Pruebas unitarias con Visual Studio .Net

Contenido

1. Código de ejemplo

El único error intencionado en este ejemplo es que, en el método Debit, "m_balance += amount" debe tener un signo menos, no más, antes del signo igual.

```
using System;

namespace BankAccountNS
{
    /// <summary>
    /// Bank Account demo class.
    /// </summary>
    public class BankAccount
    {
        private string m_customerName;

        private double m_balance;

        private bool m_frozen = false;

        private BankAccount()
        {
        }

        public BankAccount(string customerName, double balance)
        {
            m_customerName = customerName;
            m_balance = balance;
        }

        public string CustomerName
        {
            get { return m_customerName; }
        }

        public double Balance
        {
            get { return m_balance; }
        }

        public void Debit(double amount)
        {
            if (m_frozen)
            {
                throw new Exception("Account frozen");
            }

            if (amount > m_balance)
            {
                throw new ArgumentOutOfRangeException("amount");
            }

            if (amount < 0)
            {
                throw new ArgumentOutOfRangeException("amount");
            }

            m_balance += amount; // intentionally incorrect code
        }

        public void Credit(double amount)
        {
```

```
            if (m_frozen)
            {
                throw new Exception("Account frozen");
            }

            if (amount < 0)
            {
                throw new ArgumentOutOfRangeException("amount");
            }

            m_balance += amount;
        }

        public void FreezeAccount()
        {
            m_frozen = true;
        }

        public void UnfreezeAccount()
        {
            m_frozen = false;
        }

        public static void Main()
        {
            BankAccount ba = new BankAccount("Mr. Bryan Walton", 11.99
);

            ba.Credit(5.77);
            ba.Debit(11.22);
            Console.WriteLine("Current balance is ${0}", ba.Balance);
        }

    }
}
```

Amount	Cantidad
Bank Account	Cuenta bancaria
Freeze Account	Congelar cuenta
Frozen	Congelada
Balance	Saldo

2. Prepare el ejemplo

1. Abra Visual Studio.
2. En el menú **Archivo**, elija **Nuevo** y haga clic en **Proyecto**.
   Aparecerá el cuadro de diálogo **Nuevo proyecto**.
3. En **Plantillas instaladas**, haga clic en **Visual C#**.
4. En la lista de tipos de aplicación, haga clic en **Biblioteca de clases**.
5. En el cuadro **Nombre**, escriba `Bank` y, a continuación, haga clic en **Aceptar**.
6. Se crea el nuevo proyecto Bank y se muestra en el Explorador de soluciones con el archivo Class1.cs abierto en el editor de código.
7. Copie el código fuente.
8. Reemplace el contenido original de Class1.cs con el código copiado.
9. Guarde el archivo como BankAccount.cs
10. En el menú **Compilar**, haga clic en **Compilar solución**.

Ahora tiene un proyecto denominado Bank que contiene código fuente para realizar pruebas y las herramientas necesarias para ello. El espacio de nombres de Bank, **BankAccountNS**, contiene la clase pública **BankAccount** cuyos métodos probará en los procedimientos siguientes.

Este tutorial se centra en el método `Debit`. Se llama al método Debit cuando se retira dinero de una cuenta y contiene el siguiente código:

```
public void Debit(double amount)
        {
            if (m_frozen)
            {
                throw new Exception("Account frozen");
            }

            if (amount > m_balance)
            {
                throw new ArgumentOutOfRangeException("amount");
            }

            if (amount < 0)
            {
                throw new ArgumentOutOfRangeException("amount");
            }

            m_balance += amount; // intentionally incorrect code
        }
```

3. Crear un proyecto de prueba unitaria

Para crear un proyecto de prueba unitaria

1. En el menú **Archivo**, elija **Agregar** y, a continuación, elija **Nuevo proyecto...**.
2. En el cuadro de diálogo Nuevo proyecto, expanda **Instalado**, expanda **Visual C#** y, a continuación, elija **Prueba**.
3. En la lista de plantillas, seleccione **Proyecto de prueba unitaria**.
4. En el cuadro **Nombre**, escriba BankTest y elija **Aceptar**.
   El proyecto **BankTests** se agrega a la solución **Bank**.
5. En el proyecto **BankTests**, agregue una referencia a la solución **Bank**.
   En el Explorador de soluciones, seleccione **Referencias** en el proyecto **BankTests** y, después, elija **Agregar referencia...** desde el menú contextual.
6. En el cuadro de diálogo del Administrador de referencia, expanda **Solución** y active el elemento **Bank**.

