Pruebas unitarias

Índice de contenido

Si es un programador que utiliza Visual Studio Professional Edition, puede crear y ejecutar dos tipos de pruebas: unitarias y por orden. Una prueba unitaria se utiliza para comprobar que un método concreto del código de producción funciona correctamente, probar las regresiones o realizar pruebas relacionadas (buddy) o de humo. Una prueba por orden se utiliza para ejecutar otras pruebas en un orden especificado.

Visual Studio nos proporciona un sistema sencillo para poder crear nuestras pruebas unitarias. Nos facilita la creación de los proyectos de pruebas y nos genera la estructura básica que tiene que tener la prueba

Los evaluadores del equipo pueden utilizar Herramientas para pruebas Team System para crear y ejecutar pruebas. Si ejecutan una prueba unitaria en la que se produce un error, almacenan un error y se lo asignan a usted. A continuación, usted puede utilizar Visual Studio para reproducir el error ejecutando la prueba unitaria que no se ha superado.

Características de una buena prueba unitaria

Las pruebas unitarias se tienen que poder ejecutar **sin necesidad de intervención manual**. Esta característica posibilita que podamos automatizar su ejecución.

Las pruebas **unitarias tienen que poder repetirse tantas veces como uno quiera**. Por este motivo, la rapidez de las pruebas tiene un factor clave. Si pasar las pruebas es un proceso lento no se pasarán de forma habitual, por lo que se perderán los beneficios que éstas nos ofrecen.

Las pruebas unitarias deben poder **cubrir casi la totalidad del código** de nuestra aplicación. Una prueba unitaria será tan buena como su cobertura de código. La cobertura de código marca la cantidad de código de la aplicación que está sometido a una prueba. Por tanto, si la cobertura es baja, significará que gran parte de nuestro código está sin probar.

Las pruebas unitarias tienen que poder **ejecutarse independientemente** del estado del entorno. Las pruebas tienen que pasar en cualquier ordenador del equipo de desarrollo.

La ejecución de una prueba **no puede afectar la ejecución de otra**. Después de la ejecución de una prueba el entorno debería quedar igual que estaba antes de realizar la prueba.

Las diferentes **relaciones** que puedan existir entre módulos deben ser simulada para evitar dependencias entre módulos.

Es importante **conocer claramente cuál es el objetivo** del test. Cualquier desarrollador debería poder conocer claramente cuál es el objetivo de la prueba y su funcionamiento. Esto sólo se consigue si se trata el código de pruebas como el código de la aplicación.

Es importante tener en cuenta que aunque estas son las características de una buena prueba, no siempre será posible ni necesario cumplir con todas estas reglas y será la experiencia la que nos guiará en la realización de las mismas

Beneficios de las pruebas unitarias

Con las pruebas unitarias todos ganan. La **vida de desarrollador será mucho más fácil**, ya que la calidad de su código mejorará, se reducirán los tiempos de depuración y la corrección de incidencias y por tanto el cliente estará mucho más contento porque la aplicación hace lo que él quiere que haga, por lo que ha pagado.

Las pruebas **fomentan el cambio y la refactorización**. Si consideremos que nuestro código es mejorable podemos cambiarlo sin ningún problema. Si el cambio no estuviera realizado correctamente las pruebas nos avisarán de ello. Seguramente la frase “si funciona no lo toques” a más de uno familiar les resultará familiar. Si hubiera pruebas unitarias, no sería necesario pronunciarla.

Se reducen drásticamente los problemas y tiempos dedicados a la integración. En las pruebas se simulan las dependencias lo que nos permite que podemos probar nuestro código sin disponer del resto de módulos. Por experiencia puede decir que los procesos de integración son más de una vez traumáticos, dejándolos habitualmente para el final del proyecto. La frase “sólo queda integrar” haciendo referencia a que el proyecto está cerca de terminar suele ser engañosa, ya que el periodo de integración suele estar lleno de curvas.

Las pruebas **nos ayudan a entender mejor el código**, ya que sirven de documentación. A través de las pruebas podemos comprender mejor qué hace un módulo y que se espera de él.

Nos permite poder **probar o depurar un módulo sin necesidad de disponer del sistema completo**. Aunque seamos los propietarios de toda la aplicación, en algunas situaciones montar un entorno para poder probar una incidencia es más costoso que corregir la incidencia propiamente dicha. Si partimos de la prueba unitaria podemos centrarnos en corregir el error de una forma más rápida y lógicamente, asegurándonos posteriormente que todo funciona según lo esperado.

Mitos de las pruebas unitarias

Aunque los beneficios de las pruebas unitarias puedan parecer claros, no es más cierto que a día de hoy se usan en muy pocos proyectos. Pero sin son tan buenas,**¿por qué no se usan?**

Pues la razón principal es que existen bastante desconocimiento en esta materia, poca tradición y algunos falsos mitos.

Uno de los mitos es creer que escribir pruebas unitarias es escribir el doble de código; escribir el código de la aplicación y escribir el código de pruebas. Escribir una prueba nunca es escribir el doble de código, aunque lógicamente sí es escribir más código. El mito es totalmente falso.

Las pruebas siempre se deben ir escribiendo a medida que se desarrolla el software. A medida de desarrollamos vamos probando nuestro código lo que nos permite asegurar que el módulo queda terminado correctamente, libre de incidencias.

La realización de pruebas unitarias debe ser un proceso obligatorio en nuestros desarrollos y que no queden a la voluntad del desarrollador. Si el desarrollador no está habituado a su uso diario es muy fácil que tienda a evitar realizar este tipo de pruebas.

Si no estás familiarizado con el uso de pruebas unitarias la perseverancia debe ser tu gran aliado. Debes estar convencido de que el uso de pruebas unitarias es el enfoque correcto y no ser flexible; las pruebas unitarias no son opcionales y deben realizarse a medida que se desarrolla.

Dejar la implementación de pruebas para el final no es realista, ya que las pruebas nunca se llegarán a implementar y si lo hacen, serán de una baja calidad, ya que la persona que las realiza considerará que es una pérdida de tiempo, considerando que su tarea ya estaba terminada sin realizar estas pruebas.

¿Y si no escribiéramos este código cómo probamos? Hay diferentes situaciones.

La primera situación es que no se prueba. Se implementa el módulo pero no se prueba, dando por hecho que nuestra experiencia como desarrolladores hará que las incidencias sean mínimas. Cuando tenemos todos los módulos lo probamos de forma integrada a través de pruebas funcionales.

Esta situación es la situación peor con la que podemos encontrarnos, pero desgraciadamente es la más habitual. Los tiempos de integración, depuración y corrección de incidencias se multiplican. El tiempo que va desde que consideramos que el producto está terminado hasta que realmente está terminado puede llegar a ser superior al tiempo invertido en el desarrollo, y lo digo por experiencia, por mala experiencia.

