UNIVERSIDAD POLITECNICA DE VALENCIA

Prácticas

Boletín Prácticas 2 y 3

Diseño OO. Capa lógica. Diseño de clases y constructores

Ingeniería del Software
ETS Ingeniería Informática
DSIC – UPV

Curso 2020-2021

1. Objetivo

El objetivo de las próximas dos sesiones de laboratorio es obtener el código inicial de la capa lógica de la aplicación del caso de estudio. Así, a partir el diagrama de diseño que se proporcionará, los alumnos deberán implementar todas las clases indicadas empleando el lenguaje C#.

2. Conectarse al proyecto y recuperar el repositorio

Cada miembro del equipo puede conectarse desde Visual Studio al proyecto de *Azure DevOps*, para clonar el repositorio remoto en el repositorio local, en el caso de iniciar sesión en los equipos del laboratorio. Si estamos en un equipo privado en el que ya se clonó el repositorio previamente, solo será necesario sincronizarse para obtener los últimos cambios realizados. La Figura 1 resume los pasos a seguir para conectarse y clonar el repositorio (en caso de dudas, consultar el seminario 2 o el manual de la práctica 1).

Los apartados 3 y 4 de este manual realizarlos solo un miembro del equipo (por ejemplo, el responsable).

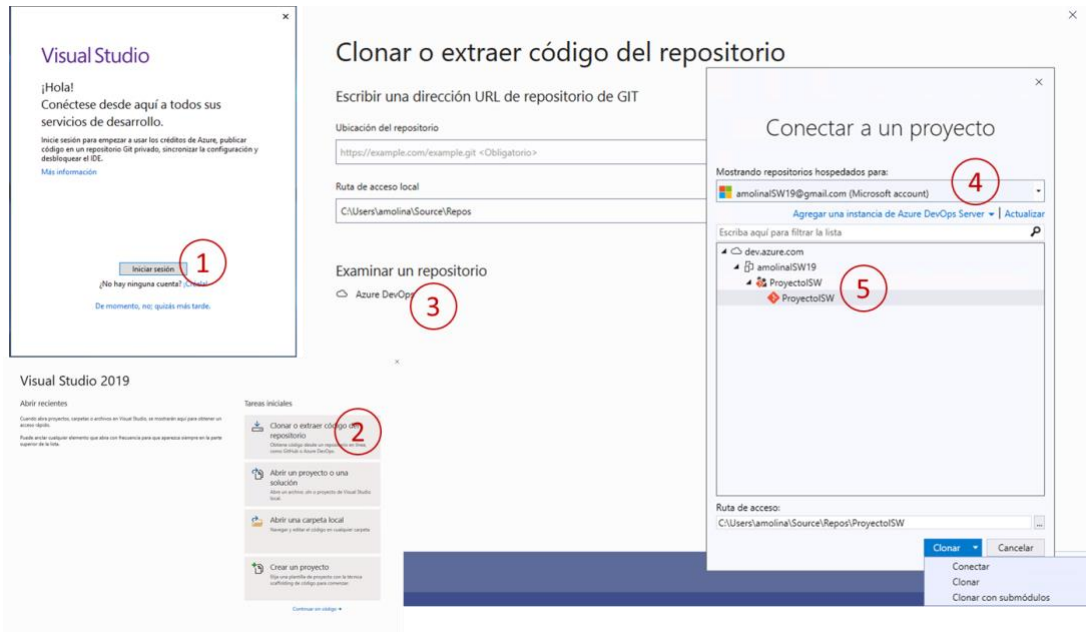

Figura 1. Pasos a seguir para conectarse y clonar el proyecto de equipo

3. Configuración inicial de la solución

Este paso lo debe hacer únicamente el responsable del equipo, a fin de evitar conflictos innecesarios en el repositorio.

Como ya se ha estudiado en las clases de teoría y seminario, la aplicación a desarrollar va a tener tres capas: Presentación, Lógica de negocio y Persistencia. En las sesiones anteriores de prácticas y seminario, ya se incorporó al proyecto una librería de clases, que además preparamos con tres *carpetas de solución* para estructurar adecuadamente nuestro código: *Library*, *Presentation* y *Testing*. A su vez, dentro de la carpeta *Library* añadimos otras dos subcarpetas denominadas *BusinessLogic* y *Persistence.* Durante esta sesión, comenzaremos a añadir código en estas carpetas. En primer lugar, comprobar que vuestra solución tiene el mismo aspecto que el indicado en la Figura 2. De no ser así, el responsable del equipo debe retomar el seminario 2 y el boletín de la práctica 1 para completar los pasos que falten a fin de obtener la estructura de proyecto requerida.

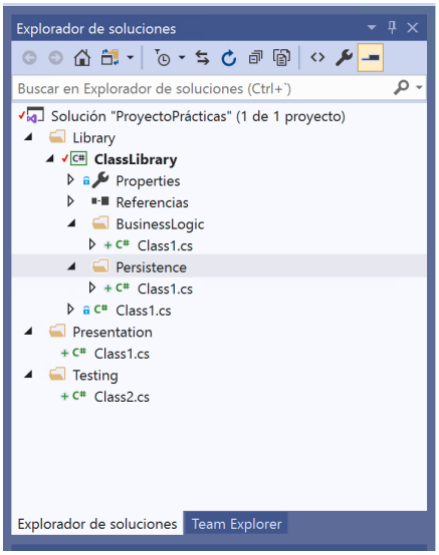

Figura 2. Configuración inicial de la solución del proyecto

Tras abrir la solución, el responsable debe crear una carpeta de soluciones denominada *Entities* dentro de la carpeta *BusinessLogic* (a través de la opción `Agregar> Nueva Carpeta` del menú contextual que se abre al pulsar el botón derecho del ratón). Además, añadiremos una clase vacía para evitar problemas con git.

