UNIVERSIDAD POLITECNICA DE VALENCIA

Laboratory

Lab Sessions 2 and 3

OO Design. Logic Layer. Constructors and Classes Design.

Ingeniería del Software

ETS Ingeniería Informática

DSIC – UPV

2019-2020 Course

1. Goal

In the following two session, 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 3 and 4 of this bulletin.

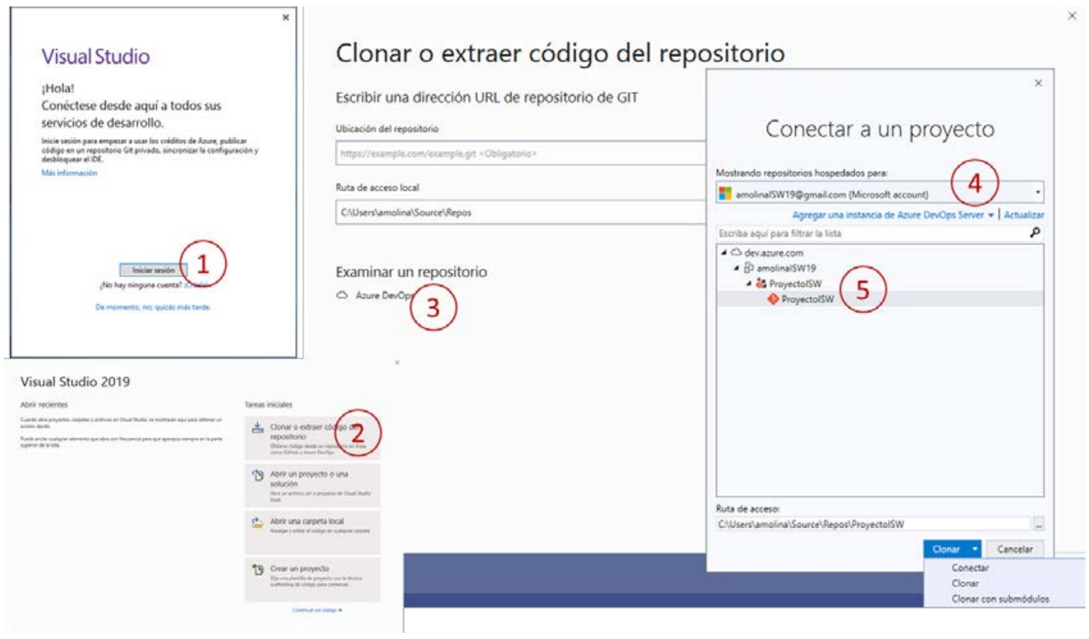
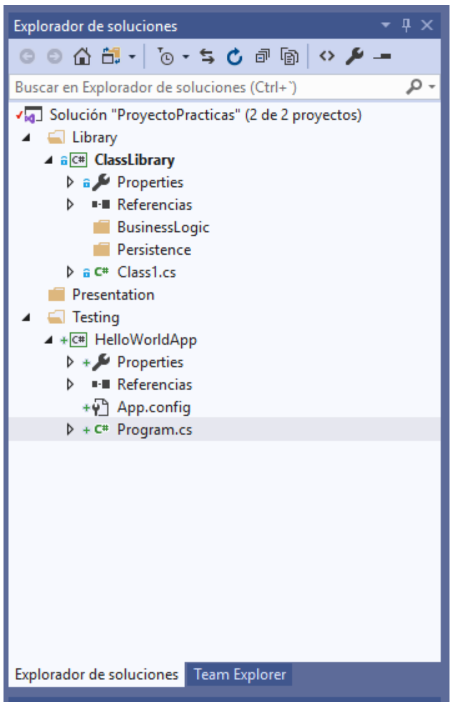

Visual Studio
¡Hola!
Conéctese desde aquí a todos sus servicios de desarrollo.
Inicie sesión para empezar a usar los créditos de Azure, publicar código en un repositorio Git privado, sincronizar la configuración y desbloquear el IDE.
Más información
Iniciar sesión
1
¿No hay ninguna cuenta?
De momento, no; quizás más tarde.
Clonar o extraer código del repositorio
Escribir una dirección URL de repositorio de GIT
Ubicación del repositorio
https://example.com/example.git <Obligatorio>
Ruta de acceso local
C:\Users\amolina\Source\Repos
Examinar un repositorio
Azure DevOps
3
Visual Studio 2019
Abrir recientes
Tareas iniciales
Clonar o extraer código del repositorio
2
Abrir un proyecto o una solución
Abrir una carpeta local
Crear un proyecto
Conectar a un proyecto
Mostrando repositorios hospedados para:
amolinaISW19@gmail.com (Microsoft account)
4
Agregar una instancia de Azure DevOps Server | Actualizar
Escriba aquí para filtrar la lista
dev.azure.com
amolinaISW19
ProyectoISW
ProyectoISW
5
Ruta de acceso:
C:\Users\amolina\Source\Repos\ProyectoISW
Clonar
Cancelar
Conectar
Clonar
Clonar con submódulos