Otra situación algo mejor, es el uso de aplicaciones de pruebas de usar y tirar. Todos hemos implementado aplicaciones tontas, generalmente de consola, para probar el módulo que estamos desarrollando y que posteriormente integraremos en un sistema mayor.

La situación no es la ideal pero realmente el desarrollo de pruebas unitarias puede tener una semejanza con este tipo de aplicaciones, siendo éste último concepto más avanzado y mejorado para que pueda ser reutilizable, independientes y completas.

Estas aplicaciones de usar y tirar nos valen para la primera vez y si somos exhaustivos nos permitirá probar bien nuestro módulo. La realidad nos dice que esta aplicación se tira y que ante cualquier modificación en el módulo nos comportamos como en la situación primera, en las pruebas con sistema completa y pruebas funcionales.

Otro mito, unido al anterior, es que desarrollar pruebas unitarias **hace que los tiempos de desarrollo se incrementen**, incrementando los costes del proyecto.

Entre desarrollar y no probar y desarrollar haciendo pruebas unitarias, reconoceré que es más rápido desarrollar sin probar…..pero creo que esto no nos vale….así que entre desarrollar con pruebas unitarias y desarrollar probando con métodos más rústicos, ésta segunda es más lenta y por tanto, más cara.

La manera más rápida de desarrollar software es desarrollarlo bien. Como hemos comentado en el punto anterior, si no desarrollamos código de calidad y no realizamos pruebas, los tiempos de desarrollo se podrán disparar enormemente; tiempo de integración, tiempo de depuración, tiempo de incidencias etc…

Pero hay un problema mayor; la pérdida de confianza del cliente. Un producto de baja calidad, lleno de incidencias es la peor tarjeta de presentación.

Por otro lado, hay que tener en cuenta que cuando más larga sea la vida de la aplicación más beneficios se obtendrán del sistema de pruebas unitarias.

Muchos productos tienen un mantenimiento evolutivo que implica el desarrollo de nuevas funcionalidad y la corrección de incidencias. Sin un proceso adecuado de pruebas unitarias, los tiempos de pruebas se dispararán en fases posteriores, ya que será necesario probar y corregir tanto las nuevas funcionalidades como las funcionalidades ya existentes.

Cierto es también que la adopción de pruebas unitarias dentro del equipo de desarrollo implica una inversión en formación. La primera vez que las empleemos llevará algo más de tiempo ya que tenemos que familiarizarnos con las herramientas y con el framework de pruebas unitarias. Este tiempo es algo habitual asociado a cualquier proceso de aprendizaje que sólo sería imputable al primer proyecto que se desarrolle.

Prueba de método

Supondremos que tenemos una clase que tiene un método suma. Este método, recibe como parámetro dos valores enteros y devuelve la suma ambos. Sí, el ejemplo es un poco tonto pero la idea es dar a conocer la herramienta, tiempo habrá para complicar el ejemplo.

```
private int sumar(int a, int b)
    {
        return (a + b);
    }
```

También se puede hacer desde el menú **Prueba → Nueva prueba**.

Una vez implementado el método necesitamos probarlo. Para ello creamos nueva primera prueba unitaria. **Si seleccionamos el método Sumar, en el menú contextual tendremos la opción de crear una prueba unitaria.***(Ilustración 1).*

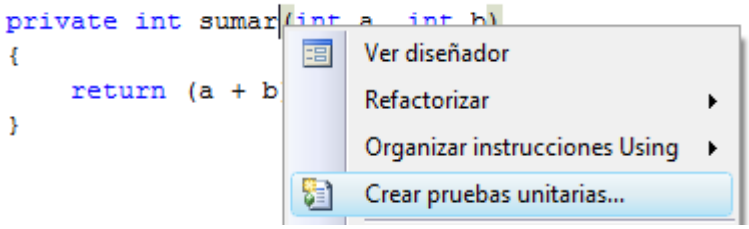

Ilustración 1: Prueba unitaria de método

La prueba la podremos crear en un nuevo proyecto de test o en uno ya existente. Como es nuestra primera prueba, lógicamente, lo incluiremos en un nuevo proyecto de test. (*Ilustración 2*).

Crearemos al menos un proyecto de Test por cada proyecto que tengas en tu aplicación, en lugar de crear un único proyecto de test que englobe todos los las pruebas de tu aplicación.