Después, debe realizar el mismo paso para la carpeta *Persistence*, añadiendo también una subcarpeta *Entities* y una clase vacía dentro.

En este momento, el responsable debe realizar un *commit* con los cambios, pulsando el botón `Cambios` del `Team Explorer`. Para ello debe de proporcionar el mensaje de texto que describe el conjunto de cambios y pulsar el botón `Confirmar`. Después, debe subir los cambios al repositorio remoto seleccionando del `Team Explorer Sincronizar> Insertar`. El resto de miembros del equipo deben sincronizar sus repositorios, a fin de comprobar que los cambios se han subido adecuadamente (`Sincronizar>Extraer`). También es posible realizar las sincronizaciones de entrada y salida a la vez escogiendo `Sincronización>Sincronizar.`

A continuación, el responsable del equipo va a cambiar el nombre y a configurar la librería de clases que añadimos en las sesiones anteriores. Para ello, el responsable del equipo, desde el `Explorador de Soluciones`, debe seleccionar la librería denominada ClassLibrary y pulsar el botón derecho del ratón. Del menú contextual que aparece, debe seleccionar la opción `Cambiar Nombre,` dándole el nuevo nombre `GestDepLibrary` (ver Figura 4).

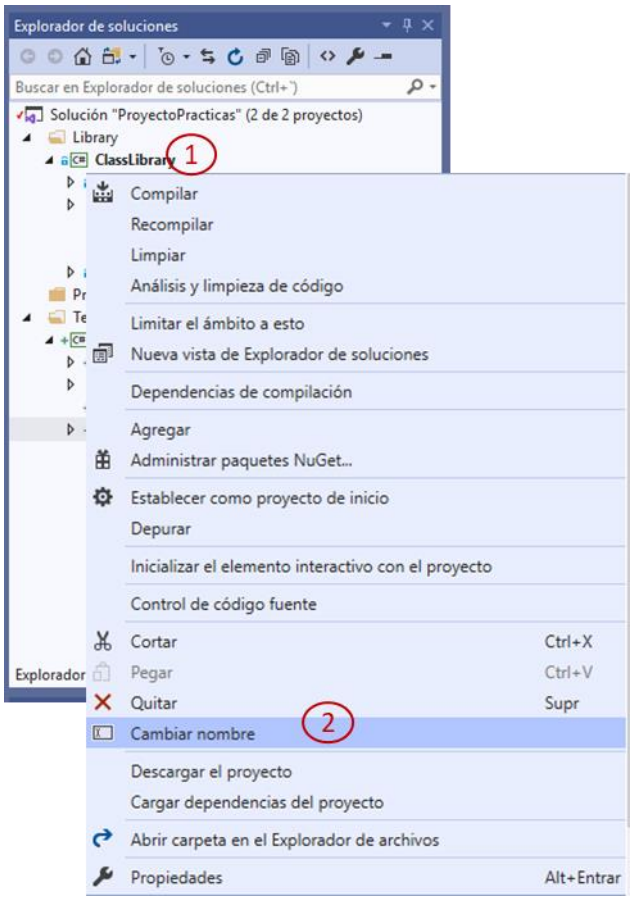

Figura 4. Cambiar el nombre de la librería

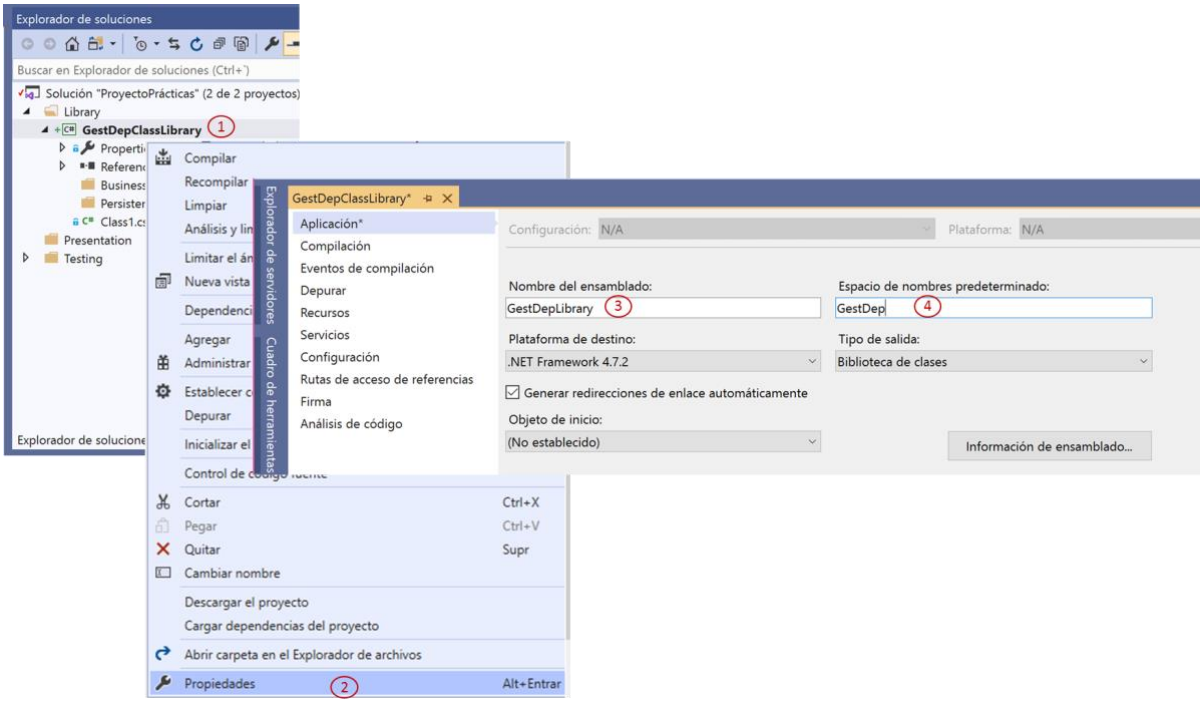

Figura 3. Configurar el espacio de nombres de la librería

Seguidamente, el responsable debe configurar el *Namespace* y el nombre del ensamblado de la librería. Para ello, debe seleccionar de nuevo la librería, y pulsar el botón derecho del ratón. Del menú contextual debe seleccionar la opción `Propiedades`. A la izquierda se abrirá la ficha de propiedades de la librería. Hay que modificar el campo `Nombre del ensamblado` por el valor: `GestDepLibrary`. Además, también debe modificar el nombre del espacio de nombres predeterminado por el valor: `GestDep`. La Figura 3 resume los pasos a seguir.

