

by Javier Carrasco

Development Environments

Block 2

Unit 5: Classes and objects. Class diagram. Specific software. Other useful diagrams

2.5.1. Classes and objects

When we talk about Object Oriented Programming (OOP), these two concepts are really closely linked, and they are, in fact, the basis of this kind of programming. Regarding languages such as Java, they are its essence, from which the whole language is built.

We define a **class** as a template to set the form of an object, by specifying the data and code that will be managed by this object. So a class is basically a set of schemas that tell us how to create an object and what it will have.

So **objects** are specific instances of a class. There is no physical representation of a class in memory until we create an object of this class.

Let's see how to define a class structure in Java.

```
class ClassName {  
    // Instance variables or attributes  
    type var1;  
    type var2;  
    // ...  
    type varN;  
  
    // Methods  
    type method1 (params) {  
        ...  
    }  
  
    type method2 (params) {  
        ...  
    }  
    // ...  
    type methodN (params) {  
        ...  
    }  
}
```

Another typical object oriented language is C++. In this case, the structure of a class would be as follows:

```
class ClassName {  
    private:  
        // Instance variables or attributes  
        type var1;  
        type var2;  
        // ...  
        type varN;  
  
    public:  
        // Methods  
        type method1 (params);  
        type method2 (params);  
        // ...  
        type methodN (params);  
}  
  
type ClassName::method1(params) {  
    ... // code of this method  
}
```

Let's illustrate how to use classes with a Java example. We will create a class that stores information about vehicles (cars, vans, trucks...).

```
class Vehicle {  
    int passengers; // number of passengers allowed  
    int fuelcap;    // fuel capacity (in liters)  
    int kml;        // fuel consumption (in km / l)  
}
```

If we want to create (instantiate) an object of this class `Vehicle`, we would use this instruction:

```
Vehicle minivan = new Vehicle();
```

After running this instruction, `minivan` becomes an instance of class `Vehicle`, so it has a physical memory location. Every time we instantiate a class, an object is created with its own values for the instance variables.

This example uses class `Vehicle` with some more lines of code.

```
class ExampleVehicle {  
  
    public static void main(String[] args) {  
        Vehicle minivan = new Vehicle();  
        int autonomy;  
  
        // Assign values to minivan instance variables  
        minivan.passengers = 5;  
        minivan.fuelcap = 20;  
        minivan.kml = 3;  
  
        /*  
         * Calculate minivan autonomy, assuming that  
         * the fuel tank is full  
         */  
        autonomy = minivan.fuelcap * minivan.kml;  
        System.out.println("Minivan vehicle can carry up to "  
            + minivan.passengers + " passengers with an autonomy of "  
            + autonomy + " km.");  
    }  
}
```

If we run the code above, we will see an output like this one:

```
Minivan vehicle can carry up to 5 passengers with an autonomy of 60 km.
```

If we think about it a bit, we can refactor the code above, so that the calculation of the autonomy can be part of the vehicle itself (part of the `Vehicle` class). This way, we can know the autonomy of any vehicle by simply calling the corresponding method. Our `Vehicle` class would be like this:

```
class Vehicle {  
  
    int passengers;  
    int fuelcap;  
    int kml;  
  
    void autonomy() {  
        System.out.println("This vehicle has an autonomy of "  
            + (fuelcap * kml) + " km.");  
    }  
}  
  
class ExampleVehicle {  
  
    public static void main(String[] args) {  
  
        Vehicle minivan = new Vehicle();  
  
        minivan.passengers = 5;  
        minivan.fuelcap = 20;  
        minivan.kml = 3;  
  
        minivan.autonomy();  
    }  
}
```

We have just added some functionality to our `Vehicle` class through `autonomy` method. Let's see now some other concepts that we need to take into account when talking about classes and objects.

2.5.1.1. Constructors

We use **constructors** to initialize the objects when we create them. They have the same name than the class to which they belong, and they are defined like a method, but they don't have any return type.

