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 2024-2025

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 deben realizarse por un solo miembro del equipo (el responsable o coordinador del equipo).

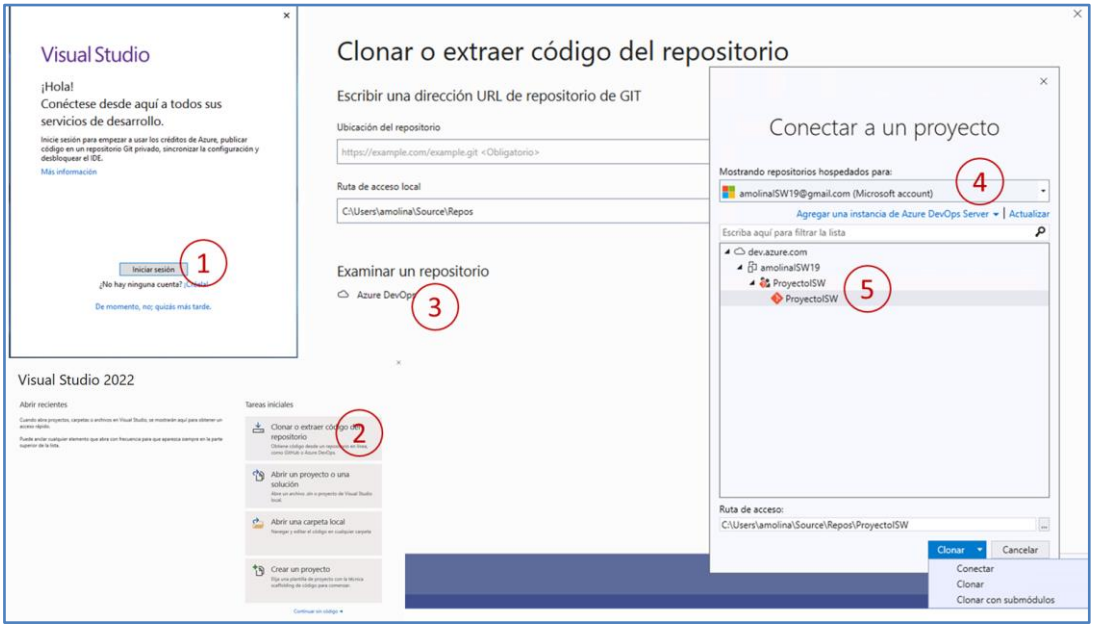

Figura 1. Pasos a seguir para conectarse y clonar el proyecto de equipo

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, comprobad 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 ST3-1 DevOps-Visual-Git y el boletín de la práctica 1 para completar los pasos que falten a fin de obtener la estructura de proyecto requerida.

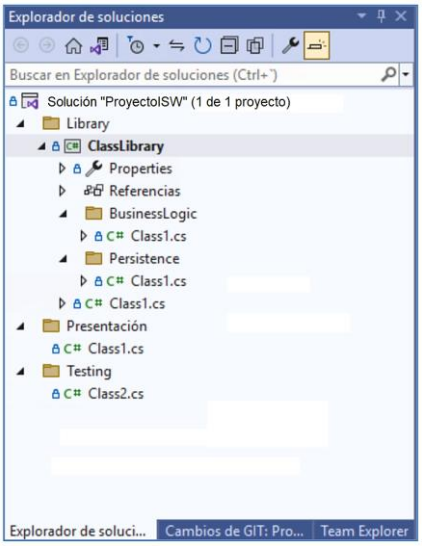

Figura 2. Configuración inicial de la solución del proyecto

Tras abrir la solución, el responsable debe crear una carpeta 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, añadirá 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 en la ventana `Cambios de GIT` tal y como se muestra en la Figura 3. Para ello debe proporcionar el mensaje de texto que describe el conjunto de cambios y elegir la opción `Confirmar todo` del desplegable. Después, debe subir los cambios al repositorio remoto haciendo un *`Insert`* . También es posible realizar estas dos operaciones a la vez con la opción `Confirmar todo e insertar` del desplegable.

El resto de miembros del equipo deben sincronizar sus repositorios, a fin de comprobar que los cambios se han subido adecuadamente con la opción `Extraer` . También es posible realizar las sincronizaciones de entrada y salida a la vez escogiendo `Confirmar todo y sincronizar` en el desplegable.

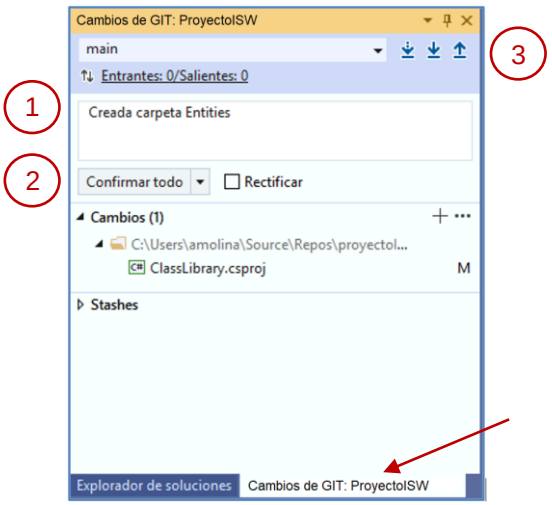

Figura 3. Insertando los cambios en el repositorio (push)

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 *`GestAcaLib`* (ver Figura 4).

