UNIVERSIDAD POLITECNICA DE VALENCIA

Laboratorio

Pruebas

Proyectos de prueba unitaria utilizando el *framework* MSTest

Ingeniería del Software

ETS Ingeniería Informática

DSIC – UPV

Contenido

1. Objetivo

Este boletín explica cómo agregar un proyecto de prueba unitaria a tu solución, y cómo puedes añadir nuevas pruebas. En la primera sección se explica cómo agregar un proyecto de pruebas unitarias existente. En la segunda parte del boletín, se describen los principales aspectos del entorno de pruebas de Microsoft que necesitas conocer para crear y ejecutar tus propias pruebas unitarias.

2. Conectarse al Proyecto y recuperar el repositorio

Cada miembro del equipo puede conectarse desde Visual Studio al proyecto Azure DevOps, para clonar el repositorio remoto en el repositorio local, en caso de iniciar sesión en los equipos del laboratorio. Si se encuentra en un ordenador privado donde se clonó previamente el repositorio, solo necesita sincronizar para obtener los últimos cambios. La Figura 1 resume los pasos a seguir para conectar y clonar el repositorio (en caso de duda, consulte el seminario 2 o el boletín de la sesión de laboratorio 1).

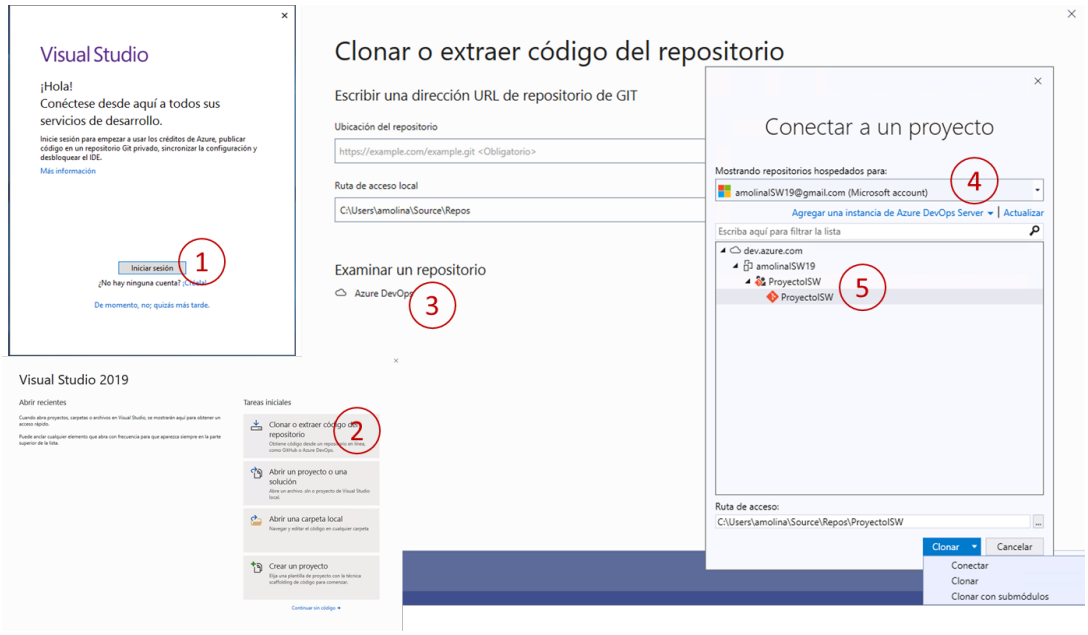


Figura 1. Pasos para conectarse y clonar el proyecto

3. Descargar y añadir el proyecto de pruebas unitarias.

Este paso debe ser realizado solo por un miembro del equipo, para evitar conflictos innecesarios en el repositorio.

3.1 Descargar el proyecto de pruebas unitarias.

En Poliformat, dentro de los recursos del laboratorio, hay una carpeta de pruebas en la que encontrarás diferentes proyectos de pruebas unitarias para probar diferentes aspectos de tu proyecto de laboratorio. El primero de ellos probará si has diseñado de manera adecuada las clases del diagrama de clases de diseño que te proporcionamos con respecto a la lógica de negocio del proyecto. Puedes encontrarlo en un archivo zip llamado GestDepLogicDesignTest.zip. Debes descargarlo y seguir estos pasos (Figure 2. Unzip and copy the projectFigure 2):

a) Descomprime el archivo GestDepLogicDesignTest.zip.
b) Copia la carpeta descomprimida en la carpeta del proyecto dentro de tu repositorio local.