4. Crear la clase de prueba

Se necesita una clase de prueba para comprobar la clase `BankAccount`. Se puede utilizar UnitTest1.cs, generado por la plantilla de proyecto, pero se debe asignar al archivo y a la clase nombres más descriptivos. Podemos hacer esto en un solo paso cambiando el nombre del archivo en el Explorador de soluciones.

Cambiar el nombre de un archivo de clase

En el Explorador de soluciones, seleccione el archivo UnitTest1.cs en el proyecto BankTests. Desde el menú contextual, elija **Cambiar nombre** y, a continuación, cambie el nombre del archivo a BankAccountTests.cs. Elija **Sí** en el cuadro de diálogo que pregunta si desea cambiar el nombre de todas las referencias del proyecto al elemento de código "UnitTest1". En este paso se cambia el nombre de la clase a `BankAccountTest`.

El archivo BankAccountTests.cs contiene ahora el siguiente código:

```
// unit test code
using System;
using Microsoft.VisualStudio.TestTools.UnitTesting;

namespace BankTests
{
    [TestClass]
    public class BankAccountTests
    {
        [TestMethod]
        public void TestMethod1()
        { }
    }
}
```

Agregar una instrucción using al proyecto en pruebas

También podemos agregar una instrucción using a la clase para poder llamar al proyecto en pruebas, sin utilizar nombres completos. En la parte superior del archivo de clase, agregue:

```
using BankAccountNS;
```

Requisitos de la clase de prueba

Los requisitos mínimos para una clase de prueba son los siguientes:

- El atributo `[TestClass]` se requiere en el marco de pruebas unitarias para código administrado de Microsoft para cualquier clase que contenga métodos de prueba unitaria que desee ejecutar en el Explorador de pruebas.
- Cada método de prueba que desee que ejecute el Explorador de pruebas debe tener el atributo `[TestMethod]`.

Puede tener otras clases de un proyecto de prueba unitaria que no tengan el atributo `[TestClass]` y puede tener otros métodos de clases de prueba que no tengan el atributo `[TestMethod]`. Puede utilizar estos otros métodos y clases en sus métodos de prueba.

5. Crear el primer método de prueba

En este procedimiento, se escribirán métodos de prueba unitaria para comprobar el comportamiento del método `Debit` de la clase `BankAccount`. El método se muestra más arriba.

Al analizar el método en pruebas, se determina que hay al menos tres comportamientos que deben comprobarse:

1. El método produce `ArgumentOutOfRangeException` si la cantidad de débito es mayor que el saldo.
2. También produce `ArgumentOutOfRangeException` si la cantidad de débito es menor que cero.
3. Si se cumplen las comprobaciones en 1.) y 2.), el método resta la cantidad del saldo de cuenta.

En la primera prueba, se comprueba que una cantidad válida (una menor que el saldo de cuenta y mayor que cero) retire la cantidad correcta de la cuenta.

Para crear un método de prueba

1. Agregue una instrucción using `BankAccountNS;` al archivo BankAccountTests.cs.
2. Agregue el siguiente método a esa clase `BankAccountTests`:

```
// unit test code
[TestMethod]
public void Debit_WithValidAmount_UpdatesBalance()
  {
      // preparación del caso de prueba
      double beginningBalance = 11.99;
      double debitAmount = 4.55;
      double expected = 7.44;
      BankAccount account = new BankAccount("Mr. Bryan Walton",
beginningBalance);

      // acción a probar
      account.Debit(debitAmount);


      // afirmación de la prueba (valor esperado Vs. Valor obtenido)
      double actual = account.Balance;
      Assert.AreEqual(expected, actual, 0.001, "Account not debited correctly");
   }
```

El método es bastante sencillo. Se configura un nuevo objeto `BankAccount` con un saldo inicial y después se retira una cantidad válida. Se utiliza el marco de pruebas unitarias de Microsoft para el método AreEqual de código administrado, para comprobar que el saldo de cierre es el esperado.