Inside the code of a constructor, we typically assign initial values to instance variables, and call any other method that may be useful at the beginning of the object's lifetime.

Let's go on with our `Vehicle` class. In this case, we are going to add a constructor with no parameters, also known as **default constructor**. We will use it whenever we need to create an object of this class.

```
class Vehicle {  
  
    int passengers;  
    int fuelcap;  
    int kml;  
  
    // Default constructor  
    Vehicle() {  
        passengers = 0;  
        fuelcap = 0;  
        kml = 0;  
    }  
  
    void autonomy() {  
        System.out.println("This vehicle has an autonomy of "  
            + (fuelcap * kml) + " km.");  
    }  
}  
  
class ExampleVehicle {  
    public static void main(String[] args) {  
        // Use default constructor to create an object  
        Vehicle minivan = new Vehicle();  
  
        minivan.autonomy();  
    }  
}
```

The output that we would get in this case is different, since we have not set the instance variables manually, but through the default constructor, so all of them are 0.

```
This vehicle has an autonomy of 0 km.
```

In some cases, we may need to assign some non-default values to the instance variables, so we use a constructor with parameters. Typically, each parameter corresponds to an instance variable.

```
class Vehicle {  
  
    int passengers;  
    int fuelcap;  
    int kml;  
  
    // Default constructor  
    Vehicle() {  
        passengers = 0;  
        fuelcap = 0;  
        kml = 0;  
    }  
  
    // Parameterized constructor  
    Vehicle(int p, int f, int k) {  
        passengers = p;  
        fuelcap = f;  
        kml = k;  
    }  
  
    void autonomy() {  
        System.out.println("This vehicle has an autonomy of "  
            + (fuelcap * kml) + " km.");  
    }  
}  
  
...
```

Using this constructor, we would create a `Vehicle` object as follows:

```
Vehicle minivan = new Vehicle(5, 20, 3);
```

We can have as many constructors as we need. All of them will have the same name (the class name), but they will have different parameters.

2.5.1.2. Access modifiers

Access control is something essential in object oriented programming, so we have to take it into account. It establishes which elements are visible from which other classes of our code. This is managed through three access modifiers: **public**, **private** and **protected** (this last one is only employed for inheritance, as we will see later).

If we assign the **public** modifier to a class member, we will be able to access this member directly from any part of the code, including any other class.

If we use the **private** modifier, we would only be able to access this member from other members of the same class. Out of this class, this member is not visible.

```
class Vehicle {
    private int passengers;
    private int fuelcap;
    private int kml;

    public Vehicle() {
        passengers = 0;
        fuelcap = 0;
        kml = 0;
    }

    public Vehicle(int p, int f, int k) {
        passengers = p;
        fuelcap = f;
        kml = k;
    }

    private int autonomy() {
        return fuelcap * kml;
    }

    public void vehicleData() {
        System.out.println("This vehicle can carry up to " + passengers
            + " passengers and it has an autonomy of " + autonomy()
            + " km.");
    }
}

class ExampleVehicle {
    public static void main(String[] args) {
        Vehicle minivan = new Vehicle(5, 20, 3);
        minivan.vehicleData(); // OK
        minivan.autonomy();    // ERROR: not visible
    }
}
```

The output of this example (as long as we comment the last line of code, which is not correct because it tries to access a private member of `Vehicle` class), is as follows:

```
This vehicle can carry up to 5 passengers and it has an autonomy of 60 km.
```

2.5.2. Class diagram

A class diagram is a graphical representation of the relationships between the classes of a software project, also representing its own structures and behaviour (not temporal behaviour, since state transition diagrams are in charge of this issue).

Now, let's see what kind of elements we can add to a class diagram: classes, attributes, methods and relationships between classes.

2.5.2.1. Class

This is the basic element, and it includes all the information about the objects of a class.

