Pruebas de Software:
Herramientas: Pruebas Unitarias

Elisa Herrmann

Ingeniería del Software de Gestión

Contenido

- ¿Qué son Pruebas Unitarias?
- Frameworks
- Ventajas
- Desventajas y limitaciones
- Mitos sobre Pruebas Unitarias
- Prácticas recomendadas en DBAccess
- NUnit
- Referencias

¿Qué son Pruebas Unitarias?

- Es un procedimiento usado para validar que un modulo o método de un objeto fuente funciona apropiadamente y en forma independiente.

- A través de ellas se verifica que cierto módulo o método se ejecuta dentro de los parámetros y especificaciones concretadas en documentos tales como los casos de uso y el diseño detallado.

- Permiten detectar efectivamente la inyección de defectos durante fases sucesivas de desarrollo o mantenimiento.

¿Qué son Pruebas Unitarias?

- Las pruebas unitarias típicamente son automatizadas, pero pueden llevarse a cabo de forma manual. Cuando son automatizadas es buena práctica que formen parte del repositorio que contiene al código probado.

- Se dice que una prueba unitaria es completa o es buena si cumple con los siguientes elementos:

  - Automática
  - Cobertura
  - Repetibles
  - Independiente

Frameworks

- Para llevar a cabo pruebas unitarias, cada organización se apoya en *frameworks* que ofrecen un conjunto completo de utilidades, motores de ejecución y reportes.

- Entre los *frameworks* más empleados destacan:
  - XUnit: JUnit, NUnit, RUnit, PHPUnit…
  - TestNG
  - CPPUnit
  - Visual Studio UnitTesting

Ventajas

Dependiendo del *framework* empleado podemos encontrar las siguientes ventajas:

- **Automatizadas**, por lo cual se hacen repetibles.
- **Fomentan el cambio**: ya que permiten probar cambios en el código y asegurar que en éstos no se hayan introducido errores funcionales; habilitan el "refactoring"del código.
- **Simplifican la integración**: permiten llegar a la fase de integración con un grado alto de seguridad sobre el código.

Ventajas

- **Documenta el código**.
- **Separa** la interfaz y la implementación.
- Los defectos están **acotados** y **fáciles de localizar**.
- Permiten al desarrollador **pensar como el consumidor del código y no como el productor**.

Desventajas y limitaciones

- No descubrirán todos los defectos del código.
- No permite determinar problemas de integración o desempeño.
- No es trivial anticipar todos los casos especiales de entradas.
- Las pruebas unitarias determinan la presencia de defectos, no la ausencia de éstos. Son efectivas al combinarse con otras actividades de pruebas.

Prácticas recomendadas

- Seguir el procedimiento de integración continua al pie de la letra, evitando hacer *commit* al repositorio si las pruebas unitarias preexistentes fallan.
- Toda falla es producida por un defecto. Antes de corregir el defecto, debe escribirse una prueba unitaria que, al fallar, compruebe que el defecto está allí, y que al pasar compruebe que el defecto fue eliminado.
- Aprovechar que se tiene la atención en esa parte del código y escribir otras pruebas que se piensen empleando las capacidades del motor de pruebas unitarias.
- Es a tiempo de diseño que debe definirse formalmente la estrategia para implementar las pruebas unitarias.

Pruebas Unitarias

- Los datos de entrada son conocidos por el Tester o Analista de Pruebas y estos deben ser preparados con minuciosidad.
- Se debe conocer de antemano que resultados debe devolver el componente según los datos de entrada utilizados en la prueba.
- Se deben comparar los datos obtenidos en la prueba con los datos esperados, si son idénticos podemos decir que el modulo supero la prueba y empezamos con la siguiente.

JUnit: Ejemplo

- Método estático que tome un *array* de enteros como argumentos y devuelva el mayor valor encontrado en el *array*

```
public class MayorNumero{
   /**
   * Devuelve el elemento de mayor valor de una lista
   *@param    list    Un array de enteros
   * @return   El entero de mayor valor de la lista*/
    public static int obtenerMayor(int lista[]){
      return 0; // Para que compile
    }
}
```

¿Qué pruebas pueden hacerse?