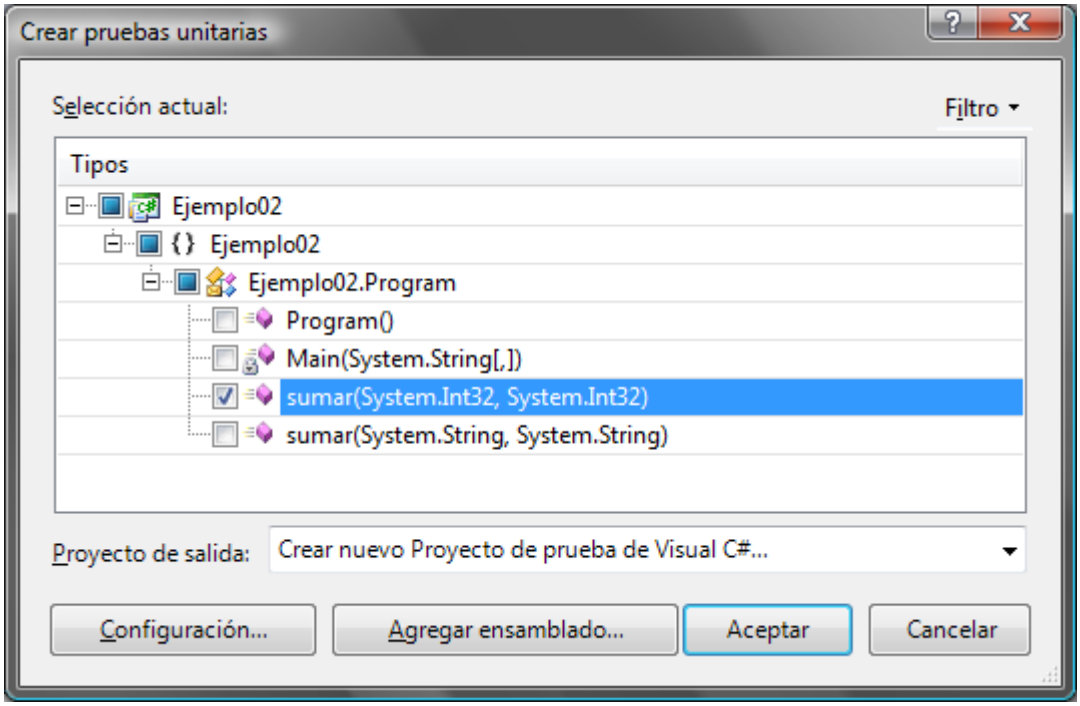

Ilustración 2: Crear prueba unitaria

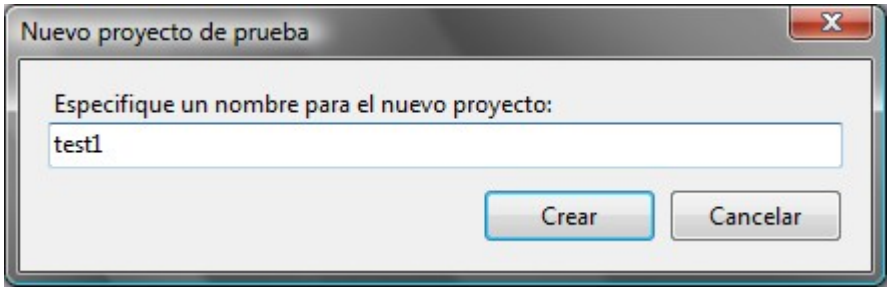

Ilustración 3: Proyecto de pruebas

El nuevo proyecto que se crea es un proyecto normal y corriente. La única peculiaridad que tiene una referencia de **Microsoft.VisualStudio.QualityTools.UnitTestFramework**. Si nos fijamos en el Explorador de soluciones podremos ver el nuevo proyecto que se ha agregado a nuestra solución. (*Ilustración 4*)

En este paso no se crea la prueba, sólo la estructura de proyectos, clases y métodos. Es labor de desarrollador, en base a las especificaciones del módulo, el desarrollo de las pruebas. Es importante que el desarrollador realice tantas pruebas como sea necesario para cubrir al máximo la funcionalidad del módulo que se prueba.

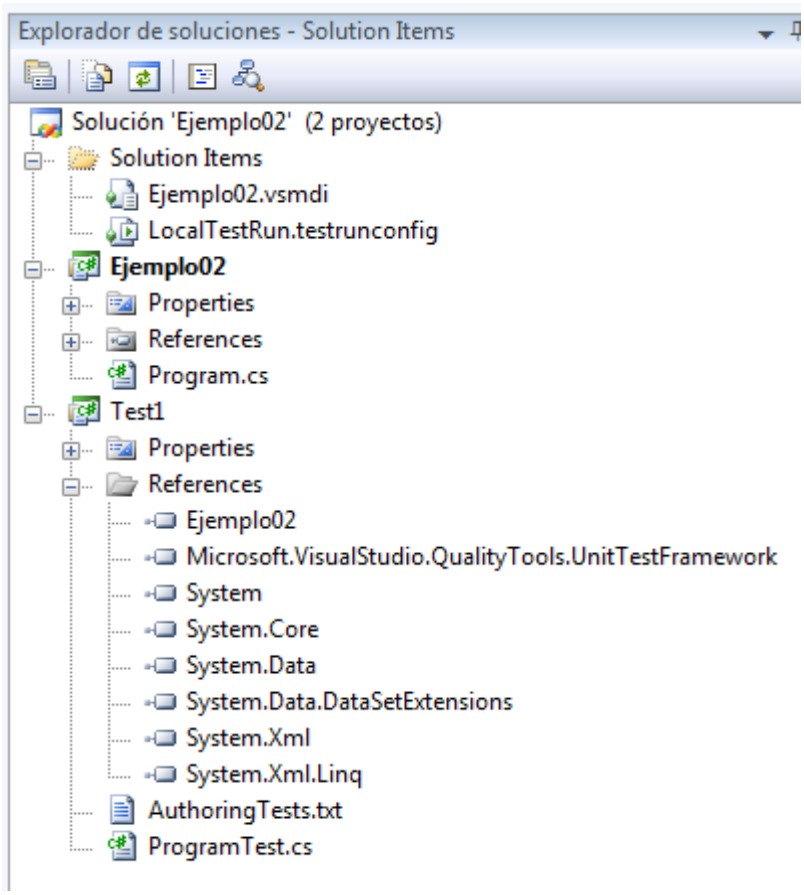

Ilustración 4: Proyecto de pruebas

Fijaros en el código del método de prueba que tanto la clase como el método son clases y métodos normales, con la única peculiaridad que el método está decorado con el atributo **TestMethod**.

```
/// <summary>
///Una prueba de sumar
///</summary>
[TestMethod()]
public void sumarTest()
{
  Program target = new Program(); // TODO: Inicializar en un valor adecuado
  int a = 0; // TODO: Inicializar en un valor adecuado
  int b = 0; // TODO: Inicializar en un valor adecuado
  int expected = 0; // TODO: Inicializar en un valor adecuado
  int actual;
  actual = target.sumar(a, b);
  Assert.AreEqual(expected, actual);
  Assert.Inconclusive("Compruebe la exactitud de este método de prueba.");
}
```

Cuando hemos creado el proyecto de Test, en la solución han aparecido dos nuevos ficheros: **Sample.vsmdi** y **LocalTestRun.testrunconfig**.

Si hacemos doble-click sobre el archivo vsmdi podremos acceder al editor de la lista de pruebas unitarias. (*Ilustración 5*). Desde aquí podremos ver, lanzar o depurar todas las pruebas unitarias que contiene la solución. Y digo que contiene la solución, porque este fichero es común para todos los proyectos de Test de la solución. Si éste contiene 2 proyectos de Test, desde la lista de pruebas unitarias veremos todas las pruebas de los dos proyectos.

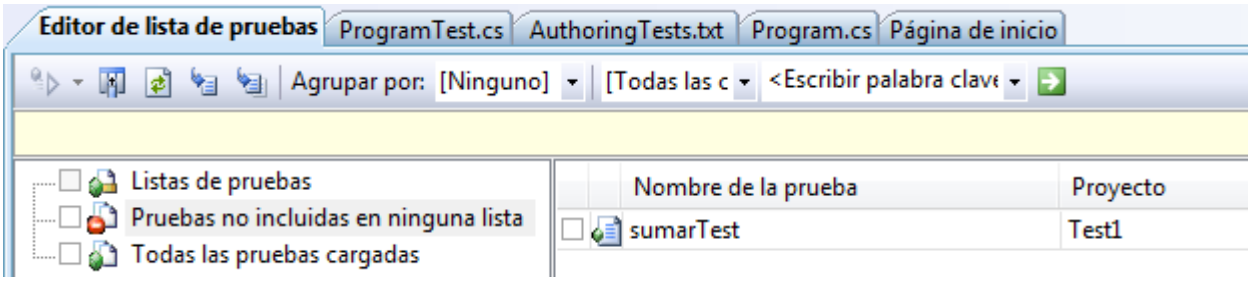

Ilustración 5: Editor de lista de pruebas

El fichero LocalTestRun.testrunconfig lo que nos va a permitir es configurar algunos aspectos relacionados con la ejecución de las pruebas, como activar el análisis de la cobertura de código. (*Ilustración 6*)