Utilizar las clases Assert

Método	Función
`Assert.AreEqual(expected, actual)`	Comprueba que expected y actual tienen el mismo valor.
`Assert.AreNotEqual( expected, actual)`	Comprueba que expected y actual NO tienen el mismo valor.
`Assert.IsTrue(bool condition)`	Comprueba que condition es true.
`Assert.IsFalse(bool condition)`	Comprueba que condition es false.
`Assert.IsNull(object anObject)`	Comprueba que el objeto anObject es nulo.
`Assert.IsNotNull(object anObject)`	Comprueba que el objeto anObject NO es nulo.
`Assert.AreSame(expected,actual)`	Comprueba si los dos argumentos están referenciando al mismo objeto.
`Assert.Greater(arg1, arg2)`	Comprueba si arg1 es mayor que arg2.

Más información en :

https://msdn.microsoft.com/es-es/library/microsoft.visualstudio.testtools.unittesting.assert.aspx

Más tipos de asserts:

https://msdn.microsoft.com/es-es/library/ms182530.aspx

Requisitos del método de prueba

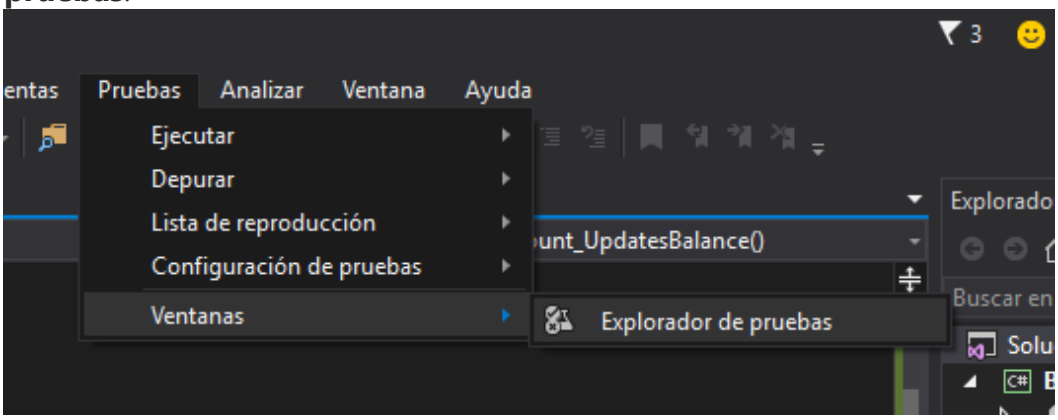
Un método de prueba debe cumplir los siguientes requisitos:

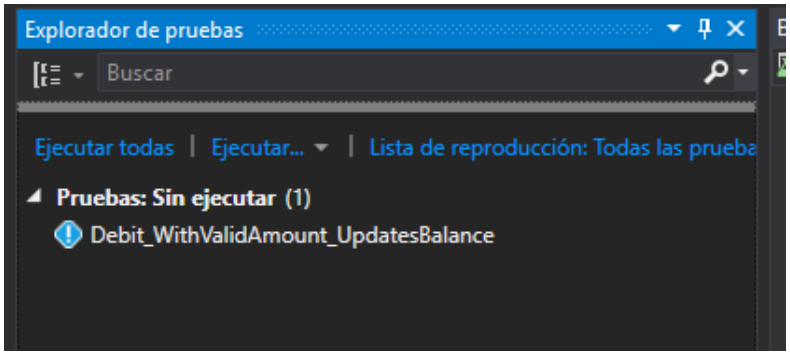
- El método se debe señalar con el atributo `[TestMethod]`.
- El método debe devolver `void`.
- El método no puede tener parámetros.

