Laura Carratalá Castillo

Documentación de pruebas unitarias

A continuación, se reflejarán de forma clara y concisa los pasos para la elaboración de las pruebas unitarias de las clases de la lógica de negocio de las 4 aplicaciones que componen nuestro proyecto de Navaja Suiza.

En primer lugar, definiremos la nomenclatura para referirnos a cada una de las pruebas. En nuestro caso hemos decidido usar el identificador de nuestro número de aplicación "App + número de aplicación", seguida de un punto, el número de la prueba y "PU" que hace referencia a "pruebas unitarias".

Pruebas unitarias para la aplicación 1:

El método que pondremos a prueba en nuestra aplicación 1 (se introduce un número y el programa devuelve si es primo o no) es, como ya hicimos en las pruebas de caja negra y caja blanca, **EsPrimo** cuyo código es:

```
/// <summary>
/// Función que verifica si un número positivo introducido es o no es primo.
/// </summary>
/// <remarks>----</remarks>
/// <param name="numeroIntroducido">Número que introduce el usuario.</param>
/// <returns>Si es primo o no.</returns>
public static bool EsPrimo(int numeroIntroducido)
{
    bool esPrimo = true;

    if (numeroIntroducido > 0)
    {
        for (int i = 2; i < numeroIntroducido && esPrimo; i++)
        {
            if (numeroIntroducido % i == 0)
            {
                esPrimo = false;
            }
        }
    }
    return esPrimo;
}
```

Puesto que ya controlamos la introducción de elementos no válidos (cadenas, símbolos, decimales, etc) con tryParse, tan solo probaremos la introducción de números enteros, ya sean positivos o negativos. A continuación, describiremos brevemente cada una de las pruebas unitarias realizadas, que se encuentran en el proyecto de pruebas TestApp1.

Clases de equivalencia:

- **numeroIntroducido:** Es un int, deberá ser mayor que 0.

App1.1PU y App1.2PU) numeroIntroducido >= 2: Introducción de un número positivo mayor o igual que 2 que sea primo. Valores límite: 2, 3. Nombre de los métodos de prueba: **NumeroEsPrimo2** y **NumeroEsPrimo3**.

App1.3PU) numeroIntroducido >= 2: Introducción de un número positivo mayor o igual que 2 que no sea primo. Valor límite: 4. Nombre del método de prueba: **NumeroNoEsPrimo4.**

App1.4PU) numeroIntroducido = 1: Probamos este valor porque se trata de un número especial, ya que en las pruebas de caja negra el programa nos indicaba que era primo, cuando no se considera como tal. Nombre del método de prueba: **NumeroNoEsPrimo1.**

App1.5PU) numeroIntroducido = 0: Probamos este valor porque se trata de un número especial, ya que en las pruebas de caja negra el programa nos indicaba que era primo, cuando no se considera como tal. Nombre del método de prueba: **NumeroNoEsPrimo0.**

App1.6PU y App1.7PU) numeroIntroducido <= MaxValue: Introducción del máximo valor que puede tomar un entero y su valor directamente inferior. Valores límite: MaxValue, MaxValue – 1. Nombre del método de prueba: **NumeroEsPrimoMaxValor** y **NumeroNoEsPrimoMaxValorMenos1.**

App1.8PU y App1.9PU) numeroIntroducido >= MinValue: Introducción del mínimo valor que puede tomar un entero y su valor directamente superior. Valores límite: MinValue, MinValue – 1. Nombre del método de prueba: **NumeroNoEsPrimoMinValor** y **NumeroNoEsPrimoMinValorMas1.**

Prueba	numero Introducido	esPrimoObtenido	Esperado	Resultado	Comentario
App1.1PU	2	true	true	Válido	Es correcto.
App1.2PU	3	true	true	Válido	Es correcto.
App1.3PU	4	false	false	Válido	Es correcto.
App1.4PU	1	true	false	No Válido	No es correcto, el 1 no se considera primo.
App1.5PU	0	true	false	No Válido	No es correcto, el 0 no se considera primo.
App1.6PU	MaxValue	true	true	Válido	Es correcto.
App1.7PU	MaxValue - 1	false	false	Válido	Es correcto.
App1.8PU	MinValue	true	false	No Válido	No es correcto, los números negativos no son primos.
App1.9PU	MinValue + 1	true	false	No Válido	No es correcto, los números negativos no son primos.