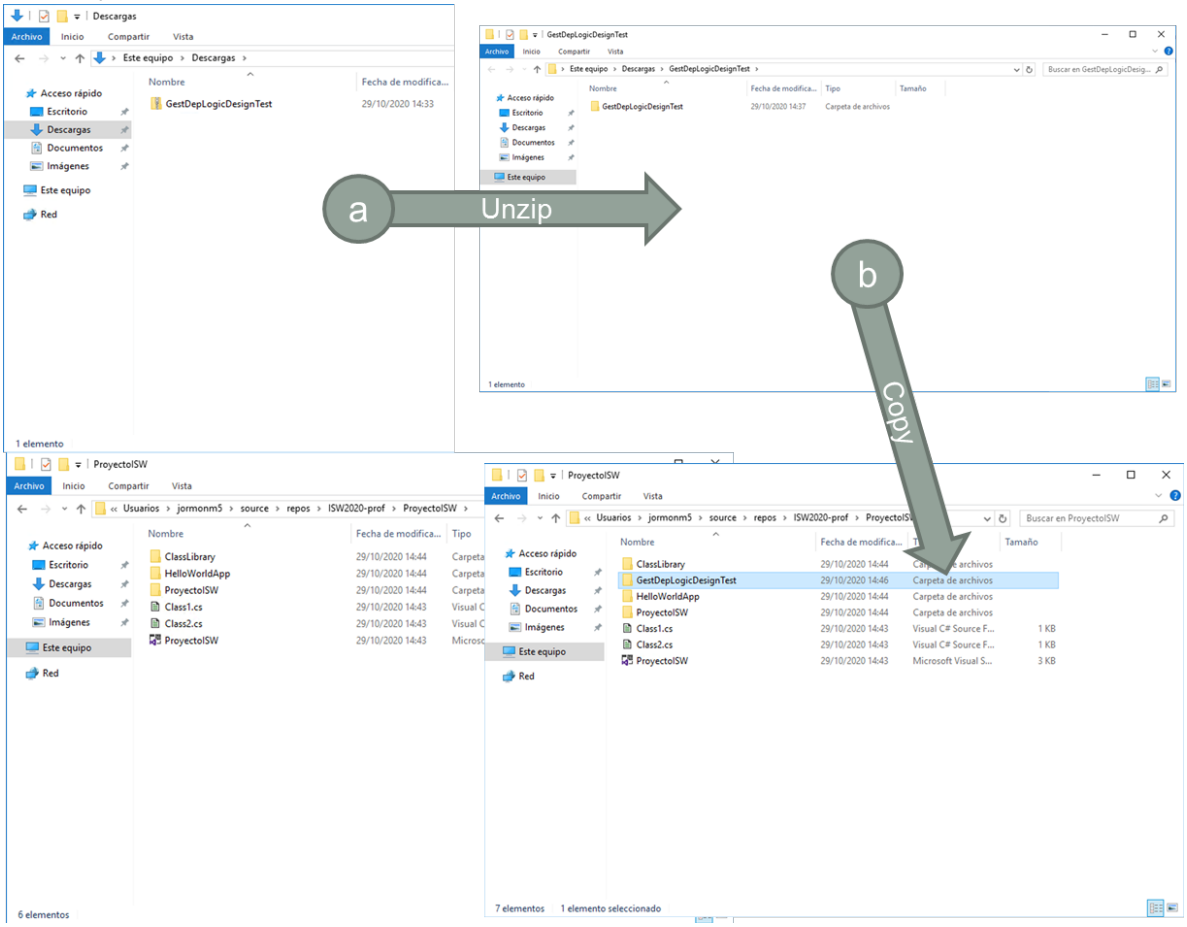


Figure 2. Unzip and copy the project

3.2 Agregar el proyecto de pruebas unitarias

Con el fin de agregar el proyecto de pruebas unitarias a la solución, debes abrir el explorador de soluciones en la vista de la solución (no en la vista de carpetas), como se puede ver en la (Figure 3-1), y seguir estos pasos

a) Seleccionar la carpeta de soluciones de Testing (Figure 3-1).
b) Botón derecho y seleccionar “Agregar” como en la Figure 3-2.
c) Seleccionar “Proyecto existente”, como en la Figure 3-3.

d) Seleccionar la carpeta con el Proyecto de pruebas unitarias y pulsar “Abrir” en el diálogo de selección de ficheros que aparecerá (Figura 4).

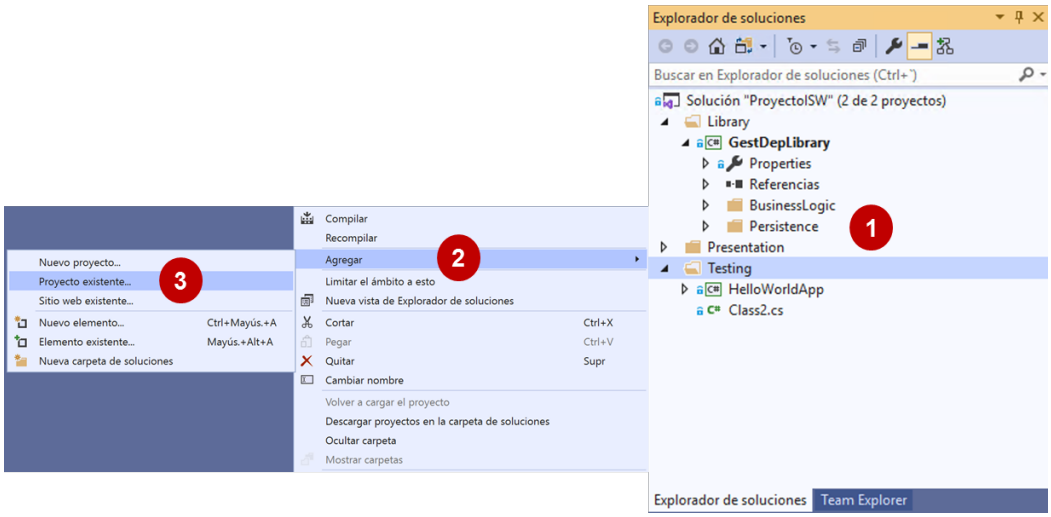


Figure 3. First steps for adding the unit test project

e) Seleccionar el fichero .csproj y pulsar “Abrir” de nuevo, again, como puedes ver en los pasos 3 y 4 de la Figura 4.

f) El proyecto de pruebas unitarias se añadirá a la solución en la carpeta de Testing, como puedes ver en la Figura 6.