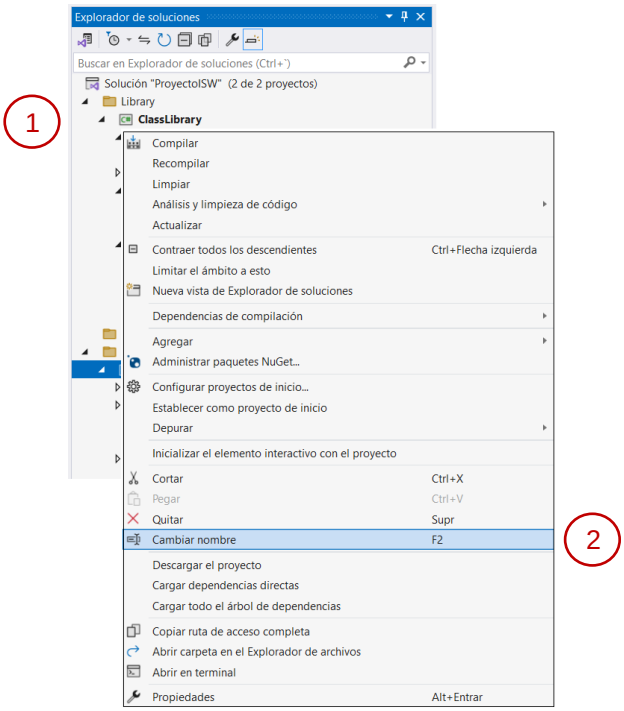

Figura 4. Cambiar el nombre de la librería

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: `GestAcaLib`. Además, también debe modificar el nombre del espacio de nombres predeterminado por el valor: `GestAca`. La Figura 5 resume los pasos a seguir.

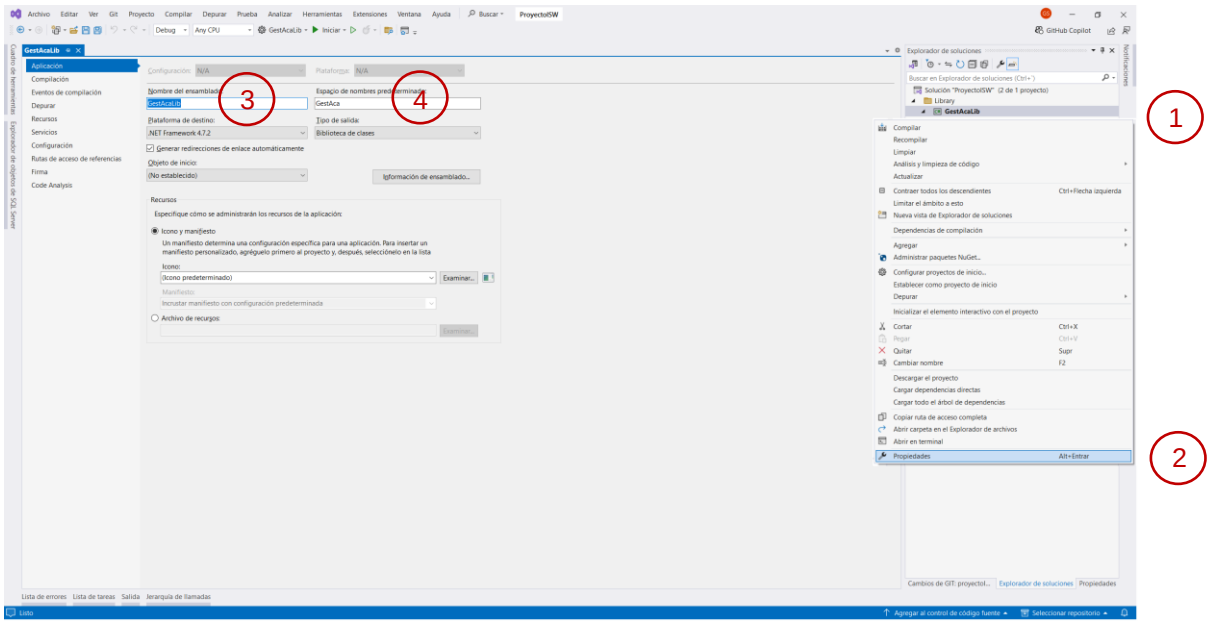

Figura 5. Cambiar el nombre de ensamblado de la librería y el nombre del espacio de nombres

Por último, el responsable eliminará la clase Class1.cs creada por defecto en la raíz de la librería de clases (la que aparece en la configuración inicial del proyecto en la Figura 2).

Llegado a este punto, el responsable debe volver a confirmar todos los cambios e insertarlos en el repositorio tal y como se explicó anteriormente (ver Figura 3). Nuevamente, los compañeros pueden recuperar dichos cambios y comprobar las modificaciones.