Por último, el responsable eliminará la clase Class1.cs creada por defecto en la raíz de la librería de clases (no las que se crearon dentro de las capetas para evitar problemas con git).

Llegado a este punto, el responsable debe volver a guardar los Cambios en un `Commit` y subirlos al repositorio remoto, accediendo a la opción `Sincronizar`. Nuevamente, los compañeros pueden recuperar dichos cambios y comprobar las modificaciones.

4. Añadir las clases del modelo de diseño

En la Figura 5 tenéis el modelo de diseño de la aplicación de gestión de actividades deportivas GestDep que debéis implementar. Está formada por nueve clases y una sola enumeración denominada `Days`. La implementación de vuestra solución debe **respetar el nombre de las clases y de los atributos**, así como reflejar fielmente las relaciones establecidas en el modelo. Las relaciones del modelo de análisis se han relajado, restringiendo la navegación en alguna de ellas. Estas restricciones de navegación deben mantenerse.

4.1 Creación de la primera clase

Vamos a añadir un fichero de clase por cada una de las clases del modelo en las carpetas que acabamos de crear. Utilizaremos la estrategia de clases parciales que nos permite C# (**`public partial class`**), de forma que la implementación de una clase puede estar repartida en más de un fichero. Usaremos clases parciales para poder separar el aspecto de persistencia del aspecto de lógica de negocio para cada clase de nuestro modelo.

Nuevamente, los siguientes pasos **solo los debe hacer el responsable del equipo** para evitar conflictos innecesarios en el repositorio. En primer lugar, debe situarse dentro de la carpeta *Persistence/Entities*. Después, debe seleccionar la clase vacía creada anteriormente y cambiarle el nombre por `Person.` Saldrá un aviso preguntando si se desean cambiar también todas las referencias en el proyecto, conteste que sí. En el editor, cambie el *namespace* y la definición de la clase, a fin de que el contenido quede como el siguiente código:

```
namespace GestDep.Entities
{
    public partial class Person
    {
    }
}
```

Vuelva a guardar los Cambios en un `Commit` y súbalos al repositorio remoto. Nuevamente, los compañeros pueden recuperar dichos cambios y comprobar las modificaciones.

