

Laboratory

Lab Sessions 2 and 3

**OO Design. Logic Layer.
Constructors and
Classes Design.**

Software Engineering
ETS Computer Science
DSIC – UPV

Year 2024-2025

1. Goal

In the following two sessions, the objective is to obtain the initial code of the logical layer of the case study application. Thus, students will have to program all the classes of the design diagram that we will provide using the language C#.

2. Connect to the Project and retrieve the repository

Each team member can connect from Visual Studio to the Azure DevOps project, to clone the remote repository into the local repository, in case of logging on to the lab computers. If you are on a private computer where the repository was previously cloned, you only need to synchronize to get the latest changes. Figure 1 summarizes the steps to follow to connect and clone the repository (if in doubt, see seminar 2 or the bulletin of the lab session 1).

Only one member of the team (e.g. the team leader) must carry out sections 0 and ¡Error! No se encuentra el origen de la referencia. of this bulletin.

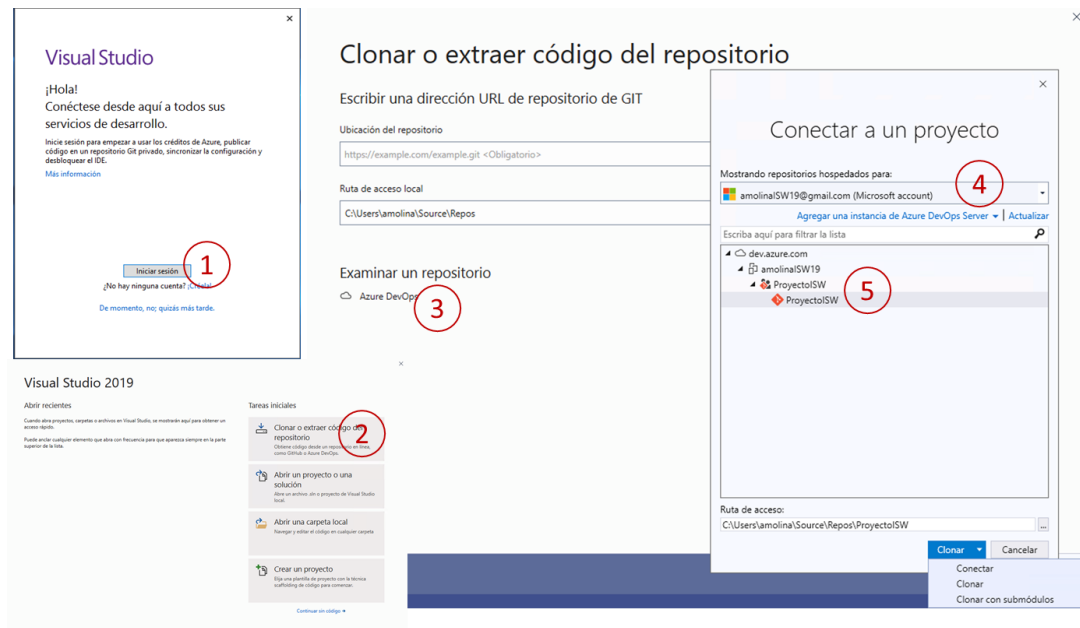


Figure 1. Steps to connect and clone the team project

3. Initial configuration of the solution

This step should be carried out only by one member of the team, in order to avoid unnecessary conflicts in the repository.

As has already been studied in the theory and seminar classes, the application to be developed will have three layers: Presentation, Business Logic and Persistence. In the previous sessions of lab and seminar, a class library was incorporated to the project. In addition, we prepared this library with three solution folders: *Library*, *Presentation* and *Testing*. At the same time, within the *Library* folder, we added two other subfolders called *BusinessLogic* and *Persistence*. During this session, we will begin to add code to these folders. Check that your solution looks like the one shown in Figure 2. If it does not, the team leader should follow seminar 2 and lab bulletin 1 to complete the missing steps in order to get the required project structure.

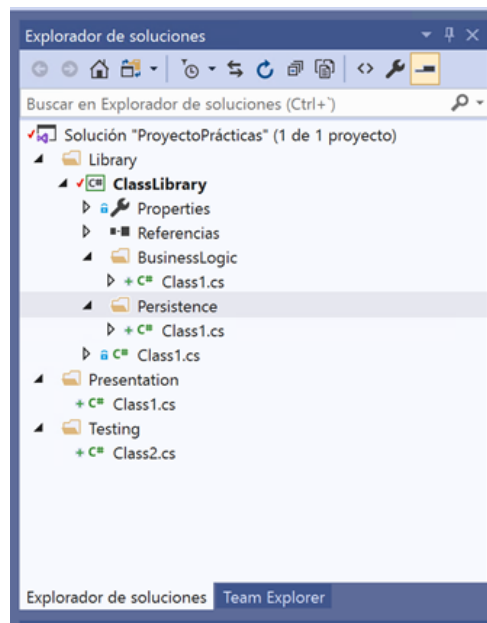




Figure 2. Initial configuration of the solution

After opening the solution, the team leader must create a solution folder named *Entities* in the *BusinessLogic* folder (press the right mouse button to open the context menu and select **Agregar > Nueva Carpeta**). **In addition, we will add an empty class to avoid problems with git.**

Afterwards, you must perform the same step for the folder *Persistence*, by also adding a subfolder *Entities* and an empty class inside.

At this point, the team leader must commit the changes. Remember that the system requires a text message that should describe the changes made.

Next, the leader should push the commit to the remote repository by selecting the option **Insert** . All the team members should synchronize their local repositories with the remote, to check that the changes have been uploaded properly (Extract ).