4. Añadir las clases del modelo de diseño

En la Figura 6 tenéis el modelo de diseño de la aplicación que debéis implementar. Está formada por ocho clases. 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. Debéis fijaros en la navegabilidad entre las relaciones del modelo, las relaciones bidireccionales tendrán que implementarse en los dos sentidos, mientras que las unidireccionales, si las hubiera, únicamente en el sentido indicado por la flecha.

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. En concreto, separaremos la implementación de una clase en dos archivos, uno en la carpeta *Persistence/Entities* que incluirá la declaración de las propiedades de la clase, y otro en la carpeta *BusinessLogic/Entities* que contendrá la lógica de la clase (constructor y métodos).

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*.

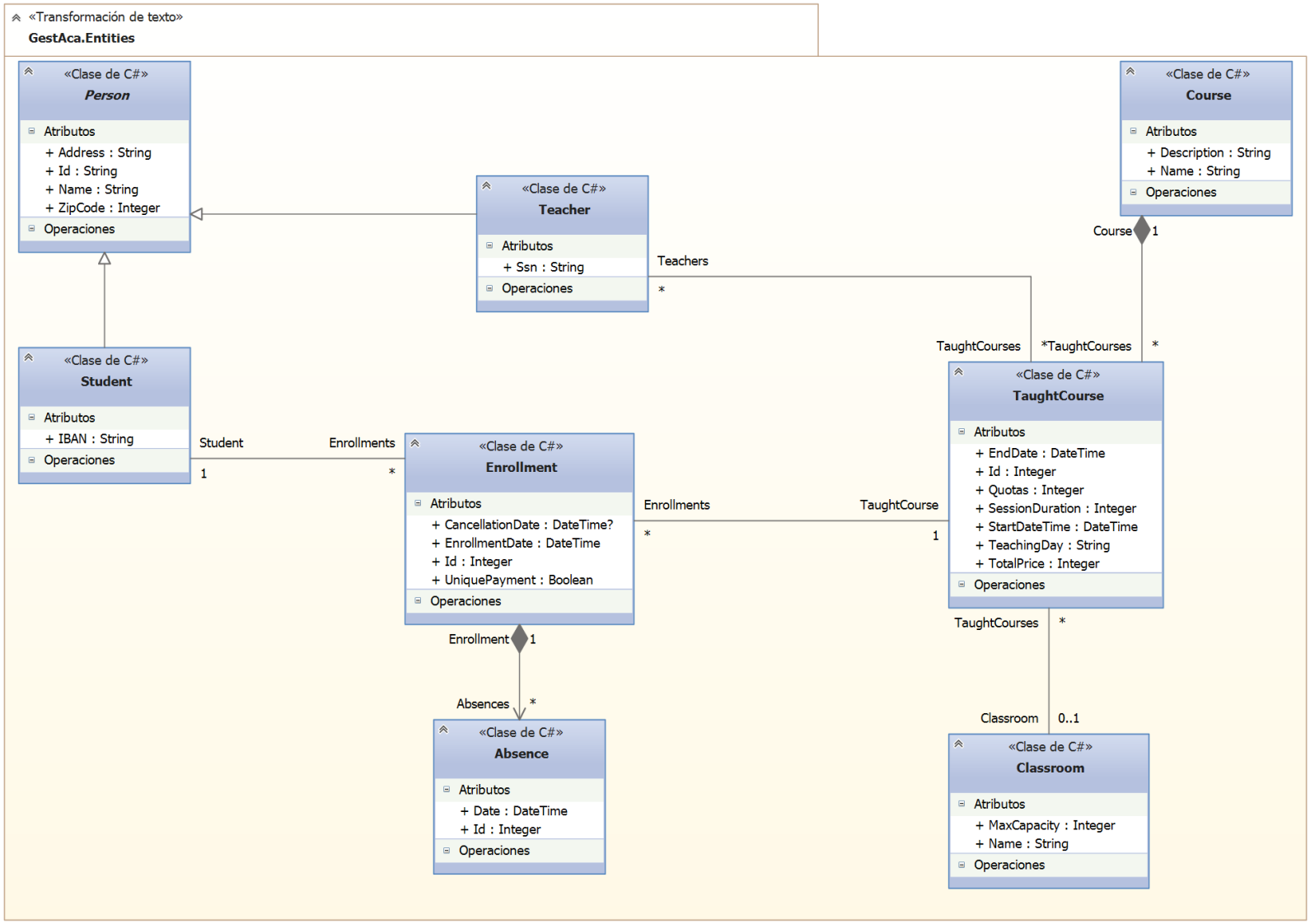

Figura 6. Diagrama de clases de diseño del caso de estudio