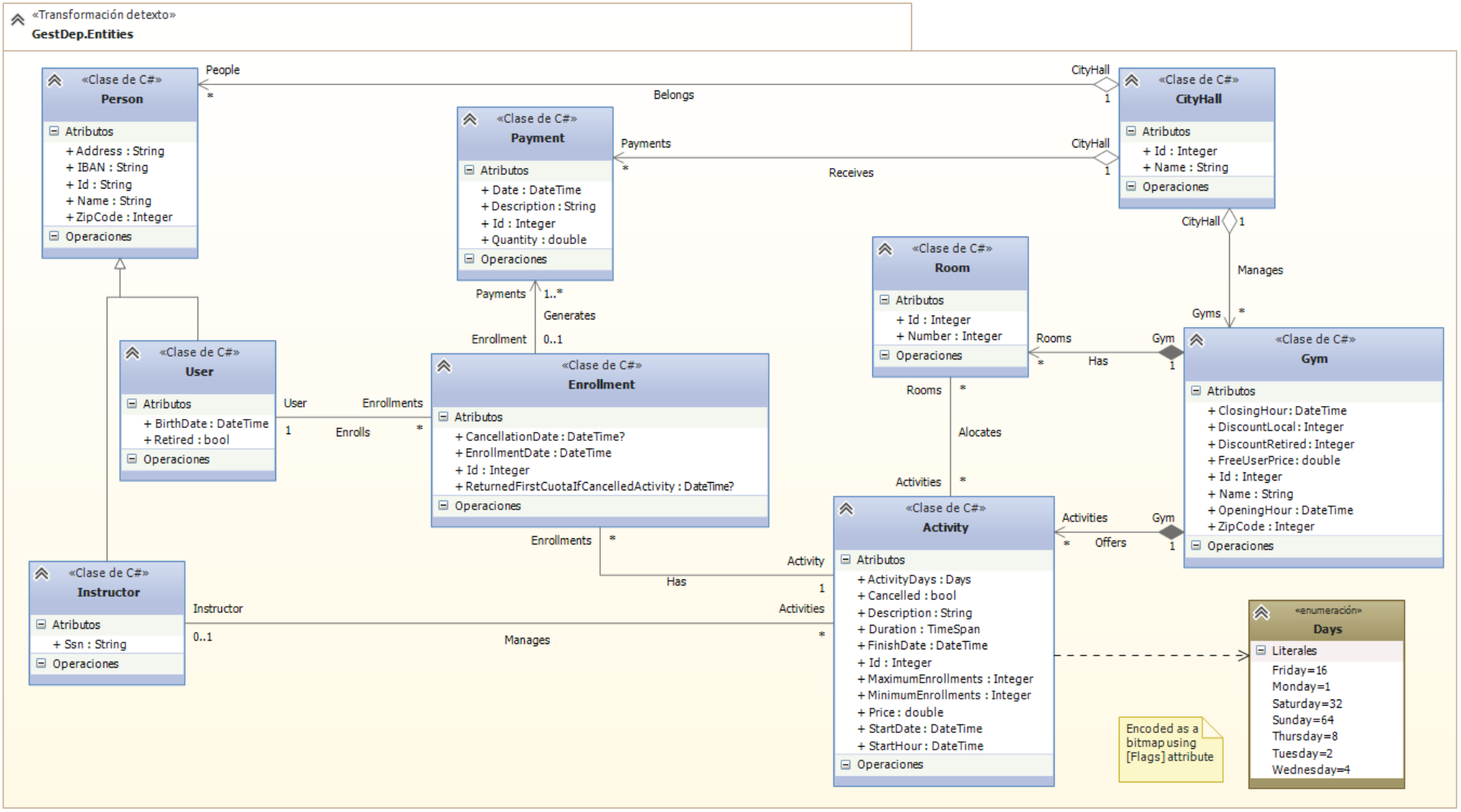

Figura 5. Diagrama de clases de diseño del sistema GestDep

4.2 Creación de la plantilla de clases

Vamos a crear una plantilla para crear todas las clases como públicas y parciales, dentro del mismo *namespace*. Para ello, el responsable del equipo:

- Selecciona del menú principal la opción `Proyecto>Exportar Plantilla`.
- Aparecerá un diálogo para elegir el tipo de plantilla. Selecciona la opción Plantilla de elemento y pulsa Siguiente.
- Se mostrará el diálogo para seleccionar el elemento a exportar. Elige la clase `Person.cs` y pulsa de nuevo `Siguiente`.
- Se visualizará el diálogo para seleccionar que referencias se desean por defecto. No marques ninguna y pulsa Siguiente.
- Aparecerá el último diálogo para indicar las opciones de la plantilla. Cambia el nombre de la plantilla por `GestDepClassDomain` y pulsa `Finalizar`.

En este momento es necesario cerrar Visual Studio y volverlo a abrir para que Visual cargue la plantilla recién creada. La Figura 6 muestra de forma resumida los pasos a seguir.

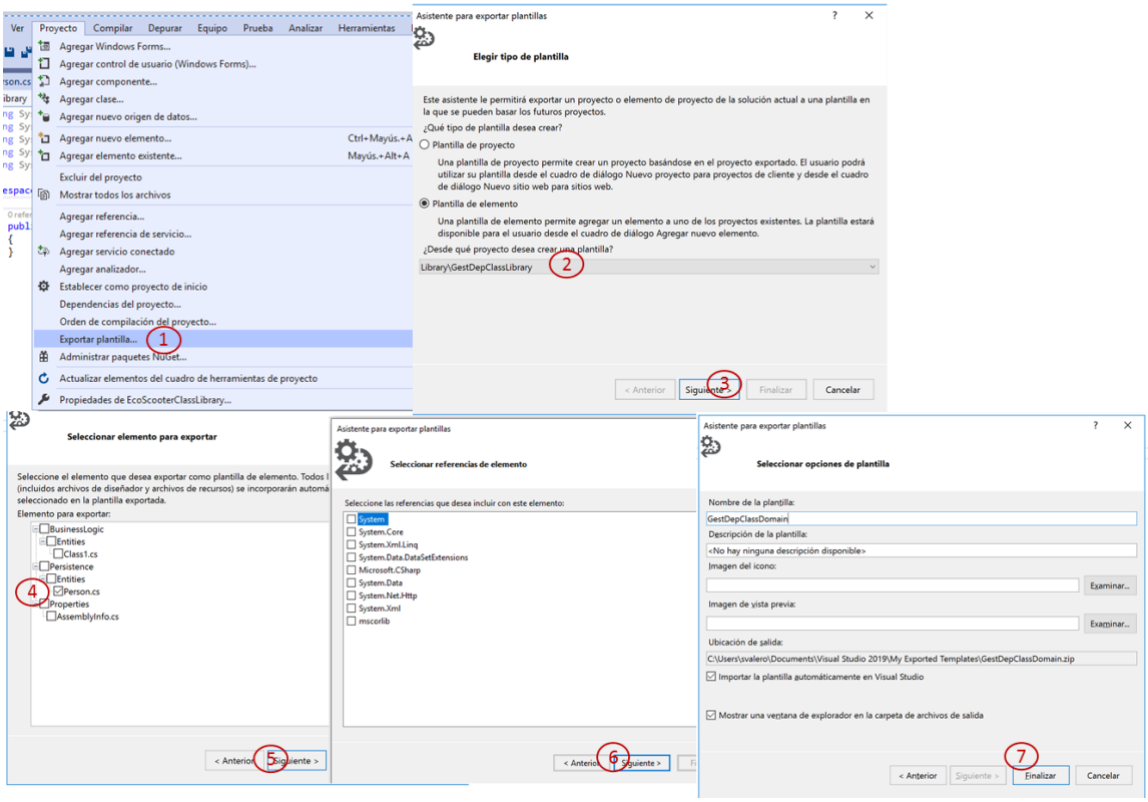

Figura 6. Creación de una plantilla para las clases del dominio

4.3 Añadir el resto de clases del modelo en Persistence/Entities

Tras abrir de nuevo el proyecto en Visual Studio, **el responsable del equipo** creará el resto de clases usando la plantilla del punto anterior, repitiendo para cada una de ellas los siguientes pasos:

- Sitúarse sobre la carpeta de soluciones *Persistence/Entities*.
- Después, abre el menú contextual con el botón derecho del ratón
- Selecciona `Agregar>Nuevo Elemento`
- Elije la opción `GestDepClassDomain` y en el apartado nombre, indica el nombre de la clase a crear (ver Figura 7)
- Pulsa `Agregar`

