



Centro Integrado de Formación Profesional
AVILÉS
Principado de Asturias

UNIDAD 8: REALIZACIÓN DE PRUEBAS

DESARROLLO DE INTERFACES

2º CURSO

C.F.G.S. DESARROLLO DE APLICACIONES MULTIPLATAFORMA

REGISTRO DE CAMBIOS

Versión	Fecha	Estado	Resumen de cambios
1.0	02/03/2024	Aprobado	Primera versión

ÍNDICE

ÍNDICE	2
UNIDAD 8: REALIZACIÓN DE PRUEBAS	3
8.1. Objetivo, importancia y limitaciones del proceso de prueba. Estrategias	3
8.2. Pruebas de integración: ascendentes y descendentes	6
8.3. Pruebas de sistema	7
8.4. Pruebas de seguridad	8
8.5. Pruebas manuales y automáticas. Herramientas software para la realización de pruebas ...	10
8.5.1. Pruebas funcionales.....	10
8.5.2. Pruebas unitarias	11
8.5.3. Herramientas software para realización de pruebas.....	20
8.6. Pruebas de regresión.....	21
8.7. Pruebas de capacidad y rendimiento. Pruebas de uso de recursos	23
8.8. Pruebas de usuario.....	25
8.9. Pruebas de aceptación	26
8.9.1. Desarrollo del plan de pruebas.....	26
8.10. Versiones alfa y beta.	27
ÍNDICE DE FIGURAS.....	27
BIBLIOGRAFÍA – WEBGRAFÍA	27

UNIDAD 8: REALIZACIÓN DE PRUEBAS

8.1. OBJETIVO, IMPORTANCIA Y LIMITACIONES DEL PROCESO DE PRUEBA. ESTRATEGIAS

Las pruebas de software son investigaciones empíricas y técnicas cuyo objetivo es proporcionar información objetiva e independiente sobre la calidad del producto desarrollado al futuro usuario y/o comprador. Son una actividad más en el proceso de calidad. Básicamente, se trata de un conjunto de actividades como una etapa más dentro del desarrollo de software.

La prueba exhaustiva de software es impracticable ya que no se pueden probar todas las posibilidades de su funcionamiento ni siquiera en programas sencillos. Por ejemplo, en una aplicación simple de suma de dos números es imposible probar todos los números existentes a sumar. El objetivo de las pruebas no es asegurar la ausencia de defectos en un software, sino demostrar que existen.

Dado que el objetivo de las pruebas es detectar defectos en el software, el hallazgo de aquellos no implica que se sea mal profesional ya que es inherente al ser humano cometer errores. Es más, el descubrimiento de un defecto es un éxito en la mejora de calidad del producto.

En la terminología de pruebas se ubica el proceso de verificación que implica la evaluación de un sistema o de uno de sus componentes para determinar si los productos de una fase dada satisfacen las condiciones impuestas al comienzo de dicha fase, es decir, responde a la pregunta “¿estamos construyendo el producto correctamente?”. Por otra parte, existe el conocido como proceso de validación el cual trata la evaluación de un sistema o de uno de sus componentes durante o al final del proceso de desarrollo para determinar si satisface los requisitos marcados por el usuario, es decir, responde a la pregunta “¿estamos construyendo el producto correcto?”.

A continuación, se verán una serie de conceptos importantes en el campo de las pruebas de software:

- Prueba (test). Actividad en la cual un sistema o uno de sus componentes se ejecuta en circunstancias previamente especificadas observando sus resultados y procediendo a su registro. Este proceso de evaluación abarca uno o varios aspectos concretos del software en desarrollo.
- Casos de prueba (test case). Conjunto de entradas, condiciones de ejecución y resultados esperados desarrollados para un objetivo particular.
- Defecto (fault, bug). Defecto en el software, por ejemplo, definición de datos o paso de procesamiento incorrectos en un programa. En el ejemplo de la suma, el software es incapaz de cargar los sumandos.

- Fallo (failure). Incapacidad de un sistema o de alguno de sus componentes para realizar las funciones requeridas dentro de los requisitos de rendimiento especificados. Por ejemplo, cargar los datos, pero no sumarlos.
- Error (error). Diferencia entre un valor calculado, observado o medio y valor verdadero, especificado o teóricamente correcto. Por ejemplo, el resultado de la suma es incorrecto.