Figure 1. Steps to connect and clone the team project

3. Initial configuration of the solution

This step should be carry 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 library of classes 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. First, check that your solution looks like the one shown in Figure 2. If it doesn't, the team leader should resume seminar 2 and lab bulletin 1 to complete the missing steps in order to get the required project structure.



```
Explorador de soluciones
Buscar en Explorador de soluciones (Ctrl+´)
Solución "ProyectoPracticas" (2 de 2 proyectos)
  Library
    ClassLibrary
      Properties
      Referencias
      BusinessLogic
      Persistence
      Class1.cs
  Presentation
  Testing
    HelloWorldApp
      Properties
      Referencias
      App.config
      Program.cs
Explorador de soluciones | Team Explorer
```

Figure 2. Initial configuration of the solution

After open the solution, the team leader have to 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`). Afterwards, you must perform the same step for the folder *Persistence*, by also adding a subfolder *Entities.* In this moment, the team leader must commit the changes, by pressing the button `Cambios` from the Team Explorer tab. Remember that the system requires a text message which describes the changes made. Then, it is possible to click on the Confirmar button.

Next, the leader should push the commit to the remote repository by selecting the option from the `Team Explorer` tab: `Sincronizar> Insertar.` All the team member should synchronize their repositories, to check that the changes have been uploaded properly (`Sincronizar>Extraer`).

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 have to 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 `EcoScooterClassLibrary` (see Figure 3)

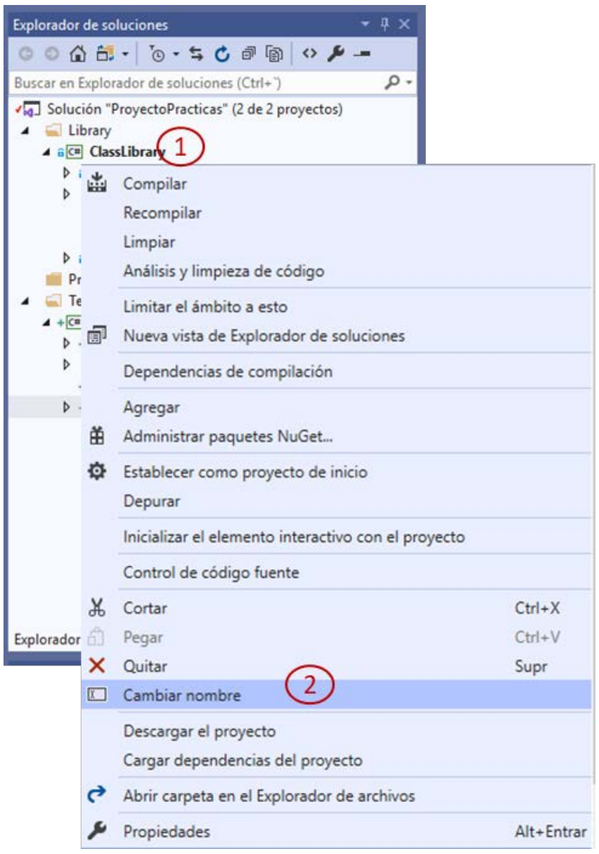
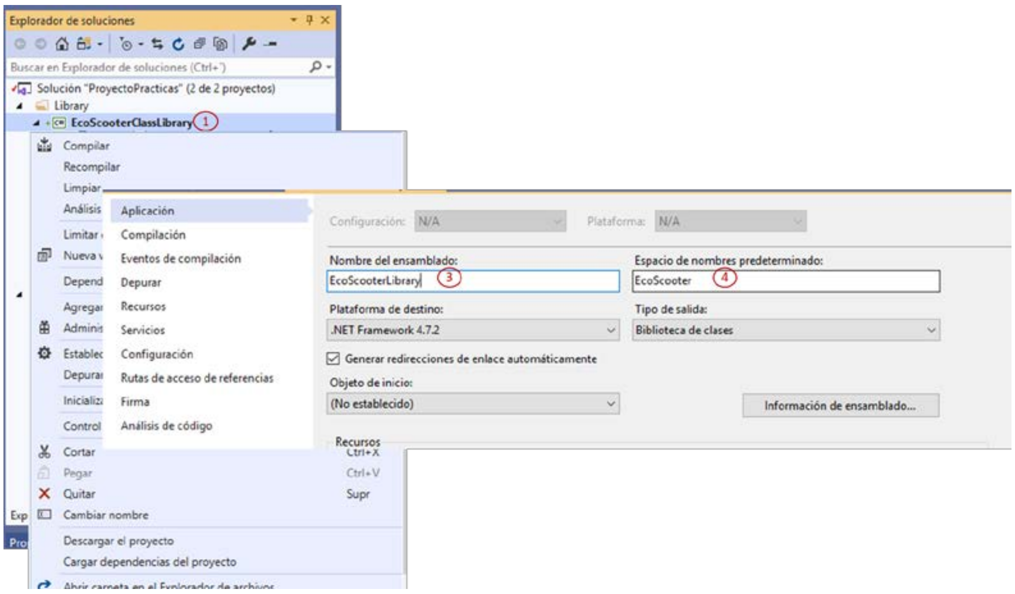