Como podemos comprobar, las pruebas App1.4PU, App1.5PU, App1.8PU y App1.9PU han resultado ser no válidas, con lo que debemos corregir el código de nuestro método para que puedan ser válidas. La corrección que hemos aplicado es la siguiente:

```
/// <summary>
/// Función que verifica si un número positivo introducido es o no es primo.
/// </summary>
/// <remarks>----</remarks>
/// <param name="numeroIntroducido">Número que introduce el usuario.</param>
/// <returns>Si es primo o no.</returns>
public static bool EsPrimo(int numeroIntroducido)
{
    bool esPrimo = true;

    if (numeroIntroducido > 1)
    {
        for (int i = 2; i < numeroIntroducido && esPrimo; i++)
        {
            if (numeroIntroducido % i == 0)
            {
                esPrimo = false;
            }
        }
    }
    else
    {
        esPrimo = false;
    }
    return esPrimo;
```

De forma que ahora la aplicación aceptará números superiores a 1 y el 1, 0 y valores negativos darán como resultado no primo. La tabla corregida queda de la siguiente manera:

Prueba	numero Introducido	esPrimoObtenido	Esperado	Resultado	Comentario
App1.1PU	2	true	true	Válido	Es correcto.
App1.2PU	3	true	true	Válido	Es correcto.
App1.3PU	4	false	false	Válido	Es correcto.
App1.4PU	1	false	false	Válido	Es correcto.
App1.5PU	0	false	false	Válido	Es correcto.
App1.6PU	MaxValue	true	true	Válido	Es correcto.
App1.7PU	MaxValue - 1	false	false	Válido	Es correcto.
App1.8PU	MinValue	false	false	Válido	Es correcto, los números negativos no son primos.
App1.9PU	MinValue + 1	false	false	Válido	Es correcto, los números negativos no son primos.

A continuación, indicaremos qué pruebas de caja negra quedan cubiertas en las pruebas unitarias (las pruebas App1.6N, App1.7N, App1.8N y App1.9N quedan abarcadas por el tryParse):

- App1.1PU y App1.2PU: Cubren la prueba App1.1N.
- App1.3PU: Cubre la prueba App1.2N.
- App1.4PU: Cubre la prueba App1.3N.
- App1.5PU: Cubre las pruebas App1.4N y App1.5N.
- App1.6PU y App1.7PU: Cubren la prueba App1.10N.
- App1.8PU y App1.9PU: Cubren la prueba App1.11N.

Pruebas unitarias para la aplicación 2:

El método que pondremos a prueba en nuestra aplicación 2 (se introduce un número y el programa devuelve la serie de múltiplos de 3 y 5 hasta dicho número) es, como ya hicimos en las pruebas de caja negra y caja blanca, **MostrarSerieMultiplos**, pero en este caso hemos realizado una refactorización previa para poder controlar en el propio método el rango de valores permitidos (entre el 1 y el 100) y poder lanzar una excepción en caso de que introduzcamos valores que se salen de los límites. El código queda de la siguiente manera:

```
/// excepción para que introduzca un número entre el 1 y el 100.</remarks>
/// <param name="numeroIntroducido">Número que introduce el usuario.</param>
/// <returns>Cadena con los múltiplos de 3 y 5.</returns>
public static string MostrarSerieMultiplos(int numeroIntroducido)
{
    string cadenaTexto = "";
    bool numeroValido = true;

    numeroValido = ValidarNumero(numeroIntroducido);

    if (numeroValido == false)
    {
        throw new ArgumentOutOfRangeException(numeroFueraDeRango);
    }
    else
    {
        for (int i = 1; i <= numeroIntroducido; i++)
        {
            if (i % 3 == 0 || i % 5 == 0)
            {
                cadenaTexto = cadenaTexto + i + ", ";
            }
        }
    }
    return cadenaTexto;
```

Puesto que ya controlamos la introducción de elementos no válidos (cadenas, símbolos, decimales, etc) con tryParse, tan solo probaremos la introducción de números enteros, ya sean positivos o negativos. A continuación, describiremos brevemente cada una de las pruebas unitarias realizadas, que se encuentran en el proyecto de pruebas TestApp2.