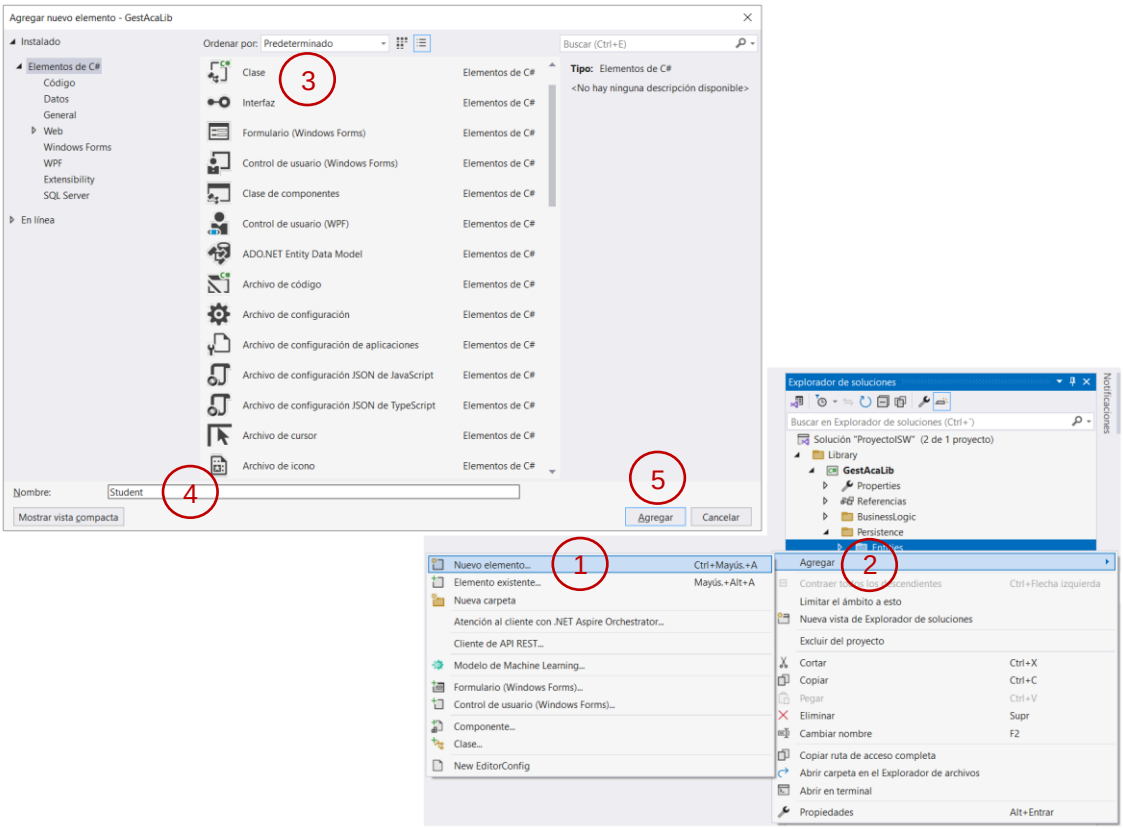

Figura 7. Agregar una nueva clase al proyecto

Los pasos para agregar una clase a la carpeta son (ver Figura 7):

1. Primero sitúate sobre la carpeta de soluciones *Persistence/Entities* y abre el menú contextual con el botón derecho del ratón
2. Selecciona `Agregar>Nuevo Elemento`
3. Elije el elemento `Clase` (es el elemento predeterminado)
4. Indica el nombre de la clase a crear (Student.cs)
5. Pulsa `Agregar`

En el editor, cambie el *namespace* y la definición de la clase, a fin de que el contenido quede como el siguiente código:

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

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

4.2 Creación de la plantilla de clases

Antes de continuar, 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 `Student.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 `GestAcaClassDomain` 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 8 muestra de forma resumida los pasos a seguir.

