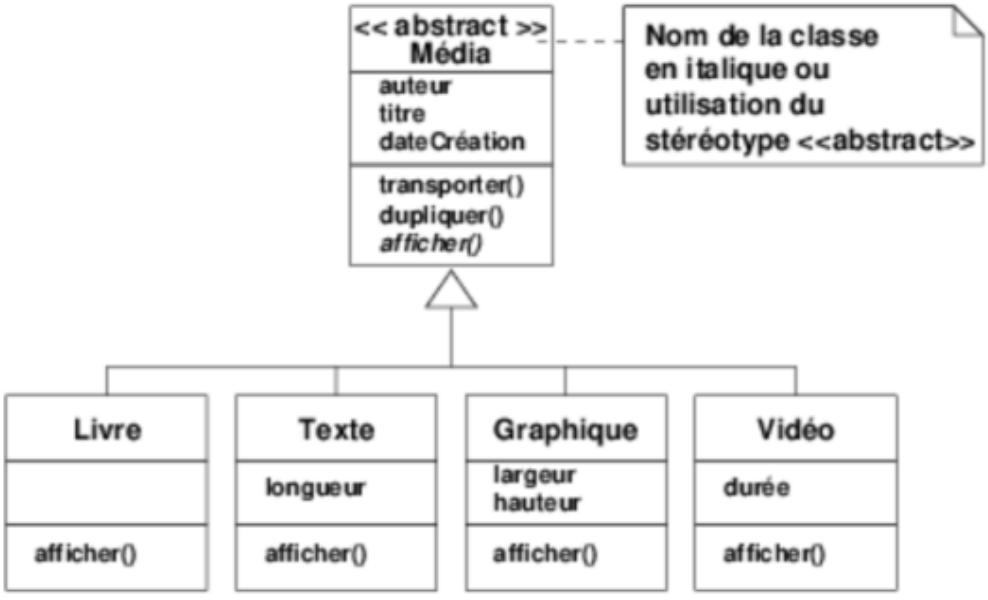

Figure 41. Example of representation of abstract class

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

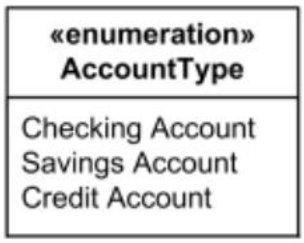

Figure 42. Example of representation of abstract class

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

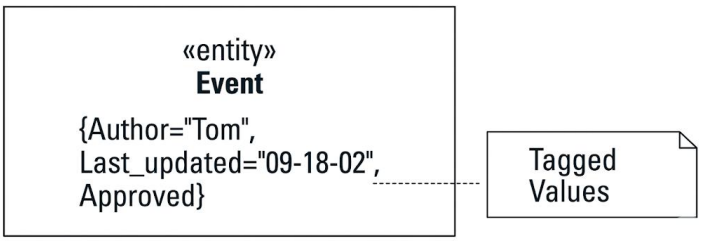

Figure 43. Example of representation of class using tagged values

- **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.

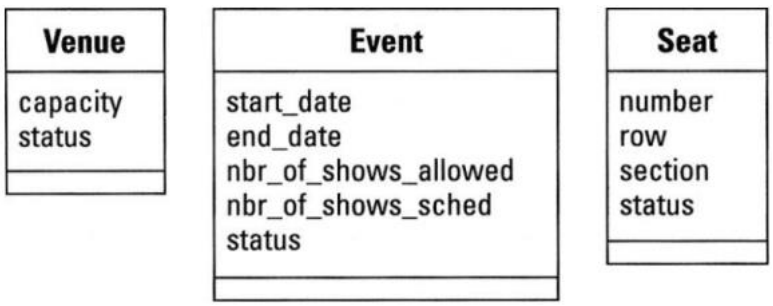

Figure 44. Example of representation of attributes in class diagram

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 (#)**.