6. Compilar y ejecutar la prueba

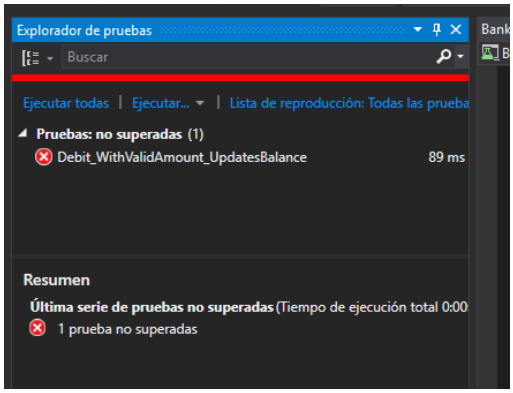
Para compilar y ejecutar la prueba

1. En el menú **Compilar**, elija **Compilar solución**.
   Si no hay ningún error, aparece la ventana UnitTestExplorer con **Debit_WithValidAmount_UpdatesBalance** incluido en el grupo **Pruebas no ejecutadas**. Si no el Explorador de pruebas aparece tras realizar una compilación correcta, elija **Prueba** en el menú, **Ventanas** y, a continuación, **Explorador de pruebas**.

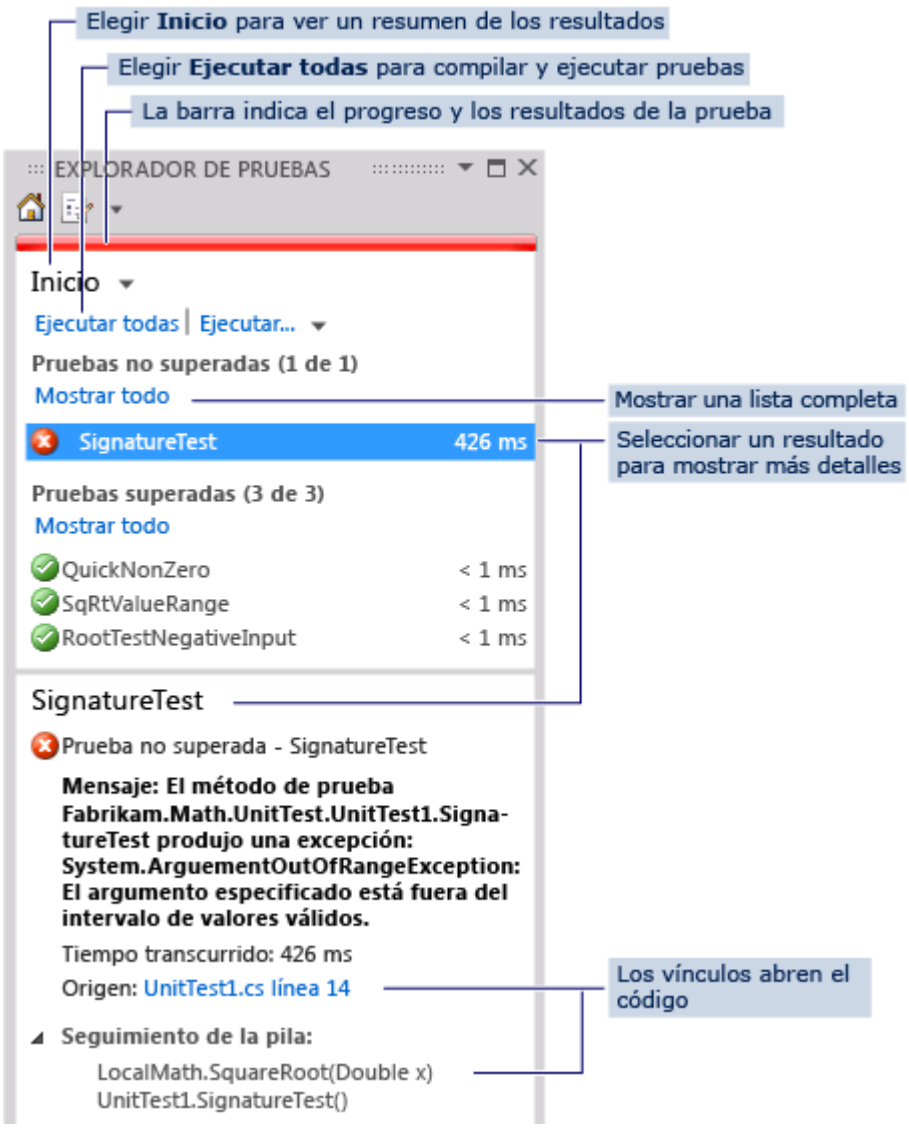




2. Elija **Ejecutar todas** para ejecutar la prueba. Mientras se ejecuta la prueba, la barra de estado en la parte superior de la ventana se anima. Al final de la serie de pruebas, la barra se vuelve verde si todos los métodos de prueba se completan correctamente o roja si no alguna de las pruebas no lo hace.
3. En este caso, la prueba no se completa correctamente. El método de prueba se mueve al grupo **Pruebas no superadas**. Seleccione el método en el Explorador de pruebas para ver los detalles en la parte inferior de la ventana.