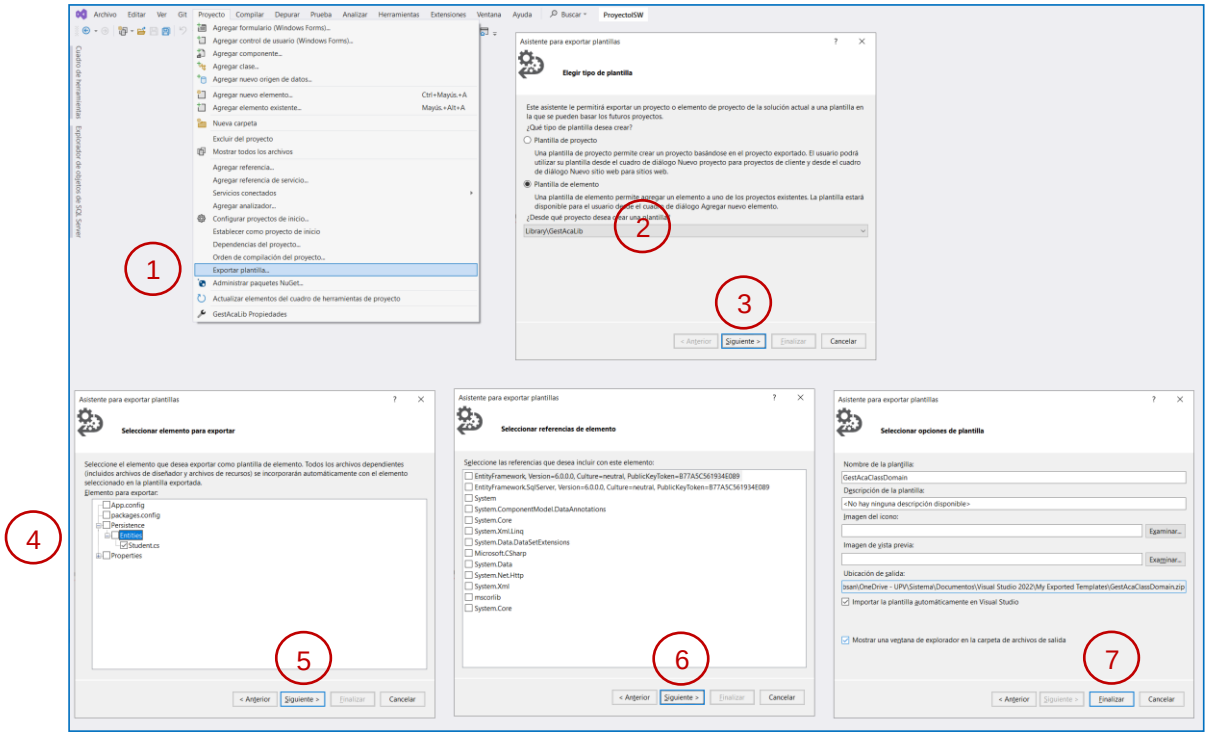

Figura 8. Creación de una plantilla para las clases del dominio

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 las clases usando la plantilla del punto anterior, repitiendo para cada una de ellas los siguientes pasos:

- Situarse 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 `GestAcaClassDomain` y en el apartado nombre, indica el nombre de la clase a crear (ver Figura 9)
- Pulsa `Agregar`

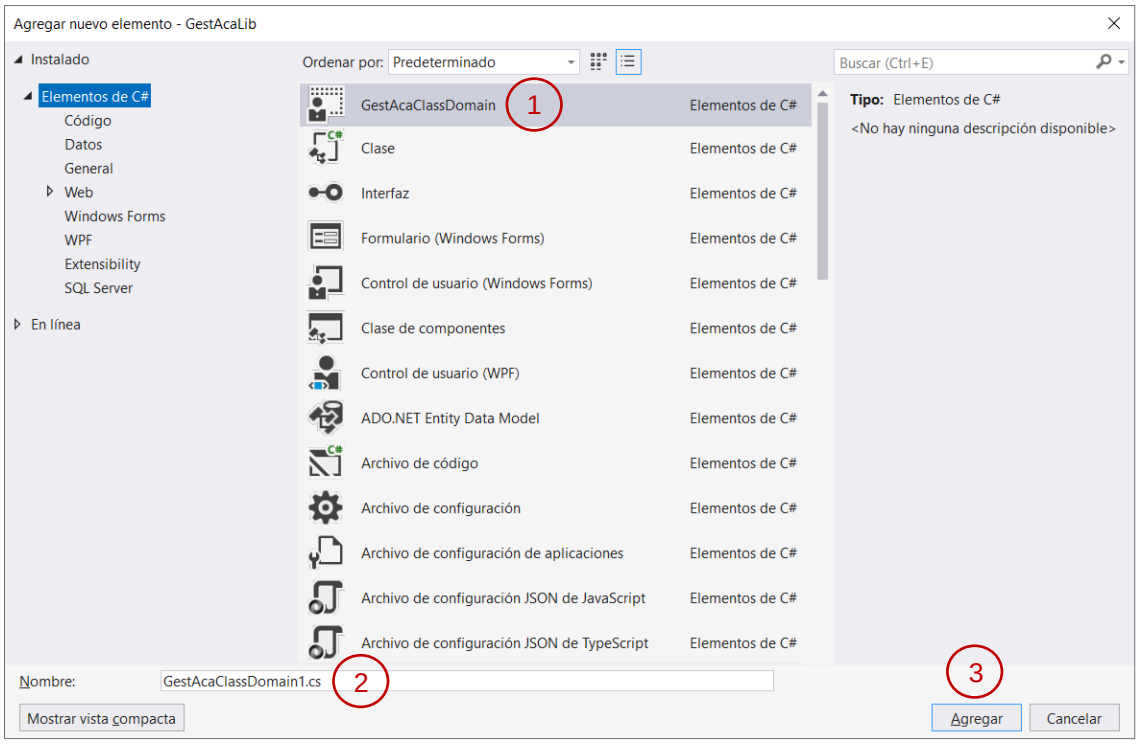

Figura 9. Usar la plantilla para crear las clases

Tras finalizar, deberéis tener como resultado el mismo contenido que puede observarse en la Figura 10.

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.

Nuevamente, debes crear un *commit* del proyecto e insertarlo en el repositorio remoto.