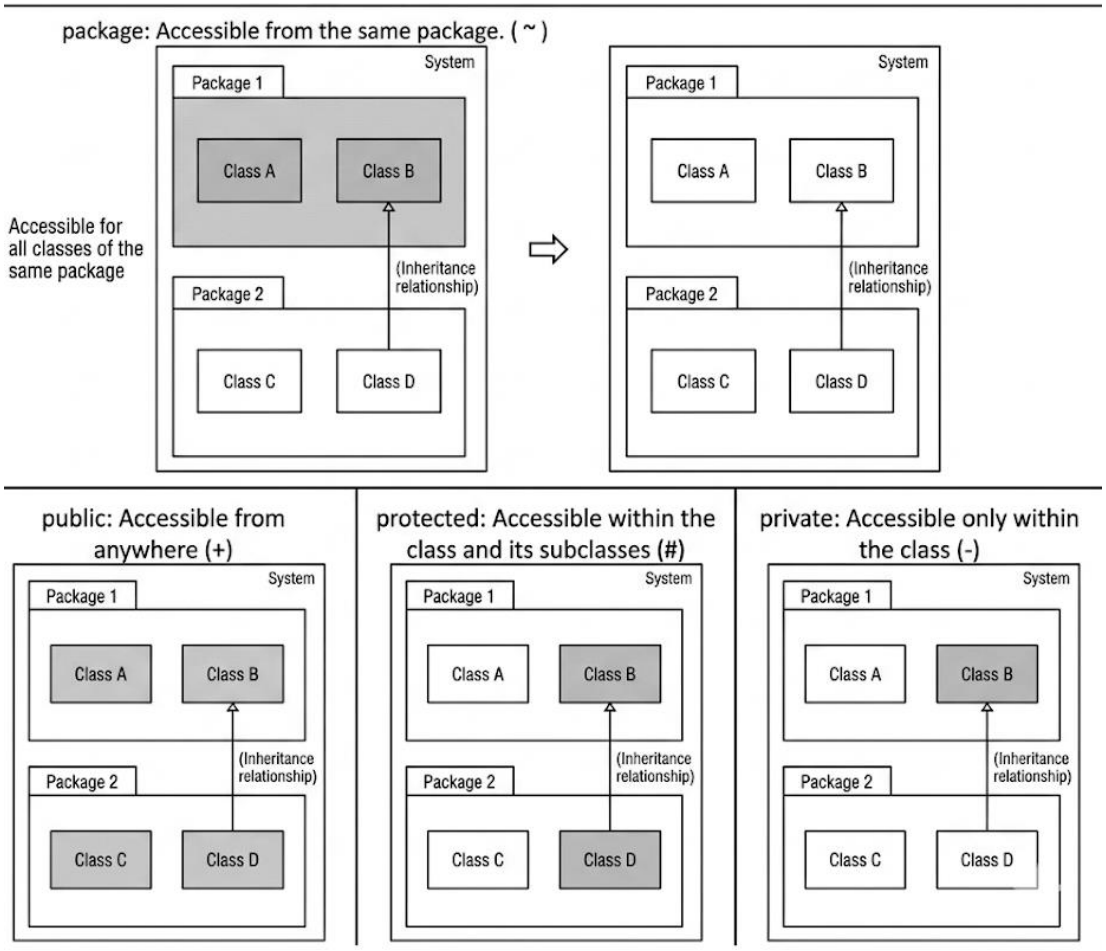

Figure 45. Different representations of attributes visibility

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

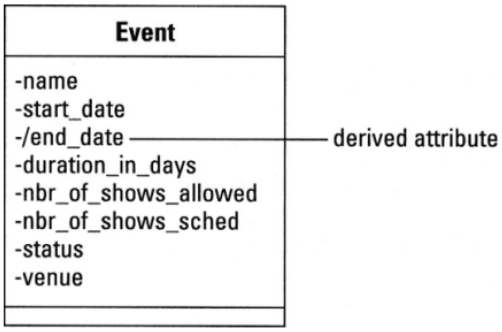

Figure 46. Example of derived attributes in a class

- **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.

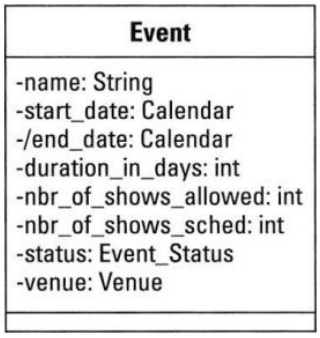

Figure 47. Example of type attributes in a class

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

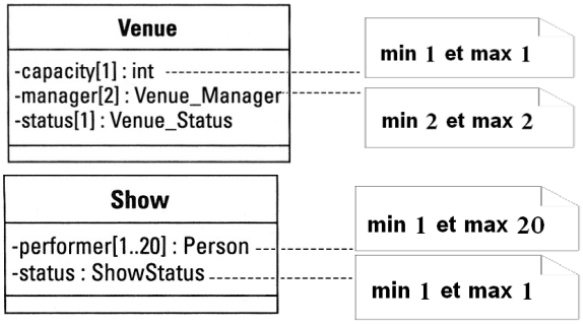

Figure 48. Example of attribute multiplicity in a class

- **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.