ClassName	In the upper part we will place the class name. Below this part, we will specify the attributes or instance variables of the class, taking into account that they can be <i>public</i> (+), <i>private</i> (-) or <i>protected</i> (#).
+Attributes	
+Methods()	

And last, in the lower part, we will add the methods that let the objects interact with other objects. They can also be *public*, *private* or *protected*.

Let's see an example of a *BankAccount* class with some attributes and methods:

BankAccount
-balance: float
+addMoney(amount:float): void
+transferMoney(amount:float): void
+checkBalance(): float

This class represents a bank account with an attribute called *balance* that stores the current amount of money that is available in the account. If we pay attention to the symbol that precedes this attribute, we can notice that it is a private attribute. Besides, the class can do some operations: add money, transfer money or check the balance. All these operations are public.

2.5.2.2. Attributes

The attributes or instance variables can be of three types, according to their visibility, as we have seen before:

- **public (+)**: visible attribute, it can be accessed from any other class.
- **private (-)**: private attribute, it can only be accessed from within current class, and can only be used by current class' methods.
- **protected (#)**: protected attribute, it can only be accessed by current class and all its subclasses (as we will see later when we talk about inheritance).

2.5.2.3. Methods

Methods show how a class will interact with its environment (other classes and objects), and what operations can be performed from that class. They can also be of three types, according to their visibility: public, private and protected.

2.5.2.4. Class relationships

Now that we know what a class is, let's see how we can establish relationships between different classes. First of all, we need to talk about the concept of **cardinality** of a relationship.

Cardinality sets the dependency level of two classes, and must be indicated at both sides of the relationship. It can be:

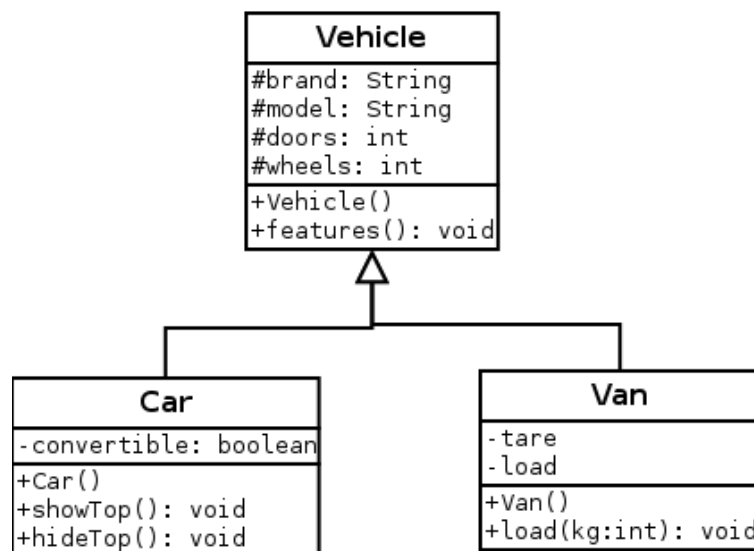
- **One to many:** specified as `1..*` or `(1..n)`
- **Zero to many:** `0..*` or `(0..n)`
- **Fixed number:** `m` (m is a given number)

Taking this into account, there can be different relationships between classes: inheritance (specialization) or associations, which can also be of different types. Let's have a look at all of them.

2.5.2.4.1. Inheritance (Specialization/Generalization)



This relationship establishes that a given class called **subclass** or **child class** inherits the attributes and methods of another class also called **superclass** or **parent class**. Besides, subclass can also have its own attributes and methods (apart from those *public* and *protected* inherited from the superclass).

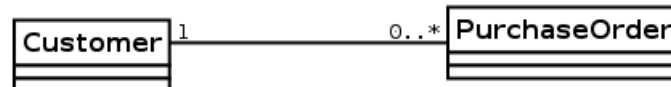


In this example, we can see that classes `Car` and `Van` inherit from `Vehicle` parent class. `Car` class has all the features inherited from `Vehicle` (brand, model, doors, wheels... and also the methods) and an additional attribute *convertible*. Regarding `Van` class, it also inherits all the attributes from `Vehicle`, and adds its own new attributes *tare* and *load*.