- Caso normal: array con valores cualesquiera
  - [3, 7, 9, 8] -> 9
- El mayor número se encuentra al principio o al final de la lista
  - [9, 7, 8] -> 9
  - [8, 7, 9] -> 9
- El mayor número está duplicado en el array
  - [9, 7, 9, 8] -> 9
- Sólo hay un elemento en el array
  - [7] -> 7
- Array compuesto por números negativos
  - [-4, -6, -7, -22] -> -4

Código de Clase a probar

```
package elementos;

public class MayorNumero {

public static int mayorNumero(int lista[]) {
int indice, max = Integer.MAX_VALUE;
for (indice = 0; indice < lista.length-1; indice++)
  {
if (lista[indice] > max) {
max = lista[indice];
}
}return max;
}

}
```

Código de Prueba

```
package elementos;
import junit.framework.TestCase;
public class MayorNumeroTest extends TestCase {
public void testSimple() {
assertEquals(9, MayorNumero.mayorNumero(new int[] {3, 7, 9, 8}));
}
public void testOrden() {
assertEquals(9, MayorNumero.mayorNumero(new int[] {9, 7, 8}));
assertEquals(9, MayorNumero.mayorNumero(new int[] {7, 9, 8}));
assertEquals(9, MayorNumero.mayorNumero(new int[] {7, 8, 9}));
}
public void testDuplicados() {
assertEquals(9, MayorNumero.mayorNumero(new int[] {9, 7, 9, 8}));
}
public void testSoloUno() {
assertEquals(7, MayorNumero.mayorNumero(new int[] {7}));
}
public void testTodosNegativos() {
assertEquals(-4, MayorNumero.mayorNumero(new int[] {-4, -6, -7, 22}));}}
```

Ejecutar Prueba

- ¿Qué está mal?

```
public static int mayorNumero(int lista[])
  {
int indice, max = Integer.MAX_VALUE;
for (indice = 0; indice < lista.length-1;
  indice++) {
if (lista[indice] > max) {
max = lista[indice];
}
}return max;
}
```

Errores

- MIN_VALUE
- Analizar los extremos: Lista.Length

Marco para desarrollar pruebas unitarias

- Pasos:
  - Importar las clases de JUNIT necesarias
  - Definir la clase de pruebas:
  - Debe extender la clase "TestCase"
  - Definir los métodos de prueba
  - Serán ejecutados automáticamente por JUNIT
  - Definir un main o ejecutar desde un IDE
    - junit.textui.TestRunner.run(<clase>)

Comprobaciones

- assertEquals (valor_esperado, valor_real);
  - Los valores pueden ser de cualquier tipo
  - Si son arrays, no se comprueban elemento a elemento, sólo la referencia
- assertTrue (condición_booleana)
- assertFalse (condición_booleana)
- assertSame (Objeto esperado, Objeto real)
  - Comprueba que son la misma referencia
- assertNotSame (Objeto esperato, Objeto obtenido)
  - Comprueba que son referencias distintas
- assertNull (Objeto)
  - Comprueba que el objeto es Null
- assertNotNull (Objeto objeto)
  - Comprueba que el objeto no es Null
- fail (string Mensaje)
  - Imprime el mensaje y falla
  - Útil para comprobar que se capturan excepciones