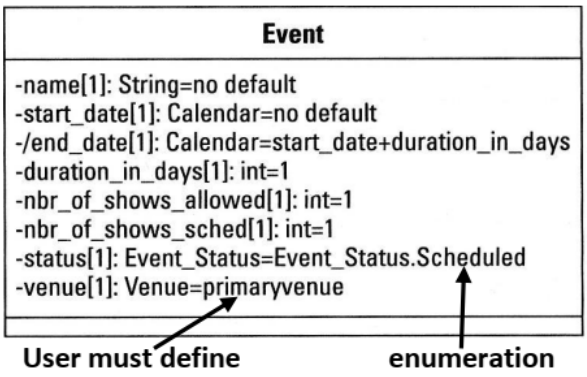

Figure 49. Example of default values of attribute in a class

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

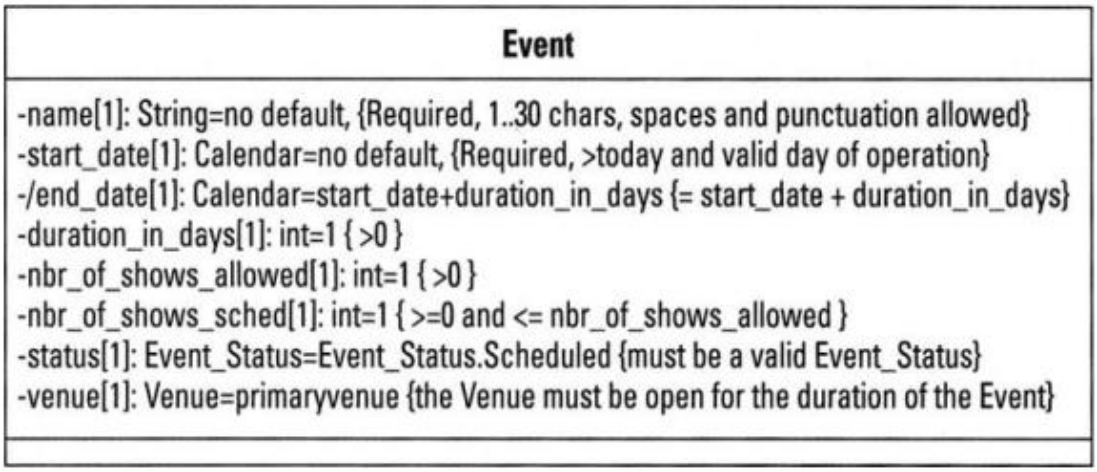

Figure 50. Example of String property of attribute in a class

- **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.

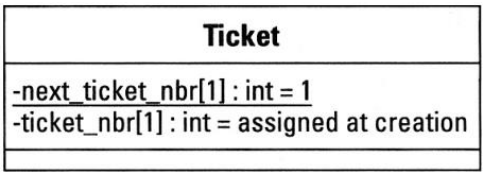

Figure 51. Example of class-level attributes

- **Operations**

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

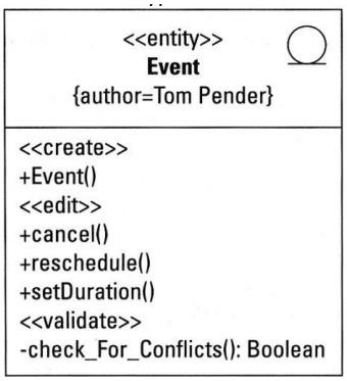

Figure 52. Example of stereotype usage in class operation

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.