Explorador de soluciones
Buscar en Explorador de soluciones (Ctrl+´)
Solución "ProyectoPracticas" (2 de 2 proyectos)
Library
ClassLibrary
1
Compilar
Recompilar
Limpiar
Análisis y limpieza de código
Limitar el ámbito a esto
Nueva vista de Explorador de soluciones
Dependencias de compilación
Agregar
Administrar paquetes NuGet...
Establecer como proyecto de inicio
Depurar
Inicializar el elemento interactivo con el proyecto
Control de código fuente
Cortar Ctrl+X
Pegar Ctrl+V
Quitar Supr
Cambiar nombre
2
Descargar el proyecto
Cargar dependencias del proyecto
Abrir carpeta en el Explorador de archivos
Propiedades Alt+Entrar

Figure 4. Change the name of the class library

In the next step, the team leader should configure the namespace and the assembly name of the library. First, the leader should select the library again, and click on the right button of the mouse. From context menu, she selects 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: `EcoScooterLibrary.` Furthermore, the leader should modify the field `Espacio de nombres prederterminado` to the value `EcoScooter.` Figure 4 summarizes the steps.



Configuración: N/A
Plataforma: N/A
Nombre del ensamblado: EcoScooterLibrary
Espacio de nombres predeterminado: EcoScooter
Plataforma de destino: .NET Framework 4.7.2
Tipo de salida: Biblioteca de clases
☑ Generar redirecciones de enlace automáticamente
Objeto de inicio: (No establecido)

Figure 3. Configure the namespace of the library

Finally, the responsible selects the default class Class1.cs and deletes it. Then, she commits and pushes the changes.

4. Add the classes of the design model

The design model of the EcoScooter management application you must implement is displayed in Figure 5. The model contains eleven classes and an enumerate type named `ScooterState.` 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. The relationships of the analysis model have been relaxed, restricting the navigation in some of them. These new navigation restriction should been maintained.

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.