Las pruebas de caja negra o integración, también conocidas como pruebas de comportamiento, se basan en las especificaciones del programa o componente sirviendo estas como instrumento para elaborar los casos de prueba sin tener en cuenta su diseño o codificación interna. El componente se ve como una “caja negra” cuyo comportamiento solo puede ser determinado estudiando sus entradas y salidas obtenidas a partir de ellas.

Las pruebas de caja blanca o estructurales se centran en atender al comportamiento interno y estructura de un programa para diseñar los casos de prueba. Se examina así la lógica interna del programa sin tener en cuenta los aspectos de rendimiento. El objetivo es diseñar casos de prueba que ejecuten al menos una vez todas las sentencias del programa y todas las condiciones, tanto en su vertiente verdadera como falsa. De estas pruebas se obtiene lo que se conoce como complejidad ciclomática de los algoritmos.

Para la realización de pruebas con éxito, se enumeran a continuación una serie de recomendaciones:

- Cada caso de prueba debe definir el resultado de salida esperado que se comparará con el resultado obtenido.
- El programador debe editar probar su propio software ya que desea (consciente e inconscientemente) demostrar que funciona sin problemas. Además, es normal que las situaciones que olvidó considerar al crear el programa queden fuera de los casos de prueba.
- Se debe inspeccionar a conciencia el resultado de cada prueba y, así, poder descubrir posibles indicios de defectos.
- Al generar casos de prueba se deben incluir tanto datos de entrada válidos y esperados como no válidos e inesperados.
- Las pruebas deben centrarse en dos objetivos:
 - Evitar los casos deseables, es decir, los no documentados ni diseñados cuidadosamente
 - No deben hacerse planes de prueba suponiendo que no va a haber apenas defectos en los programas, lo que deriva en dedicar escasos recursos al diseño de pruebas.
- Probar si el software no hace lo que debe.
- Probar si el software hace lo que debe, es decir, si provoca efectos secundarios adversos.
- La experiencia suele indicar que donde hay un defecto se encuentran otros, es decir, la probabilidad de descubrir nuevos defectos es proporcional al número de defectos ya descubiertos.

- Las pruebas son una tarea tanto o más creativa que el desarrollo de software en contra de la creencia de que son una tarea destructiva y rutinaria.
- Es interesante planificar y diseñar las pruebas para poder detectar el máximo número y variedad de defectos con el mínimo consumo de tiempo y esfuerzo.

A continuación, se enumeran las tareas a realizar para probar un software siguiendo un orden establecido:

- Diseño de las pruebas. Identificación de la técnica o técnicas de pruebas que se van a utilizar. Distintas técnicas de pruebas ejercitan diferentes criterios sirviendo de guía para realizarlas.
- Generación de los casos de prueba. Estos elementos representan los datos que se usarán como entrada para ejecutar el software a probar. Concretamente, determinan un conjunto de entradas, condiciones de ejecución y resultados esperados para un objetivo particular.
- Definición de procedimientos de prueba. Especificación de cómo se va a llevar a cabo el proceso, quién lo va a realizar, cuándo se hará, etc.
- Ejecución de la prueba. Se aplican los casos de prueba generados anteriormente e identificar los posibles fallos producidos al comparar resultados obtenidos como esperados
- Realización de un informe de la prueba. Con el resultado de ejecución de las pruebas, qué casos pasaron satisfactoriamente cuáles no y qué fallos se detectaron.

8.2. PRUEBAS DE INTEGRACIÓN: ASCENDENTES Y DESCENDENTES

Aunque los módulos de un programa funcionen correctamente por separado, es preciso probarlos conjuntamente. Un módulo puede tener un efecto adverso o inadvertido sobre otro módulo. Los métodos, cuando se combinan, pueden producir un resultado indeseado; la imprecisión aceptada de forma individual puede crecer hasta niveles inaceptables al combinar los módulos. Por tanto, se hace necesario probar el software ensamblando los módulos probados anteriormente, lo que constituye el objetivo de las pruebas de integración.