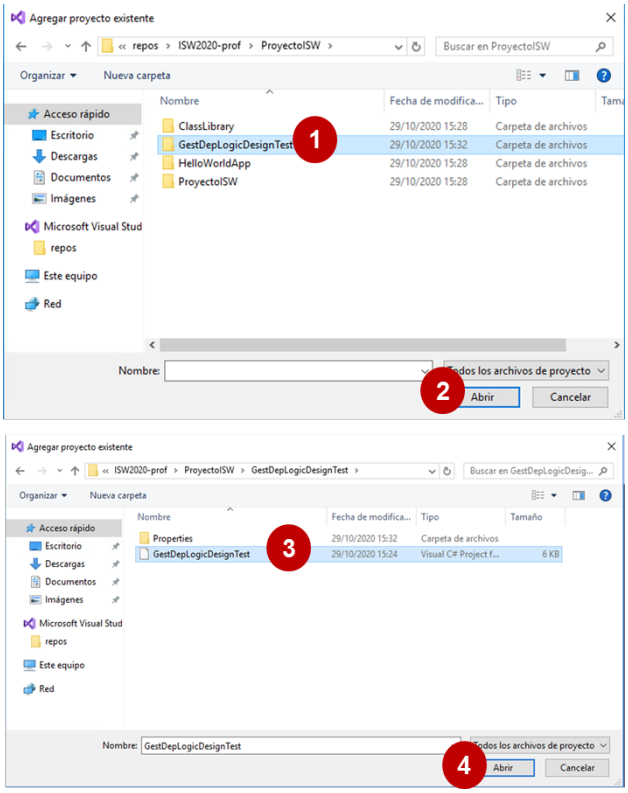


Figura 4. Seleccionar el proyecto de pruebas unitarias para ser añadido a la solución.

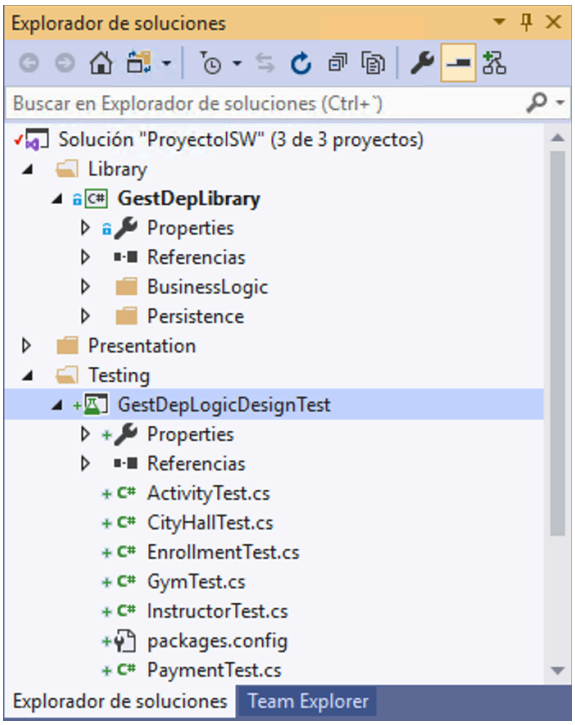


Figura 5. Proyecto de pruebas unitarias añadido a la solución.

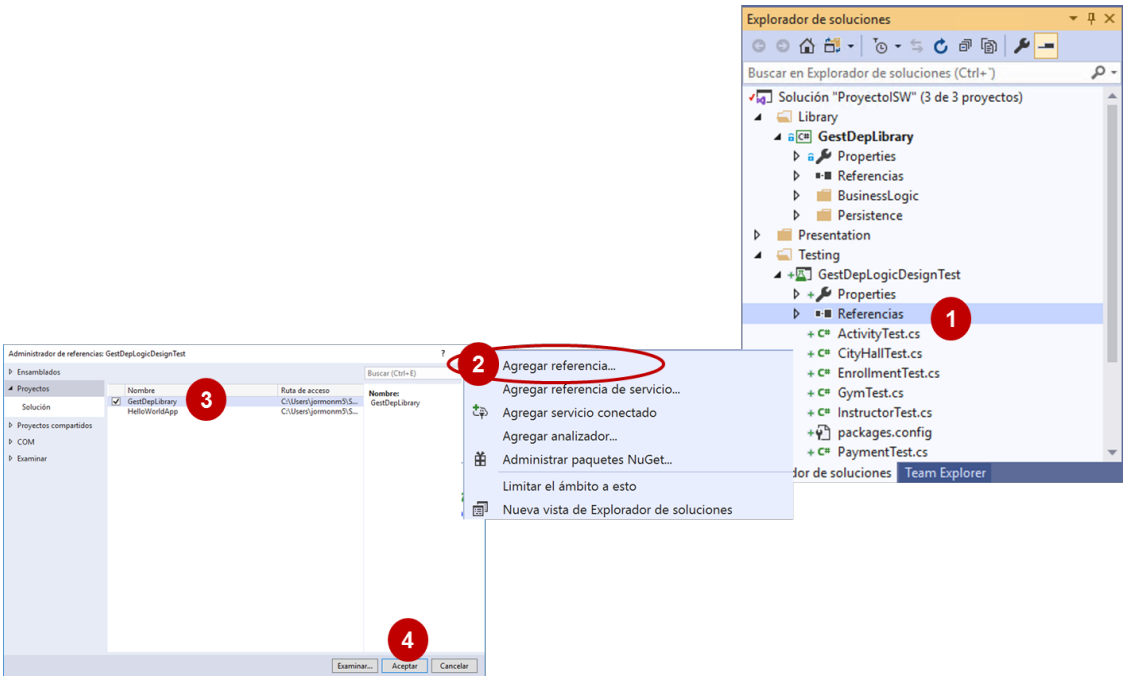


Figura 6. Añadiendo la referencia a la librería de clases que se va a probar

3.3 Configurar el proyecto de pruebas unitarias

Después de agregar el proyecto de prueba unitaria, es necesario comprobar si la referencia a la biblioteca de clases que queremos probar está bien establecida, desplegando el apartado “Referencias”. Si ve alguna advertencia junto al nombre de la clase de la biblioteca, debe agregar la referencia nuevamente. Siga estos pasos (Figura 7):