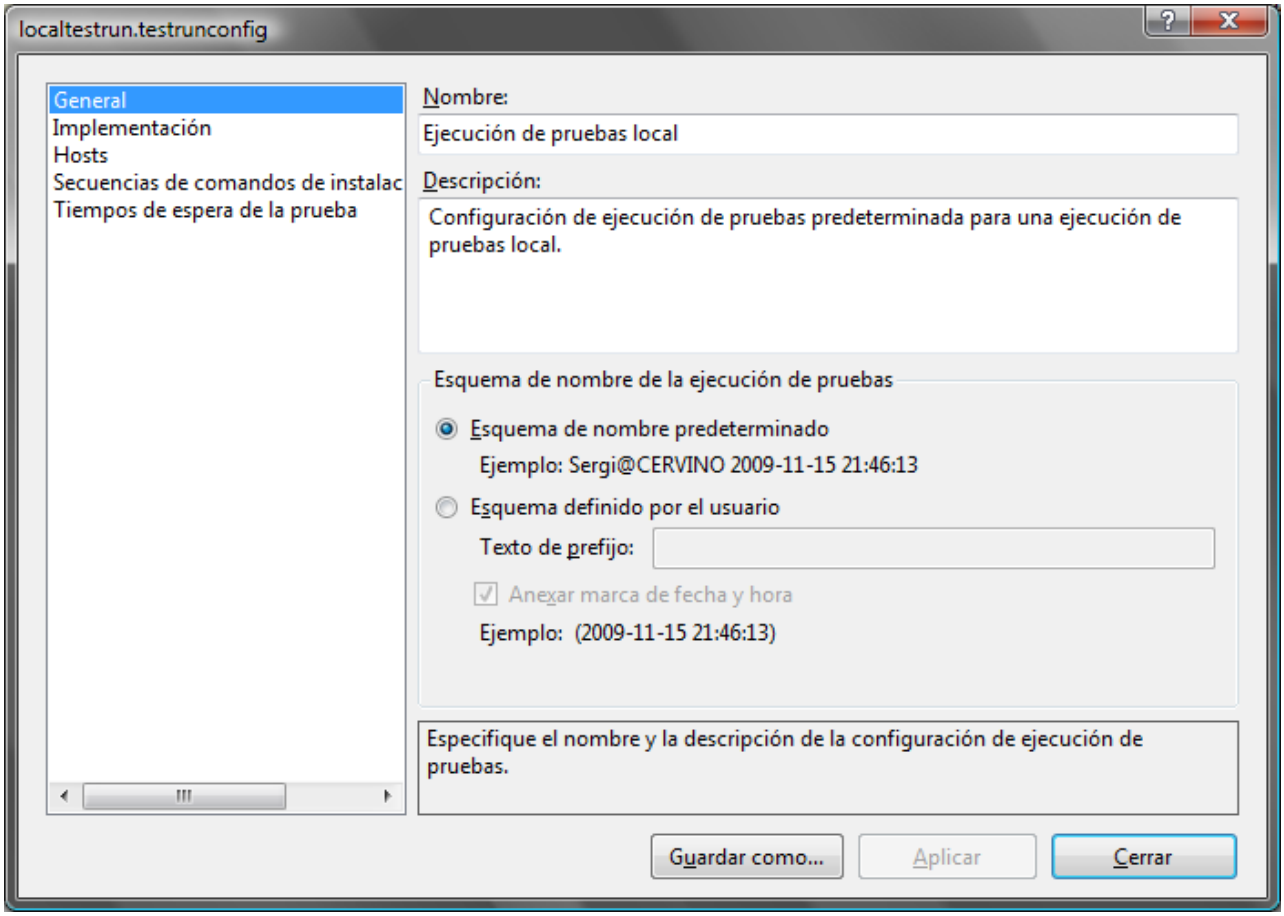

Ilustración 6: Configurar pruebas

Una vez que tenemos la estructura de nuestro método de prueba tendremos que implementar la prueba.

La idea es muy sencilla. Si pasamos 1 y 2 en los parámetros de entrada el método sumar nos tiene que devolver 3. Es decir, sabiendo los parámetros de entrada y lo que hace el método que estamos probando, sabremos lo que tiene que devolver.

El objetivo de la prueba es comprobar que el método Sumar hace exactamente lo que se espera de él. Si el método Sumar en lugar de 3 devolviese 7, porque se ha equivocado al sumar, la prueba unitaria fallaría.

Una vez hecha prueba recuerda eliminar la sentencia **Assert.Inconclusive**. Esta sentencia se incluye cuando se crea la prueba e indica que el resultado de la prueba no es concluyente. De esta manera se marcan las pruebas que no están implementados.

```
[TestMethod()]
public void sumarTest()
{
    Program target = new Program(); // TODO: Inicializar en un valor adecuado
    int a = 1; // TODO: Inicializar en un valor adecuado
    int b = 2; // TODO: Inicializar en un valor adecuado
    int expected = 3; // TODO: Inicializar en un valor adecuado
    int actual;
    actual = target.sumar(a, b);
    Assert.AreEqual(expected, actual);
}
```

Una vez, hecha la prueba sólo tenemos que ejecutarla. Desde la ventana de lista de pruebas podemos ejecutar todas las pruebas o sólo aquellas que seleccionemos. En la parte inferior, en la ventana de resultados de la prueba veremos el resultado de la ejecución de los mismos. (*Ilustración 7*)