Existe una tendencia que intenta lo que se conoce como integración no incremental, la cual consiste en combinar todos los módulos y probar el programa en su conjunto. Esta forma de trabajar puede dar lugar a un resultado algo caótico con un gran conjunto de fallos y la consecuente dificultad para identificar qué módulos los han provocado. Para contrarrestar esto, existe otro tipo de integración incremental en la que el programa se prueba en pequeñas porciones en las que los fallos son más fáciles de detectar. Existen dos tipos de integraciones incrementales: ascendentes y descendentes.

La integración incremental ascendente opera del siguiente modo:

- Se combinan módulos de bajo nivel en grupos que realicen una función específica
- Se escribe un controlador que coordine la entrada y salida de los casos de prueba
- Se prueba el grupo
- Se eliminan los controladores y se combinan los grupos moviéndose hacia arriba por la estructura del programa.

En cuanto a la descendente, su operativa sigue los pasos a continuación:

- Se usa el módulo de control principal como controlador de la prueba creando resguardos para todos los módulos directamente subordinados al principal. Estos resguardos son módulos que a su vez simulan los que utiliza el módulo que se está probando.
- Dependiendo del enfoque e integración elegido, se van sustituyendo uno a uno los resguardos subordinados por los módulos reales. Enfoques posibles pueden ser primero-en-profundidad o primero-en-anchura.
- Se llevan a cabo pruebas cada vez que se integra un nuevo módulo.
- Tras terminar cada conjunto de pruebas, se reemplaza otro resguardo con el módulo real.

8.3. PRUEBAS DE SISTEMA

Este tipo de pruebas tiene como misión probar profundamente el sistema para verificar que se han integrado adecuadamente todos los elementos del sistema (hardware, software de terceros, etc.) y que estos realizan las funciones adecuadas. Concretamente, es preciso comprobar:

- El cumplimiento los requisitos funcionales establecidos
- El funcionamiento y rendimiento de las interfaces hardware, software y de usuario
- La adecuación de la documentación del usuario
- Rendimiento y respuesta en condiciones límite y de sobrecarga

Para la generación de casos de prueba de sistema se utiliza el enfoque de caja negra. Este tipo de pruebas se suelen hacer en el entorno del desarrollador, denominadas Pruebas Alfa y, seguidamente, en el entorno del cliente denominadas Pruebas Beta.

Se distinguen los siguientes tipos de prueba:

- Pruebas de comunicación. Comprueban que las interfaces tanto remotas como locales funcionan adecuadamente.
- Pruebas de rendimiento. Comprueban los tiempos de respuesta de la aplicación.
- Pruebas de recuperación. Se fuerza el fallo del software para comprobar su capacidad de recuperación.
- Pruebas de volumen. Se comprueba su funcionamiento con cantidades próximas a la capacidad total de procesamiento.
- Pruebas de sobrecarga. Similares a las anteriores, pero en el límite de su capacidad.
- Pruebas de tensión. Similares a las de volumen restringiendo el tiempo disponible (equivalente a determinar la velocidad de trabajo).
- Pruebas de disponibilidad de datos. Comprueban si tras la recuperación el sistema mantuvo la integridad de los datos.
- Pruebas de operación. Comprueban la correcta implementación de los procedimientos de operación incluyendo la planificación y control de trabajos, arranque y reinicio del sistema, etc.
- Pruebas de entorno. Comprueban la interacción con otros sistemas.
- Pruebas de usabilidad. Comprueban la usabilidad del sistema, tal como se vio en la unidad correspondiente.
- Pruebas de almacenamiento. Se comprueba la cantidad de memoria principal y secundaria que usa el programa y el tamaño de archivos temporales.
- Pruebas de configuración. Se intenta comprobar el software en distintos escenarios de configuración que abarcan el tipo de hardware, sistema operativo, si hay o no un antivirus.
- Pruebas de documentación. Hacen referencia tanto a documentación técnica para desarrolladores como a la del usuario final.

8.4. PRUEBAS DE SEGURIDAD

Los problemas de seguridad en las aplicaciones que desarrolla una empresa repercuten directamente en su imagen ante el mercado afectando fuertemente en su negocio. La correcta identificación y corrección de los posibles problemas de seguridad en una etapa temprana del desarrollo permite ahorrar trabajo, reducir los costes y aumentar la calidad de la aplicación final, mejorando el desempeño global.