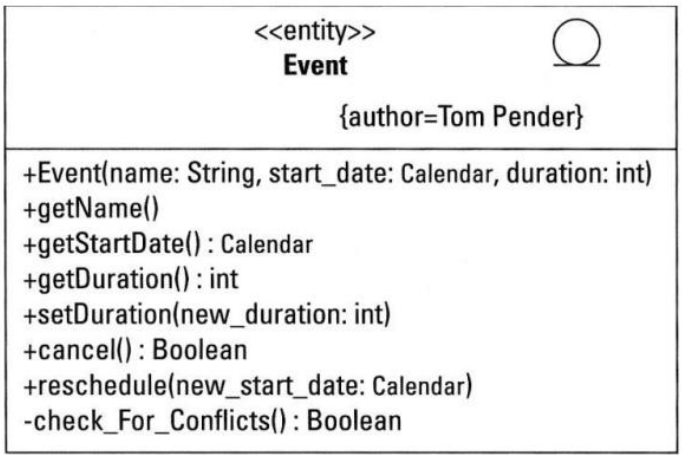

Figure 53. Example of operations in class operation

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.

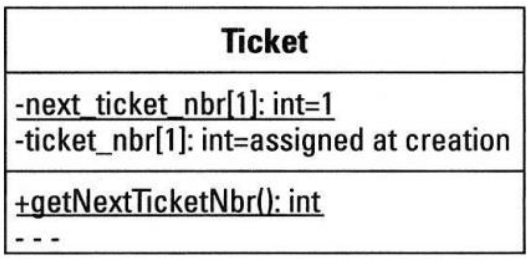

Figure 54. Example of static operations in class operation

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*.

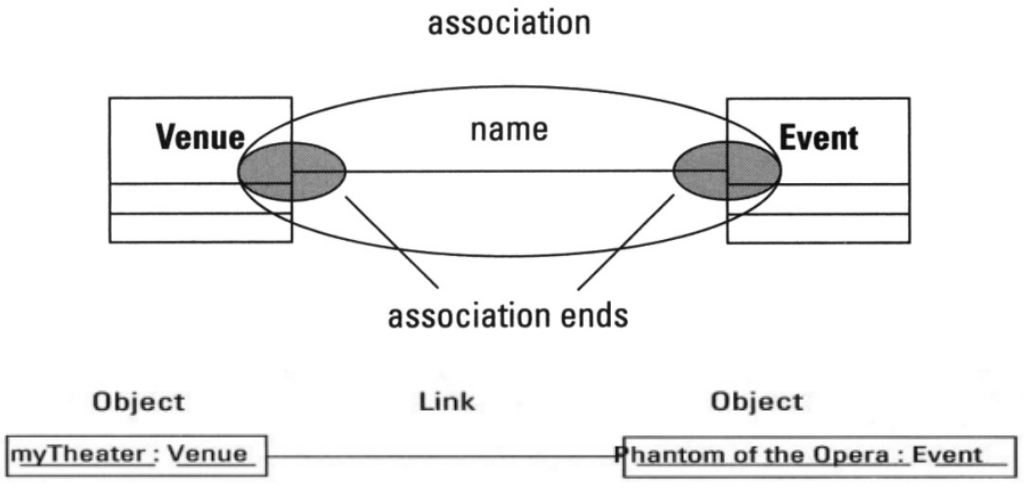

Figure 55. Example of binary association

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.

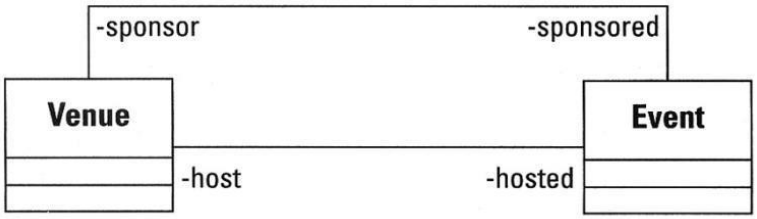

Figure 56. Example of a role name in a binary association

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.

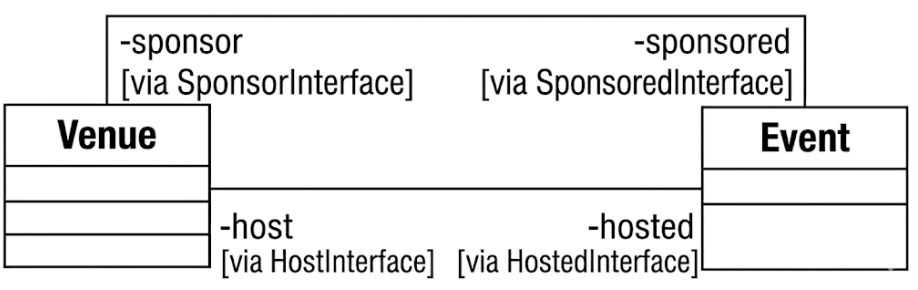

Figure 57. Example of interface specifier in a binary association

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**.

