chapitre 4 class diagram

Soumia LAYACHI

26 octobre 2025

1 Introduction

Introduction The class diagram is considered the most important diagram in object-oriented modeling ; it is the only mandatory diagram in such modeling. While the use case diagram shows a system from the perspective of the actors, the class diagram shows its internal structure. It provides an abstract representation of the system's objects that will interact to realize the use cases. It is a static view, as the temporal factor in the system's behavior is not taken into account.

2 Objects

An object is an abstraction of a real-world element. It has information, such as name, surname, address, etc., and behaves according to a set of operations that apply to it. In addition, a set of attributes characterize the state of an object, and we have a set of operations (methods) that allow us to act on the behavior of our object. An object is an instance of a class, and a class is an abstract data type, characterized by properties (its attributes and methods) common to objects, which allows the creation of objects possessing these properties. Object=identity+state(attributes)+behavior (methods)

- Identity : The object has an identity, which allows it to be distinguished from other objects, regardless of its state. This identity is generally constructed using an identifier that naturally arises from the problem (for example, a product can be identified by a code, a car by a serial number, etc.)
- Attributes : These are the data characterizing the object. They are variables storing information about the object's state.
- Methods (sometimes called member functions) : An object's methods characterize its behavior, that is, the set of actions (called operations) that the object is capable of performing. These operations allow the object to react to external requests.

Note : Thinking about the object requires a change of mindset !

3 Classes

3.1 Notions of class and class instance

A class is the structure of an object, that is, the declaration of the set of entities that will make up an object. A class is an abstract data type characterized by properties (attributes and operations) common to its objects and a mechanism for creating objects having these properties.

$$\text{Class} = \text{instantiation} + \text{attributes (instance variables)} + \text{operations}$$

— Instantiation : An object has an identity that distinguishes it from other objects, regardless of its state. Instantiation represents the relationship between an object and its class of membership that allowed it to be created.
— Attributes (also called instance variables) : They have a name and either a base type (simple or constructed) or a class (the attribute references an object of the same or another class).
— Operations (sometimes called methods) : These are the operations applicable to an object of the class. They can modify all or part of the state of an object and return values calculated from this state.
— Every object-oriented system is organized around classes.
— A class is the formal description of a set of objects having common semantics and characteristics.

3.2 Graphical representation

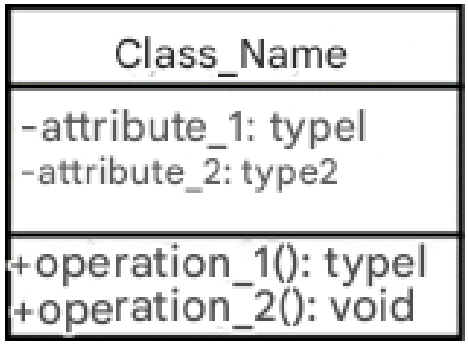

FIGURE 1 – UML representation of a class.

A class is a workbook. It is represented by a rectangle divided into three to five compartments. The first indicates the name of the class, the second its attributes, and the third its operations. A responsibility compartment can be added to list the set of tasks to be performed by the class. An exceptions

compartment can also be added to list the exceptional situations to be handled by the class.

3.3 Encapsulation, visibility, interface

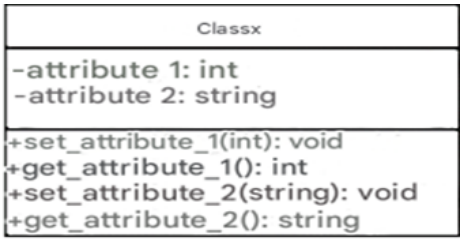

FIGURE 2 – Good practices regarding attribute manipulation.

Encapsulation allows you to define visibility levels for elements within a container. Public or + : Any element that can see the container can also see the specified element. Protected or # : Only an element within the container or one of its descendants can see the specified element. Private or - : only an element located in the container can see the element. Package or ~ or nothing : only an element declared in the same package can see the element. In a class, the visibility marker is located at the level of each of its characteristics (attributes, association ends and operation). It allows to indicate if another class can access it.

3.4 Name of a class

The name of the class must evoke the concept described by the class. It begins with a capital letter. To indicate that a class is abstract, the keyword abstract must be added.

3.5 Attributes

3.5.1 Class attributes

Attributes define information that a class or object needs to know. They represent the data encapsulated in the objects of that class. Each of these pieces of information is defined by a name, a data type, and a visibility and can be initialized. The attribute name must be unique within the class.

3.5.2 Class attributes(Static Variables)

A class attribute is therefore not a property of an instance, but a property of the class and access to this attribute does not require the existence of an instance. Graphically, a class attribute is underlined.

3.5.3 Derived attributes

Derived attributes can be calculated from other attributes and calculation formulas. During design, a derived attribute can be used as a marker until you can determine the rules to apply to it. Derived attributes are symbolized by adding a "/" in front of their name.

3.6 Methods

3.6.1 Class Method

In a class, an operation (same name and same types of parameters) must be unique. When the name of an operation appears several times with different parameters, we say that the operation is overloaded.