La prueba de seguridad intenta verificar los mecanismos de protección incorporados en el sistema. Estas pruebas van desde la introducción de datos incorrectos o inapropiados, intentos de acceso no autorizados, pruebas de control de integridad de datos, etc.

En el ámbito de la seguridad de aplicaciones, se pueden distinguir tres tipos de pruebas: **estáticas, dinámicas e interactivas.**

Las **pruebas estáticas de seguridad** de aplicaciones, también conocidas por su acrónimo **SAST (Static Application Security Testing)** son **pruebas de caja blanca** que se basan en el análisis del código fuente, por lo que no pueden encontrar problemas en tiempo de ejecución ni probar dependencias externas.

Usan reconocimiento de patrones de las funciones y propiedades internas con lo que se consigue un análisis de lenguaje muy técnico y una conclusión matemática. Se suelen usar en muestras de aplicaciones sospechosas de ser muy dañinas y de fácil propagación o en desarrollo de nuevas aplicaciones para tener una primera impresión de posibles mejoras.

Las **herramientas de pruebas dinámicas (Dynamic Application Security Testing - DAST)** se despliegan después de completar y compilar una aplicación. Este tipo de herramientas no se ocupa tanto de las vulnerabilidades dentro del código, ya que se entiende que una herramienta SAST ya las ha eliminado, sino que actúan como un probador externo tratando de atacar una aplicación. Un ejemplo de uso de ataque puede basarse en utilizar las interfaces HTTP y HTML expuestas en una web. Este tipo de herramientas incluso pueden ser configuradas para que busquen vulnerabilidades a ataques frecuentes en sectores específicos como el financiero.

Además de lo ya citado, la diferencia sustancial con respecto a las estáticas es que estas necesitan de un soporte específico para su lenguaje de programación, mientras que las herramientas DAST en su mayoría no lo requieren, aunque también pueden ser capaces de trabajar con el código fuente con el fin de localizar vulnerabilidades.

Aunque se puede usar exclusivamente una herramienta DAST o SAST, lo normal en la actualidad es que para una organización o empresa sea más seguro para las organizaciones implementar ambas o trabajar con una herramienta que tenga ambos componentes.

Las pruebas interactivas son también conocidas por el acrónimo IAST (Interactive Application Security Testing). Son aquellas pruebas que se realizan mientras la aplicación es ejecutada directamente por un ser humano. Principalmente se usan para detectar fallos en el rendimiento y combinaciones no estándar del uso de los controles, los cuales se van informando en tiempo real, por lo que son más eficaces que las que se han visto previamente.

8.5. PRUEBAS MANUALES Y AUTOMÁTICAS. HERRAMIENTAS SOFTWARE PARA LA REALIZACIÓN DE PRUEBAS

Las pruebas manuales se realizan ejecutando el software, introduciendo datos de entrada e inspeccionando la salida. Es un proceso sencillo que no requiere aprendizaje previo. Además, se ejecutan exactamente como lo haría el usuario final, pero tienen algunos inconvenientes:

- Son repetitivas. Cada vez que se cambia o añade una nueva funcionalidad o se corrige un error, se necesitan volver a probar las aplicaciones para asegurarse de que no se ha producido un error por causa de ese cambio.
- Susceptible de error. Precisamente por ser repetitivas, posiblemente se pasen por alto detalles o que no se prueben funcionalidades aparentemente no relacionadas con el cambio, pero realmente afectadas.
- Solo se prueba lo que se ve. Esto significa que solo se prueba indirectamente aquello que no se ve y es, especialmente significativo en el caso de aplicaciones en la parte del servidor. Se necesita que funcionen correctamente la capa de presentación y negocio, pero sobre todo aquellas residentes en el servidor.

Las pruebas automatizadas se integran en las metodologías modernas de una forma integral en el desarrollo del software. Este tipo de pruebas es repetible y portable. Repetible significa que la prueba se puede repetir indefinidamente produciendo los mismos resultados y portable a que otras personas pueden ejecutar la prueba y verificar que el software pasa todas las pruebas. Por otra parte, la prueba evoluciona con el software de forma que, si los requisitos y funcionalidades cambian, la prueba también debe hacerlo. Realizando pruebas automáticas, se pueden tener dos tipos: prueba funcional y unitaria.