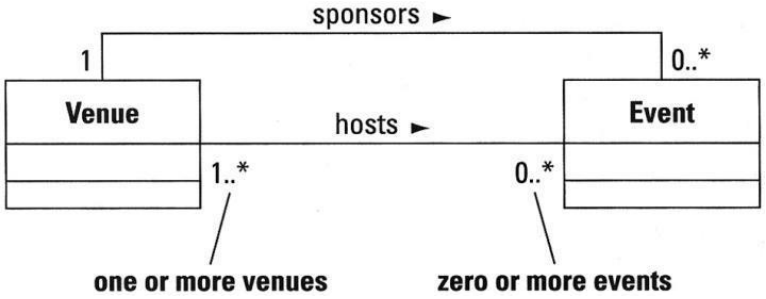

Figure 58. Example of multiplicity in a binary association

- **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.

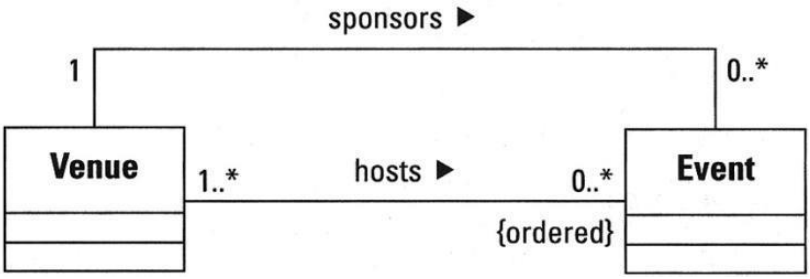

Figure 59. Example of order in a binary association

- **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).

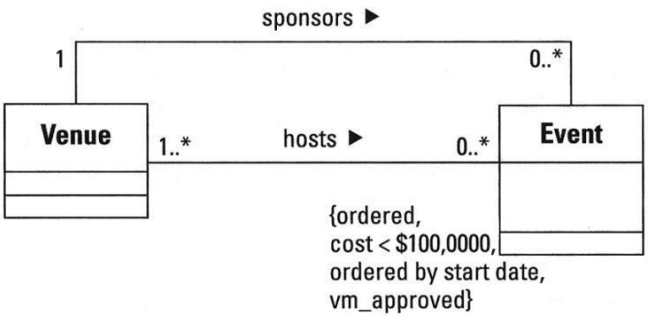

Figure 59. Example of constraints in a binary association

- **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).

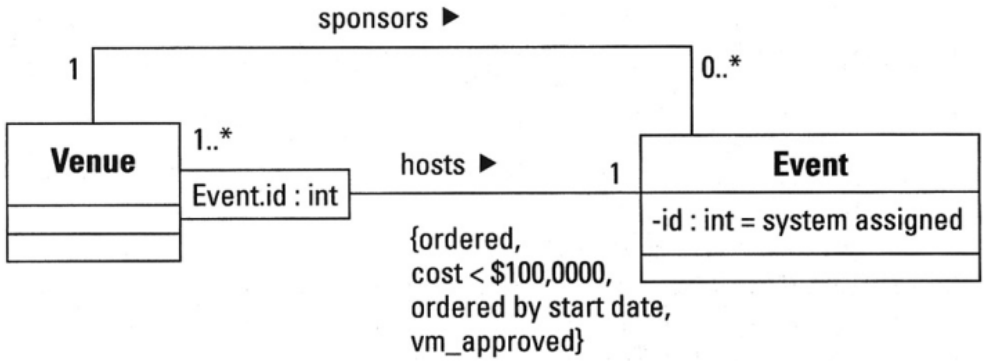

Figure 60. Example of qualifier in a binary association

- **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.

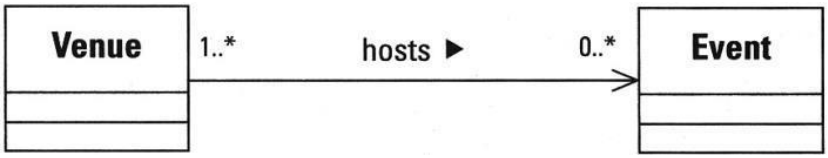

Figure 61. Example of navigability in a binary association

- **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.