a) Seleccionar “Referencias”.
b) Seleccionar la opción “Agregar referencia”.
c) Pulsar sobre el nombre de tu librería de clases (GestDepLibrary in este ejemplo).
d) Pulsar el botón “Aceptar”.
e) Seleccionar el proyecto de test y, desde el menú contextual, la opción “Compilar” (ver Figura 7-2). Si aparecen errores de compilación, debes comprobar uno a uno el orden de los parámetros en los constructores que has definido. Modifica tu código para que el orden de los parámetros coincide con las cabeceras utilizadas en los test.
f) Si aparecen avisos sobre las referencias, pulsa el botón Refrescar (Figura 7-3). Ten paciencia, a veces necesita algún tiempo para actualizarse.

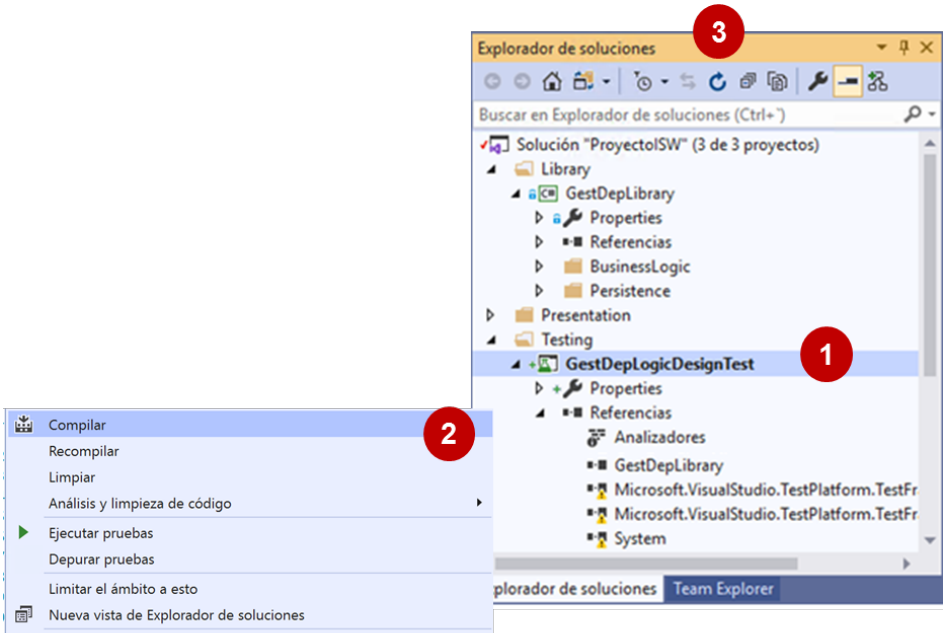


Figura 7. Compilar el proyecto de pruebas unitarias

3.4 Probar la librería de clases

Si despliegas el contenido del proyecto de prueba unitaria, podrás ver una clase de prueba unitaria para cada clase de la biblioteca. Además, verás otra clase llamada *TestData* que contiene los datos utilizados para crear y comparar valores en las pruebas.

Para cada clase, podrás ver dos pruebas: una para probar el constructor sin parámetros; y otra para probar el constructor con parámetros. Estas pruebas verifican si el orden de los parámetros es el esperado y si se han inicializado correctamente las propiedades de los objetos con ellos. Además, verifican si se han inicializado las listas.

Para ejecutar las pruebas, solo es necesario seleccionar el proyecto de prueba y luego la opción “Ejecutar pruebas” del menú contextual (ver Figura 8.). Puedes ver el resultado de las diferentes pruebas en la ventana “Explorador de pruebas”. Puedes abrirlo desde el menú Ver en caso de que no esté abierto de forma predeterminada.

Las pruebas que pasan todas las comprobaciones aparecen en verde en el explorador de pruebas; de lo contrario, cuando no superan ninguna marca, aparecen en rojo. Puedes seleccionar una prueba fallida para saber qué verificación ha fallado, y también, el mensaje de error proporcionado por los programadores de pruebas unitarias sobre el error. Hemos agregado diferentes consejos para ayudarte a corregir los errores en su código. Además, puede verse la línea que contiene la verificación fallida en la clase de prueba unitaria (consulte la Figura 8.).