3.6.2 Class method (static method)

As with class attributes, it is possible to declare class methods. A class method can only manipulate class attributes and its own parameters. This method does not have access to class attributes (i.e., instances of the class). Accessing a class method does not require the existence of an instance of that class. Graphically, a class method is highlighted.

3.6.3 Abstract methods and classes

A method is said to be abstract when we know its header, but not how it can be implemented. A class is said to be abstract when it defines at least one abstract method or when a parent class contains an abstract method that has not yet been implemented. A pure abstract class has only abstract methods. In object-oriented programming, such a class is called an interface. To indicate that a class is abstract, you must add the keyword abstract after its name.

3.7 Relationships between classes

Class diagrams express the static structure of the system. They describe the set of classes and their associations. A class describes a set of objects (instances of the class). An association expresses a bidirectional semantic connection between classes . An association describes a set of links (instances of the association). The role describes one end of an association. Cardinalities (or multiplicity) indicate the number of instances of one class for each instance of the other class.

3.7.1 Association

An association indicates that there can be links between instances of the associated classes. How should an association be modeled ? More precisely, what is the difference between the two diagrams in the following figure ?

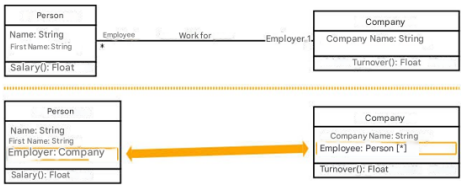

FIGURE 3 – Two ways to model an association.

In the first version, the association appears clearly and constitutes a distinct entity. In the second, the association is manifested by the presence of two attributes in each of the related classes. The question of whether to model associations as such has long been debated. UML decided on the first version, because it is more at a conceptual level (as opposed to the implementation level) and greatly simplifies the modeling of complex associations (such as many-to-many associations, for example).

3.7.2 Association Termination and its Properties

The possession of an association end by the classifier at the other end of the association can be specified graphically by adding a small solid circle (dot) between the association end and the class. It is not essential to specify the possession of association ends. In this case, the possession is ambiguous. On the other hand, if possession of association ends is indicated, all association ends are unambiguous : the presence of a dot implies that the association end belongs to the class at the other end, the absence of the dot implies that the association end belongs to the association.

FIGURE 4 – Using a small filled circle to specify the owner of an association termination.

The diagram specifies that the vertex association end belongs to the Polygon class while the polygon association end belongs to the Defined By association. A property is a structural characteristic. In the case of a class, properties are made up of the attributes and any association endings that the class possesses. In the case of an association, properties are made up of the association endings that the association possesses.

A property can be set by the following elements : name, visibility, multiplicity, navigability.

3.7.3 Binary and n- ary association

Binary association A binary association is materialized by a solid line between the associated classes.

FIGURE 5 – Example of binary association.

n-ary association An n-ary association links more than two classes. An n-ary association is represented by a large diamond with a path leading to each participating class. The name of the association, if any, appears near the diamond. Note : The multiplicity associated with an association, aggregation, or composition term declares the number of objects that may occupy the position defined by the association term. Examples of multiplicity include :

— exactly one : 1 or 1..1 ;
— several :*or 0..* ;
— at least one : 1..* ;
— from one to six : 1..6.

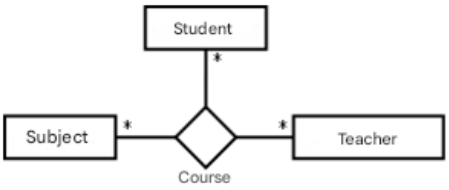

FIGURE 6 – Example of association.

An n-ary association links more than two classes. An n-ary association is represented by a large diamond with a path leading to each participating class. The name of the association, if any, appears near the diamond. Note : The multiplicity associated with an association, aggregation, or composition term declares the number of objects that may occupy the position defined by the association term. Examples of multiplicity include : • exactly one : 1 or 1..1 ; • several :*or 0..* ; • at least one : 1..* ; • from one to six : 1..6.

3.7.4 Class-association

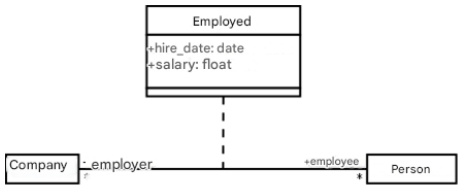

FIGURE 7 – Example of class - association.

Sometimes, an association must have properties. For example, the Employs association between a company and a person has the salary and hiring date as

properties. Indeed, these two properties belong neither to the company, which can employ several people, nor to the people, who can have several jobs. They are therefore properties of the Employs association. Since associations cannot have properties, a new concept must be introduced to model this situation : that of association class. An association class is characterized by a discontinuous line between the class and the association it represents.

3.7.5 Aggregation and composition

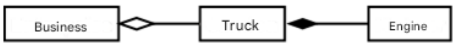

FIGURE 8 – Example of aggregation and composition relationship.

Aggregation When we want to model a whole/part relationship where a class constitutes a larger element (whole) composed of smaller elements (part), we must use an aggregation. An aggregation is an association that represents a structural or behavioral inclusion relationship of an element in a set. Graphically, an empty diamond is added to the side of the aggregate. Unlike a simple association, an aggregation is transitive.