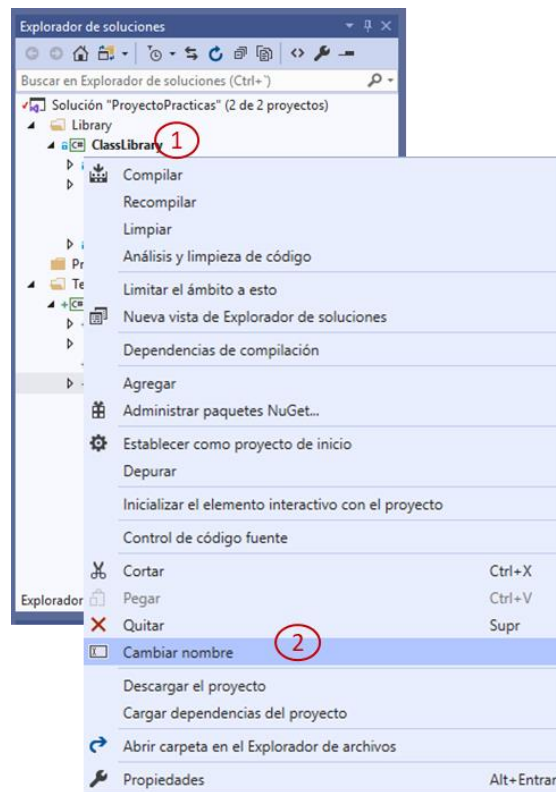


Figure 3. Change the name of the class library

Now, the team leader will change the name and configure the library of classes we added in previous sessions. For this purpose, the team leader must select the library named ClassLibrary from the Explorador de Soluciones tab and clicks the right button of the mouse. From the context menu, the team leader must select the option Cambiar Nombre, naming it *GestAcaLib* (see Figure 3)

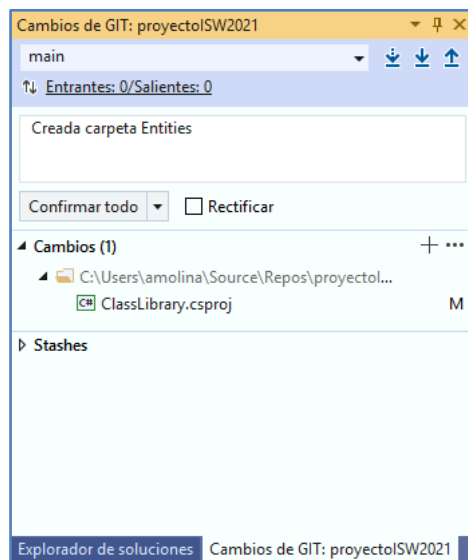


Figure 4. Commit and Inserting changes in the repository (push)

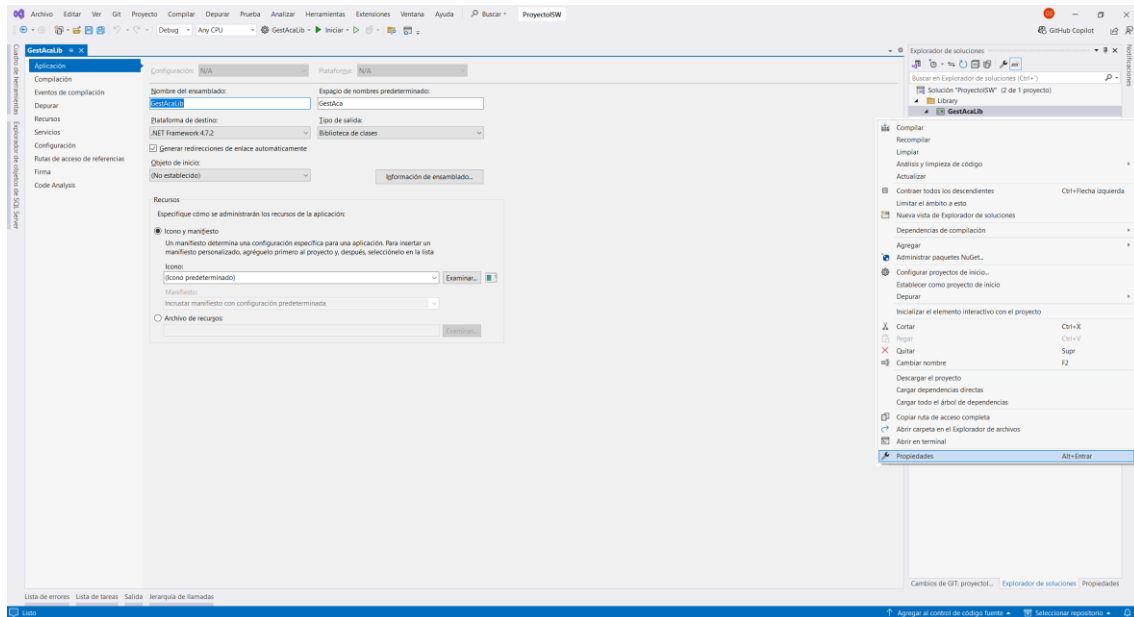


Figure 5. Changing assembly name and namespace

In the next step, the team leader should configure the namespace and the assembly's name of the library. Firstly, the leader should select the library again, and click on the right button of the mouse. From context menu, they select the option: *Propiedades*. On the left, the properties sheet of the library will open. The field *Nombre del ensamblado* should be updated to the value: *GestAcaLib*. Furthermore, the leader should modify the field *Espacio de nombres predefinido* to the value *GestAca*. Figure 5 summarizes the steps.

Finally, the team leader deletes the default class *Class1.cs* from the root of the library (but not the ones inside the folders created in the previous steps). Then, they commit and push the changes. Again, colleagues can retrieve these changes and check the modifications.

4. Add the classes of the design model

The design model of the application you must implement is displayed in Figure 7. The model contains eight classes. The implementation of your solution **must respect the name of the classes and their attributes**, as well as faithfully reflect the relationships established in the model. Attention must be paid to the navigability between the relationships. Bi-directional relationships must be implemented in both directions, while unidirectional ones, if any, will only be implemented in the direction indicated by the arrow.

4.1 First class creation

We are going to add a class file for each one of the classes of the model in the folders we have just created. We will use the strategy of partial classes that allows us C# (**public partial class**), so that the implementation of a class can be distributed in more than one file. We will use partial classes to be able to separate the persistence aspect from the business logic aspect for each class of our model. Specifically, we will separate the implementation of the class in two files: one in the *Persistence/Entities* folder, which will include the class's properties declaration, and the other in the *BusinessLogic/Entities* folder, which will contain the class's logic (constructors and methods).