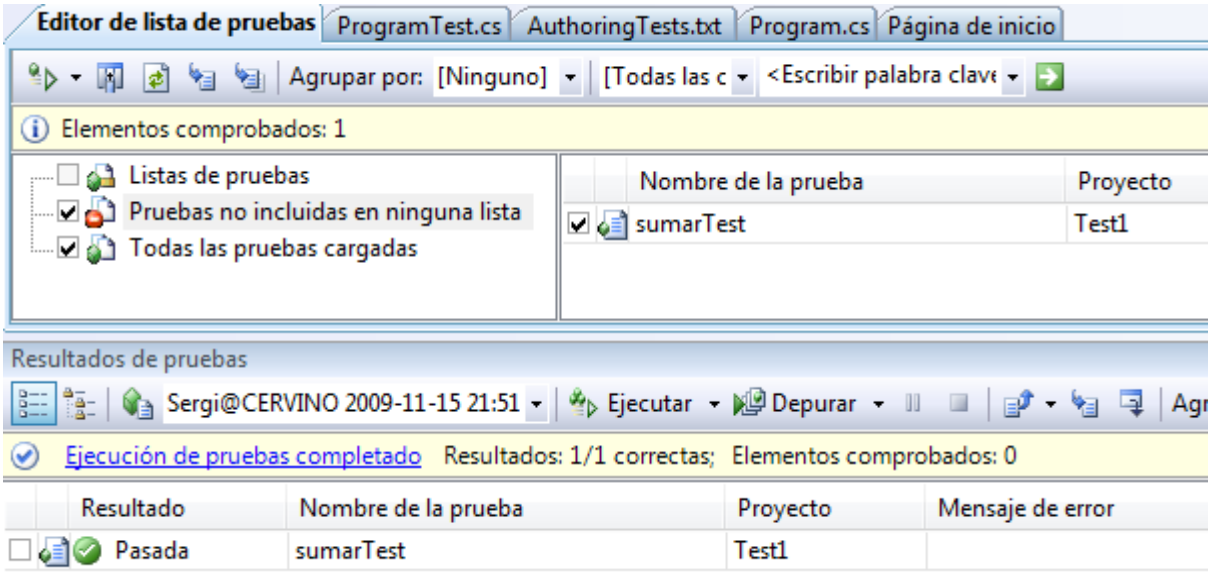

Ilustración 7: Ejecución de pruebas

En la prueba podéis ver cómo usando la sentencia **Assert.AreEqual** se comprueba si el resultado del método Sumar corresponde con el resultado esperado.

La clase **Assert** es parte del Framework y que se incluye dentro de la referencia Microsoft.VisualStudio.QualityTools.UnitTestFramework. Esta clase nos proporciona algunos métodos para comprobar los resultados de los métodos que están sometidos a la prueba; IsFalse, IsNull, AreSame, etc

Parece por tanto, que toda prueba debería terminar con una clausula de este tipo para comprobar si la prueba ha sido correcta o no. Pues bien, esto no es así.

Aunque seguramente usaremos alguno de los métodos de la clase Assert en la mayoría de las situaciones no es de uso obligatorio. Si no usamos esta clausula, la ejecución de la prueba dará un resultado correcta siempre y cuando la ejecución de la misma no genere una excepción.

Por ejemplo, vamos a modificar nuestro método suma para que sólo sume valores positivos y si alguno de los parámetros que recibe es menor que 0, que devuelva una excepción.

```
public int sumar(int a, int b)
{
if (a < 0 || b < 0)
   throw new ArgumentException();
   return (a + b);
}
```

En este caso, nos podría interesar hacer una prueba dónde comprobemos que no se genera una excepción cuando los parámetros sean mayores que 0. En esa prueba no sería necesario incluir ninguna sentencia de comprobación. Si le pasamos valores positivos la prueba será correcta siempre y cuando no genera una excepción.

Del mismo modo, nos interesará hacer una prueba para comprobar que la validación se hace correctamente y que se devuelve la excepción ArgumentException en este caso
.
En este caso, deberemos decorar el método de prueba con el atributo ExpectedException. Con este atributo estamos especificando que el resultado esperado de la ejecución de la prueba es la excepción ArgumentException. Si no se genera la excepción o la excepción no es del tipo esperado, la prueba fallará.

```
/// <summary>
///Otra prueba de sumar
///</summary>
[TestMethod()]
[ExpectedException(typeof(System.ArgumentException))]
public void sumarTest2()
{
    Program target = new Program(); // TODO: Inicializar en un valor adecuado
    int a = -1; // TODO: Inicializar en un valor adecuado
    int b = -2; // TODO: Inicializar en un valor adecuado
    int actual;
    actual = target.sumar(a, b);
}
```

Métodos de inicialización

Para terminar con esta introducción hablaremos sobre los métodos de inicialización. En las clases que contienen las pruebas unitarias existen cuatro métodos qué tienen una finalidad especial y que están decorados con un atributo que determina su objetivo.

El atributo **ClassInitialize** identifica un método que contiene código que debe ejecutarse antes de que se ejecute cualquiera de las pruebas de la clase y para asignar los recursos que utilizará la clase de pruebas.

El atributo **TestInitialize** identifica un método que contiene código que se ejecuta antes de cada prueba que contiene la clase y se usa para asignar y configurar los recursos que necesitan todas las clase de la prueba.

El atributo **ClassCleanup** identifica un método que se utilizará después de la ejecución de todas las pruebas de la clase y para liberar los recursos obtenidos por la prueba.

El atributo **TestCleanup** contiene código que se ejecutará después de ejecutar cada prueba y se puede usar la liberar los recursos obtenidos durante la prueba.

```
#region Atributos de prueba adicionales
 //
 //Puede utilizar los siguientes atributos adicionales mientras escribe sus
 // pruebas:
 //
 //Use ClassInitialize para ejecutar código antes de ejecutar la primera prueba
 // en la clase
 //[ClassInitialize()]
 //public static void MyClassInitialize(TestContext testContext)
 //{
 //}
 //
 //Use ClassCleanup para ejecutar código después de haber ejecutado todas las
 // pruebas en una clase
 //[ClassCleanup()]
 //public static void MyClassCleanup()
 //{
 //}
 //
 //Use TestInitialize para ejecutar código antes de ejecutar cada prueba
 //[TestInitialize()]
 //public void MyTestInitialize()
 //{
 //}
 //
 //Use TestCleanup para ejecutar código después de que se hayan ejecutado todas
 // las pruebas
 //[TestCleanup()]
 //public void MyTestCleanup()
 //{
 //}
 //
 #endregion
```

El uso de estos métodos no es para nada obligatorio y sólo deberemos emplearlos en las situaciones que lo necesitemos. Por defecto, al generar la prueba, estos métodos están comentados.

Estos cuatro métodos nos ayudarán a cumplir con estos dos objetivos ya que nos podrán ayudar a preparar el entorno y los recursos necesarios para la ejecución de las pruebas y a restaurar el entorno y liberar los recursos una vez ejecutadas.

Pruebas unitarias en web

Para probar nuestros sitios web el procedimiento es un poco diferente. Se puede resumir en tres pasos:

1. Crear un sitio Web ASP.NET dentro de su solución Visual Studio.
2. Agregue una clase al proyecto de sitio Web
3. Generar una prueba unitaria a partir de esa clase.

Creamos un sitio web

Añadimos una nueva clase. Pinchando con el botón derecho encima del proyecto, elegimos Agregar nuevo elemento y en la ventana elegimos clase. (*Ilustración 8*)

Si no tenemos la carpeta App_Code Visual Studio nos mostrará una ventana preguntándonos si queremos crear la carpeta. (*Ilustración 9*)
No puede generar pruebas desde el código en un archivo .aspx o en una carpeta distinta de la carpeta App_Code.

Una vez hecho esto tendremos nuestra clase **Class1.cs** dentro de la carpeta **App_Code** de nuestra solución (*Ilustración 10*)

Si el nuevo archivo de clase no está abierto, para abrirlo haga doble clic en él en el Explorador de soluciones.