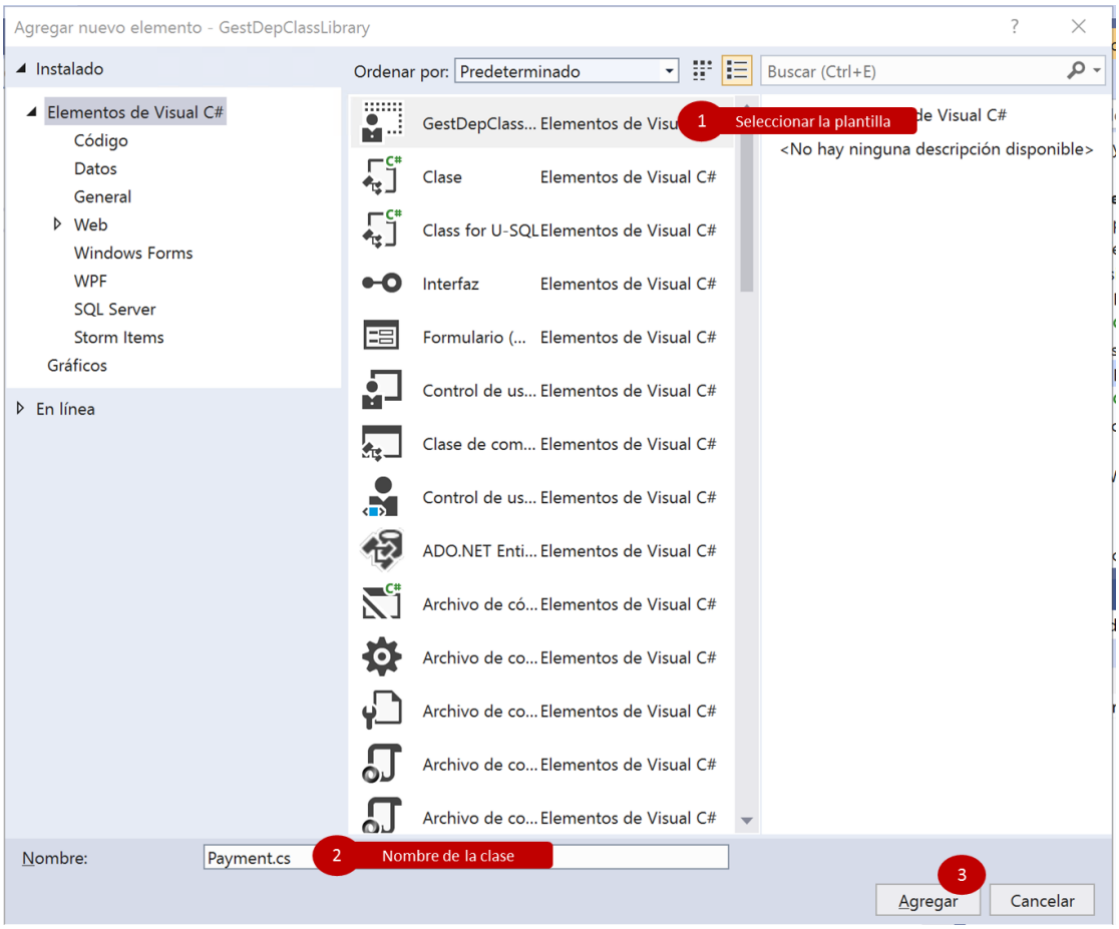

Figura 7. Usar la plantilla para crear las clases

Tras finalizar, deberéis tener como resultado el mismo contenido que puede observarse en la Figura 8. Nuevamente, es un buen momento para salvar los cambios en el proyecto y sincronizarlos con el resto de los miembros del equipo.

4.4 Añadir las clases a BusinessLogic.Entities

Tal y como se indicó anteriormente, vamos a implementar las clases usando clases parciales, de forma que el código correspondiente a la capa de persistencia estará situado en las clases que están dentro de la carpeta *Persistence/Entities*, mientras que el resto del código se situará en *BusinessLogic/Entities.* **El responsable del equipo** va a copiar todas las clases recién creadas (que aún están sin código) a la carpeta *Persistence/Entities.* Para ello, solo tiene que seleccionar todas las clases, y desde el menú contextual seleccionar la opción copiar. Después, se sitúa en *BusinessLogic/Entities* y desde el menú contextual selecciona la opción pegar. Además, debe eliminar la clase vacía `Class1.cs` que se creó para evitar los errores de Git. Nuevamente, debes crear un *commit* del proyecto e insertarlo en el repositorio remoto.

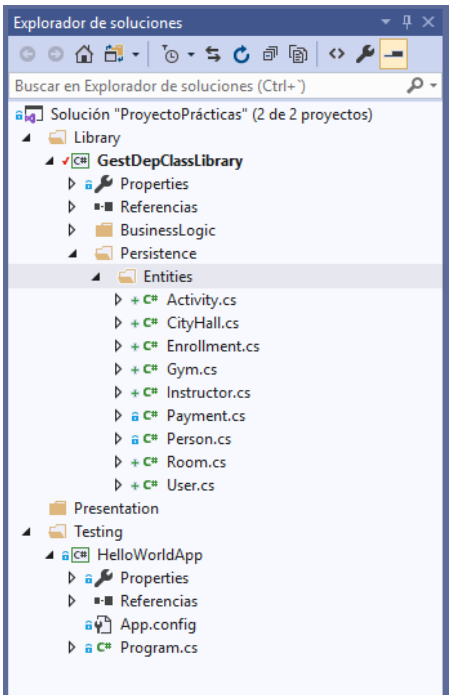

Figura 8. Clases vacías dentro de Persistence.Entities

4.5 Añadir el tipo numerado

Solo nos queda un elemento por crear, el tipo numerado. Para ello, el responsable se sitúa dentro de la carpeta *Persistence/Entities,* y desde el menú contextual selecciona la opción `Agregar>Nuevo Elemento`. En este caso, debe seleccionar la opción `Archivo de código` y nombrarlo como `Days.`