The **next steps should only be done by the team leader** to avoid unnecessary conflicts in the repository. First, the leader should open the context menu in the *Persistence/Entities* folder (by clicking on the right button of the mouse).

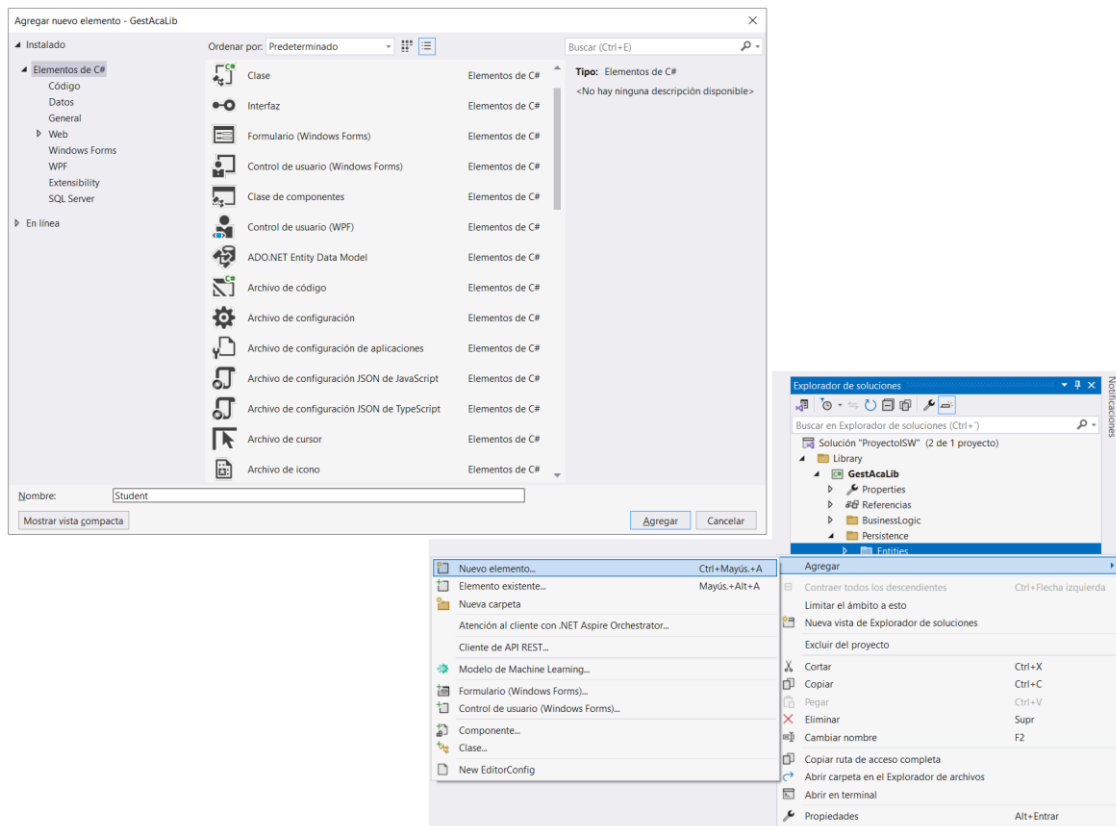


Figure 6: steps to create a new class.

The steps to add a new class are (see Figure 6):

1. Right-click over *Persistence/Entities* solutions folder to open the context menu.
2. Select **Agregar > Nuevo elemento**.
3. Choose the element “Class”, which is the default one.
4. Write the name of the new class.
5. Click on “Agregar”.

In the editor, the leader will change the namespace and the definition of the class, so that the content is like the following code:

```
namespace GestAca.Entities
{
    public partial class Subject
    {
    }
}
```

After that, commit the changes again and push them to the remote repository. The teammates can pull these changes again and check them.