Clases de equivalencia:

- **numeroIntroducido**: Es un int.

App2.1PU) 5 < numeroIntroducido <= 100: Introducción de un número positivo mayor que 5 y menor o igual que 100. Valores límite: 6, 100. Nombre del método de prueba: **SerieMultiplosCorrecta6**. No es necesario probar con el 100 puesto que en el documento de pruebas de caja negra se ve que si es válido.

App2.2PU) 3 <= numeroIntroducido <= 5: Introducción de un número positivo mayor o igual que 3 y menor o igual que 5. Valores límite: 3, 4, 5. Nombre del método de prueba: **SerieMultiplosCorrecta5**. No es necesario probar con el 3 ni con el 4 puesto que en el documento de pruebas de caja negra se ve que si son válidos y además el 3 si aparece en la cadena (el 4 no porque no es múltiplo ni de 3 ni de 5).

App2.3PU) 1 <= numeroIntroducido < 3: Introducción de un número positivo mayor o igual que 1 y menor que 3. Valores límite: 1, 2. Nombre del método de prueba: **SerieMultiplosCorrecta1**. No es necesario probar con el 2, puesto que en los resultados se ve que no sale en la cadena al no ser múltiplo de 3 ni de 5. Con lo que con esta prueba se cubren los dos casos.

App2.4PU) numeroIntroducido = 0: Introducción de un número que esté directamente por debajo del límite de valores permitidos. Nombre del método de prueba: **SerieMultiplosIncorrecta0**.

App2.5PU) numeroIntroducido = 101: Introducción de un número que esté directamente por encima del límite de valores permitidos. Nombre del método de prueba: **SerieMultiplosIncorrecta101**.

App2.6PU y App2.7PU) numeroIntroducido <= MaxValue: Introducción del máximo valor que puede tomar un entero y su valor directamente inferior. Valores límite: MaxValue, MaxValue – 1. Nombre del método de prueba: **SerieMultiplosIncorrectaMaxValor** y **SerieMultiplosIncorrectaMaxValorMenos1.**

App2.8PU y App2.9PU) numeroIntroducido >= MinValue: Introducción del mínimo valor que puede tomar un entero y su valor directamente superior. Valores límite: MinValue, MinValue + 1. Nombre del método de prueba: **SerieMultiplosIncorrectaMinValor** y **SerieMultiplosIncorrectaMinValorMas1.**

Prueba	numero Introducido	cadenaTexto Obtenida	cadenaTexto Esperada	Resultado	Comentario
App2.1PU	6	“3, 5, 6, “	“3, 5, 6, “	Válido	Es correcto.
App2.2PU	5	“3, 5, ”	“3, 5, ”	Válido	Es correcto.
App2.3PU	1	“”	“”	Válido	Es correcto.
App2.4PU	0	ArgumentOutOfRangeException	ArgumentOutOfRangeException	Válido	Es correcto.
App2.5PU	101	ArgumentOutOfRangeException	ArgumentOutOfRangeException	Válido	Es correcto.
App2.6PU	MaxValue	ArgumentOutOfRangeException	ArgumentOutOfRangeException	Válido	Es correcto.
App2.7PU	MaxValue - 1	ArgumentOutOfRangeException	ArgumentOutOfRangeException	Válido	Es correcto.
App2.8PU	MinValue	ArgumentOutOfRangeException	ArgumentOutOfRangeException	Válido	Es correcto.
App2.9PU	MinValue + 1	ArgumentOutOfRangeException	ArgumentOutOfRangeException	Válido	Es correcto.