Creamos el método sumar

```
public int sumar(int a, int b)
{
    return a + b;
}
```

Haga clic con el botón secundario en la clase del archivo de clase y, a continuación, haga clic en **Crear pruebas unitarias**

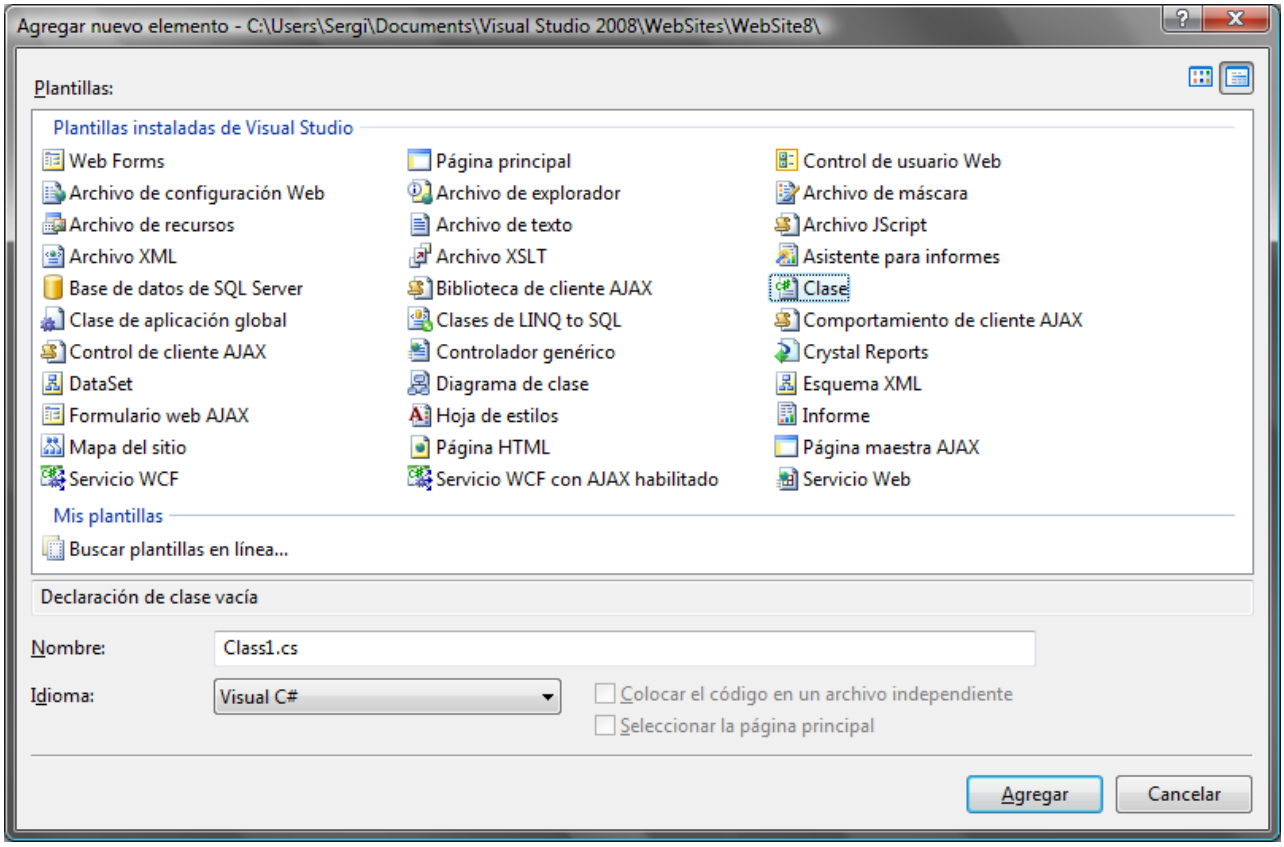

Ilustración 8: Crear nuevo elemento

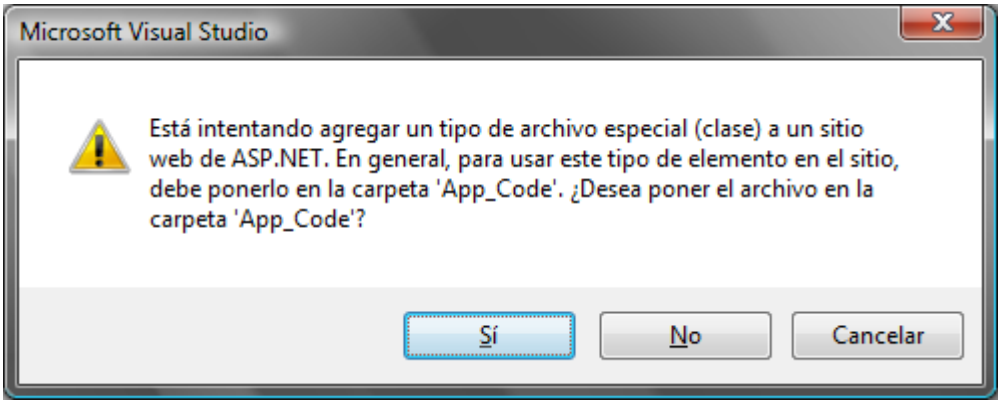

Ilustración 9: Crear carpeta App_Code

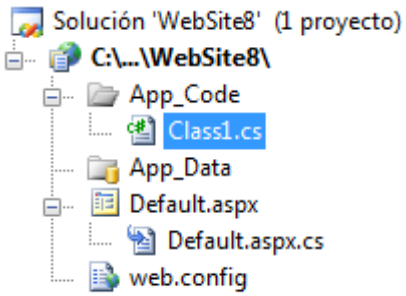

Ilustración 10: Nueva clase

Aparecerá el cuadro de diálogo Crear pruebas unitarias que vimos en el apartado anterior

Confirme que están seleccionados los métodos, las clases o los espacios de nombres para los que desea generar las pruebas.

Acepte el Proyecto de salida predeterminado o seleccione un proyecto nuevo.

La nueva prueba unitaria de ASP.NET se agrega a un archivo en su proyecto de prueba.

Para ver la prueba unitaria, abra el archivo de prueba y desplácese hasta el final. Se han especificado automáticamente los atributos necesarios para ejecutar una prueba unitaria como una prueba unitaria de ASP.NET.

Configurar una prueba unitaria de ASP.NET

Puede convertir una prueba unitaria ya existente en una prueba unitaria de ASP.NET mediante su configuración, es decir, si asigna valores a determinados atributos personalizados de la prueba. Estos valores se establecen en el archivo de código que contiene la prueba unitaria.