2.5.2.4.2. Association



This relationship lets us link two classes that collaborate between them, so that an object of a class has an object (or some objects) of the other class. It is not a strong relationship, this is, the lifetime of one side does not depend on the other side.



In this example we can see a **Customer** class that has many **Purchase orders** (or none). A purchase order can only be associated to one customer.

There can be an arrow at one of the edges of the association. In this case, the arrow indicates the **navigability** of the association. It indicates that we can only get the objects of the edge pointed to by the arrow from the other edge.



In the image above, class B2 is *navigable* from class A2, so we can access B2 object(s) contained in A2 class, but we can't get A2 elements present in B2. If the arrow is not present, then navigability is not specified: we can either navigate both sides of the association.

2.5.2.4.3. Aggregation and composition

There are two special types of associations in which one class is a whole which contains the other class.

A **composition** is a special type of association in which the lifetime of an object depends on another object that includes it (there is a *strong* relationship between them). In other words, if we delete the main object that includes the other one, then the included object is also deleted. It is represented by a black diamond next to the main including class.

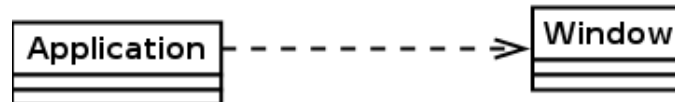
An **aggregation** is a special type of association in which an object is part of the other object (as composition is), but its lifetime does not depend on the main containing object (it is a *weaker* relationship). It is represented by an empty diamond next to the main including class.



In this example we can see a **Store** that has **Customers** and **Accounts**. The diamonds are next to the main containing class (*Store*), and there is a composition with the **Account** class (so, if the store is deleted, its accounts will also be deleted), and an aggregation with the **Customer** class (so that customers will not be deleted, and can be linked to other stores).

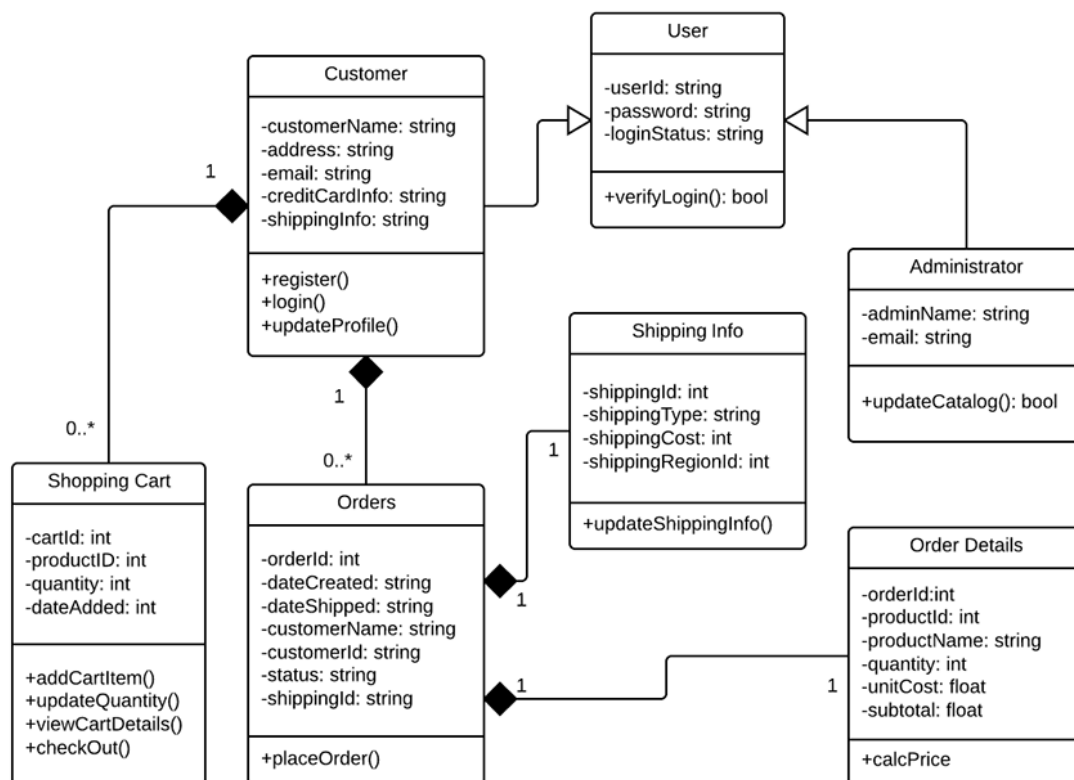
2.5.2.4.4. Dependency, instantiation or usage