A continuación, indicaremos qué pruebas de caja negra quedan cubiertas en las pruebas unitarias (las pruebas App2.6N, App2.7N, App2.8N y App2.9N quedan abarcadas por el tryParse):

- App2.1PU: Cubre la prueba App2.1N.
- App2.2PU: Cubre la prueba App2.2N.
- App2.3PU: Cubre las pruebas App2.3N.
- App2.4PU: Cubre la prueba App2.4N.
- App2.5PU: Cubre las pruebas App2.1N y App2.5N.
- App2.6PU y App2.7PU: Cubren las pruebas App2.10N.
- App2.8PU y App2.9PU: Cubren las pruebas App2.11N.

Pruebas unitarias para la aplicación 3:

El método que pondremos a prueba en nuestra aplicación 3 (se introduce un vector de 10 números separados por comas y devuelve el vector modificado con los valores repetidos cambiados por -1 y el número de repeticiones totales) es **ComprobarValoresEntrada**. Hemos realizado una refactorización previa con respecto a la versión anterior para poder controlar mejor los valores obtenidos y las excepciones que se pueden obtener, ya sea porque se introducen elementos no válidos (cadenas, signos, etc) o porque no se introduce la cantidad de números indicada. El código queda de la siguiente manera:

```
public static int[] ComprobarValoresEntrada(string cadenaValores, out int modificaciones)
{
    string[] numerosIntroducidos = cadenaValores.Split(',');
    bool elementoValido = true;
    const int kTamanyo = 10;
    int[] vectorNumeros = new int[kTamanyo];

    if (numerosIntroducidos.Length != kTamanyo)
    {
        throw new ArgumentException(cantidadNumerosNoValida);
    }
    else
    {
        for (int i = 0; i < kTamanyo && elementoValido; i++)
        {
            int numero;
            elementoValido = int.TryParse(numerosIntroducidos[i], out numero);

            if (elementoValido)
            {
                vectorNumeros[i] = numero;
            }
        }

        if (!elementoValido)
        {
            throw new Excepciones.ArgumentoNoValidoException(elementoNoValido);
        }
        modificaciones = ModificarRepetidos(vectorNumeros);
    }
    return vectorNumeros;
}
```

A continuación, describiremos brevemente cada una de las pruebas unitarias realizadas, que se encuentran en el proyecto de pruebas TestApp3.

Clases de equivalencia:

App3.1PU) cadenaValores = "1,2,4,4,5,6,7,1,9,7": Introducción de una cadena de valores válidos, en este caso queremos verificar que obtenemos el **vector** con los números repetidos modificados por -1. Nombre del método de prueba: **VectorModificadoCorrecto**.

App3.2PU) cadenaValores = "1,2,4,4,5,6,7,1,9,7": Introducción de una cadena de valores válidos, en este caso queremos verificar que obtenemos el **número** de valores modificados. Nombre del método de prueba: **NumValoresModificadosCorrecto**.

App3.3PU) cadenaValores = "1,2,4,4,5,6,7,1": Introducción de una cadena de valores no válida, faltan dos números por introducir. Nombre del método de prueba: **VectorModificadoFaltanNumeros**.

App3.4PU) cadenaValores = "1,2,4,4,5,6,7,1,9,7,2,1,8": Introducción de una cadena de valores no válida, sobran tres números. Nombre del método de prueba: **VectorModificadoSobranNumeros**.

App3.5PU) cadenaValores = "1,2,4,4,5,6,7,1,9,t": Introducción de una cadena de valores no válidos, se ha introducido un elemento no válido (t). Nombre del método de prueba: **VectorModificadoElementoNoValido**.

Prueba	cadena Valores	valores Modificados	vector Obtenido	vector Esperado	Resultado	Comentario
App3.1PU	"1,2,4,4,5,6,7,1,9,7"	3	{ 1, 2, 4, -1, 5, 6, 7, -1, 9, -1 }	{ 1, 2, 4, -1, 5, 6, 7, -1, 9, -1 }	Válido	Es correcto.
App3.3PU	"1,2,4,4,5,6,7,1"	0	ArgumentException	ArgumentException	Válido	Es correcto, faltan números por introducir.
App3.4PU	"1,2,4,4,5,6,7,1,9,7,2,1,8"	0	ArgumentException	ArgumentException	Válido	Es correcto, se han introducido números de más.
App3.5PU	"1,2,4,4,5,6,7,1,9,t"	0	ArgumentoNoValidoException	ArgumentoNoValidoException	Válido	Es correcto, se ha introducido algún elemento no numérico.