Para poder establecer los atributos personalizados, primero debe agregar una referencia al espacio de nombres que admite los atributos personalizados; éste es el espacio de nombres Microsoft.VisualStudio.TestTools.UnitTesting.Web. Con esta referencia en contexto, IntelliSense puede ayudarle a establecer los valores de los atributos.

Cuando se genera una prueba unitaria de ASP.NET, estos atributos se definen automáticamente.

Para configurar una prueba unitaria de ASP.NET

Atributos de una prueba ASP.NET.

[TestMethod]

Dado que todas las pruebas unitarias requieren el atributo [TestMethod], este atributo ya estará definido.

[UrlToTest()]

Esta es la dirección URL que se prueba al ejecutar esta prueba unitaria; por ejemplo, [UrlToTest(“http://localhost/WebSites/Default.aspx”)]

[HostType()]

Utilice [HostType(“ASP.NET”)]. Las pruebas normalmente se ejecutan bajo el proceso del host de VSTest, pero las pruebas unitarias de ASP.NET deben ejecutarse bajo el proceso del host de ASP.NET.

[AspNetDevelopmentServerHost()]

Especifica la configuración que debe utilizarse cuando un servidor de desarrollo de ASP.NET es el servidor host para la prueba

Ejemplo

Ejemplo de prueba unitaria del método suma en ASP.NET, utilizando el servidor de desarrollo

```
/// <summary>
///Una prueba de sumar
///</summary>
[TestMethod()]
[HostType("ASP.NET")]
[AspNetDevelopmentServerHost("%PathToWebRoot%\\WebSite8", "/WebSite8")]
[UrlToTest("http://localhost/WebSite8")]
public void sumarTest()
{
 Class1_Accessor target = new Class1_Accessor();

 int a = 1;
 int b = 2;
 int expected = 3;
 int actual;
 actual = target.sumar(a, b);
 Assert.AreEqual(expected, actual);
}
```

Ejemplo.
Para probar un sitio Web que se ejecuta bajo IIS, utilice únicamente los atributos TestMethod, HostType y UrlToTest:

```
[TestMethod()]
[HostType("ASP.NET")]
[UrlToTest("http://localhost:25153/WebSite1")]
```

Configurar pruebas unitarias de ASP.NET utilizando configuraciones de ejecución

Puede especificar las opciones de configuración en una configuración de ejecución que corresponden a los atributos utilizados por pruebas unitarias de ASP.NET. Una vez especificados estos atributos en una configuración de ejecución, las opciones se aplicarán cuando ejecute cualquier prueba unitaria de ASP.NET, siempre que la configuración de ejecución esté activa.

Nota:
Sólo puede estar activo un conjunto de opciones de configuración para las pruebas unitarias de ASP.NET: configuración de atributos u opciones de configuración de ejecución, pero nunca una combinación de estas dos. Las opciones de configuración de ejecución tienen precedencia sobre los atributos, si están presentes. Esto significa que aun cuando sólo especifique una configuración de ASP.NET en la configuración de ejecución, cualquier configuración de ASP.NET especificada como atributos se omitirá.

Para configurar pruebas unitaria de ASP.NET mediante la configuración de ejecución

1. Abra un archivo de configuración de ejecución. (*Ilustracion 11*).
2. En la página Host, establezca el Tipo de host en ASP.NET.

Se mostrarán opciones adicionales, algunas de los cuales corresponden a los atributos que se pueden especificar en código, como URL para probar. Los que se describieron en el apartado anterior.

Cuando termine de configurar valores en la página Host, haga clic en Guardar y, a continuación, haga clic en Aceptar.

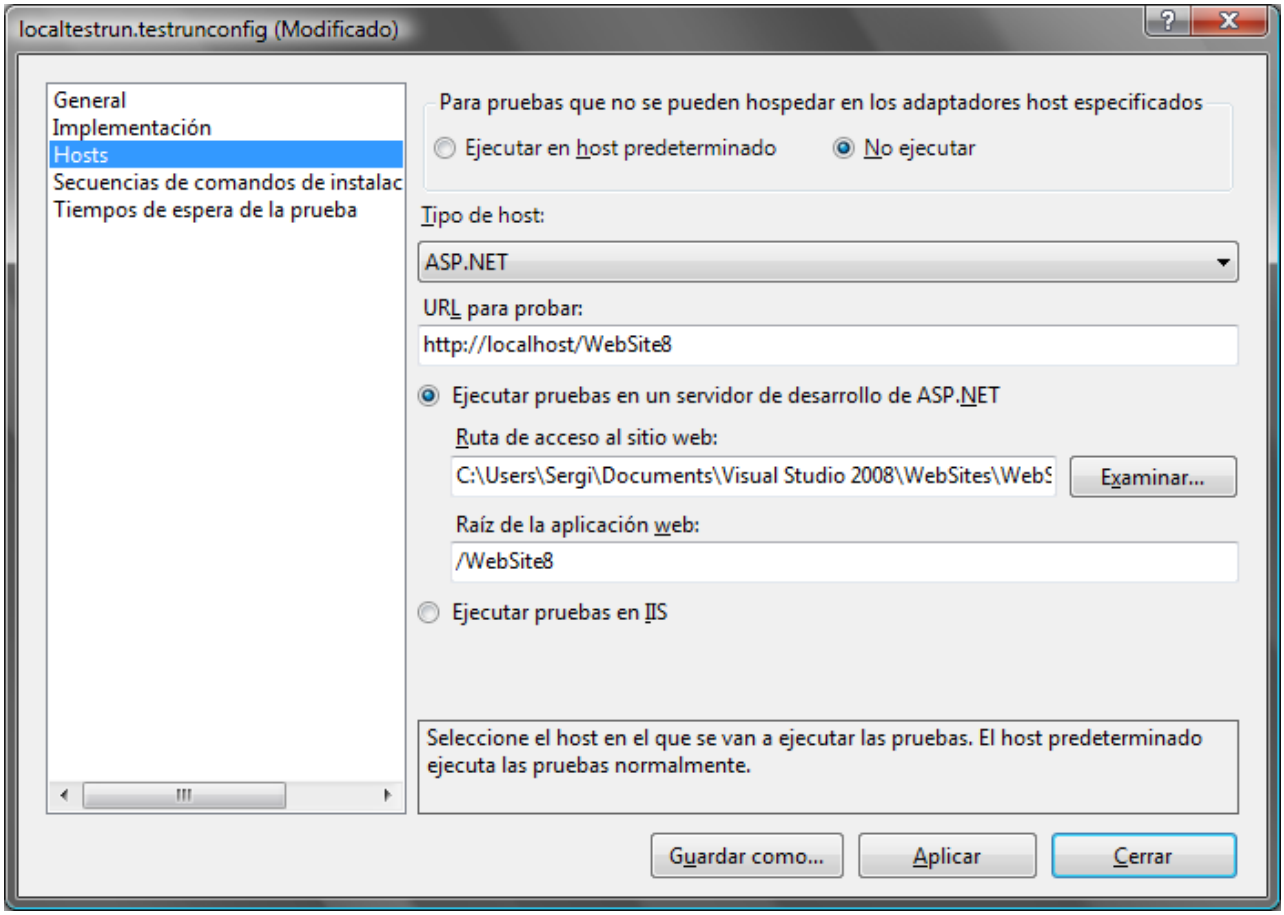

Ilustración 11: Configuración de ejecución

Orígenes de datos para alimentar las pruebas

En algunas ocasiones, para poder probar de forma completa un módulo es necesario probar muchas variantes en los parámetros de entrada.