Uso de Comprobaciones

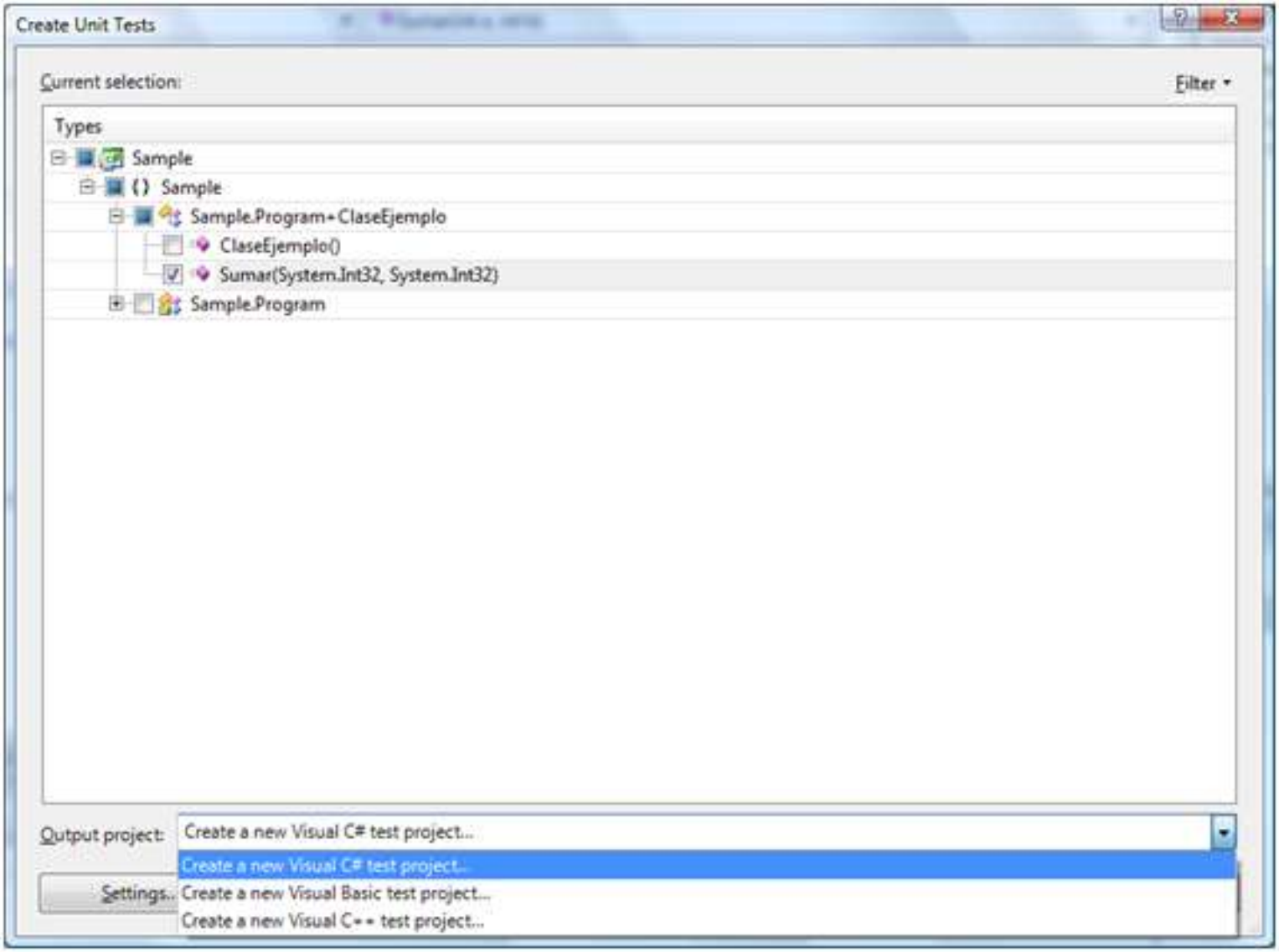
- En una función de prueba se pueden poner tantos métodos de comprobación como sean necesarios para implementar el caso de prueba concreto.
- En general hay que comprobar que un método lanza todas las excepciones que se han declarado en el mismo cuando debe. Y que no las lanza cuando no hay motivo para ello. Esta es la utilidad del método fail.