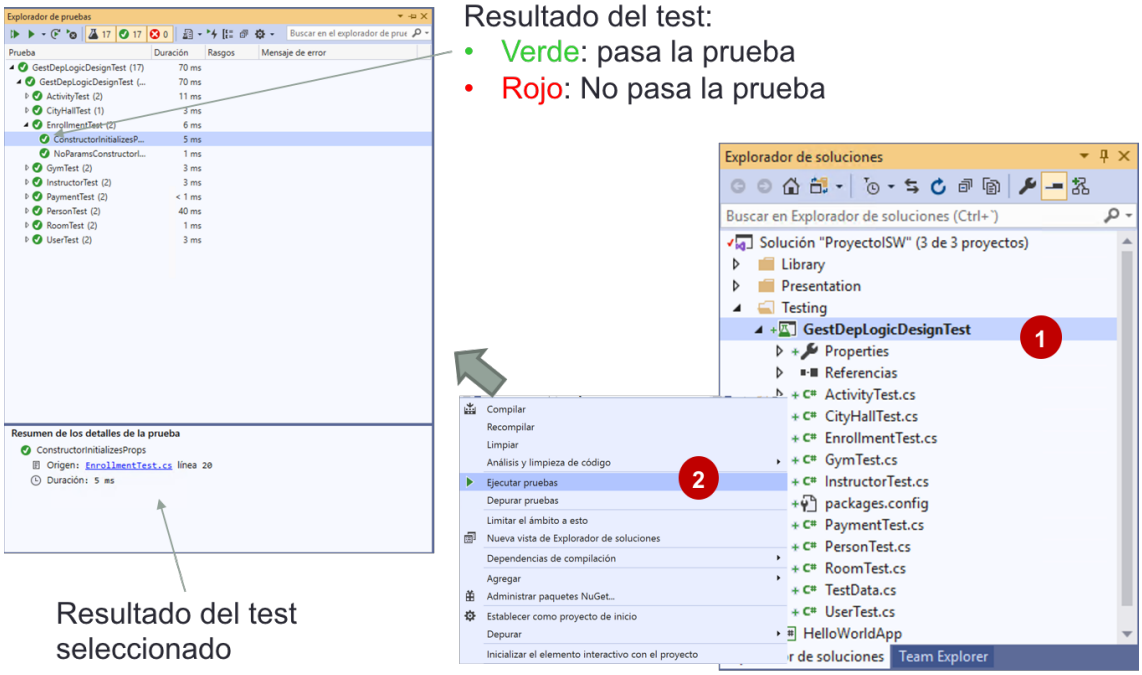


Figura 8. Ejecutando pruebas unitarias

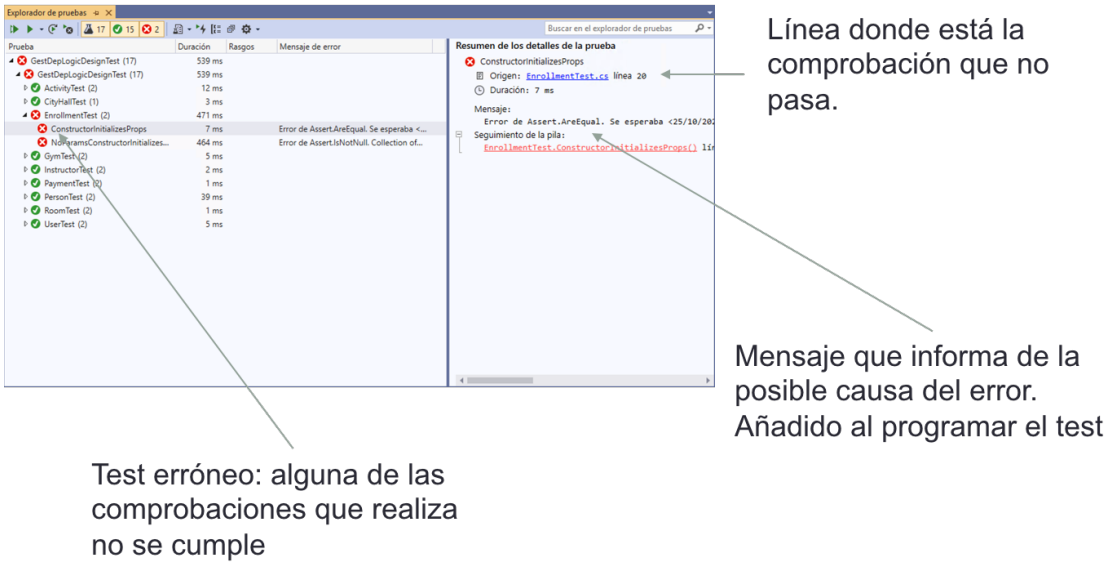


Figura 9. Prueba fallada: información proporcionada

4. Añadiendo nuevas pruebas

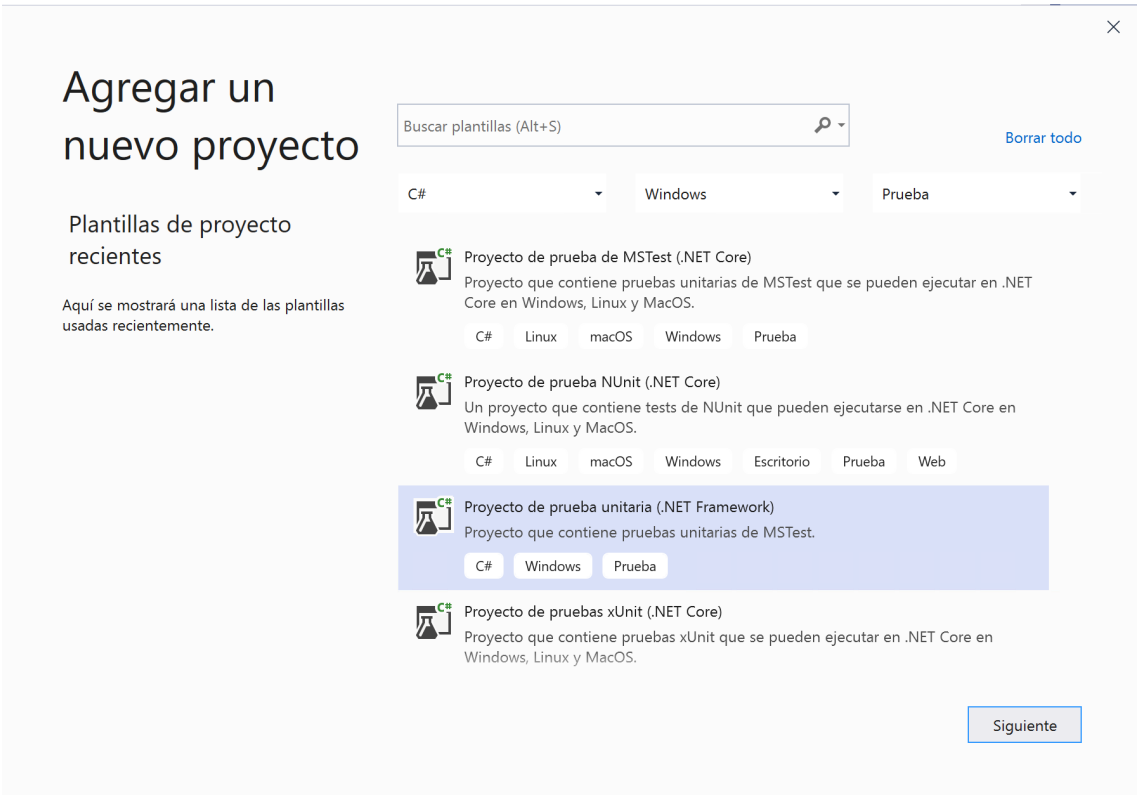


Figura 10. Proyecto de prueba unitaria .NET Framework

En Visual Studio, es posible definir y ejecutar pruebas unitarias para encontrar errores y asegurarse de que su código funcione correctamente mediante el marco de pruebas unitarias MSTest. Una buena política de desarrollo es ejecutar todas las pruebas unitarias del proyecto después de cualquier cambio en el código, de manera que cada miembro del equipo solamente sincronice (*push*) sus cambios cuando las pruebas unitarias se ejecuten sin errores.

Una prueba unitaria debe probar solo una funcionalidad de su código, evitando las interacciones entre otras funcionalidades tanto como sea posible. Por lo tanto, si la prueba detecta un error, puede asegurarse de que este error se debe a la funcionalidad que está probando y no al efecto de otra parte de su código.