Por ejemplo, si tenemos nuestra ya famoso método Sumar, con podría interesar probar el método con diferentes parámetros de entrada, para comprobar que realmente suma bien en todas las situaciones.

Una primera aproximación podría ser escribir tantas pruebas como necesitemos, cambiando en cada prueba los parámetros de entrada. Una solución poco adecuada.

La solución, más adecuada, cuando las N pruebas sólo cambian en los datos de entrada, **es usar la característica que nos ofrece Visual Studio para generar una única prueba y cargar los diferentes escenarios de datos desde un origen de datos**.

El proceso para configurar un origen de datos para una prueba es muy sencillo. Desde la ventana que muestra la lista de pruebas unitarias de la solución, seleccionaremos la prueba unitaria que nos interese.

Una vez seleccionada podremos ver sus propiedades. A través de la propiedad "**Cadena de conexión a datos**" podremos especificar el origen de datos.(*Ilustración 12*)

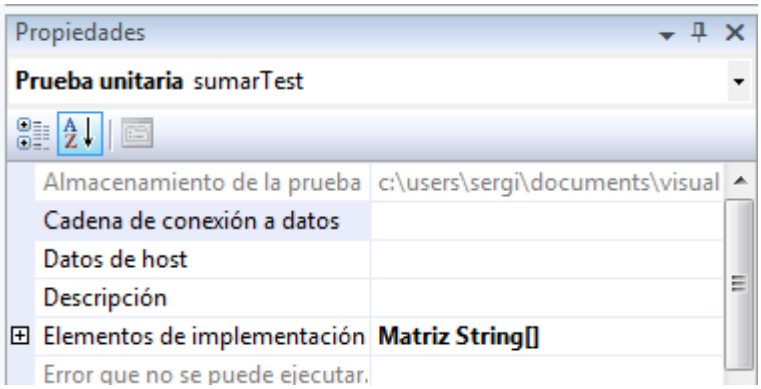

Ilustración 12: Propiedades prueba unitaria

Si pinchamos sobre esta opción nos aparecerá un asistente que nos guiará paso a paso para especificar el origen de datos, que puede ser una base de datos, un fichero CSV o un fichero XML (*Ilustración 13*)

Seleccionamos un fichero CSV que tiene tres valores separados por comas. Los dos primeros son los valores a sumar, siendo el tercero el resultado esperado

Se nos añade a la prueba la linea:

```
[DataSource("Microsoft.VisualStudio.TestTools.DataSource.CSV",
     "|DataDirectory|\\datosSuma.csv", "datosSuma#csv",
     DataAccessMethod.Sequential), DeploymentItem("Test1\\datosSuma.csv"),
     TestMethod()]
```

El atributo DataSource indica el origen de datos que se empleará en la prueba, así como la forma de acceder al mismo.

El atributo DeployementItem permite especificar ficheros adicionales que se incluirán entre los ficheros que se usarán para la prueba. En este caso se indica que se va a incluir el fichero datosSuma.csv, que es el fichero que contiene todos los valores posibles que queremos probar como parámetros de entrada.

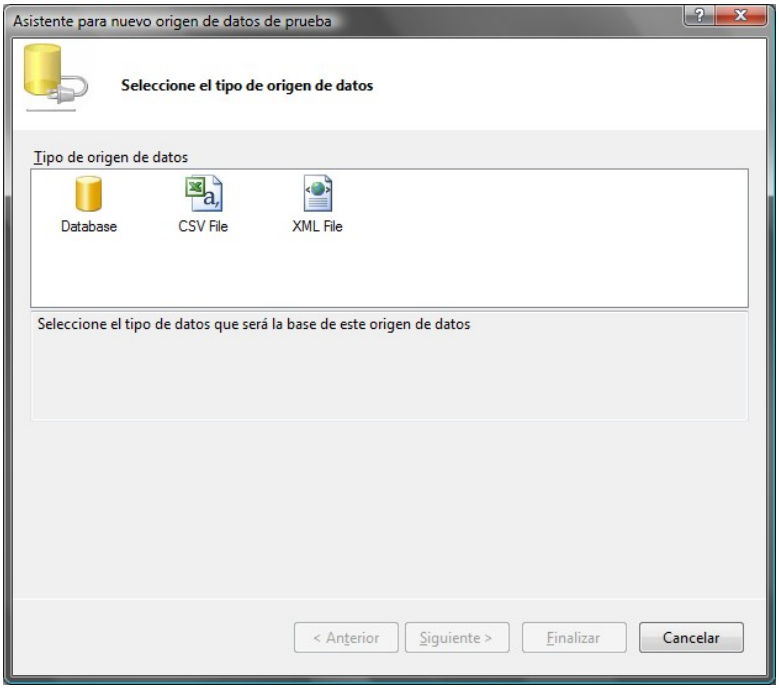

Ilustración 13: Asistente origen de datos

Una vez hecha esta operación, tendremos que modificar nuestra prueba para que sea capaz de usar el origen que acabamos de añadir.

A través de la variable de clase **TestContext** que posee la clase de pruebas podremos acceder a los datos, usando la propiedad **DataRow**.

La prueba unitaria queda de la siguiente manera:

```
/// <summary>
///Una prueba de sumar
///</summary>
[DataSource("Microsoft.VisualStudio.TestTools.DataSource.CSV",
       "|DataDirectory|\\datosSuma.csv", "datosSuma#csv",
      DataAccessMethod.Sequential), DeploymentItem("Test1\\datosSuma.csv"),
          TestMethod()]
public void sumarTest()
{
    Program target = new Program();
    int a = (int)this.TestContext.DataRow[0];
    int b = (int)this.TestContext.DataRow[1];
    int expected = (int)this.TestContext.DataRow[2];
    int actual;
    actual = target.sumar(a, b);
    Assert.AreEqual(expected, actual);
}
```

Para que esto funcione hay que añadir una referencia a System.Data.

¡Se acabó!

No se vayan todavía, ¡¡¡ aun hay más !!!

- interfaz de usuario ASP.NET
- Capa de acceso a datos
- pruebas de servicios web
- cobertura de código

... en próximas ediciones !!!