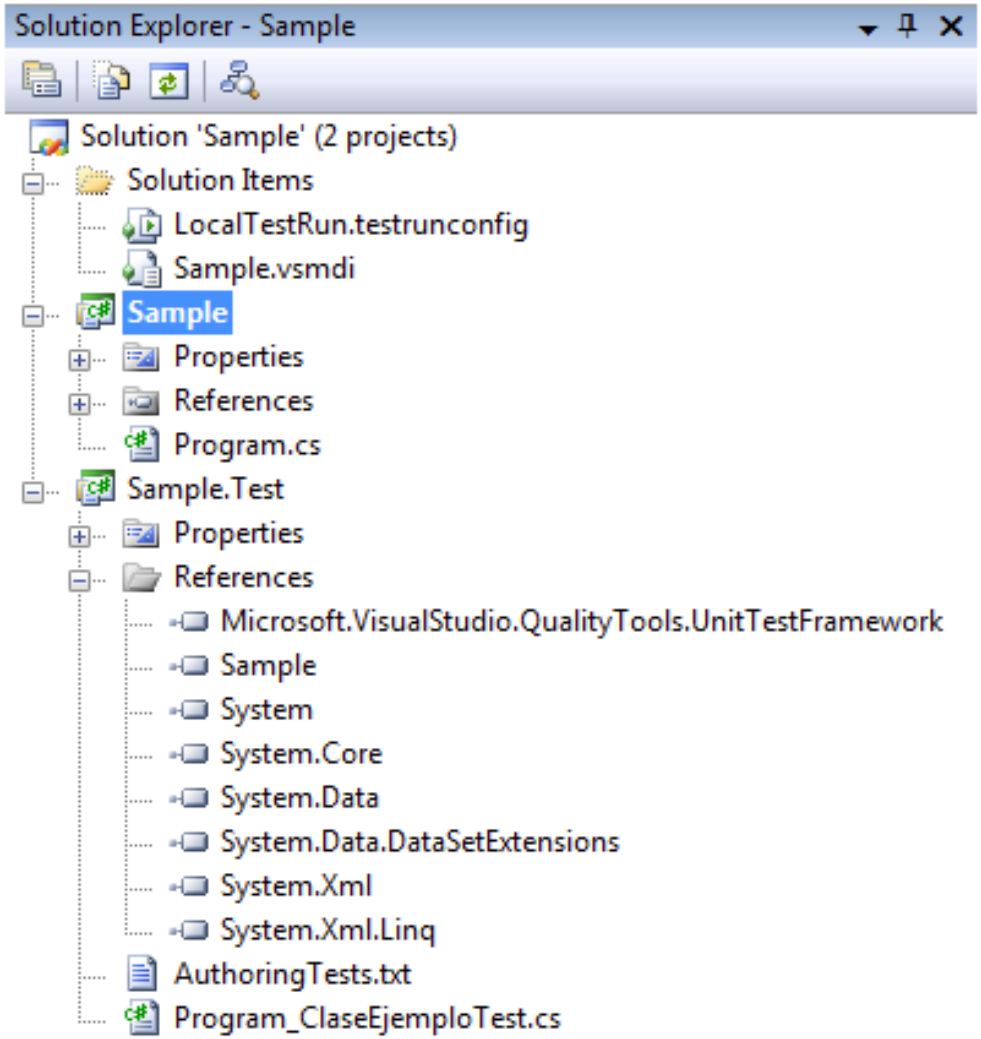
```
public void
  testExcepcionOrdenarListaNula( )  {try
  {ordena_lista(null);
  fail("Debería haber lanzado una
  excepción");} catch (RuntimeException
  e) { }}
```

Visual Studio UnitTesting

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

- Recomendación: crear al menos un proyecto de Test por cada proyecto, en lugar de crear un único proyecto de test que englobe todas las pruebas.

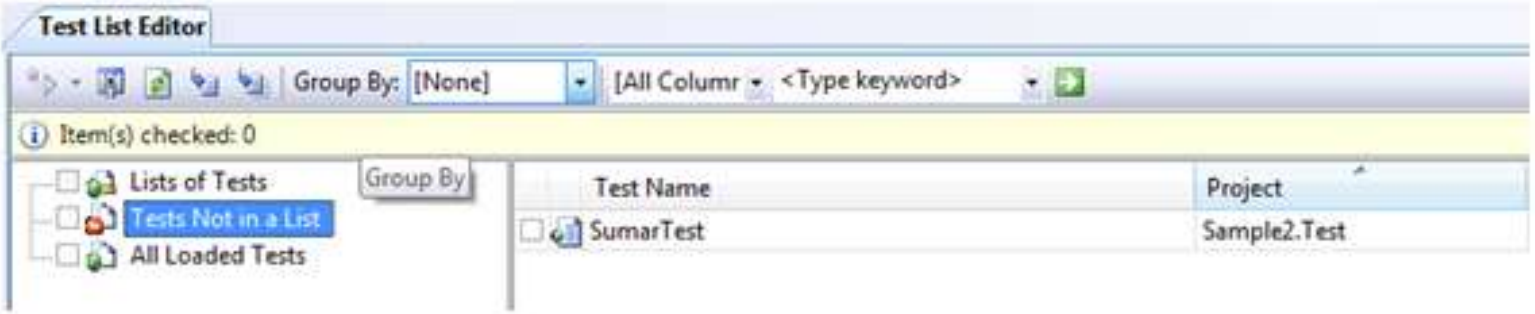

Create Unit Tests
Current selection:
Filter
Types
Sample
Sample
Sample.Program+ClaseEjemplo
ClaseEjemplo()
Sumar(System.Int32, System.Int32)
Sample.Program
Output project:
Create a new Visual C# test project...
Create a new Visual C# test project...
Create a new Visual Basic test project...
Create a new Visual C++ test project...
Settings...