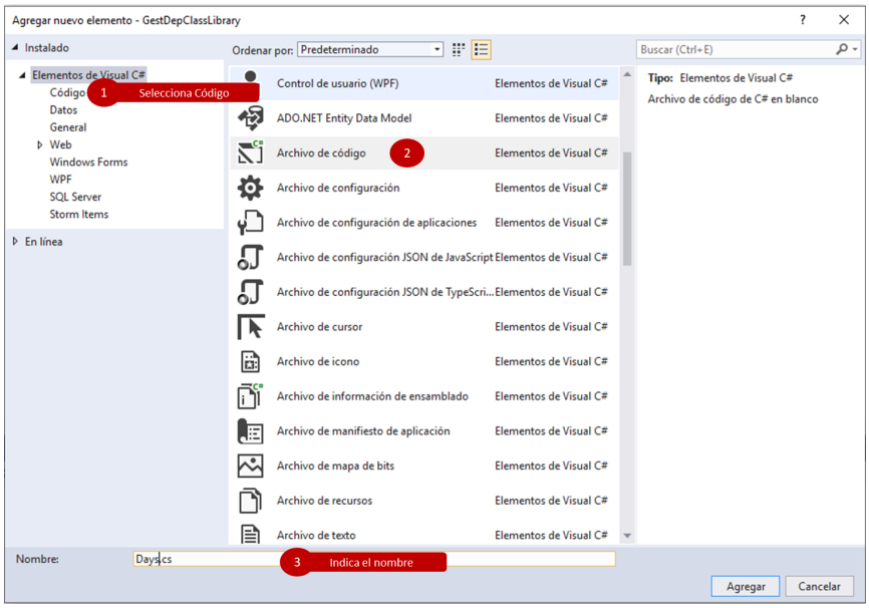

Figura 9. Agregar un archivo de código para implementar el tipo enumerado

El código de Days debe editarse para que sea igual al siguiente:

```
using System;

namespace GestDep.Entities
{
    [Flags]
    public enum Days {
        None = 0,
        Mon = 1,
        Tue = 2,
        Wed = 4,
        Thu = 8,
        Fri = 16,
        Sat = 32,
        Sun = 64
    }
}
```

Mediante la instrucción `[Flags]` indicamos que `Days` es un tipo enumerado que se puede tratar como un campo de bits. Por ejemplo, la siguiente sentencia inicializaría la variable myDays con los días lunes, miércoles y viernes, al poner a 1 dichas posiciones.

```
Days myDays = Days.Mon | Days.Wed | Days.Fri;
```

Con este paso ya tenemos todos los elementos necesarios para empezar a completar el código de las clases. Así, en este momento la solución debe tener el mismo aspecto que el de la Figura 10. El responsable guarda los cambios y los sincroniza con el repositorio remoto. El resto de compañeros debe sincronizar sus repositorios extrayendo todos los cambios realizados

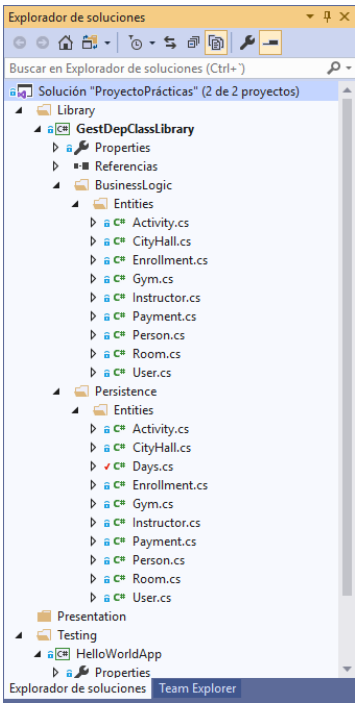

Figura 10. Estado de la solución tras crear todas las clases vacías y el tipo enumerado

5. Implementar el aspecto de persistencia de las clases

A partir de este punto, **todos los miembros el equipo pueden participar en la implementación**, repartiendo el trabajo para que cada miembro trabaje de forma independiente en cada uno de los ficheros para evitar conflictos Se recomienda salvar los cambios y sincronizarlos por cada clase finalizada.

El aspecto de persistencia lo vamos a implementar en las clases alojadas en *Persistence/Entities.* En cada una de ellas hay que añadir las propiedades indicadas en el diseño más las propiedades necesarias para representar las asociaciones del diagrama, siguiendo para ello las pautas explicadas en el tema 5 de teoría. En estas pautas se explica cómo debemos fijarnos en las cardinalidades máximas de las asociaciones, para saber si una asociación se convierte en una propiedad (cardinalidad máxima de 1) o en una colección (cardinalidad máxima de n). Además, debemos fijarnos en las restricciones de navegabilidad del diagrama de diseño, puesto que no todas las asociaciones son bidireccionales.

Por otra parte, las propiedades que aparecen en el diseño (los ***atributos***) se declararan como **public**. Las propiedades para representar las ***relaciones*** las declararemos como **public virtual**, puesto que es un requisito que necesitaremos para implementar la capa de persistencia.

A modo de ejemplo, vamos a proporcionaros el código completo de tres de las clases del modelo: `Person`, `User` y `Enrollment`, explicando las pautas de diseño seguidas en la implementación de cada una de ellas. El resto de clases, deben ser diseñadas por el equipo.