8.5.1. PRUEBAS FUNCIONALES

Una prueba funcional está basada en la ejecución, revisión y retroalimentación de las funcionalidades previamente diseñadas para el software. Las pruebas funcionales se hacen mediante el diseño de modelos de prueba que buscan evaluar cada una de las opciones con las que cuenta el paquete informático.

El objetivo de la prueba funcional es validar cuándo el comportamiento observado del software probado cumple o no con sus especificaciones. La prueba funcional toma el punto de vista del usuario. En este tipo de pruebas, las funciones se comprueban ingresando las entradas y examinando las salidas, la estructura del programa raramente se tiene en cuenta.

Para realizar este tipo de pruebas, se analizan las especificaciones para obtener los casos de prueba. Técnicas como partición de equivalencia, análisis del valor límite, grafos causa efecto y conjetura de errores son adecuadas para este tipo de pruebas. Además, se deben considerar condiciones no válidas e inesperadas de la entrada y tener en cuenta que la definición del resultado esperado es una parte vital de un caso de prueba. El propósito de este tipo de pruebas es mostrar discrepancias con la especificación, no demostrar que el programa cumple con ella.

8.5.2. PRUEBAS UNITARIAS

Las pruebas unitarias prueban una unidad de trabajo que implementan una operación lógica accesible mediante la interfaz pública. En primer lugar, hay que tener en cuenta que una prueba unitaria no tiene barra libre para acceder a cualquier parte del sistema de información. A continuación, se pueden ver una serie de restricciones y características que debe cumplir una prueba de esta tipología. En definitiva, una prueba unitaria:

- No accede a la base de datos subyacente
- No accede al sistema de archivos
- No accede a la red
- No requiere una configuración de entorno

Los Frameworks de gestión y creación de pruebas unitarias más conocidos pertenecen a la familia xUnit. Por ejemplo, en Java se utiliza JUnit, en .NET NUnit, PHPUnit para PHP y para Smalltalk está SUnit. Otro muy conocido es unittest, propio de Python.

Una prueba unitaria sigue una estructura conocida como triple A:

- Arrange: Se prepara el contexto de la prueba
- Act: Se ejecuta el método que se está probando
- Assert: Verifica que ocurrió lo que se esperaba

Para ilustrar este concepto, primero se va a crear una aplicación sin ningún tipo de Framework de pruebas unitarias. Por tanto, se creará un proyecto de consola que se puede nombrar como **StringTests**. La plantilla de aplicación de consola contiene un "Hello World". La idea de este proyecto sin Framework es que muestre el mensaje "Pruebas pasadas con éxito" si se ha pasado el test o que muestre una excepción en caso contrario. Si se ha llegado hasta el Console.WriteLine, es que las pruebas se han pasado correctamente.

```
Console.WriteLine("Pruebas pasadas con éxito");
```

Por tanto, antes de esta línea, se introducirán las llamadas a los métodos que realizan las pruebas. La primera prueba comprobará que la cadena "Hola" tiene una longitud 4. El método de test unitario se compondrá de las tres etapas de la triple A: Arrange, Act y Assert. Por ejemplo, se podría construir de este modo:

```
static void TestLenStringHelloIs4()
{
    // Arrange
    string hello = "Hola";

    // Act
    int helloLength = hello.Length;

    // Assert
    if (helloLength != 4) throw new Exception("Prueba fallida");
}
```

Se ve primero que se prepara el contexto mediante un Arrange que inicializa la variable hello con el valor "Hola", se actúa obteniendo la longitud, que es lo que interesa en este caso y por fin se verifica si el resultado es el esperado comprobando si la longitud es distinta de 4, que es cuando se lanzará la excepción. Si ahora se antepone la llamada de este método al Console.WriteLine, se podrá ver que el sistema finaliza con un mensaje "Pruebas pasadas con éxito".

```
1 // See https://aka.ms/new-console-template for more information
2 TestLenStringHelloIs4();
3 Console.WriteLine("Pruebas pasadas con éxito");
4 static void TestLenStringHelloIs4()
5 {
6     // Arrange
7     string hello = "Hola";
8
9     // Act
10    int helloLength = hello.Length;
11
12    // Assert
13    if (helloLength != 4) throw new Exception("Prueba fallida");
14 }
15
16
17
```

Consola de depuración de Microsoft Visual Studio

Pruebas pasadas con éxito

1. Prueba unitaria de comprobación de longitud

A partir de aquí puede forzarse el fallo del método. Por ejemplo, si se cambia el valor de la variable hello y este no tiene cuatro caracteres, el sistema lanzará una excepción al no pasar la prueba. También si se modifica el Assert y se verifica una longitud distinta (por ejemplo, 5).