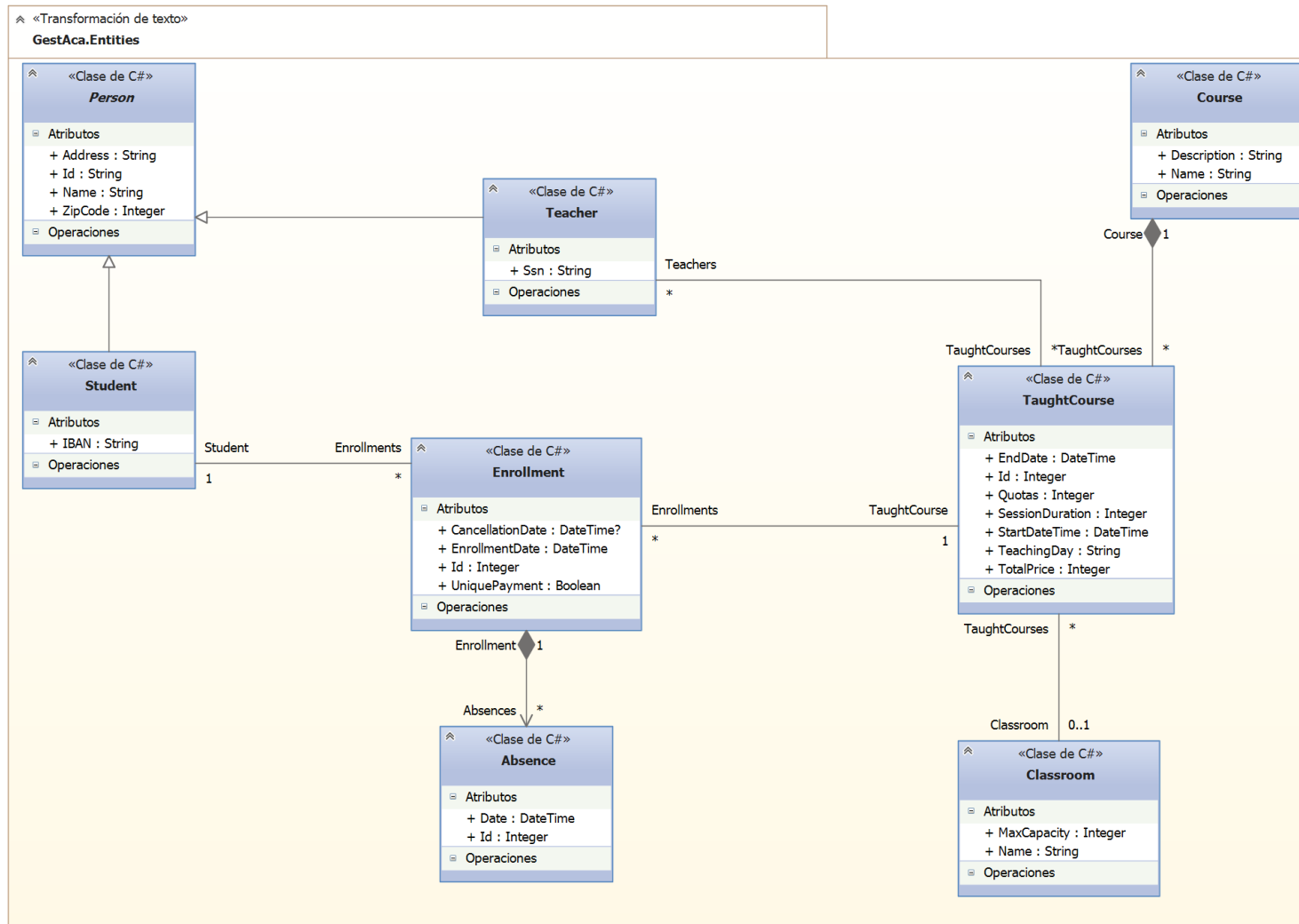


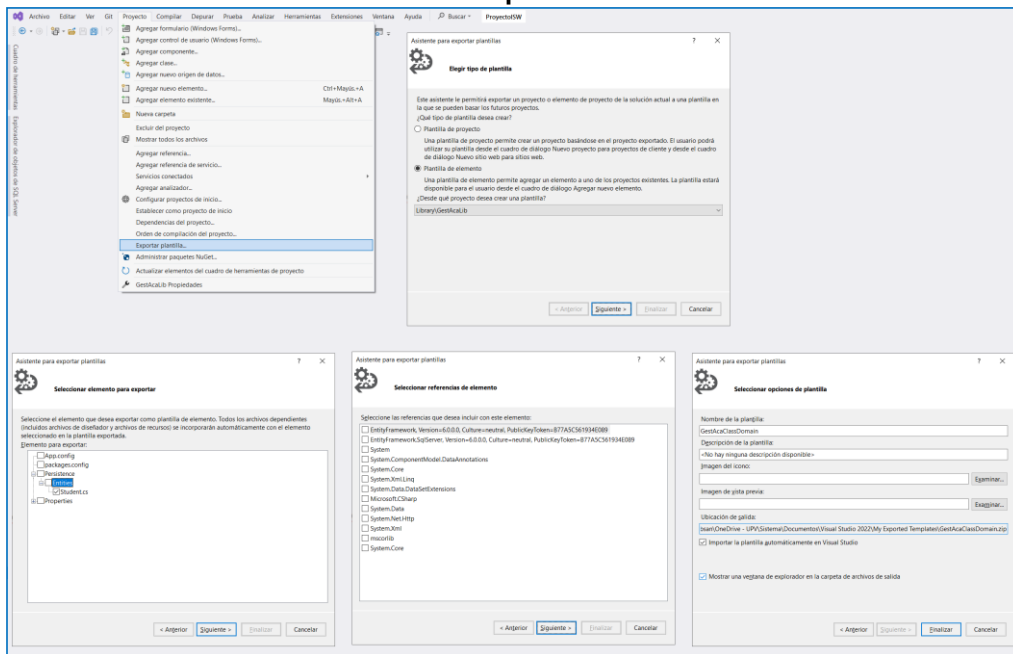
Figure 7. Design class diagram of the case study

4.2 Creation of a class template

Before we continue, we are going to create a template to create all classes as public and partial, within the same namespace. For this, the team leader:

- Selects from the main menu the option: Proyecto>Exportar Plantilla.
- A dialog appears to select the type of the template. Selects the option: Plantilla de elemento, then clicks on Siguiente.
- A dialog is shown to select the element to export. Selects the class Subject.cs, then clicks on Siguiente again.
- Now, a dialog to select the default references is displayed. No option should be selected. Clicks on Siguiente.
- A last dialog is shown to indicate the template options. The leader should change the name of the template by GestAcaClassDomain and then, clicks on Finalizar.

In this moment it is necessary to close and restart Visual Studio to load the new template.



shows an overview of these steps.