En nuestro caso, cada método de nuestras clases representa una funcionalidad a probar. Por ejemplo, en el primer conjunto de pruebas unitarias que le proporcionamos, hemos agregado una prueba unitaria para cada constructor que se haya implementado.

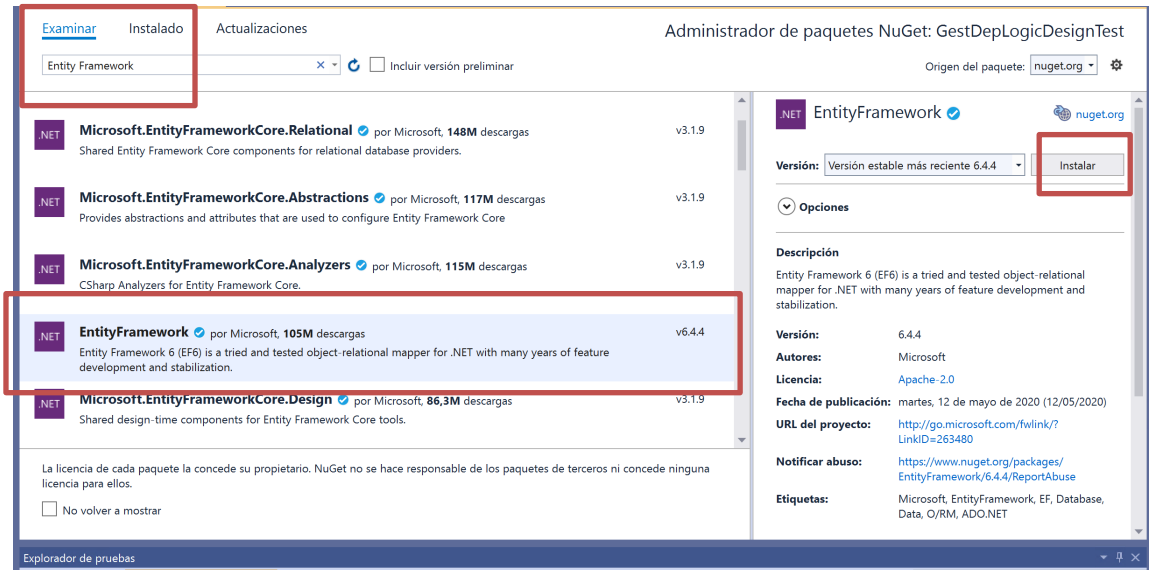
4.1 Añadiendo y configurando un proyecto de prueba unitaria.

MSTest[1] proporciona diferentes marcos para probar los diferentes tipos de proyectos que puede crear en Visual Studio. Por lo tanto, debe seleccionarse como proyecto de prueba unitaria el compatible con el proyecto que está desarrollando. En nuestro caso, debemos seleccionar como tipo de proyecto de prueba unitaria .NET, como el seleccionado en la Figura 10.

[1] https://docs.microsoft.com/es-es/visualstudio/test/getting-started-with-unit-testing?view=vs-2019

Una vez creado, debes añadir la referencia al proyecto que se quiere probar con la opción "`Añadir Referencia`", seleccionando el proyecto o librería de clases que contiene el código que quieres probar, como explicamos en la sección 3.3. Además, también tendrás que añadir todos los paquetes *NuGet* necesarios para ejecutar tu código.