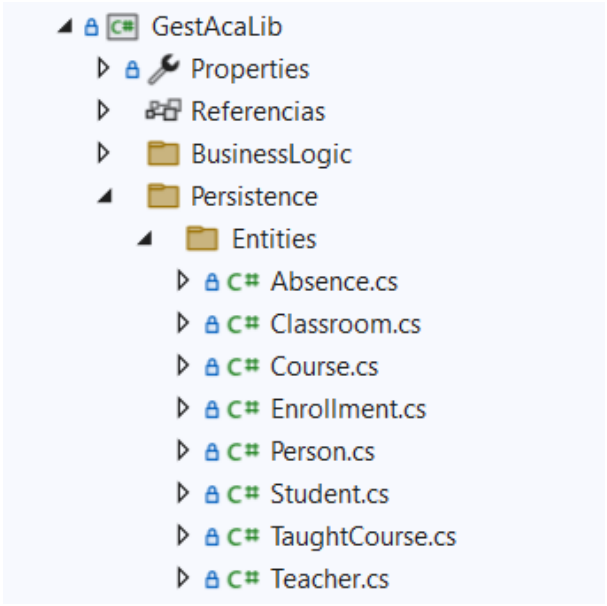

Figura 10. Clases vacías creadas en Persistence, Entities

4.5 Añadir el tipo numerado

Si el diagrama de clases incluyera algún **tipo enumerado** deberíamos crearlo también. En el caso de estudio GestAca no hay ningún enumerado, por lo tanto, no hemos de hacer nada más. Si hubiera un enumerado, por ejemplo, un enumerado denominado Auhorized, con tres posibles valores (Yes, No, Pending), en este caso, deberíamos de crear el tipo enumerado `Authorized`. Para ello, dentro de la carpeta *Persistence/Entities,* y desde el menú contextual se selecciona la opción `Agregar>Nuevo Elemento`. En este caso, debe seleccionar la opción `Archivo de código` y nombrarlo, `Authorized.cs` (ver Figura 8).

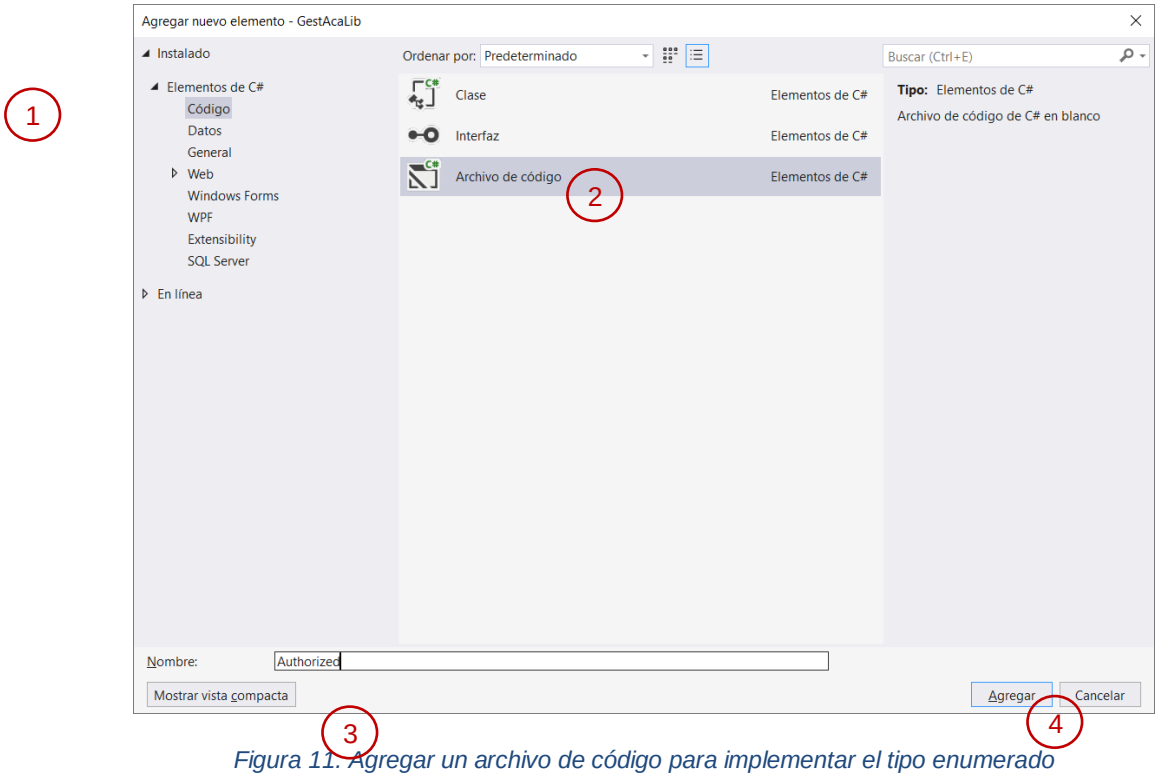

Figura 11. Agregar un archivo de código para implementar el tipo enumerado

El código de `Authorized` debería editarse añadiendo los valores permitidos, p.e.:

```
using System;

namespace GestAca.Entities
{

    public enum Authorized : int
    {
       Yes,
       No,
       Pending,
    }

}
```

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 12.