7. Usar el Explorador de pruebas



- **Para ver una lista completa de pruebas:** elija **Mostrar todo** en cualquier categoría.
- **Para ver los detalles de un resultado de pruebas:** seleccione la prueba en el Explorador de pruebas para ver detalles tales como los mensajes de excepción en el panel de detalles.
- **Para navegar hasta el código de una prueba:** haga doble clic en la prueba en el Explorador de pruebas o elija **Abrir prueba** en el acceso directo.
- **Para depurar una prueba:** abra el acceso directo de una o varias pruebas y, a continuación, elija **Depurar pruebas seleccionadas**.

8. Corrija el código y vuelva a ejecutar las pruebas

Analizar los resultados de pruebas

El resultado de la prueba contiene un mensaje que describe el error. Para el método `AreEquals`, el mensaje muestra lo que se esperaba (el parámetro *Expected<XXX>*) **y lo que se recibió realmente (el parámetro *Actual<YYY>***). Se esperaba una disminución en el saldo en comparación con el inicial, pero, en cambio, ha aumentado en la cantidad retirada.

Un nuevo examen del código Debit muestra que la prueba unitaria ha logrado encontrar un error. La cantidad retirada se agrega al saldo de cuenta en lugar de ser restarse.

Corregir el error

Para corregir el error, reemplace simplemente la línea:

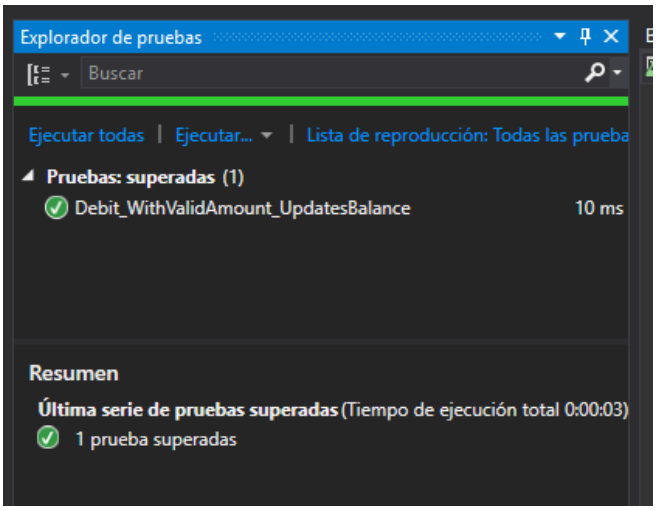
```
m_balance += amount;
```

por:

```
m_balance -= amount;
```

Vuelva a ejecutar la prueba

En el Explorador de pruebas, elija **Ejecutar todas** para volver a ejecutar la prueba. La barra de color rojo o verde se vuelve verde y la prueba se mueve al grupo de **Pruebas superadas**.



9. Actividad

Estudia el método debit, sus clases de equivalencia, fronteras y prepara los diferentes casos de prueba. Impleméntalos usando métodos de prueba en la clase BankAccountTests.
Ejecuta las pruebas.

Nota:
Deja los casos que deben producir una exception:

```
if (m_frozen)
{
    throw new Exception("Account frozen");
}

if (amount > m_balance)
{
    throw new ArgumentOutOfRangeException("amount");
}

if (amount < 0)
{
    throw new ArgumentOutOfRangeException("amount");
}
```

10. Pruebas que esperan excepciones.

Analizar los problemas

Después de crear un método de prueba para confirmar que se reste correctamente una cantidad válida en el método `Debit`, podemos volver a los casos restantes del análisis original:

1. El método produce `Exception` si la cuenta está congelada.
2. El método produce `ArgumentOutOfRangeException` si la cantidad de débito es mayor que el saldo.
3. También produce `ArgumentOutOfRangeException` si la cantidad de débito es menor que cero.