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). From the context menu, she selects the option: `Agregar > Clase.` Give it the name `Person.` In the editor, the leader will change the namespace and the definition of the class, so that the content is like the following code:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace EcoScooter.Domain
{
    public partial class Person
    {
    }
}
```

After, commit the changes again (`Confirmar` option) and push them to the remote repository. The team mates can pull these changes again and check them.

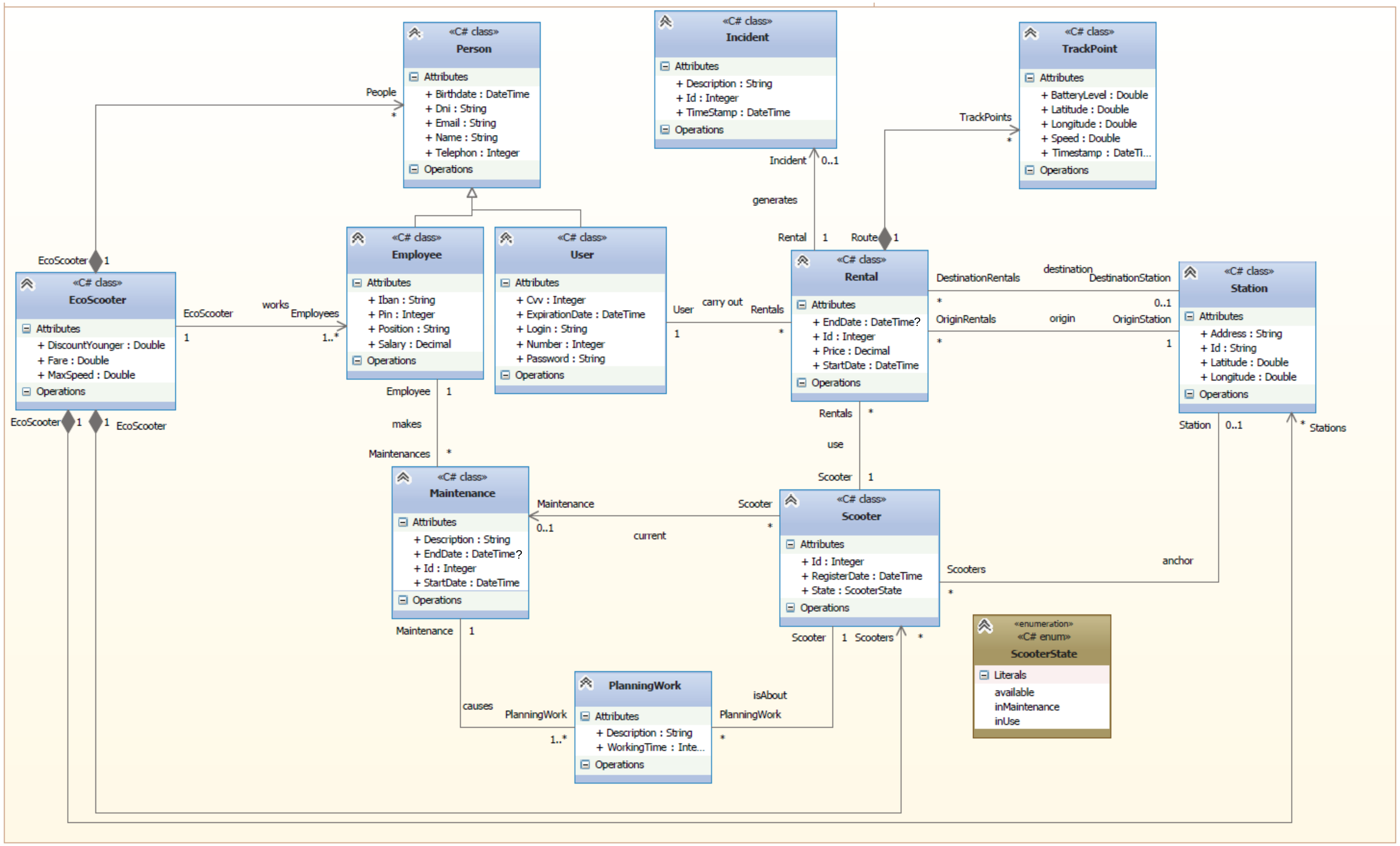


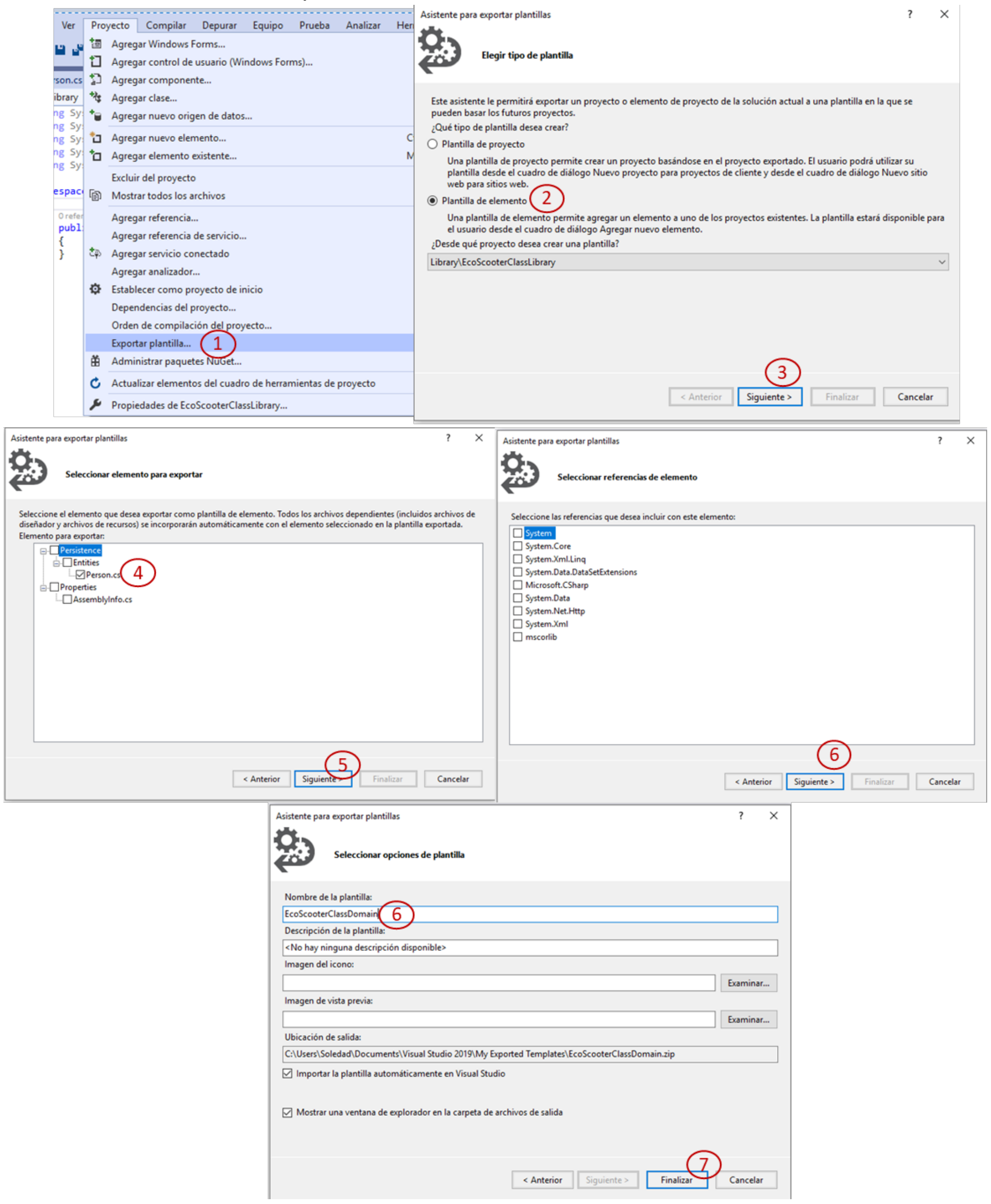
Figure 5. Design model of the EcoScooter management application

4.2 Creation of a class template

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 `Person.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 `EcoScooterClassDomain` and then, clicks on `Finalizar`.

In this moment it is necessary to close and restart Visual Studio to load the new template. Figure 6 shows an overview of these steps.


Ver
Proyecto
Compilar
Depurar
Equipo
Prueba
Analizar
Agregar Windows Forms...
Agregar control de usuario (Windows Forms)...
Agregar componente...
Agregar clase...
Agregar nuevo origen de datos...
Agregar nuevo elemento...
Agregar elemento existente...
Excluir del proyecto
Mostrar todos los archivos
Agregar referencia...
Agregar referencia de servicio...
Agregar servicio conectado
Agregar analizador...
Establecer como proyecto de inicio
Dependencias del proyecto...
Orden de compilación del proyecto...
Exportar plantilla...
1
Administrar paquetes NuGet...
Actualizar elementos del cuadro de herramientas de proyecto
Propiedades de EcoScooterClassLibrary...
2
3
4
5
6
6
7
< Anterior
Siguiente >
Finalizar
Cancelar
Examinar...

Asistente para exportar plantillas

Elegir tipo de plantilla

Este asistente le permitirá exportar un proyecto o elemento de proyecto de la solución actual a una plantilla en la que se pueden basar los futuros proyectos.

¿Qué tipo de plantilla desea crear?

○ Plantilla de proyecto

Una plantilla de proyecto permite crear un proyecto basándose en el proyecto exportado. El usuario podrá utilizar su plantilla desde el cuadro de diálogo Nuevo proyecto para proyectos de cliente y desde el cuadro de diálogo Nuevo sitio web para sitios web.

● Plantilla de elemento

Una plantilla de elemento permite agregar un elemento a uno de los proyectos existentes. La plantilla estará disponible para el usuario desde el cuadro de diálogo Agregar nuevo elemento.

¿Desde qué proyecto desea crear una plantilla?

Library\EcoScooterClassLibrary

Asistente para exportar plantillas

Seleccionar elemento para exportar

Seleccione el elemento que desea exportar como plantilla de elemento. Todos los archivos dependientes (incluidos archivos de diseñador y archivos de recursos) se incorporarán automáticamente con el elemento seleccionado en la plantilla exportada.