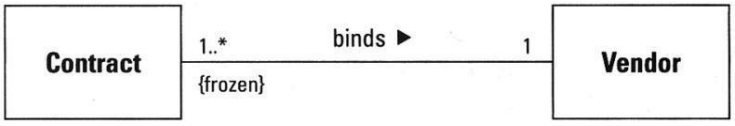

Figure 62. Example of changeability in a binary association

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.

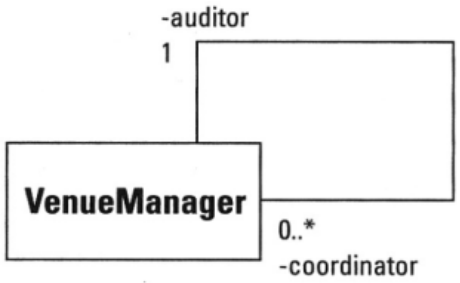

Figure 63. Example of reflexive association

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.

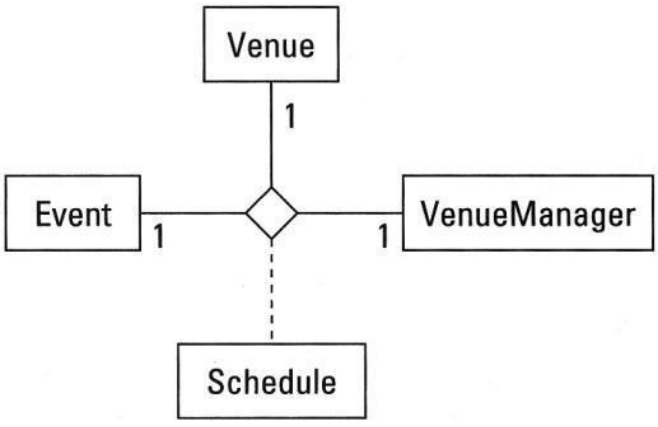

Figure 64. Example of n-ary association

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.

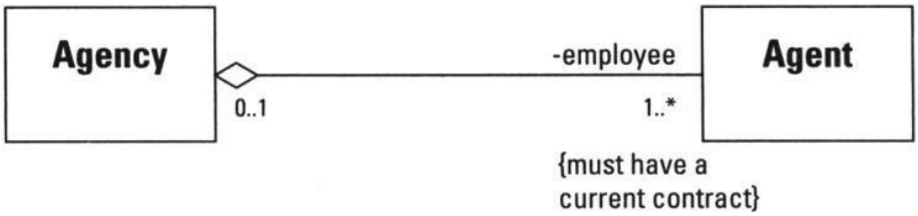

Figure 65. Example of aggregation association

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.

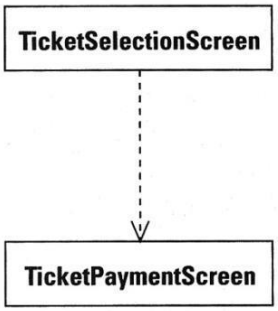

Figure 66. Example of dependency relationships

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

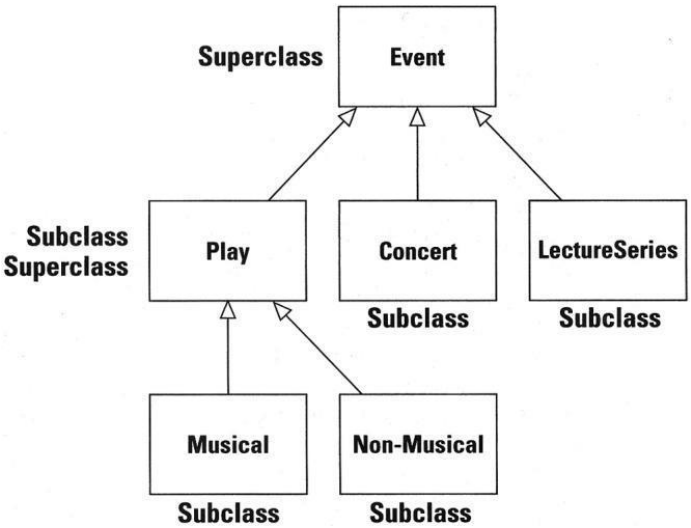

Figure 67. Example of generalization relationships

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).