Crear los métodos de prueba

Se usa el atributo ExpectedExceptionAttribute para validar que se ha producido la excepción correcta. El atributo hace que la prueba dé un error a menos que se produzca una `Exception`.

```
//unit test method
[TestMethod]
[ExpectedException(typeof(Exception))]
public void Debit_WhenBankAccountIsFrezee_ShouldThrowException()
 {
       // preparación del caso de prueba
       double beginningBalance = 11.99;
       double debitAmount = -100.00;
       BankAccount account = new BankAccount("Mr. Bryan Walton",
beginningBalance);
       account.FreezeAccount();

       // acción a probar
       account.Debit(debitAmount);

       // la afirmación es manejado por el atributo ExpectedException
}
```

Pruebas que esperan excepciones: ExpectedException

Este atributo se utiliza para comprobar si se inicia una excepción esperada.

- El método de prueba es correcto si se inicia la excepción esperada.
- Se producirá un error en la prueba si la excepción iniciada hereda de la excepción esperada.

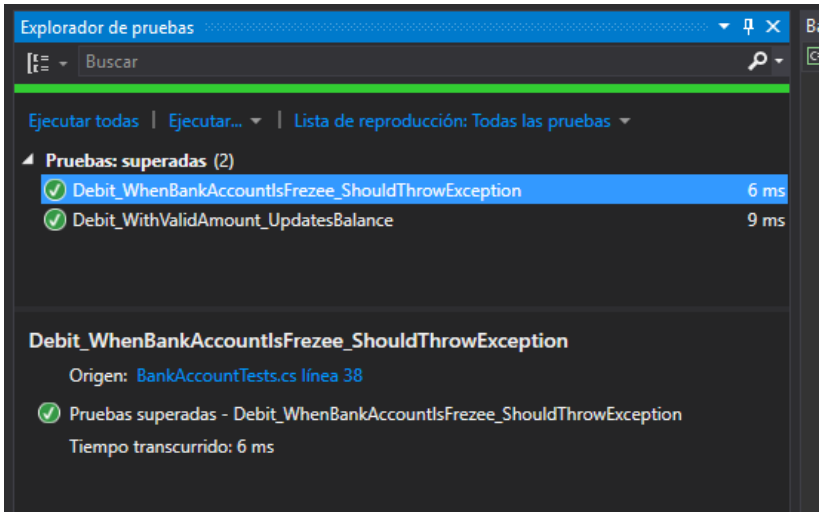
- Si una prueba tiene un atributo ExpectedException y una instrucción Assert, cualquiera de los dos puede provocar un error en la prueba.
    - Para determinar si el error fue provocado por el atributo o por la instrucción, haga doble clic en el resultado de la prueba para abrir la página de resultados para esa prueba. Para obtener más información acerca de los resultados de pruebas, vea Test Results Reported.

- Este atributo puede especificarse en un método. Solo puede haber una instancia de este atributo en un método.

https://msdn.microsoft.com/es-es/library/microsoft.visualstudio.testtools.unittesting.expectedexceptionattribute(v=vs.120).aspx

Vuelva a ejecutar la prueba

En el Explorador de pruebas, elija **Ejecutar todas** para volver a ejecutar la prueba. La barra de color rojo o verde se vuelve verde y la prueba se mueve al grupo de **Pruebas superadas**.



11. Actividades

Crea las pruebas oportunas para los casos `amount>m_balance` y `amount<0`.

12. Utilice pruebas unitarias para mejorar el código

Los dos últimos métodos de prueba también son algo problemáticos.

No podemos estar seguros de qué condición del código en pruebas se inicia cuando se realiza cada serie de pruebas. Sería útil tener alguna manera de diferenciar las dos condiciones. Cuanto más se piensa en el problema, más evidente resulta que sabiendo qué condición se ha infringido aumentaría la confianza en las pruebas.

Esta información también sería útil, muy probablemente, al mecanismo de producción que controla la excepción cuando la inicia el método en pruebas. Generar más información cuando el método inicie una excepción ayudaría a todos los componentes relacionados, pero el atributo `ExpectedException` no puede proporcionar esta información.