Composition Composition, also called composite aggregation, describes a structural containment between instances. Thus, the destruction of the composite object implies the destruction of its components. An instance of the part always belongs to at most one instance of the composite element : the multiplicity of the composite side must not be greater than 1 (i.e. 1 or 0..1). Graphically, we add a solid diamond (◆) on the aggregate side.

The concepts of aggregation and especially of composition pose numerous problems in modeling and are often the subject of quarrels among experts and sources of confusion.

3.7.6 Generalization and Inheritance

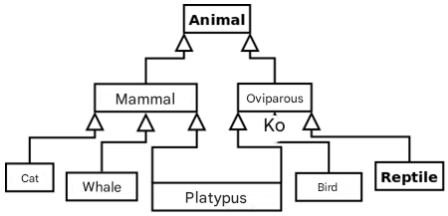

FIGURE 9 – Part of the animal kingdom described with multiple inheritance.

Generalization describes a relationship between a general class (base class or parent class) and a specialized class (subclass). The specialized class is fully

consistent with the base class, but includes additional information (attributes, operations, associations). The symbol used for the inheritance or generalization relationship is an arrow with a solid line whose tip is a closed triangle designating the most general case. The main properties of the inheritance are : • the child class has all the characteristics of its parent classes, but it cannot access the private characteristics of the latter ; • a child class can redefine (same signature) one or more methods of the parent class. Unless otherwise indicated, an object uses the most specialized operations in the class hierarchy ; • all associations of the parent class apply to derived classes ; • A class can have several parents, which is called multiple inheritance. The C++ language is one of the object languages that allows its effective implementation, while the Java language does not.

3.7.7 Dependence

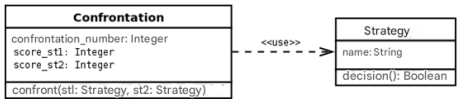

FIGURE 10 – Example of a dependency relationship.

A dependency is a unidirectional relationship expressing a semantic dependency between elements of the model. It is represented by a directed dashed line. It indicates that modifying the target may imply a modification of the source. The dependency is often stereotyped to better explain the semantic link between the elements of the model. A dependency is often used when one class uses another as an argument in the signature of an operation.

3.7.8 Interfaces

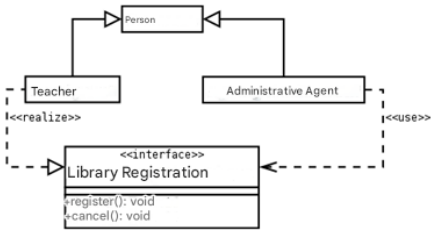

FIGURE 11 – Example of a diagram implementing an interface.

An interface is represented as a class except for the absence of the keyword abstract (because the interface and all its methods are, by definition, abstract)

and the addition of the stereotype «interface> >. An interface must be implemented by at least one class and can be implemented by several. Graphically, this is represented by a broken line ending in a triangular arrow and the stereotype "realize". A class can very well realize several interfaces. A class (client class of the interface) can depend on an interface (required interface). This is represented by a dependency relationship and the "use" stereotype.

3.8 Object diagram

An object diagram represents objects and their relationships to provide a fixed view of the state of a system at a given time. The class diagram models rules and the object diagram models facts.

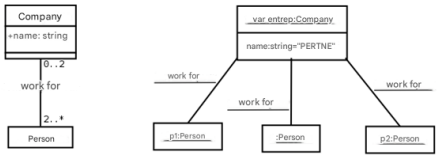

FIGURE 12 – Example of a class diagram and associated object diagram. Graphically, an object

is represented as a class. However, the operations compartment is not useful. In addition, the name of the class of which the object is an instance is preceded by a « : » and is underlined. To differentiate objects of the same class, their identifier can be added before the class name. Finally, the attributes receive values. When some attribute values of an object are not filled in, we say that the object is partially defined. In an object diagram, the relationships in the class diagram become links. The generalization relationship has no instance, so it is never represented in an object diagram. Graphically, a link is represented as a relationship, but if there is a name, it is underlined. Naturally, multiplicities are not represented.

Example

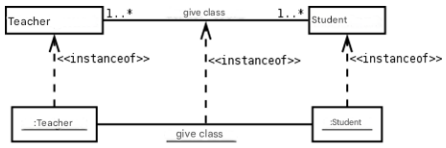

FIGURE 13 – Instantiation dependency between workbooks and their instances.

The instantiation dependency relationship (stereotyped «instance of») describes the relationship between a workbook and its instances. In particular, it connects links to associations and objects to classes.

Références

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

[2] Benoit Charroux, Aomar Osmani, Yann Therry-Mieg, *UML 2 – Synthèse et exercices*, Pearson Édition française, ISBN 2-7440-7124-2, 2005.

[3] G. Booch et al., *Object-Oriented Analysis and Design with Applications*, Addison-Wesley, 2007.

[4] Laurent Audibert, *Cours UML 2.0*, disponible sur : `http://www.developpez.com`