Elemento para exportar:

- ☐ Persistence
  - ☐ Entities
    - ☑ Person.cs
- ☐ Properties
  - ☐ AssemblyInfo.cs

Asistente para exportar plantillas

Seleccionar referencias de elemento

Seleccione las referencias que desea incluir con este elemento:

- ☐ System
- ☐ System.Core
- ☐ System.Xml.Linq
- ☐ System.Data.DataSetExtensions
- ☐ Microsoft.CSharp
- ☐ System.Data
- ☐ System.Net.Http
- ☐ System.Xml
- ☐ mscorlib

Asistente para exportar plantillas

Seleccionar opciones de plantilla

Nombre de la plantilla: EcoScooterClassDomain

Descripción de la plantilla: <No hay ninguna descripción disponible>

Imagen del icono:

Imagen de vista previa:

Ubicación de salida: C:\Users\Soledad\Documents\Visual Studio 2019\My Exported Templates\EcoScooterClassDomain.zip

☑ Importar la plantilla automáticamente en Visual Studio

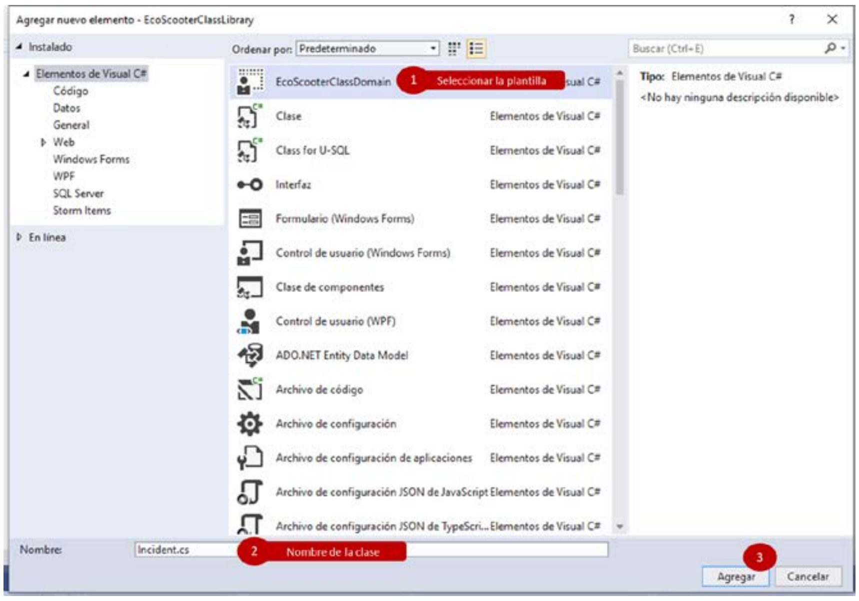
☑ Mostrar una ventana de explorador en la carpeta de archivos de salida

Figure 6. Creating a template for domain classes

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

After opening again 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:

- Goes to the solution folder *Persistence/Entities*.
- Opens the context menu by clicking the right button of the mouse.
- Selects `Agregar>Nuevo Elemento`
- Chooses the option:`EcoScooterClassDomain.` In the field name, indicates the name of the new class (see Figure 7)
- Clicks on `Agregar`



Agregar nuevo elemento - EcoScooterClassLibrary

Instalado
- Elementos de Visual C#
  - Código
  - Datos
  - General
  - Web
  - Windows Forms
  - WPF
  - SQL Server
  - Storm Items
- En línea

Ordenar por: Predeterminado

Buscar (Ctrl+E)

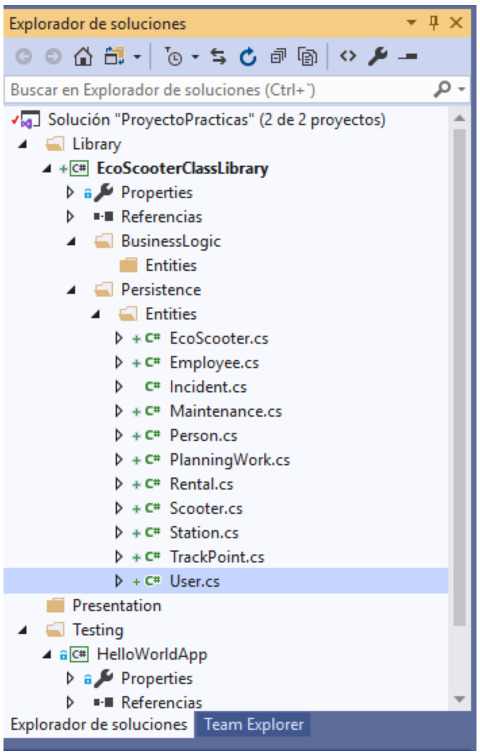
EcoScooterClassDomain	Elementos de Visual C#
Clase	Elementos de Visual C#
Class for U-SQL	Elementos de Visual C#
Interfaz	Elementos de Visual C#
Formulario (Windows Forms)	Elementos de Visual C#
Control de usuario (Windows Forms)	Elementos de Visual C#
Clase de componentes	Elementos de Visual C#
Control de usuario (WPF)	Elementos de Visual C#
ADO.NET Entity Data Model	Elementos de Visual C#
Archivo de código	Elementos de Visual C#
Archivo de configuración	Elementos de Visual C#
Archivo de configuración de aplicaciones	Elementos de Visual C#
Archivo de configuración JSON de JavaScript	Elementos de Visual C#
Archivo de configuración JSON de TypeScri...	Elementos de Visual C#

Tipo: Elementos de Visual C#
<No hay ninguna descripción disponible>

Nombre: Incident.cs

Figure 7. Using the template to create classes

As a result, you should have got the same content that can be observed in the Figure 8. Commit changes and synchronize.



Explorador de soluciones

Buscar en Explorador de soluciones (Ctrl+´)