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

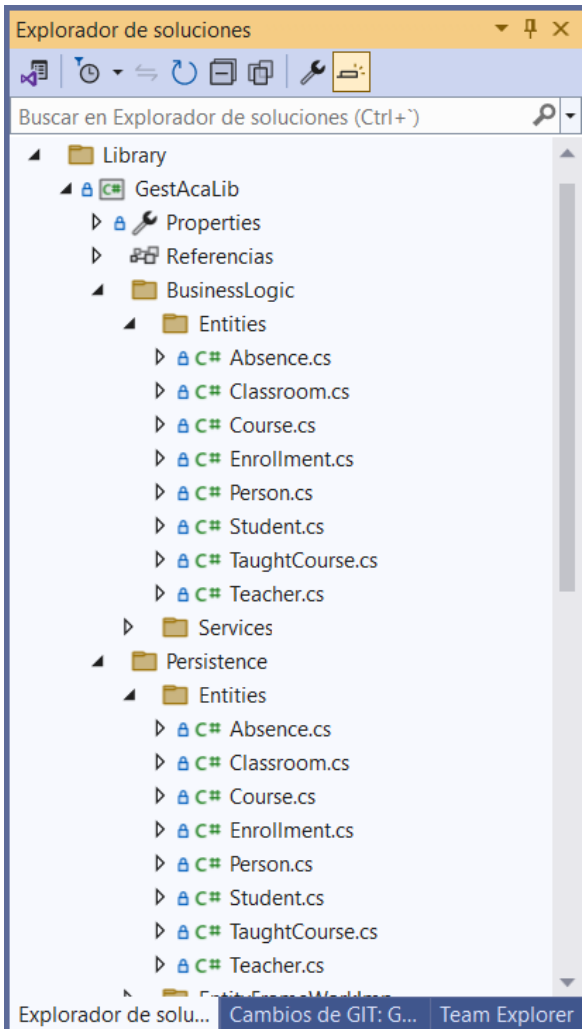

Figura 12. Estado de la solución tras crear todas las clases vacías

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, deberíamos fijarnos en las restricciones de navegabilidad del diagrama de diseño en caso de que todas las asociaciones no fueran 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.

EJEMPLO: Caso Estudio TarongISW

Vamos a utilizar un ejemplo (caso de estudio TarongISW) para explicar los pasos a seguir para implementar las propiedades de las clases aplicando las pautas de diseño. En concreto, dado el diagrama de clases de la Figura 13, se implementarán las clases *Contract* y *Temporary*.

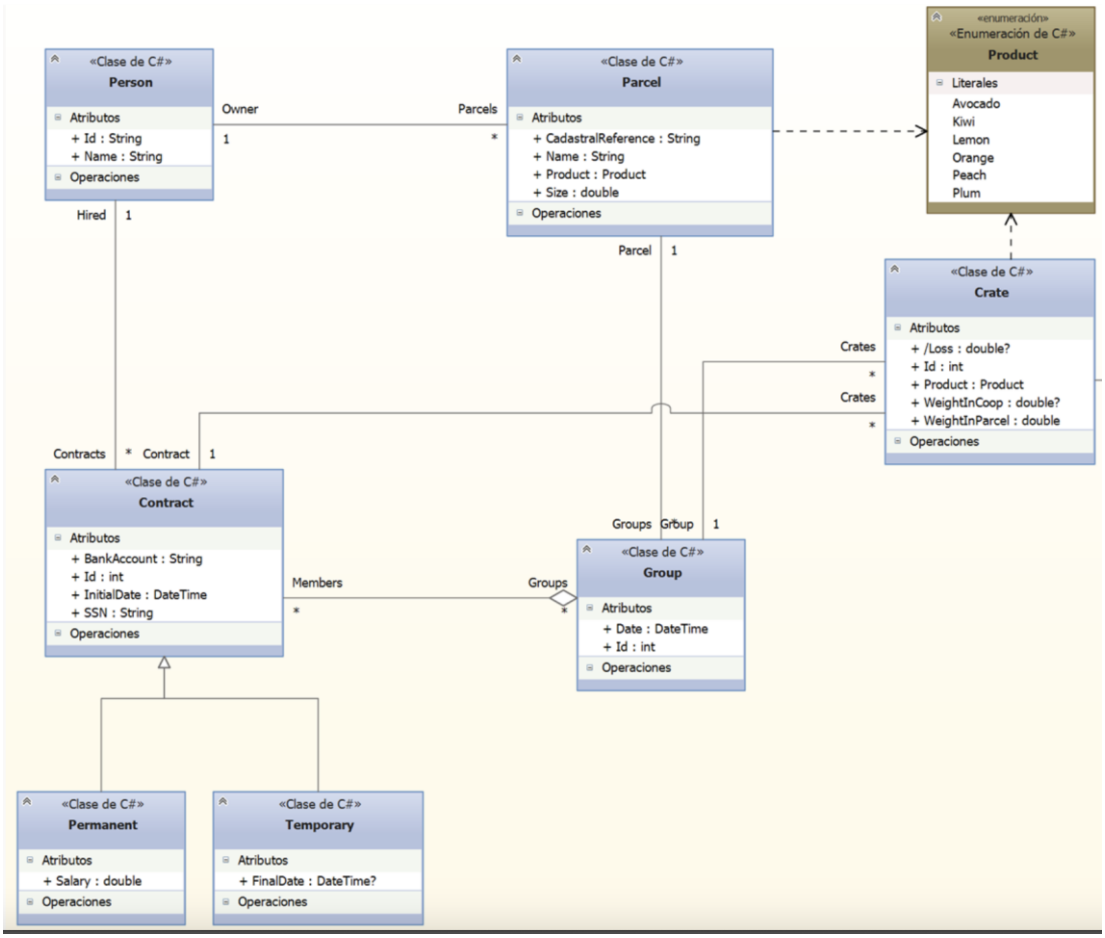

Figura 13. Diagrama de clases de ejemplo del caso de estudio TarongISW

De forma similar, el equipo tendrá que implementar las clases de su caso de estudio (cada miembro del equipo se responsabilizará de implementar unas clases).