Al examinar de nuevo el método en pruebas, se puede ver que ambas instrucciones condicionales utilizan un constructor `ArgumentOutOfRangeException`, que toma su nombre del argumento como parámetro:

```
throw new ArgumentOutOfRangeException("amount");
```

En una búsqueda por MSDN Library detectamos que existe un constructor que proporciona información mucho más completa. ArgumentOutOfRangeException`(String, Object, String)` incluye el nombre del argumento, su valor y un mensaje definido por el usuario. Se puede refactorizar el método en pruebas para utilizar este constructor. Incluso mejor, podemos utilizar miembros de tipo que se encuentran disponibles públicamente para especificar los errores.

13. Refactorizar el código en pruebas

Refactorizar

- Refactorizar es realizar modificaciones en el código con el objetivo de mejorar su estructura interna, sin alterar su comportamiento externo.
- Refactorizar no es una técnica para encontrar y corregir errores en una aplicación, puesto que su objetivo no es alterar su comportamiento externo.
- Cambio realizado a la estructura interna del software para hacerlo… más fácil de comprender y más fácil de modificar sin cambiar su comportamiento observable.

Primero se definen dos constantes para los mensajes de error en el ámbito de la clase:

```
// class under test
public const string DebitAmountExceedsBalanceMessage = "Debit amount exceeds
balance";

public const string DebitAmountLessThanZeroMessage = "Debit amount less than
zero";
```

A continuación, se modifican las dos instrucciones condicionales en el método `Debit`:

```
if (amount > m_balance)
{
 throw new ArgumentOutOfRangeException("amount", amount,
DebitAmountExceedsBalanceMessage);
}

if (amount < 0)
{
 throw new ArgumentOutOfRangeException("amount", amount,
DebitAmountLessThanZeroMessage);
}
```

Refactorizar los métodos de prueba

En el método de prueba, primero quitamos el atributo `ExpectedException`.

En su lugar, se captura la excepción y se comprueba que se haya iniciado en la instrucción condicional correcta. Sin embargo, ahora debemos decidir entre dos opciones para comprobar las condiciones restantes.

Por ejemplo, en el método `Debit_WhenAmountIsMoreThanBalance_ShouldThrowArgumentOutOfRange`, podemos realizar una de las siguientes acciones:

- Asegurar que la propiedad `ActualValue` de la excepción (el segundo parámetro del constructor de `ArgumentOutOfRangeException`) es mayor que el saldo inicial. Esta opción requiere probar la propiedad `ActualValue` de la excepción con la variable `beginningBalance` del método de prueba y, también, requiere que se compruebe que `ActualValue` es mayor que cero.
- Asegurar que el mensaje (el tercer parámetro del constructor) incluye el `DebitAmountExceedsBalanceMessage` definido en la clase `BankAccount`.

El método StringAssert.Contains del marco de pruebas unitarias de Microsoft permite comprobar la segunda opción sin los cálculos necesarios de la primera opción.

Un segundo intento de revisar `Debit_WhenAmountIsMoreThanBalance_ShouldThrowArgumentOutOfRange` podría ser similar a:

```
[TestMethod]
public void Debit_WhenAmountIsMoreThanBalance_ShouldThrowArgumentOutOfRange()
{
        // arrange
        double beginningBalance = 11.99;
        double debitAmount = 20.0;
        BankAccount account = new BankAccount("Mr. Bryan Walton",
beginningBalance);
        // act

        try
        {
                account.Debit(debitAmount);
        }
        catch (ArgumentOutOfRangeException e)
        {

        // assert
        StringAssert.Contains(e.Message, BankAccount.
DebitAmountExceedsBalanceMessage);
        }
  }
```

Vuelva a probar, reescriba y vuelva a analizar

Cuando se vuelven a probar los métodos de prueba con valores diferentes, descubrimos lo siguiente:

1. Si se captura el error correcto usando una aserción `debitAmount` mayor que el saldo, la comprobación `Contains` se supera, la excepción se omite y el método de prueba se completa correctamente. Este es el comportamiento que deseamos.
2. Si se usa `debitAmount` menor que 0, la comprobación no se puede realizar porque se devuelve un mensaje de error incorrecto. La comprobación tampoco se puede realizar si se introduce una excepción temporal `ArgumentOutOfRange` en otro punto del método bajo la ruta de acceso del código de prueba. Esto también es correcto.
3. Si el valor de `debitAmount` es válido (es decir, menor que el saldo pero mayor que cero), no se detecta ninguna excepción, por lo que la comprobación nunca se detecta. El método de prueba se completa correctamente. Esto no es bueno, porque se desea que el método de prueba no se supere si no se produce ninguna excepción.