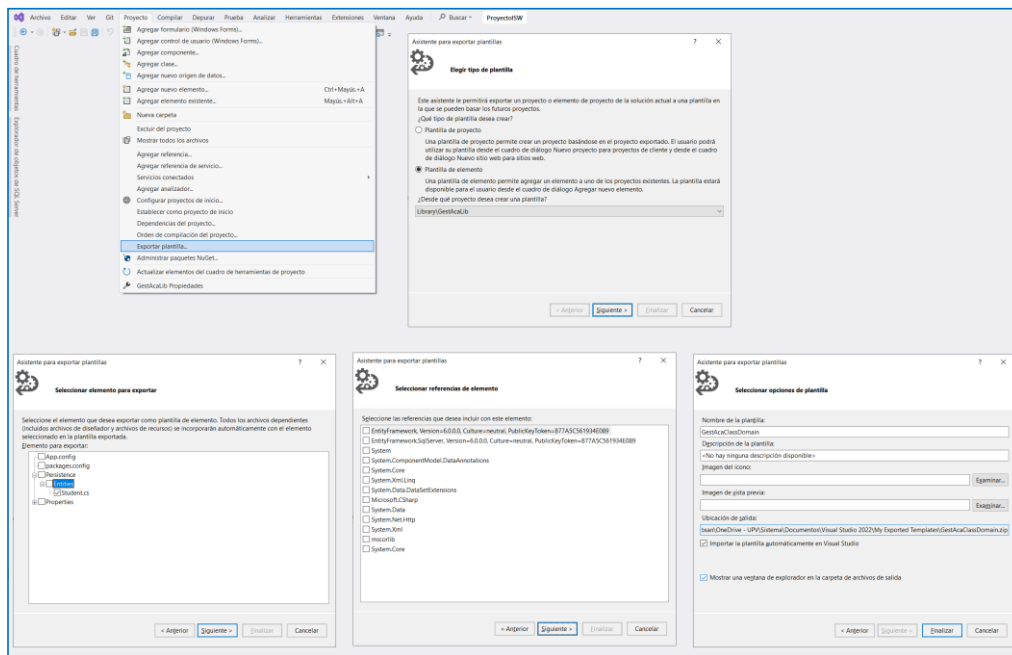


Figure 8. Creating a template for domainclasses

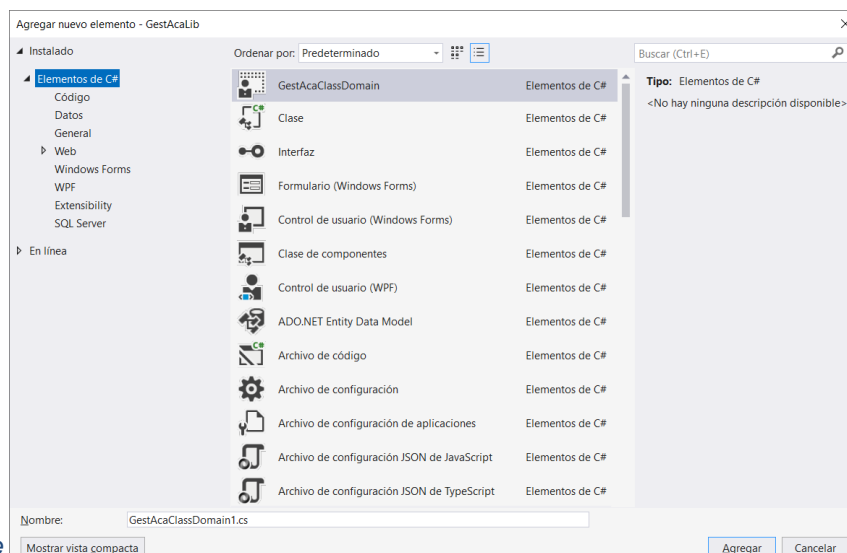
4.3 Add the rest of the model classes inside Persistence/Entities

After reopening the Project in Visual Studio, **the team leader will create** the rest of the classes using the new template, repeating for each one of them the following steps:

- Go to the solution folder *Persistence/Entities*.
- Open the context menu by clicking the right button of the mouse.
- Select **Agregar>Nuevo Elemento**

Choose the option: GestAcaClassDomain. In the field name, indicates the name of the

new class (see



- Figure 9)
- Click on **Agregar**

As a result, you should have got the same content that can be observed in Figure 9. Commit changes and synchronize with the rest of the team.

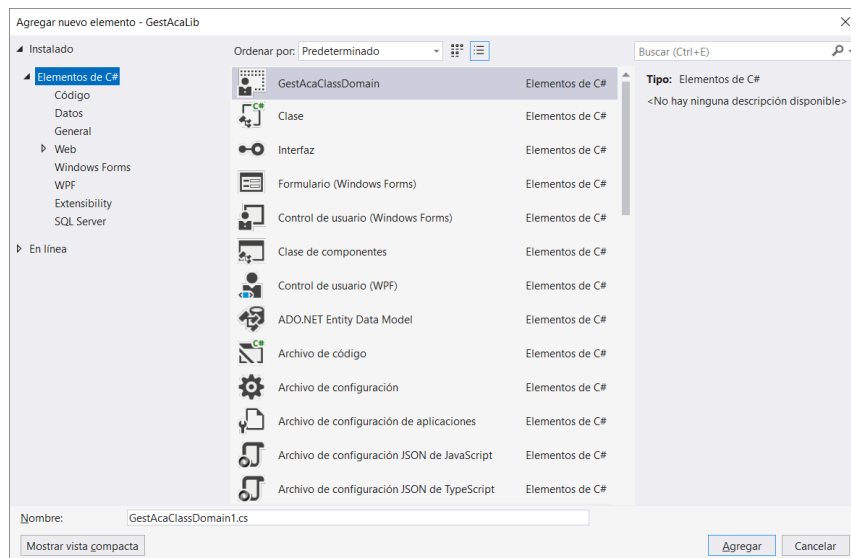


Figure 8. Using the template to create classes

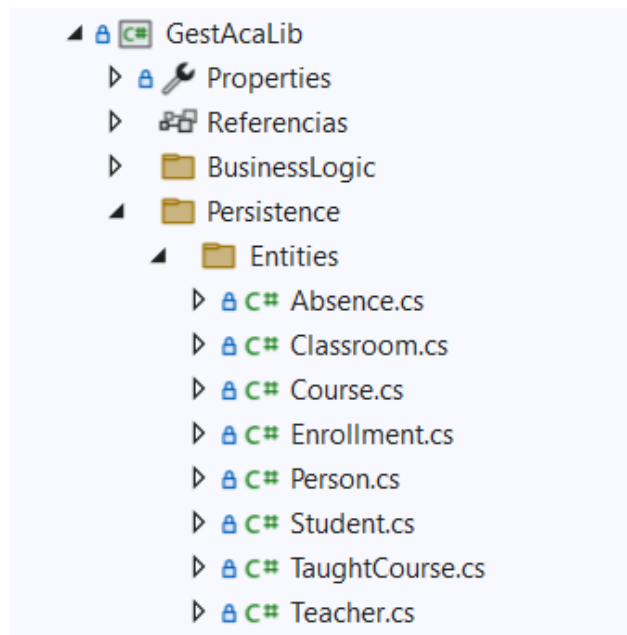


Figure 9. Empty classes inside Persistence.Entities

4.4 Add the classes into BusinessLogic.Entities

As indicated above, we are going to implement the classes using partial classes, so that the code corresponding to the persistence layer will be located in the classes that are inside the *Persistence/Entities* folder, while the rest of the code will be located in *BusinessLogic/Entities*.