This is a usage relationship, this is, a class uses another class in some part of its code. This is the weakest relationship, and it is represented with a discontinuous arrow: $\text{---} \Rightarrow$, the direction of the arrow tells us which class is being used (the one pointed by the arrow).



This example shows how dependency works in a class diagram. **Application** class instantiates a **Window** object, so the instantiation of this object depends on the behaviour of the *Application* object. You should also take into account that the instantiated object (the window) is not stored in the object that instantiates it (the application).

To finish with this section, let's see the complete class diagram of a simple online shop.



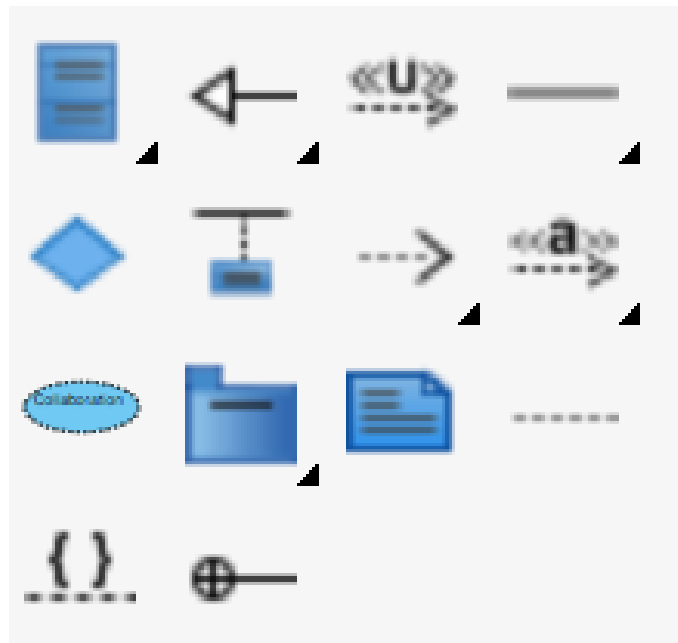
It has two types of users: customers and system administrators. Customers can have many shopping carts and orders. Each order has its own shipping info and order details, which is composed of products with their prices and quantities.

2.5.3. Specific software

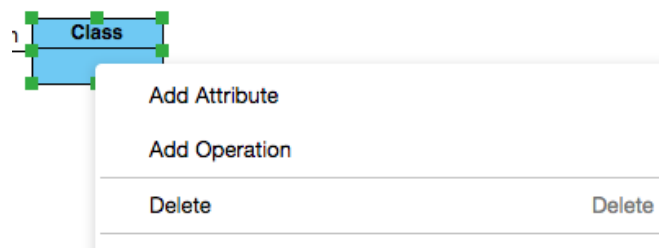
Regarding the software that we can use to create class diagrams, we are going to rely on **Visual Paradigm Online** or **Modelio** again, as we did in previous units. You should be familiar with these tools by now, so let's check the main elements that we need to manage in order to create class diagrams with them.

2.5.3.1. Visual Paradigm Online

Regarding **Visual Paradigm Online**, this is the toolbox for class diagrams:



1. First icon lets us create **classes**. We can set the class name by just typing it once the class has been selected. If we right click on the class, there are two context menus called *Add Attribute* and *Add Operation* to add a class attribute and a method, respectively.