El tercer hecho es un error en el método de prueba. Para intentar resolver el problema, se agrega una validación Fail al final del método de prueba para controlar el caso donde no se produce ninguna excepción.

Pero, al volver a examinar, se muestra que se produce un error en la prueba si se detecta la excepción correcta. La instrucción catch restaura la excepción, el método continúa ejecutándose y produce errores en la nueva validación.

Para resolver este nuevo problema, se agrega una instrucción `return` después de `StringAssert`. Al volver a examinar se confirma que hemos corregido los problemas. La versión final de `Debit_WhenAmountIsMoreThanBalance_ShouldThrowArgumentOutOfRange` tiene el siguiente aspecto:

```
[TestMethod]
public void Debit_WhenAmountIsMoreThanBalance_ShouldThrowArgumentOutOfRange()
{
 // arrange
 double beginningBalance = 11.99;
 double debitAmount = 20.0;
 BankAccount account = new BankAccount("Mr. Bryan Walton", beginningBalance);
            // act
 try
 {
       account.Debit(debitAmount);
 }
 catch (ArgumentOutOfRangeException e)
 {
 // assert
       StringAssert.Contains(e.Message, BankAccount.
DebitAmountExceedsBalanceMessage);
       return;
 }
 Assert.Fail("No exception was thrown.");
}
```

En esta sección final, el trabajo que se hizo al mejorar el código de prueba condujo a métodos de prueba más eficaces e informativos. Pero, lo que es más importante, el análisis adicional también condujo a mejoras en el código del proyecto en pruebas.

14. Actividades

Modifica de forma similar las pruebas para controlar la excepción que contiene el valor:

`DebitAmountLessThanZeroMessage`.

15. Agregar pruebas para ampliar el intervalo de entradas

1. Para mejorar su confianza en que el código funcione en todos los casos, agregue pruebas para in intervalo más amplio de valores de entrada.

> **Sugerencia**
>
> Evite modificar las pruebas existentes que se completan correctamente. En su lugar, agregue nuevas pruebas. Cambie las pruebas existentes solo si cambian los requisitos de usuario. Esta directiva ayuda a garantizar que no se pierda la funcionalidad existente mientras se trabaja para ampliar el código.

Ejemplo extraido de otro proyecto.

```
[TestMethod]
public void RooterValueRange()
{
  // Create an instance to test:
  Rooter rooter = new Rooter();
  // Try a range of values:
  for (double expectedResult = 1e-8;
      expectedResult < 1e+8;
      expectedResult = expectedResult * 3.2)
  {
    RooterOneValue(rooter, expectedResult);
  }
}

private void RooterOneValue(Rooter rooter, double expectedResult)
{
  double input = expectedResult * expectedResult;
  double actualResult = rooter.SquareRoot(input);
  Assert.AreEqual(expectedResult, actualResult,
      delta: expectedResult / 1000);
}
```

16. Enlaces de interés

Tutorial: Crear y ejecutar pruebas unitarias en código administrado

https://msdn.microsoft.com/es-es/library/ms182532.aspx#BKMK_Prepare_the_walkthrough

Inicio rápido: Desarrollo basado en pruebas con el Explorador de pruebas

https://msdn.microsoft.com/es-es/library/hh212233.aspx

Usar cobertura de código para determinar la cantidad de código que se está probando

https://msdn.microsoft.com/es-es/library/dd537628.aspx

Ejecutar pruebas unitarias con el Explorador de pruebas

https://msdn.microsoft.com/es-es/library/hh270865.aspx

#100devdays Pruebas Unitarias

https://channel9.msdn.com/Blogs/DevWow/-100devdays-Pruebas-Unitarias

Vídeo:"Getting started with unit testing": Part1:Haz click aquí, Part2: Haz click aquí, Part3: Haz click aquí