```
Solución "ProyectoPracticas" (2 de 2 proyectos)
  Library
    EcoScooterClassLibrary
      Properties
      Referencias
      BusinessLogic
        Entities
      Persistence
        Entities
          EcoScooter.cs
          Employee.cs
          Incident.cs
          Maintenance.cs
          Person.cs
          PlanningWork.cs
          Rental.cs
          Scooter.cs
          Station.cs
          TrackPoint.cs
          User.cs
  Presentation
  Testing
    HelloWorldApp
      Properties
      Referencias
```

Explorador de soluciones | Team Explorer

Figure 8. 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 *Persistence/Entities.* So, she only must select all the classes, and from the context menu, selects the option copy. Afterwards, inside *BusinessLogic/Entities,* she selects paste from the context menu. Then, the leader adds a new commit and pushes it to the remote repository.

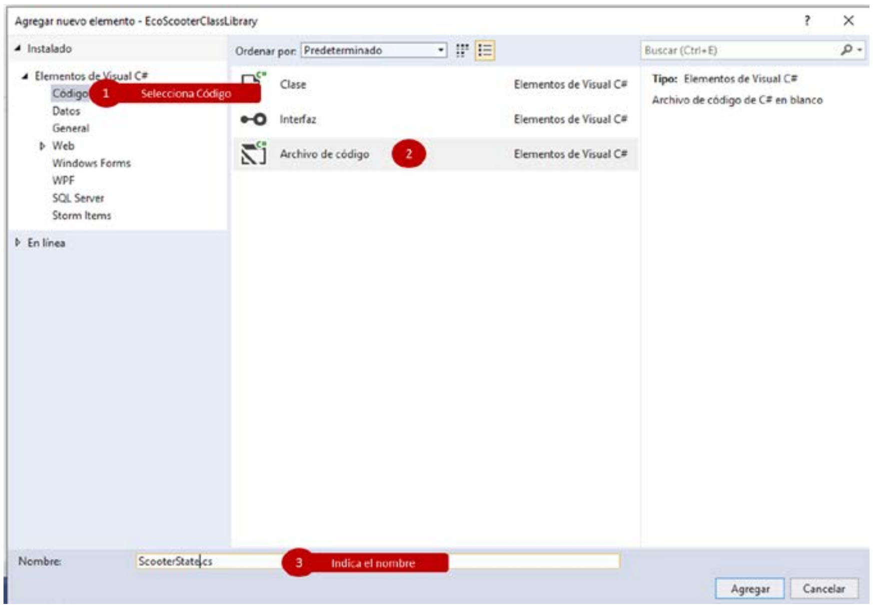
4.5 Add the enumerate type

In order to create the enumerate type, the leader goes to the folder *Persistence/Entities and,* from the context menu, she selects the option `Agregar>Nuevo Elemento`. In this case, we need to select the option `Archivo de código.` Name the new file as `ScooterState.cs`.(see Figure 9).

The leader should edit the code to be as the following:

```
namespace EcoScooter.Domain
{

    public enum ScooterState : int
    {
        inUse,
        available,
        inMaintenance
    }
}
```


Agregar nuevo elemento - EcoScooterClassLibrary

Instalado
Elementos de Visual C#
Código
Datos
General
Web
Windows Forms
WPF
SQL Server
Storm Items
En línea

Ordenar por: Predeterminado

Clase	Elementos de Visual C#
Interfaz	Elementos de Visual C#
Archivo de código	Elementos de Visual C#

Buscar (Ctrl+E)
Tipo: Elementos de Visual C#
Archivo de código de C# en blanco

Nombre: ScooterState.cs

Figure 10. Adding the enumerate type file

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 10 . The team leader saves all the changes, creates a new commit and pushes it. All team members should synchronize their repositories.

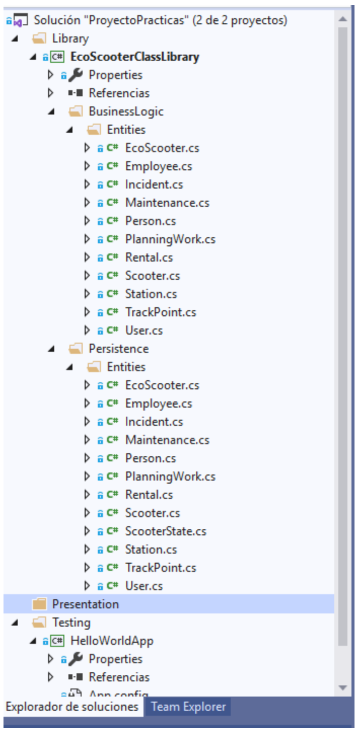

```
Solución "ProyectoPracticas" (2 de 2 proyectos)
  Library
    EcoScooterClassLibrary
      Properties
      Referencias
      BusinessLogic
        Entities
          EcoScooter.cs
          Employee.cs
          Incident.cs
          Maintenance.cs
          Person.cs
          PlanningWork.cs
          Rental.cs
          Scooter.cs
          Station.cs
          TrackPoint.cs
          User.cs
      Persistence
        Entities
          EcoScooter.cs
          Employee.cs
          Incident.cs
          Maintenance.cs
          Person.cs
          PlanningWork.cs
          Rental.cs
          Scooter.cs
          ScooterState.cs
          Station.cs
          TrackPoint.cs
          User.cs
  Presentation
  Testing
    HelloWorldApp
      Properties
      Referencias
      App.config
```

Figure 9.Solution status after creating all empty classes and the enumerated type.

o

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 to commit and synchronize at each finished class.

We are going to program the persistence aspect in the classes located in the folder *Persistence/Entities.* We are going to 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 maxim 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 as **public** the properties of the design model (the **attributes** are modelled as properties in this design model). However, we must declare as **public virtual** all the properties which represent the **associations**, because it is a requisite of the persistence layer.

As an example, we provide you the complete code of three of the classes of the model: `Person`, `Employee` and `Maintenance.` Also, we explain all the design guidelines followed in the implementation of each one of them. **The rest of the classes must be programmed by the team**.

5.1 Programing the class *Person*

Person is a generalization which represents the common elements between the classes *Employee* and *User.* There is an association between *Person* and *EcoScooter,* but is not navigable from *Person,* thus *EcoScooter* must not be present in its design. So, only a property for each one of properties described in the model appears in the code (in alphabetical order).

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace EcoScooter.Entities
{
    public partial class Person
    {
        public DateTime Birthdate
        {
            get;
            set;
        }
        public string Dni
        {
            get;
            set;
        }
        public string Name
        {
            get;
            set;
        }
        public string Email
        {
º                get;
            set;
        }
        public int Telephon
        {
            get;
            set;
        }
    }
}
```

5.2 Programming the class Employee

Empoyee is a specialization of *Person*, so we program *Employee* as a class which inherits from *Person* (`public partial class Employee : Person`). In addition, we add a property for each one of the properties described in the model, with the indicate type. Regarding its associations, *Employee* is related with *EcoScooter* and with *Maintenance.* From Employee is not possible to achieve EcoScooter, because the navigation is restricted, thus it does not appear in the design.