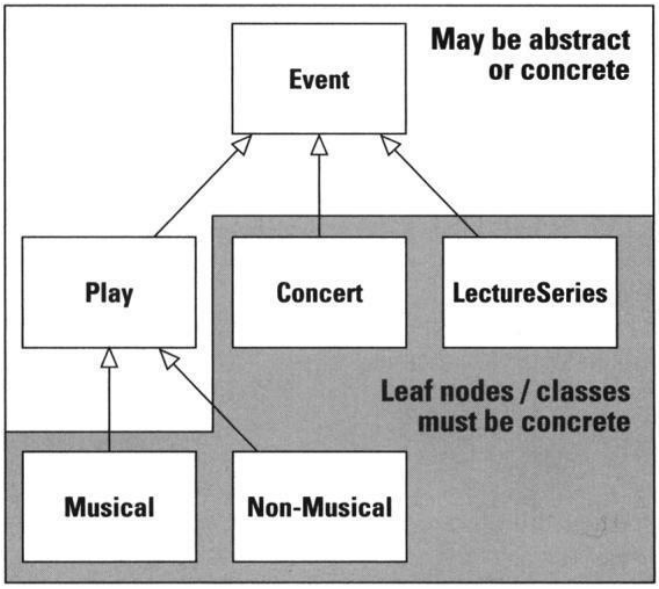

Figure 68. Example of difference between leaf and nodes in a generalization relationships

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.

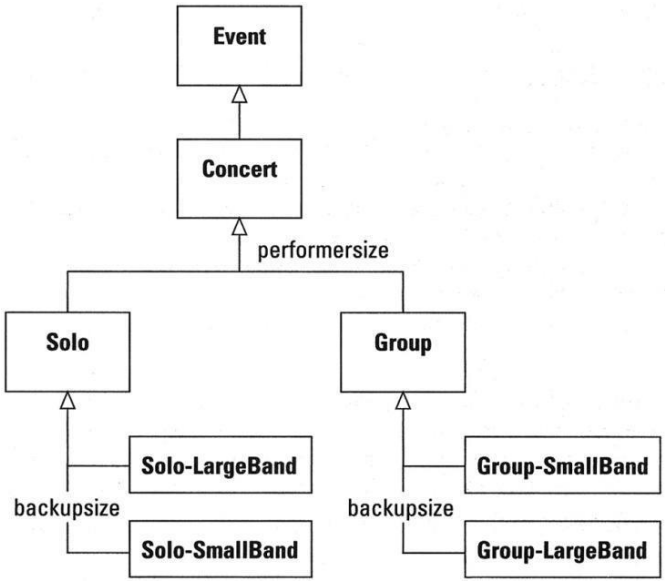

Figure 68. Example of generalization identification

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.

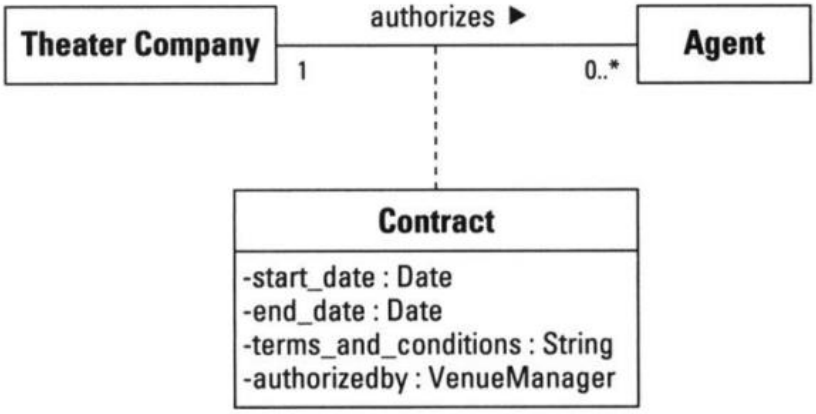

Figure 69. Example of class association

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

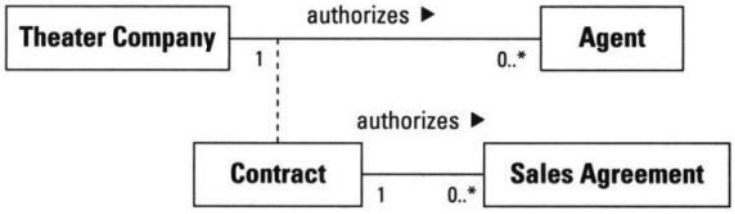

Figure 70. Example of class nature of a class association