The **team leader will copy** all the new empty classes into the folder *BusinessLogic/Entities*. When they perform that operation, they add a new commit and push to the remote repository.

4.5 Add the enumerate type (optional)

If the class diagram had any enumerated type in it, it should be added to the project too. In order to do so, the leader goes into the folder *Persistence/Entities* and, from the context menu, they select the option *Agregar>Nuevo Elemento*. In this case, we need to select the option *Archivo de Código*, and give it a name, such as, for instance, *Authorized.cs*.(see Figure 10).

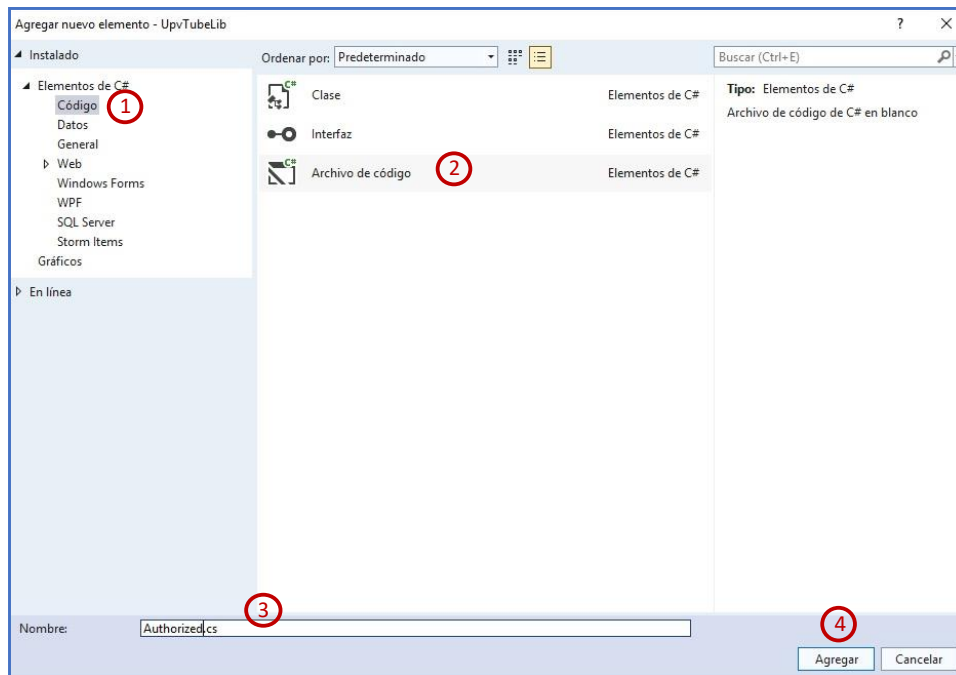


Figure 10 Adding code file for enumerated type

The Product code would be modified to include the allowed values, for example:

```
using System;

namespace GestAca.Entities
{
    public enum Authorized : int
    {
        Yes,
        No,
        Pending,
    }
}
```

With this step, we already have all the necessary elements to begin to complete the code of the classes. Thus, in this moment the solutions must look the same as Figure 12. The team leader saves all the changes, creates a new commit and pushes it. All team members should synchronize their repositories.

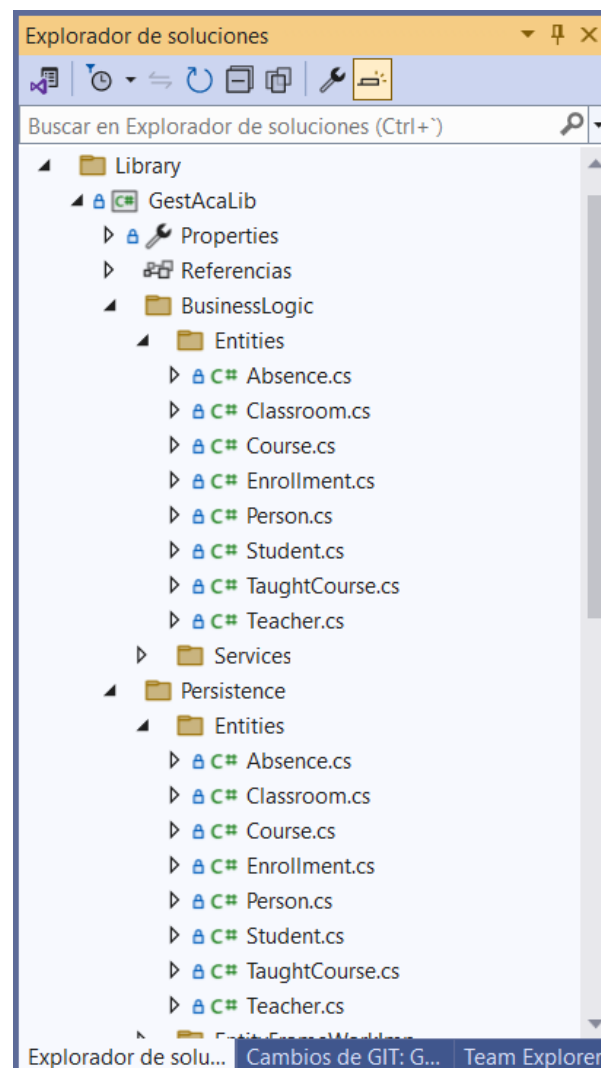


Figure 11. Solution status after creating all empty classes.

5. Programming the persistence aspect of the classes

All members can participate in the implementation from this point on, distributing the work so that each member works independently in each of the files to avoid conflicts. We recommend you commit and synchronize at each finished class.

We are going to write the persistence code in the classes located in the folder *Persistence/Entities*. We will add the properties indicated in the design plus the properties necessary to represent the associations of the diagram in each of the classes. To do it, we will follow the guidelines explained in unit 5. These guidelines explain how we should look at the maximum cardinality of the relationships, to know if an association becomes a property (maximum cardinality of 1) or if it becomes a collection (maximum cardinality of n). In addition, we must look at the navigability restrictions of the design diagram, as not all the relationships are bidirectional.