On the contrary, the relationship between *Employee* and *Maintenance* is navigable from *Employee* and the maximum cardinalty is N (* according to the diagram notation), so we add a collection of type *Maintenance* in *Employee* named *Maintenances.*

First, we add the specific properties of the class, and after them, the properties which derive from the associations of the class with other entities of the model. In the following code you can observe the final result.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace EcoScooter. Entities
{
    public partial class Employee : Person
    {
        public string Iban
        {
            get;
            set;
        }

        public int Pin
        {
            get;
            set;
        }

        public string Position
        {
            get;
            set;
        }

        public Decimal Salary
        {
            get;
            set;
        }

         /*Associations*/
        public virtual ICollection<Maintenance> Maintenances
        {
            get;
            set;
        }

    }
}
```

5.4 Programming the class *Maintenance*

First, we add the properties described in the model, with the specified type. In this case there is a property with a data type that ends with "?:"

```
EndDate: 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 also null.[1]. In our example, the date of completion (`EndDate`) is of type `DateTime`, that 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 terminated. 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

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? EndDate = null;


if (EndDate.HasValue)
    Console.WriteLine(EndDate.Value);
else
    Console.WriteLine("La fecha de cancelación no tiene valor.");
```

With respect to associations, the Maintenance class is related to three classes: *Employee*, *Scooter* and *PlanningWork*. The association between Maintenance and Scooter is not navigable from *Maintenance*, so we should not add anything to its design. The relationship between *Employee* and *Maintenance* is navigable from this class, so we must observe the maximum cardinality, which in this case is 1. Therefore, we must add a property of type *Employee* in the design of the class. Finally, the relationship between *Maintenance* and *PlanningWork* is navigable, having a maximum cardinality of N. Therefore, we must add a *PlanningWork* type collection to the class design. Next, you can see the resulting code.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace EcoScooter.Entities
{
    public partial class Maintenance
    {

        public string Description
        {
            get;
            set;
        }
        public DateTime? EndDate
        {
            get;
            set;
        }
        public int Id
        {
            get;
```

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

```
            set;
        }
        public DateTime StartDate
        {
            get;
            set;
        }
        /*Relaciones*/
        public virtual Employee Employee
        {
            get;
            set;
        }
        public virtual ICollection<PlanningWork> PlanningWork
        {
            get;
            set;
        }

    }
}
```

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, in addition to initializing the collections. The **order of the parameters** of the constructor must be: **first**, the **own properties in alphabetical order**; **secondly**, the objects it receives to be assigned to the properties added 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: **first**, 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: `Person`, `Employee` and `Maintenance`,.

6.1 Programming the constructors of thec class *Person*

`Person` has no collection, so its constructor without parameters has no sentence. The constructor with parameters receives all the necessary values to instantiate its properties. Note that they are received in **alphabetical order**.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace EcoScooter. Entities
{
    public partial class Person
    {
        public Person()
        {
        }
        public Person(DateTime birthDate, String dni, String email, String name, 
int telephon)
        {
            Birthdate = birthDate;
            Dni = dni;
            Email = email;
            Name = name;
            Telephon = telephon;
        }
    }
}
```

6.2 Programming the constructors of the class *Employee*

`Employee` 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 inherit properties in first place, in alphabetical order. Secondly, it receives the own values, also alphabetically.

With regard to associations of this class, their relationship with `EcoScooter` is not navigable, so they do not have to handle any object of this class. Regarding its relationship with Maintenance, we must look at the minimum cardinality. In this case it is zero, so we do not have to pass any object of the `Maintenance` class as a parameter in the constructor.

Both in the constructor without parameters and in the constructor with parameters, we must initialize the `Maintenances` collection, creating an empty list.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace EcoScooter.Entities
{
    public partial class Employee : Person
    {
        public Employee() : base ()
        {
            Maintenances = new List<Maintenance>();
        }

        public Employee(DateTime birthDate, String dni, String email, String 
name, int telephon, String iban, int pin, String position, Decimal salary ) : 
base(birthDate, dni, email, name, telephon)
        {
            Iban = iban;
            Pin = pin;
            Position = position;
            Salary = salary;

            Maintenances = new List<Maintenance>();
        }
    }
}
```

6.3 Programming the constructors of the class *Maintenance*

`Maintenance` has four properties to initialize, in addition to a collection of the `PlanningWork` class. Thus, the constructor without parameters must initialize this collection to an empty list. The constructor with parameters must also perform this initialization, so it calls its constructor without parameters in order to avoid duplicating code (using `this()`). With respect to the constructor with parameters, it receives the four values corresponding to their properties, again in alphabetical order. As the `EndDate` property can receive a null value in the creation of the object, so the type of the parameter in the constructor header must be `DateTime ?.`

With regard to its association, we look at its minimum cardinality, being 1 to 1. Therefore, we have to relax at one of the ends of the relationship, and we have decided to relax on this side (so you must ensure by code the consistency of the information), so that no object of the `PlanningWork` class will be passed in the constructor.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace EcoScooter. Entities
{
    public partial class Maintenance
    {
        public Maintenance()
        {
            PlanningWork = new List<PlanningWork>();
        }
        public Maintenance(String description, DateTime? endDate, int id,
DateTime startDate) : this()
        {
            Description = description;
            endDate = EndDate;
            Id = id;
            StartDate = startDate;
        }
    }
}
```

7. Work to be delivered

By the end of the two sessions, the groups must have completed the implementation of the 11 classes of the model, as well as the type listed. The properties of the classes must be programmed in the files located in *Persistence/Entities*. The constructors must be programmed 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.
- Other 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 following two sessions, the code of each team should pass some tests, so if the above **guidelines are not follow**, the code **will no pass the tests**.