Ahora puede crearse otra prueba que verifique si la cadena hello contiene la letra H. Por tanto, se volverá a crear otro método con la misma estructura, pero distinta ejecución.

```
static void TestStringHelloContainsH()
{
    // Arrange
    string hello = "Hola";

    // Act
    bool helloContainsH = hello.Contains("H");

    // Assert
    if (!helloContainsH) throw new Exception("Prueba fallida");
}
```

De igual modo, ahora se comprueba que la palabra en sí contiene la letra y si no es así, la prueba fallará. Por supuesto, también debe incluirse una llamada a este método antes del `Console.WriteLine`.

Hasta aquí se ha sobrevivido sin necesidad de un Framework y puede seguir haciéndose si se ajusta el código para hacerlo más elegante. Para eso, se añadirá una clase estática que agrupe las funcionalidades relacionadas con el `Assert`. Por supuesto, se llamará con ese mismo nombre. Por tanto, en el proyecto hay que acudir a su menú contextual y lanzar **Agregar > Clase** dándole el nombre `Assert.cs` y con el siguiente código:

```
public static class Assert
{
    public static void That(bool condition)
    {
        if (!condition) throw new Exception("Error en Assert");
    }
}
```

Ahora ya se puede tener un código más elegante en ambos métodos de prueba, pero hay que recordar que ahora se comprueba que la condición se cumple y no lo contrario:

```
static void TestLenStringHelloIs4(){
    // Arrange
    string hello = "Hola";
    // Act
    int helloLength = hello.Length;
    // Assert
    Assert.That(helloLength == 4);
}
static void TestStringHelloContainsH(){
    // Arrange
    string hello = "Hola";
    // Act
    bool helloContainsH = hello.Contains("H");
    // Assert
    Assert.That(helloContainsH);
}
```

Hasta ahora se ha visto una forma artesanal de construir pruebas que podría mejorarse con reflexión, teniendo en cuenta que se pueden diseñar métodos que, por ejemplo, contengan un patrón de nombrado (por ejemplo, incluir la palabra `Test`) y lanzar la batería de pruebas con un bucle que los recorra y los ejecute sin tener que preocuparse de incluirlos explícitamente. Ese, por cierto, es el mecanismo que usan los Frameworks que se han visto hasta ahora. En .NET, tal como ya se adelantó, el Framework de pruebas se llama [NUnit](#) y no deja de ser una versión portada del [JUnit](#) de Java.

Para poder utilizarlo, se va a agregar el paquete correspondiente mediante el gestor de paquetes Nuget. Se necesitarán `NUnit` y `NUnit3TestAdapter`.



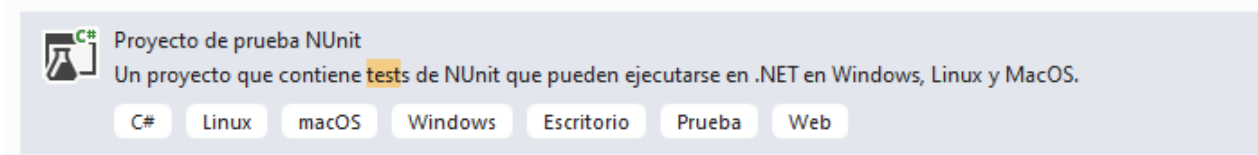
NUnit por Charlie Poole, Rob Prouse, 326M descargas
NUnit is a unit-testing framework for all .NET languages with a strong TDD focus.



NUnit3TestAdapter por Charlie Poole, Terje Sandstrom, 246M descargas
NUnit3 adapter for running tests in Visual Studio and DotNet. Works with NUnit 3.x, use the NUnit 2 adapter for 2.x tests.

2. NUnit en Nuget

A continuación, habría que añadir un proyecto de pruebas dentro de la solución en la que esté el proyecto con el desarrollo principal. En Visual Studio, agregando el proyecto e introduciendo como patrón de búsqueda test, se puede obtener el tipo de proyecto requerido.