Solution Explorer - Sample
Solution 'Sample' (2 projects)
Solution Items
LocalTestRun.testrunconfig
Sample.vsmdi
Sample
Properties
References
Program.cs
Sample.Test
Properties
References
Microsoft.VisualStudio.QualityTools.UnitTestFramework
Sample
System
System.Core
System.Data
System.Data.DataSetExtensions
System.Xml
System.Xml.Linq
AuthoringTests.txt
Program_ClaseEjemploTest.cs

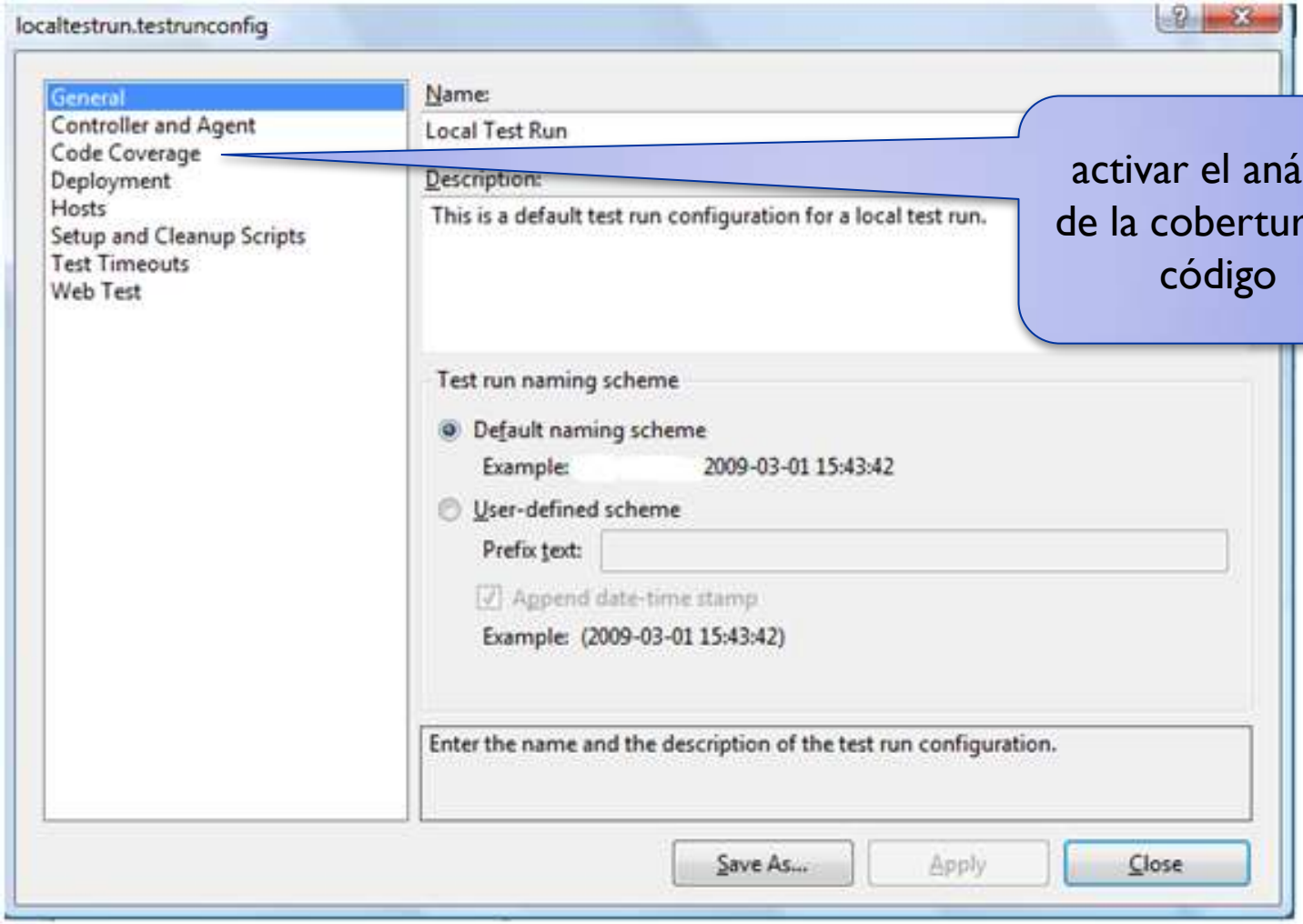
Código del método de prueba

```
///A test for Sumar ///</summary>
[TestMethod()]
public void SumarTest() {
    ClaseEjemplo target = new ClaseEjemplo(); int a
    = 0;
    int b = 0;
    int expected = 0;
    int actual;
    actual = target.Sumar(a, b);
    Assert.AreEqual(expected, actual);
    Assert.Inconclusive("Verify the correctness of this
    test method."); }
```