5.1 Implementación de la clase *Person*

La clase *Person* es una generalización que representa los elementos comunes entre la clase *Instructor* y la clase *User*. Existe una asociación entre *Person* y *CityHall*, pero no es navegable desde *Person*, por lo que no debe aparecer en el diseño de esta clase. Así, en el código aparece una propiedad por cada una de las propiedades descritas en el modelo (añadidas en orden alfabético).

```
namespace GestDep.Entities
{
    public partial class Person
    {
        public string Address
        {
            get;
            set;
        }
        public string IBAN
        {
            get;
            set;
        }

        public string Id
        {
            get;
            set;
        }

        public string Name
        {
            get;
            set;
        }
        public int ZipCode
        {
            get;
            set;
        }
     }
}
```

5.2 Implementación de la clase *User*

La clase *User* es una especialización de la clase *Person*, por lo que implementaremos *User* como una clase que hereda de *Person* (`public partial class User : Person`). Además, añadiremos una propiedad por cada una de las propiedades descritas en el modelo, con el tipo indicado. Con respecto a sus asociaciones, *User* se relaciona con *Enrollment*. La relación con *Enrollment* es navegable y su cardinalidad máxima es de N (* según la notación empleada por el modelo), por lo que debemos añadir una colección de tipo *Enrollment* en esta clase, que denominaremos *Enrollments*. Nótese que en primer lugar hemos añadimos las propiedades propias, y tras ellas, las propiedades derivadas de las asociaciones de la clase con otras entidades del modelo. Seguidamente adjuntamos el código.

```
namespace GestDep.Entities
{
    public partial class User : Person
    {
        public DateTime BirthDate {
            get;
            set;
        }

        public bool Retired {
            get;
            set;
        }

        /* Associations*/
        public virtual ICollection<Enrollment> Enrollments {
            get;
            set;
        }
    }
}
```

5.3 Implementación de la clase *Enrollment*

En primer lugar, añadiremos una propiedad por cada una de las propiedades descritas en el modelo, con el tipo indicado En este caso tenemos una propiedad en la que aparece un "?" después del tipo de datos:

```
CancellationDate: DateTime?
```

Esto permite que el valor del atributo cuyo tipo de datos no es un objeto puede admitir valores nulos. Las referencias a objetos pueden ser nulas, pero no los valores (tipos primitivos y *struct*,). Esta notación permite hacerlos también nulos[1]. En nuestro ejemplo, la fecha de cancelación (`CancellationDate`) es de tipo `DateTime`, que es un tipo *struct*. Ese atributo puede que nunca tenga un valor de fecha válida y que sea nulo hasta que un empleado indique que ha terminado con el mantenimiento. Para permitir un valor nulo en esa fecha se define del tipo `DateTime?`. De forma similar, cuando en el constructor se utiliza un parámetro de ese tipo, hay que definirlo de tipo `DateTime?`, y así se le puede pasar el valor `null`.

Para comprobar si una variable definida de esta manera tiene un valor `null` se utiliza `HasValue`, y para acceder al valor `Value`, como ilustra el siguiente ejemplo:

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

```
            DateTime? EndDate = null;


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

Con respecto a las asociaciones, la clase *Enrollment* se relaciona con tres clases: *User, Payment* y *Activity* y todas ellas son navegables. Comencemos por la relación entre *Enrollment* y *Activity*, en este caso la cardinalidad máxima es de 1, por lo que añadiremos una propiedad de tipo `Activity`. Con respecto a la relación entre *Enrollment* y *Payment,* vemos que la cardinalidad máxima es N, así que tendremos que añadir una colección de tipo `Payment` que denominaneros `Payments`. Finalmente, en la asociación entre *Enrollment* y *User*, podemos observar que la cardinalidad máxima es de 1, por lo que añadiremos una propiedad de tipo `User` en el diseño de la clase.

Seguidamente, podéis observar el código resultante.

```
namespace GestDep.Entities
{
    public partial class Enrollment
    {
        public DateTime? CancellationDate
        {
            get;
            set;
        }
        public DateTime EnrollmentDate
        {
            get;
            set;
        }

        public int Id
        {
            get;
            set;
        }

        public DateTime? ReturnedFirstCuotaIfCancelledCourse
        {
            get;
            set;
        }

        //Associations
        public virtual Activity Activity
        {
            get;
            set;
        }
        public virtual ICollection<Payment> Payments
        {
            get;
            set;
        }

        public virtual User User
        {
            get;
            set;
        }
    }
}
```

6. Implementar los constructores de las clases en BusinessLogic

El código de los constructores de las clases debe implementarse en los ficheros de las clases alojados en BusinessLogic/Entities. Nuevamente, todos los miembros el equipo pueden colaborar en el desarrollo, pero debe dividirse el trabajo de forma que **cada uno trabaje en una clase diferente para no generar conflictos.** Se recomienda crear commits y sincronizar por cada clase finalizada.

Toda clase de nuestro sistema **tendrá dos constructores**, uno sin parámetros y otro con todos los parámetros necesarios. El constructor sin parámetros simplemente inicializará las colecciones que tenga la clase creando **colecciones** vacías. En cambio, el segundo constructor con parámetros, además de inicializar las colecciones, debe recibir el valor de todas las propiedades que necesitan ser inicializadas. No añadiremos al constructor el atributo **Id si es de tipo Integer.** Usaremos **Entity Framework (EF)** para la persistencia de la información y EF **se encarga de** dar valor a los **atributos ID de tipo numérico** de manera automática en el momento en el que se persiste el objeto por primera vez. Por lo tanto, no debemos dar valor de forma manual a esos valores, ya que EF quien le asignará el valor en cuando lo persista por primera vez.

El **orden de los parámetros** en el constructor será en **primer lugar los parámetros propios en orden alfabético** y **después** los **objetos** que tiene que recibir para asignarlos a las propiedades añadidas a causa de las **asociaciones** de la clase con otras, también en orden alfabético. En el caso de clases **que hereden de otras**, en primer lugar, pasaremos el bloque de los **parámetros que recibe la clase padre, ordenados de forma alfabética**. En segundo lugar, **el bloque con los parámetros propios** (también ordenados) y **por último** las propiedades añadidas a causa de las **asociaciones**.

Hay que recordar que para el caso de las clases que poseen atributos “nullables” (i.e con “`?`”), el constructor con parámetros debe declarar también el **tipo de datos con “`?`**” para que haya concordancia de tipos.

A modo de ejemplo, seguidamente se proporcionará el código de las mismas clases usadas de ejemplo en el punto anterior: `Person`, `User` y `Enrollment`.

6.1 Implementación de los constructores de la clase *Person*

`Person` no tiene ninguna colección, por lo que su constructor sin parámetros no tiene ninguna sentencia. El constructor con parámetros recibe todos los parámetros necesarios para instanciar sus propiedades, incluido el Id, puesto que es de tipo `string`. Nótese que son recibidos en **orden alfabético.**