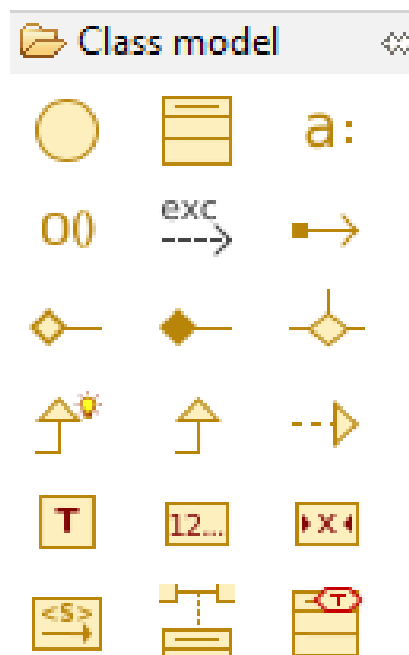
- 2nd icon lets us join two classes with a **generalization/inheritance** relationship.
- 4th icon lets us define **associations** between classes. From the arrow icon in the upper toolbar we can define its navigability.
 - We can also set the multiplicity of both sides of the association by double clicking on the place in which we want to place the text.
- 5th icon (first icon of the second row) lets us define **aggregations** and **compositions**. We just need to join the diamond to an association line, and we can change the background color of the diamond from the upper toolbar. Also, we can define aggregations or compositions from the 4th icon, by clicking on the bottom right edge and choosing the appropriate type of association.



- 7th icon (third icon of the second row) lets us define **dependency** associations
- The rest of icons provided by this tool are not going to be used.

2.5.3.2. Modelio


In **Modelio**, once we have chosen the *class diagram* option when creating a new diagram, this is the main toolbox for creating them:



- 2nd icon lets us define classes. If we double click on the class we can change some basic properties, such as the class name.
- 3rd and 4th icons let us add attributes and methods to a class, respectively. We just need to select the icon and then click on the class we want to place the attribute/method on. Again, if we double click on the attribute/method, we can change its properties: visibility, attribute/method name, and even parameter names and types, regarding methods.

Property	Value
Name	Attribute
Type	 string
Visibility	Public
Multiplicity min	1
Multiplicity max	1
Value	
Access mode	Read/Write
Type constraint	
Abstract	<input type="checkbox"/>
Class	<input type="checkbox"/>
Derived	<input type="checkbox"/>

- 6th icon lets us define associations. We just select this tool and then click on both classes that we want to associate. If we double click on the association line, we can change its properties: navigability, association type (including aggregations and compositions), multiplicity of both sides and so on.

Property	From: Class1	To: Class
Association name		
Navigable	<input checked="" type="checkbox"/>	<input type="checkbox"/>
Role	class1	
Target	 Class1	<null>
Association type	Association	Association
Multiplicity min	0	0
Multiplicity max	1	1
Visibility	Public	N/A
Is changeable	<input checked="" type="checkbox"/>	N/A
Accessors	Read/Write	N/A
Abstract	<input type="checkbox"/>	N/A

- We can also establish an aggregation and composition from the 7th and 8th icons, respectively.
- Finally, we can set generalizations or inheritance from the 11th icon.
- Apparently, there's no specific tool for dependency relationships, but we can use any discontinuous arrow to represent them, such as the 12th icon.

Proposed exercises:

2.5.3.1. Represent in a **class diagram** the following specification:

We must represent the main features and classes of ships. As attributes, we need to store the **name**, the number of **anchors**, the **material** with which it is built (wood, metal, fiberglass...) and its **length** (in metres). We are going to focus on four ship types:

- **Container ship**, for which we must also store its capacity in **TEU**, a unit of measurement which is equivalent to one container.
- **Bulk carrier**, for which we need to know the number of load **hatches** (it is usually an odd number).
- **Tanker**, for which we need to know if it has **double helmet** or not, and its **capacity** in tons.
- **Fishing boat**, for which we will store the **type of fishing** (commercial, artisan or sport).