Por ejemplo, en caso de que agregue pruebas para verificar la implementación de la capa de persistencia de los proyectos de laboratorio, deberá agregar el paquete NuGet Entity Framework. Para agregar paquetes NuGet, puede seleccionar el proyecto y luego seleccionar la opción "`Administrar paquetes NuGet`" del menú contextual. Puede buscar el paquete NuGet requerido desde la pestaña "`Explorar`" y luego instalarlo haciendo clic en el botón "Instalar" (instalar) , como puede ver en la Figurea 11.



Figurea 11. Añadiendo el paquete NuGet EntityFramework al proyecto

Finalmente, si utilizas EntityFramework, necesitarás también añadir la cadena de conexión a la base de datos gestionada por la capa de persistencia en el archivo app.config, por ejemplo:

```
  <connectionStrings>
    <clear />
    <add name="GestDepDBConnection"
connectionString="Server=(localdb)\mssqllocaldb;Database=GestDepDB;Trusted_Connection=True;MultipleActiveResultSets=true" providerName="System.Data.SqlClient" />
  </connectionStrings>
```

4.2 Añadiendo una prueba unitaria al proyecto

Puedes organizar tus pruebas unitarias por clase, servicio o función de su proyecto, de manera que puedes añadir tantas clases de pruebas unitarias a tu programa como necesites para probar los diferentes métodos. Como ejemplo, hemos proporcionado un proyecto de prueba de unidad organizado por clases, en el que puedes ver tantas clases de prueba unitaria como clases tenemos en el proyecto.

Para añadir una nueva clase de prueba unitaria, solo tienes que seleccionar el proyecto y escoger la opción "`Agregar>Prueba Unitaria`", como muestra la Figura 12.

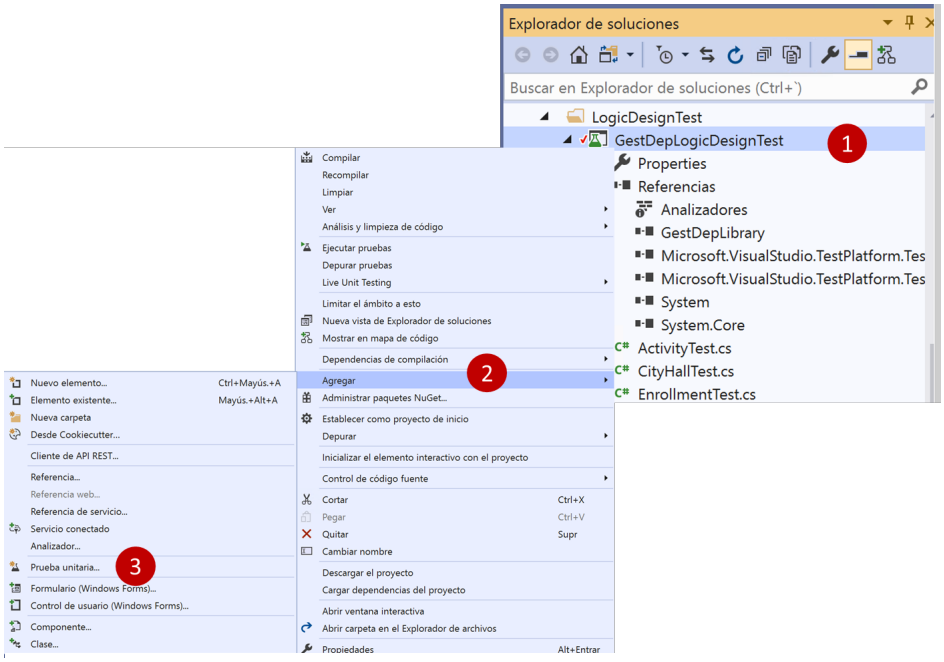


Figura 12. Añadiendo una clase de test unitaria en el proyecto de prueba.

Se añade una nueva clase al proyecto de prueba con el código que se observa a continuación:

```
using System;
using Microsoft.VisualStudio.TestTools.UnitTesting;

namespace GestDepLogicDesignTest
{
    [TestClass]
    public class UnitTest1
    {
        [TestMethod]
        public void TestMethod1()
        {
        }
    }
}
```

Cada método que se agrega con el decorador `[TestMethod]` en la parte superior se interpreta como una prueba realizada dentro del conjunto de pruebas. El entorno lo ejecutará de forma independiente.

4.3 Escribiendo pruebas unitarias

Como puedes encontrar en la documentación[2] de Microsoft, las pruebas unitarias siguen normalmente el patrón AAA Arrange-Act-Assert:

- **Sección Arrange**: inicializa los objetos y establece los valores de los datos que se pasan al método que se va a probar.
- **Sección Act**: invoca el método en pruebas con los parámetros organizados.
- **Sección Assert**: comprueba si la acción del método en pruebas se comporta de la forma prevista.