```
namespace GestDep.Entities
{
    public partial class Person
    {
        public Person() {
        }

        public Person(string address, string iban, string id, string name, int
zipCode) {
            Address = address;
            IBAN = IBAN;
            Name = name;
            ZipCode = zipCode;
            Id = id; // //DNI value. It is a String, so EE doesn't manage it.

        }
    }
}
```

6.2 Implementación de los constructores de la clase User

`User` hereda de `Person`, por lo que sus constructores llamarán a los constructores de su clase padre para que realice la inicialización de las propiedades heredadas, usando la sentencia `base()`. Además, en el constructor con parámetros, en primer lugar, pasaremos en orden alfabético los parámetros que recibe para instanciar sus propiedades heredadas y en segundo lugar los parámetros propios (también en orden alfabético).

Esta clase se relaciona con la clase `Enrollment` y debemos fijarnos **en la cardinalidad mínima** de la asociación. En este caso es de cero, por lo que tampoco tenemos que pasar ningún objeto de la clase `Enrollment` como parámetro en el constructor.

Tanto en el constructor sin parámetros como en el constructor con parámetros, debemos inicializar la colección `Enrollments`, creando una lista vacía.

```
namespace GestDep.Entities
{
    public partial class User
    {
        public User() {

            Enrollments = new List<Enrollment>();

        }

        public User(string address, string iban, string id, string name,
                    int zipCode, DateTime birthDate, bool retired) :
               base ( address,  iban,  id,  name,  zipCode) {

            BirthDate = birthDate;
            Retired = retired;

            Enrollments = new List<Enrollment>();

        }
    }
}
```

6.3 Implementación de los constructores de la clase *Enrollment*

La clase `Enrollment` tiene una colección de `Payment`, por lo que el constructor sin parámetros deberá inicializarla. El constructor con parámetros también debe hacer esta inicialización, por lo que llamará a `this()` para evitar duplicidades de código.

Por otra parte, la clase `Enrollment` tiene cuatro atributos propios que inicializar. Sin embargo, la propiedad `CancellationDate` siempre tendrá un valor nulo cuando creemos una matrícula, pues de ser cancelada, lo hará en un momento posterior. Por ello no tiene sentido crear en este caso un constructor que reciba ese parámetro, añadiendo la inicialización a NULL directamente en el código del constructor. De forma similar, `ReturnedFirstCuotaIfCancelledCourse` también tendrá un valor de NULL cuando creemos una matrícula. Además, la propiedad Id es un entero y lo controlará EE

Con respecto a sus asociaciones, nos fijamos en su **cardinalidad mínima**, siendo en todos los casos de **1**. Por lo tanto, debemos pasar en el constructor un objeto de cada clase con las que se relaciona: `Activity`, `Payment` y `User`.

```
namespace GestDep.Entities
{
    public partial class Enrollment
    {
        public Enrollment() {
           Payments = new List<Payment>();
        }

       public Enrollment(DateTime enrollmentDate, Activity activity,
            Payment payment, User user): this(){

            CancellationDate = null;
            EnrollmentDate = enrollmentDate;
            ReturnedFirstCuotaIfCancelledCourse = null;
            //Id is managed by EE
            Activity = activity;
            Payments.Add(payment);
            User = user;

       }
}
```

6.4 Reutilizar código usando `this`

En las clases con colecciones, es necesario inicializarlas tanto en el constructor sin parámetros como en el constructor con parámetros. Por ello, es posible reutilizar código llamando al constructor sin parámetros desde el constructor con parámetros, tal y como se muestra en el siguiente ejemplo. Nótese que esto no es posible cuando existe herencia, puesto que se debe llamar a `base`.

```
    public partial class Vehicle
    {
        public Vehicle () {
            Wheels = new List<Wheel>();
        }
        public Vehicle (string tradeMark): this() {
            //Wheels is initiated in Vehicke()
            TradeMark = tradeMark;

        }
    }
```

6.5 Consideración a la hora de implementar *Activity*

La propiedad `Cancelled` de `Activity` también estará siempre a `false` cuando se creen objetos de esta clase, puesto que las actividades al crearse no estarán canceladas. Por ello, al igual que con Enrollment, el constructor no recibirá un parámetro para instanciar su valor, sino que directamente se instanciará a `false` en su código.

7. Trabajo a entregar

Al finalizar las dos sesiones, los grupos deben haber finalizado la implementación de las 9 clases del modelo, así como del tipo enumerado. Las propiedades de las clases deben estar implementadas en los ficheros situados en Persistence/Entities. Los constructores de las clases deben estar implementados en BusinessLogic/Entities. Además, por cada clase debemos tener dos constructores:

- Uno sin parámetros, que en el caso de que la clase tenga colecciones las inicialice.

- Otro con los parámetros necesarios, donde estos parámetros siguen el orden establecido en el apartado 6. Además, estos constructores también inicializan las colecciones que contenga.

Al finalizar la práctica, el código de cada equipo deberá pasar una serie de tests, **por lo que, si no se siguen** las pautas anteriores, **el código no pasará** las pruebas.