On the other hand, we must declare the properties of the design model (the **attributes** are modelled as properties in this design model) as **public**. Furthermore, we must declare all the properties which represent **associations** as **public virtual**, because it is a requirement for the implementation of the persistence layer.

EXAMPLE (TarongISW Case Study)

The following code is an example to explain the steps to follow in the implementation of the classes' properties by following the design patterns. In this example, given the class diagram in

Figure 12, we will implement the classes *Contract* and *Temporary*. You will have to implement the classes of this year's case study in a similar way, with each team member implementing some of the classes.

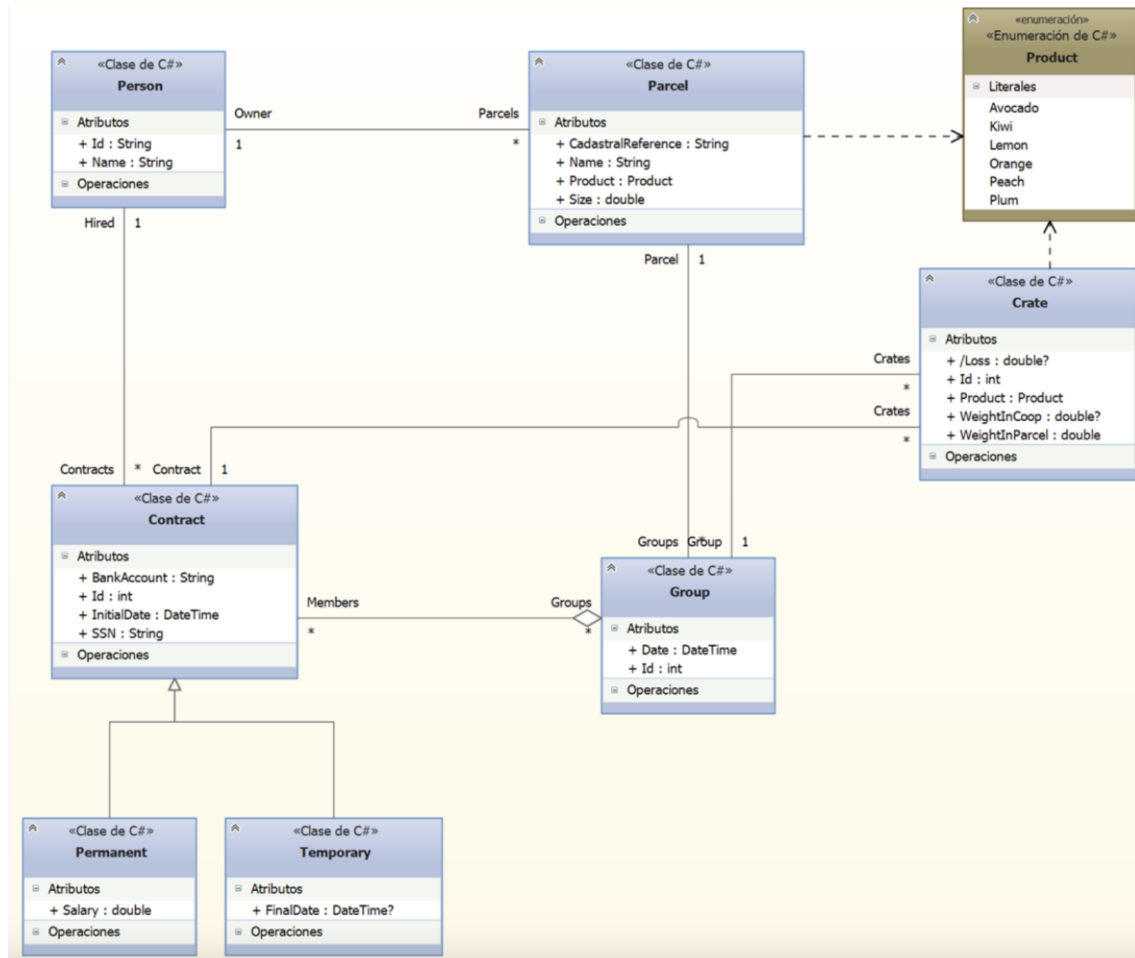


Figure 12: TarongISW class diagram (example)

5.1. *Contract* class implementation

Contract is a generalization which represents the common elements between the classes *Permanent* and *Temporary*. In the code there is a property per attribute in the model. It is related with classes *Group* and *Crate* by means of relationships with maximum cardinality n (*) that are implemented with collections *Groups* and *Crates*, respectively. Observe that the names of the collections are the same as the role names in the model. It is also related with class *Person* by means of an association with cardinality 1, implemented with a property *Hired* of type *Person*.

So, only a property for each one of attributes specified in the model appears in the code (in alphabetical order). Observe that we define first the properties of the class are defined, and then the properties that are derived from the relationships according to the model.

```

namespace TarongISW.Entities
{
    public partial class Contract
    {
        public string BankAccount
        {

```

```

        get;
        set;
    }

    public int Id
    {
        get;
        set;
    }

    public DateTime InitialDate
    {
        get;
        set;
    }

    public string SSN
    {
        get;
        set;
    }

    public virtual Person Hired
    {
        get;
        set;
    }

    public virtual ICollection<Crate> Crates
    {
        get;
        set;
    }

    public virtual ICollection<Group> Groups
    {
        get;
        set;
    }
}

```

5.1 Programming the class *Temporary*

Temporary is a specialization of *Contract*, so we program *Temporary* as a class which inherits from *Contract* (`public partial class Temporary: Contract`). In addition, we add a property for each one of the properties described in the model, with the indicated type.

In this case we only have a property *FinalDate* with “?” after the data type:

`FinalDate: DateTime?`

This allows the value of the attribute whose data type is not an object to admit null values. References to objects can be null, but not values (primitive types and struct). This notation allows

to make them null too¹. In our example, the contract date of completion (FinalDate) is of type DateTime, which is a *struct* type. That attribute may never have a valid date value and may be void until an employee indicates that maintenance has been completed. To allow a null value on that date you define the type as DateTime?. Likewise, if a parameter of this type is to be used in the constructor, it must be defined as DateTime? and it would be possible to pass the null value.