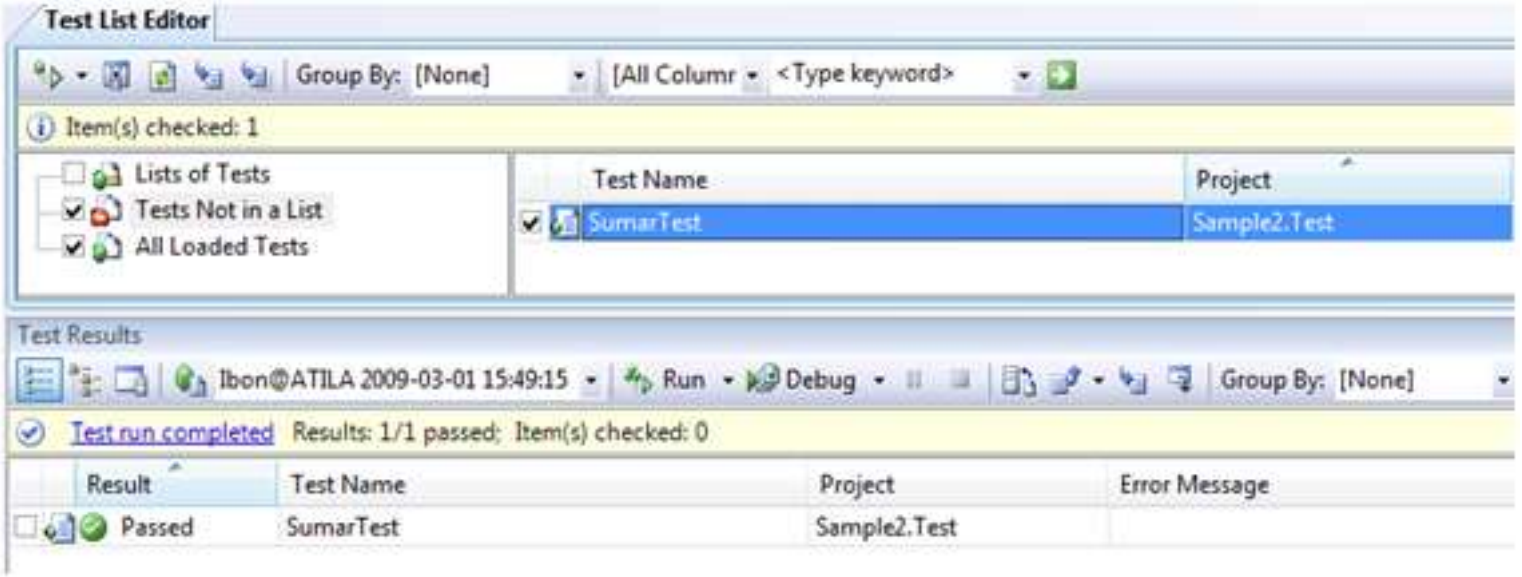
Sample.vsmdi



LocalTestRun.testrunconfig



Ejecutar Prueba



% de Cobertura

Code Coverage Results

2008-12-08 19:50:48

Hierarchy	Not Covered (Blocks)	Not Covered (% Blocks)	Covered (Blocks)	Covered (% Blocks)
Ibon@ATILA 2008-12-08 19:50:48	95	32,42 %	198	67,58 %
BL.dll	75	74,26 %	26	25,74 %
BL	29	74,36 %	10	25,64 %
CustomerBL	29	74,36 %	10	25,64 %
BL.ERPService	46	74,19 %	16	25,81 %
Customer	10	45,45 %	12	54,55 %
ERPServiceClient	14	77,78 %	4	22,22 %
SampleFaultException	22	100,00 %	0	0,00 %
DAL.dll	7	5,51 %	120	94,49 %
DAL	7	5,51 %	120	94,49 %
Helpers.dll	4	8,00 %	46	92,00 %
Helpers	4	8,00 %	46	92,00 %
Mail	4	8,00 %	46	92,00 %
ServiceDemo.dll	9	60,00 %	6	40,00 %
ServiceSample	9	60,00 %	6	40,00 %

```
public class ClaseEjemplo
{
    public int Sumar(int a, int b)
    {
        if (a < 0 || b < 0)
            throw new ArgumentException();

        return a + b;
    }
}
```

Otras Pruebas

- Pruebas de Aislamiento mediante Mock Objects: JMock y EasyMock.
- Pruebas de Aplicaciones que acceden a Bases de Datos: DBUnit.
- Pruebas de Documentos: XMLUnit.
- Pruebas de Aplicaciones Web :
  - HttpUnit
  - HtmlUnit
  - JWebUnit