5.1 Implementación de la clase *Contract*

La clase *Contract* es una generalización que tiene las propiedades comunes de los dos tipos de contratos representados por las clases *Permanent* y *Temporary*. Así, en el código aparece una propiedad por cada una de las propiedades descritas en el modelo. Se relaciona mediante con las clases *Group* y *Crate* mediante asociaciones con cardinalidad máxima n (*) que se implementan con las colecciones `Groups` y `Crates`, respectivamente. Obsérvese que el nombre de las colecciones coinciden con los roles de la correspondiente relación en el modelo. También se relaciona con la clase *Person* mediante una asociación con cardinalidad 1, que se implementa con la propiedad `Hired` de tipo `Person`.

Nótese que primero se han añadido las propiedades propias de la clase y luego las propiedades derivadas de las relaciones con otras clases según el modelo.

```
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;
        }
}
```

5.2 Implementación de la clase *Temporary*

La clase *Temporary* es una especialización de la clase *Contract*, por lo que la implementaremos como una clase que hereda de `Contract` (`public partial class Temporary : Contract`). Esta clase no se relaciona con ninguna otra clase, por lo que únicamente tendríamos que añadir una propiedad por cada una de las propiedades descritas en el modelo, con el tipo indicado.

En este caso solamente tenemos la propiedad *FinalDate* en la que aparece un "?" después del tipo de datos:

```
FinalDate: DateTime?
```

Esto permite que el valor del atributo cuyo tipo de datos no es un objeto pueda 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 finalización de contrato (`FinalDate`) es de tipo `DateTime`, que es un tipo *struct*. Ese atributo puede que no tenga un valor de fecha válida hasta que finalice el contrato temporal. 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`. Seguidamente adjuntamos el código de la clase *Temporary*.

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

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

```
            DateTime? FinalDate = null;
            if (FinalDate.HasValue)
                Console.WriteLine(FinalDate.Value);
            else
                Console.WriteLine("La fecha de finalización no tiene valor.");
```

(Este código no debe incluirse ahora, solamente si se necesita durante la implementación de la lógica de la aplicación)

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

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

con parámetros, además de inicializar las colecciones, debe recibir el valor de todas las propiedades que necesitan ser inicializadas.

IMPORTANTE:

No añadiremos al constructor el atributo **Id si es de tipo Integer.** Más adelante 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 será 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**. Esto facilitará la ejecución de proyectos de pruebas que serán proporcionados más adelante.

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: `Contract` y `Temporary`.

6.1 Implementación de los constructores de la clase *Contract*

`Contract` tiene dos colecciones, `Crates` y `Groups`, que deben inicializarse en el constructor sin parámetros. El constructor con parámetros recibe todos los parámetros necesarios para instanciar sus propiedades, excepto el Id, puesto que es de tipo `int`. Nótese que son recibidos en **orden alfabético.** La cardinalidad mínima para las relaciones con *Group* y *Crate* es 0 por lo que no es necesario asignar ningún elemento inicialmente a las colecciones y, por tanto, no es necesario pasar al constructor ningún parámetro. Es decir, no es necesario asignar incialmente a un contrato un grupo o un cajón. La cardinalidad con *Person* es 1 por lo que, cuando se crea un contrato, este debe relacionarse necesariamente con una persona[2]. Para ello tenemos que pasar el correspondiente valor al constructor en el parámetro `hired`. El código resultante sería este:

```
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 se le da valor a Id porque se lo dará EF
              this.BankAccount = bankAccount;
              this.InitialDate = initialDate;
```

[2] En el caso de que tuvieramos cardinalidad mínima distinta de 0 en los dos lados de la relación sería necesario relajar alguno de los construtores, tal y como se explica en el Tema 5 de teoría.

```
            this.SSN = SSN;

            // Relaciones a 1
            Hired = hired;

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

6.2 Implementación de los constructores de la clase *Temporary*

`Temporary` hereda de `Contract`, 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 si lo tuviera (también en orden alfabético).

Si hay atributos cuyo valor no se conoce en el momento de crear el objeto no se pasarán como parámetro al constructor. Son los que están marcados en el diagrama de diseño con el símbolo ?. En ese caso, dentro del constructor podrán inicializarse a un valor por defecto o nulo. En la clase `Temporary`, el valor de `FinalDate` no se conoce en el momento de instanciar el objeto y se inicializa a `null` en el 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 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()`. El código de la clase `Contract` utilizando `this()` sería el siguiente, donde puedes ver que ya no aparece en el constructor con parámetros la inicialización de las listas:

```
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): this()
        {
            // No se le da valor a Id porque se lo dará EF
            this.BankAccount = bankAccount;
            this.InitialDate = initialDate;
            this.SSN = SSN;

            // Relaciones a 1
            Hired = hired;

        }
}
```

7. Trabajo a entregar

Al finalizar las dos sesiones, los grupos deben haber finalizado la implementación de todas las 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 incluirá la inicialización de las colecciones en el caso de que la clase las tenga.
- 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** y no podrá continuar con la implementación de la capa de persistencia.