Next, we show the code of class *Temporary*.

```
namespace TarongISW.Entities
{
    public partial class Temporary : Contract
    {
        public DateTime? FinalDate {
            get;
            set;
        }
    }
}
```

To check if a variable defined in this way has a null value, the method HasValue is used, and to access to the content the method Value, as the following example illustrates:

```
DateTime? FinalDate = null;

if (FinalDate.HasValue)
    Console.WriteLine(FinalDate.Value);
else
    Console.WriteLine("The contract end date is not valid.");
```

(This code is not to be included at this stage; it will have to be included if required during the business logic layer implementation).

6. Programming the constructors of the classes in *BusinessLogic*

The code of the constructors must be programmed in the files of the classes located at BusinessLogic/Entities. Again, all team members can collaborate in the development, but the work must be distributed so that **each one works in a different class**. This is the best way to **not generate conflicts**. It is recommended to create a commit and synchronize when you finish a class.

Each class will have **two constructors**, one without parameters and other with all the necessary parameters. The constructor without parameters only initializes the collections of the class by creating empty collections. On the other hand, the second constructor with parameters, besides initializing the collections, must receive the value of all the properties that need to be initialized. We will not add the **Id attribute** to the constructor if it is of **the Integer type**. We will use **the Entity Framework (EF)** for the information persistence and **EF initializes the value to the ID attributes of numerical type automatically** when the object is persisted for the first time. Therefore, we should not give value to those values manually, since EF will assign that value when the object persists for the first time.

¹ <https://docs.microsoft.com/es-es/dotnet/csharp/programming-guide/nullable-types/using-nullable-types>

The **order of the parameters** of the constructor must be: **firstly**, its **own properties in alphabetical order**; **secondly**, the objects needed because of the **associations**, also in **alphabetical order**.

When the **class inherits** from another one, the order of the parameters of the constructor must be: **firstly**, the **parameters** from the **base** class, in **alphabetical order**; **secondly**, the set of the **own parameters** (also alphabetically ordered); and **lastly**, the value of the properties added because of the **associations**.

It should be remembered that in the case of classes that have "nullable" attributes (i.e with "?"), the constructor with parameters must also declare **the type of data with "?"** so that there is concordance of types.

As an example, the code of the same example classes used in the previous point will be provided below: Contract and Temporary.

6.1 Programming the constructors of the class *Contract*

Contract has two collections, Crates and Groups, that must be initialized in the constructor without parameters. The constructor with parameters receives all the necessary values to initialize the properties, except the Id, because its type is int, as has been explained above. Note that they are received in **alphabetical order**. The minimum cardinality for the relationships with *Group* and *Crate* is 0 so it is not necessary to insert any element initially in the collections nor pass any parameter in the constructor. That is, it is not necessary to initially assign a group or a crate to a contract. The cardinality of the relationship with *Person* is 1 and, thus, when a contract is created it must be related with a person². To do so we must pass the corresponding parameter hired in the constructor. The corresponding code would be as follows:

² In case we had minimum cardinality different than 0 on both sides of the relationship, it would be necessary to relax one of the constructors as it is explained in chapter 5.

```

namespace TarongISW.Entities
{
    public partial class Contract
    {
        public Contract()
        {
            // Colecciones
            Crates = new List<Crate>();
            Groups = new List<Group>();
        }

        public Contract(string bankAccount, DateTime initialDate, string SSN, Person
hired)
        {
            // No value for Id because Entity Framework will assign it
            this.BankAccount = bankAccount;
            this.InitialDate = initialDate;
            this.SSN = SSN;

            // Relationships with cardinality 1
            Hired = hired;

            // Collections
            Crates = new List<Crate>();
            Groups = new List<Group>();
        }
    }
}

```

6.2 Programming the constructors of the class Temporary

Temporary inherits from Person, so its constructors call the constructors of its base class to initialize the inherit properties, using the sentence `base()`. In addition, the constructor with parameters receives the values of the inherited properties first, in alphabetical order. Then, it receives its own values, also in alphabetical order.

If there are attributes whose value is not known when the object is to be created, they will not be passed in the constructor. **Those attributes are marked with a “?”** at the end of their type in the design diagram and should be initialized as null in the constructor. In the class Temporary, the FinalDate is unknown when the object is to be initialized, so it will be set to null in the constructor and will not be passed in the constructor.

```

namespace TarongISW.Entities
{
    public partial class Temporary
    {
        public Temporary() {
        }

        public Temporary(string bankAccount, DateTime initialDate, string SSN,
Person hired):base(bankAccount, initialDate, SSN, hired)
        {
            this.FinalDate = null;
        }
    }
}

```

6.3 Reuse code using this

In classes with collections, it is necessary to initialise them both in the constructor without parameters and in the constructor with parameters. Therefore, it is possible to reuse code by calling the constructor without parameters from the constructor with parameters, as shown in the following example. Note that this is not possible when there is inheritance, since we must call the base method.

```
namespace TarongISW.Entities
{
    public partial class Contract
    {
        public Contract()
        {
            // Collections
            Crates = new List<Crate>();
            Groups = new List<Group>();
        }

        public Contract(string bankAccount, DateTime initialDate, string SSN, Person
        hired): this()
        {
            // No value for Id because Entity Framework will assign it
            this.BankAccount = bankAccount;
            this.InitialDate = initialDate;
            this.SSN = SSN;

            // Relationships with cardinality 1
            Hired = hired;
        }
    }
}
```

7. Work to be delivered

By the end of the two sessions, the groups must have completed the implementation of all classes in the model, as well as the enumerated types if there were any. The properties of the classes must be declared in the files located in *Persistence/Entities*. The constructors must be implemented in *BusinessLogic/Entities*. In addition, for each class we must have two constructors:

- One without parameters, which in case the class has collections initializes them.
- Another with parameters, where these parameters follow the order explained in section 6. In addition, these constructors also initialize the collections it contains.

At the end of the next two sessions, the code of each team should pass some tests, so if the above **guidelines are not followed**, the code **will not pass the tests and you will not be able to carry on to persistence implementation**.