3. Proyecto de prueba NUnit

Dado que al proyecto principal se le llamó StringTests y que en este caso debería tener un nombre que contenga también la cadena Tests, se le puede llamar **StringTestsNUnit** para destacar que en este proyecto se usa dicho Framework.

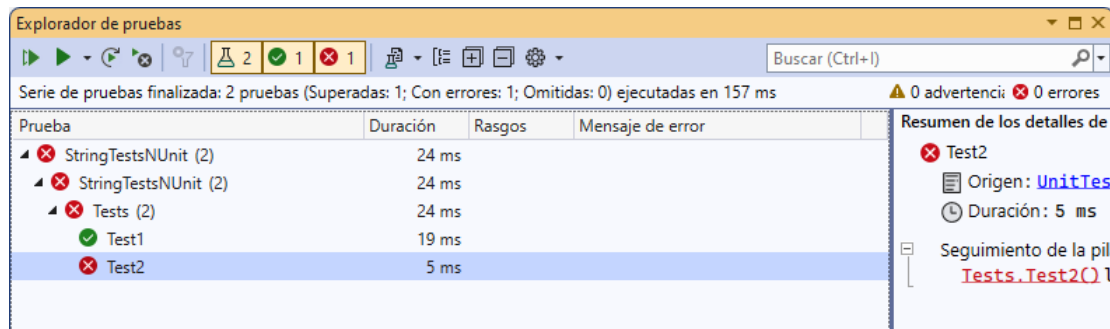
Una vez creado, el proyecto añade un fichero plantilla con una prueba unitaria llamada UnitTest1.cs y el siguiente código:

```
namespace StringTestsNUnit
{
    public class Tests
    {
        [SetUp]
        public void Setup()
        {
        }
        [Test]
        public void Test1()
        {
            Assert.Pass();
        }
    }
}
```

En principio, lo que más interesa son las anotaciones de tipo Test, las cuales delimitan que el método al que se refieren es una prueba. Para ver el funcionamiento de estas pruebas, se va a crear otra con el siguiente código:

```
[Test]
public void Test2()
{
    Assert.Fail();
}
```

Como puede verse, la diferencia con la anterior es que se cambia el método llamado con Assert de Pass a Fail. En definitiva, se fuerza el fallo de la prueba. Para ver cómo se lanzan estas pruebas, hay que acudir al menú contextual del proyecto de pruebas y elegir la opción **Ejecutar pruebas**. La salida en este caso será como se muestra en la figura y se podrá comprobar cómo se pasa la primera prueba mientras que la segunda produce un fallo.



4. Ejecución de pruebas

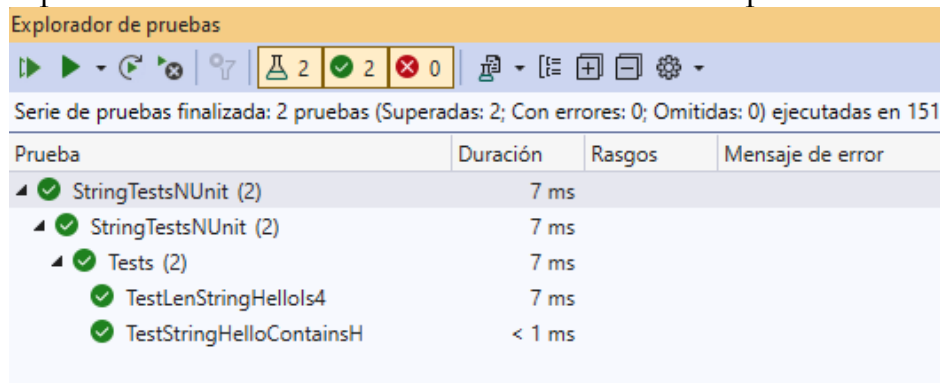
Bien, ahora se pueden trasladar las pruebas realizadas anteriormente para que sigan el formato de NUnit. Su código se muestra a continuación y se explicará posteriormente.

```
[Test]
public void TestLenStringHelloIs4()
{
    // Arrange
    string hello = "Hola";
    // Act
    int helloLength = hello.Length;
    // Assert
    Assert.That(helloLength, Is.EqualTo(4));
}

[Test]
public void TestStringHelloContainsH()
{
    // Arrange
    string hello = "Hola";
    // Act
    bool helloContainsH = hello.Contains("H");
    // Assert
    Assert.IsTrue(helloContainsH);
}
```