2.5.3.2. Represent the following specification in a **class diagram**:

Let's suppose that we want to develop a video game, and we need to think about the characters of this videogame. We are going to define a **Character** class, that will have some attributes such as the **age**, **height**, **genre** and **life level**. Depending on the role of this character in the game, it can be:

- **Soldier**, whose additional attribute will be the **weapon** that he will be carrying (sword, bow...). There are two main types of soldiers:
 - **Bowman**, whose additional attributes will be the **movement type** and the **damage** caused.
 - **Knight**, with a **movement type**, **damage** caused and life level of its **horse**.
- **Citizen**, whose single additional attribute will be his **tool** (hammer, hoe...).
 - **Farmer**, whose additional properties will be the **movement type**, the **job** (gather, plant...) and the **work level**, that will indicate how much it will take to finish a task.
 - **Miner**, whose properties will be the **movement type**, the **job** to be done (chop, demolish, extract...), and the **fatigue level**, that will tell us when he will be exhausted.

2.5.3.3. This is another exercise from previous units that has also been adapted to this one. Try to define the class diagram for it.

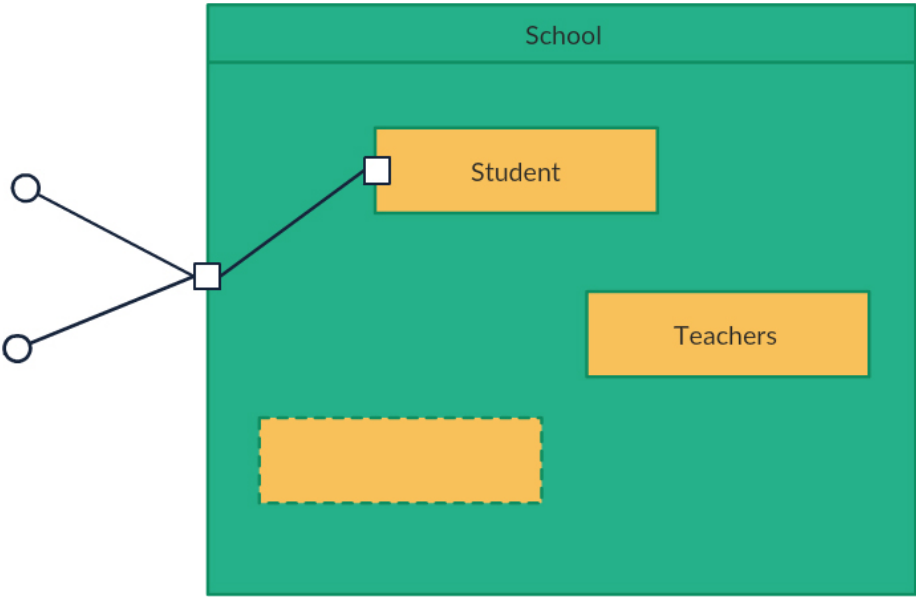
A blog has three types of users: administrators, editors and visitors. For all of them we will save their login and password. Administrators can register other users, editors can publish posts, and visitors can comment them. Besides, we will have an additional user type, called *author* which is a subtype of editor that can create and publish posts, but he must be always under the supervision of an editor.

2.5.3.4. Try now with the class diagram of this exercise from previous units, which has been adapted to this unit to include the attributes of the involved classes:

A cultural organization is focused on the loan of two type of objects: music discs and books. For both of them we store some general information, such as the *id*, title and author. Regarding books, we also store the number of pages, and for music discs we are interested in the record company. There are many users that come to this organization, for whom we store its *id* (DNI) and name. They can ask for books and music discs (up to 5 objects simultaneously), for which we will store the start and end date.

2.5.4. Other useful diagrams

Composite structure diagram: They show the internal structure of a class.



Object diagram: Similar to the class diagram, but they show the relationships between the objects of the application at a given moment, with concrete values for their attributes.