[2] https://docs.microsoft.com/es-es/visualstudio/test/unit-test-basics?view=vs-2019

Puedes observar estas tres secciones AAA en el siguiente código de ejemplo:

```
[TestMethod]
public void ConstructorInitializesProps()
{   //Arrange
   const string EXPECTED_PERSON_ID = "94814560G";
   const string EXPECTED_PERSON_ADDRESS = "Camí de Vera";
   const string EXPECTED_PERSON_NAME = "Nom de prova";
   const string EXPECTED_PERSON_IBAN = "ES6621000418401234567891";
   const int EXPECTED_PERSON_ZIP_CODE = 46021;

   //Act
   Person person = new Person(EXPECTED_PERSON_ADDRESS, EXPECTED_PERSON_IBAN,
EXPECTED_PERSON_ID, EXPECTED_PERSON_NAME, EXPECTED_PERSON_ZIP_CODE);

   //Assert
    Assert.AreEqual(EXPECTED_PERSON_ADDRESS, person.Address, "Address not
properly initialized. Check the order of the parameters and the assignment.");
    Assert.AreEqual(EXPECTED_PERSON_ID, person.Id, "Id not properly initialized.
Check the order of the parameters and the assignment.");
    Assert.AreEqual(EXPECTED_PERSON_IBAN, person.IBAN, "IBAN not properly
initialized. Check the order of the parameters and the assignment.");
    Assert.AreEqual(EXPECTED_PERSON_NAME, person.Name, "Name not properly
initialized. Check the order of the parameters and the assignment.");
    Assert.AreEqual(EXPECTED_PERSON_ZIP_CODE, person.ZipCode, "Zip code not
properly initialized. Check the order of the parameters and the assignment.");
}
```

Tenemos disponible varios tipos de Assert, pero los más utilizados son:

- `Assert.AreEqual(ExpectedValue, ResultValue)`
- `Assert.IsNull(ResultValue)`
- `Assert.IsNotNull(ResultValue)`
- `Assert.IsTrue(BooleanExpression)`
- `Assert.IsFalse(BooleanExpression)`
- `Assert.Fail()`: causa que la prueba falle
- `Assert.TrowsException`: comprueba que una excepción que se espera que sea lanzada por el método en pruebas, realmente se lance.

Así, una prueba unitaria falla si alguno de los *asserts* definidos no se cumple. Cuando un assert falla, el resto no se ejecuta.

4.4 Refactorizando pruebas

En la mayoría de los casos, es necesario inicializar los mismos objetos y establecer el mismo valor de los datos que se pasan a los métodos bajo prueba. Por lo tanto, es posible escribir la sección *Arrange* solo una vez, refactorizando nuestras pruebas. Por ejemplo, en la prueba del proyecto que se le proporcionó, es posible encontrar una clase llamada DataTest, que crea todos los objetos y establece los valores de diferentes variables estáticas que luego se utilizan en todas las pruebas unitarias.

Además, a veces es necesario realizar una configuración inicial de la prueba para garantizar que el resultado no se vea afectado por pruebas anteriores (por ejemplo, limpiar una base de datos). De la misma forma, quizás sea necesario realizar alguna acción para limpiar el entorno de ejecución. El *framework* MSTest nos ayuda a través de dos decoradores diferentes:

```
[TestInitialize]
[TestCleanup]
```

Si sitúas un decorador `TestInitialize` en la parte superior de un método en una clase de unidad, este método se ejecutará antes de cada ejecución de prueba de unidad. De manera similar, cuando decoras un método con `TestCleanup`, este método se ejecuta después de cada método de prueba de la clase. Por ejemplo, en el siguiente código se diseña una clase de prueba base, que define un método `TestInitialize` que creará un objeto DAL antes de cada prueba unitaria. También contiene un método `TestCleanup` que eliminará todos los datos agregados a la base de datos después de cada prueba unitaria. Esta clase `BaseTest` no contiene ningún método, pero se puede usar para extenderla y crear una clase de prueba con ambos métodos definidos.

```
    [TestClass]
    public class BaseTest
    {
        protected private EntityFrameworkDAL dal;

        [TestInitialize]
        public void IniTests()
        {
            dal = new EntityFrameworkDAL(new GestDepDbContext());
            dal.RemoveAllData();


        }
        [TestCleanup]
        public void CleanTests()
        {
            dal.RemoveAllData();
        }

    }
```

En este otro ejemplo, se crea una clase de prueba extendiendo `BaseTest`. Contiene un método de prueba unitario llamado `StoresInitialData()` que prueba si el objeto `persona` persiste correctamente. Tenga en cuenta que también se ejecutará el método InitTests () antes que `StoresInitialData()` y `CleanTests()` se ejecutará después.

```
public class PersonPersistenceTest : BaseTest
{
   [TestMethod]
   public void StoresInitialData()
  {
      /*Arrange
            - BaseTest.IniTests() is run before this code
            - Static Objects and variables used to create the object are inicialized
in the class TestData
      */

      //Act
       Person person = new Person(TestData.EXPECTED_PERSON_ADDRESS,
TestData.EXPECTED_PERSON_IBAN, TestData.EXPECTED_PERSON_ID,
TestData.EXPECTED_PERSON_NAME, TestData.EXPECTED_PERSON_ZIP_CODE);

      dal.Insert(person);
      dal.Commit();

     //Assert
     Person personDAL = dal.GetAll<Person>().First();
    Assert.AreEqual(TestData.EXPECTED_PERSON_ADDRESS, personDAL.Address, "Address not properly
stored.");
```

```
      Assert.AreEqual(TestData.EXPECTED_PERSON_ID, personDAL.Id, "Id not properly stored. ");
      Assert.AreEqual(TestData.EXPECTED_PERSON_IBAN, personDAL.IBAN, "IBAN not properly stored.
");
      Assert.AreEqual(TestData.EXPECTED_PERSON_NAME, personDAL.Name, "Name not properly stored.
");
      Assert.AreEqual(TestData.EXPECTED_PERSON_ZIP_CODE, personDAL.ZipCode, "Zip code not
properly stored.");

       //After assert: BaseTest.CleanTests() is run
   }
}
```