Prueba	cadena Valores	valores ModificadosObtenidos	Valores Modificados Esperados	Resultado	Comentario
App3.2PU	"1,2,4,4,5,6,7,1,9,7"	3	3	Válido	Es correcto.

Pruebas unitarias para la aplicación 4:

El método que pondremos a prueba en nuestra aplicación 4 (se introduce un vector de 10 números separados por comas y devuelve el vector al revés) es **ComprobarValoresEntrada**. Hemos realizado una refactorización previa con respecto a la versión anterior para poder controlar mejor los valores obtenidos y las excepciones que se pueden obtener, ya sea porque se introducen elementos no válidos (cadenas, signos, etc) o porque no se introduce la cantidad de números indicada. El código queda de la siguiente manera:

```
public static int[] ComprobarValoresEntrada(string cadenaValores)
{
    string[] numerosIntroducidos = cadenaValores.Split(',');
    bool elementoValido = true;
    const int kTamanyo = 10;
    int[] vectorLeido = new int[kTamanyo];
    int[] vectorAlReves;

    if (numerosIntroducidos.Length != kTamanyo)
    {
        throw new ArgumentException(cantidadNumerosNoValida);
    }
    else
    {
        for (int i = 0; i < kTamanyo && elementoValido; i++)
        {
            int numero;

            elementoValido = int.TryParse(numerosIntroducidos[i], out numero);

            if (elementoValido)
            {
                vectorLeido[i] = numero;
            }
        }

        if (!elementoValido)
        {
            throw new Excepciones.ArgumentoNoValidoException(elementoNoValido);
        }
        vectorAlReves = InvertirVector(vectorLeido);
    }
    return vectorAlReves;
}
```

A continuación, describiremos brevemente cada una de las pruebas unitarias realizadas, que se encuentran en el proyecto de pruebas TestApp4.

Clases de equivalencia:

App4.1PU) cadenaValores = “1,2,3,4,5,6,7,8,9,10”: Introducción de una cadena de valores válidos. Nombre del método de prueba: **VectorRevesCorrecto**.

App4.2PU) cadenaValores = “1,2,3,4,5,6,7,8”: Introducción de una cadena no válida, faltan números por introducir. Nombre del método de prueba: **VectorRevesFaltanNumeros**.

App4.3PU) cadenaValores = “1,2,3,4,5,6,7,8,9,10,11,12,13”: Introducción de una cadena no válida, sobran números. Nombre del método de prueba: **VectorRevesSobranNumeros**.

App4.4PU) cadenaValores = “1,2,3,4,5,6,7,8,9,t”: Introducción de una cadena de valores válidos, se ha introducido un elemento no válido (t). Nombre del método de prueba: **VectorRevesElementoNoValido**.

Prueba	cadenaValores	vectorAlRevesObtenido	vectorAlRevesEsperado	Resultado	Comentario
App4.1PU	“1,2,3,4,5,6,7,8,9,10”	{ 10, 9, 8, 7, 6, 5, 4, 3, 2, 1 }	{ 10, 9, 8, 7, 6, 5, 4, 3, 2, 1 }	Válido	Es correcto.
App4.2PU	“1,2,3,4,5,6,7,8”	ArgumentException	ArgumentException	Válido	Es correcto, faltan números por introducir.
App4.3PU	“1,2,3,4,5,6,7,8,9,10,11,12,13”	ArgumentException	ArgumentException	Válido	Es correcto, se han introducido números de más.
App4.4PU	“1,2,3,4,5,6,7,8,9,t”	ArgumentoNoValidoException	ArgumentoNoValidoException	Válido	Es correcto, se ha introducido algún elemento no numérico.