Los cambios realizados son muy pequeños. De hecho, se podría haber dejado el Assert.That de la clase que se creó de forma artesanal para realizar pruebas. En cualquier caso, NUnit dispone de métodos más precisos que afinan mucho más en las comprobaciones y permiten una mayor variedad de opciones; basta con mirar las opciones que muestra el autocompletado al introducir la clase Assert.

La ejecución de pruebas vuelve a ser correcta con las nuevas incorporaciones:



Prueba	Duración	Rasgos	Mensaje de error
StringTestsNUnit (2)	7 ms		
StringTestsNUnit (2)	7 ms		
Tests (2)	7 ms		
TestLenStringHellols4	7 ms		
TestStringHelloContainsH	< 1 ms		

5. Ejecución de pruebas de cadena

Al ejecutar una prueba se puede dar cualquiera de estos cuatro posibles resultados:

- Exitoso o pasado (pass).
- Fallido (fail)
- Inconcluso (error no esperado)
- Ignorado (su ejecución no ha sido considerada)

Es interesante analizar lo que significa que el resultado de una prueba es inconcluso. Este tipo de resultado tiene mucho más sentido en otro tipo de pruebas como son las de integración. Por ejemplo, si se va a probar que funciona una consulta, pero no se llega a poder conectar a la base de datos, no se puede asegurar que la consulta falla porque aún no se ha llegado a ejecutar. En ese caso, se estaría ante un resultado inconcluso. Se puede forzar dicho estado con `Assert.Inconclusive()`

El resultado ignorado puede ser muy útil si se desea que alguna prueba no se tenga en cuenta en la batería mientras se está desarrollando algo o porque se desea desactivar de forma temporal. Hay dos formas de ignorar una prueba. La primera es sencillamente lanzar un `Assert.Ignore()` dentro de su código y la segunda es modificar la anotación `Test` por la siguiente:

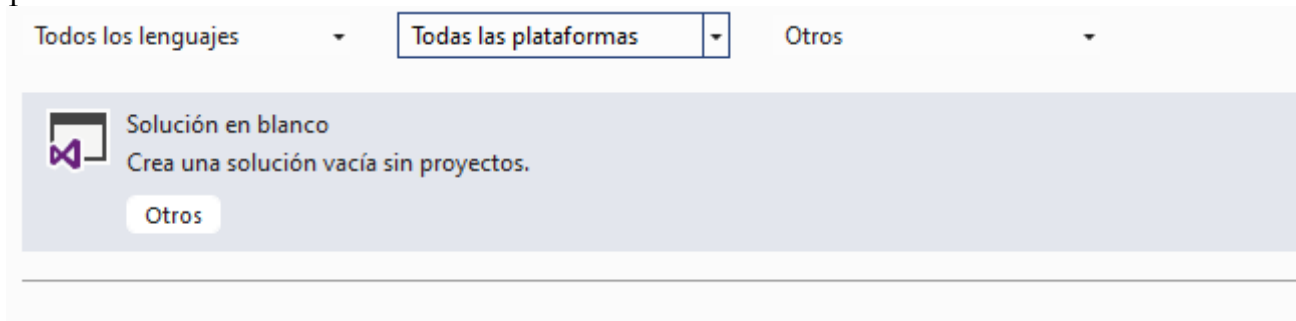
```
[Test, Ignore("Desactivado temporalmente por ...")]  
public void TestIgnored()  
{  
    // Código de la prueba  
}
```

En este caso ni se entra en el código de la prueba; se ignora completamente.

Hasta ahora se han realizado ejemplos ilustrativos, pero de muy poco recorrido, ya que se han hecho sobre un modelo teórico que no tiene aplicación práctica. A continuación, se va a ver un ejemplo sencillo que podría asemejarse a casos reales.

Supóngase que se va a probar una biblioteca de clases que recoge la lógica de negocio de una aplicación de gestión bancaria. Aquí solo se probará que sus clases están operativas y

funcionan correctamente. Dado que se entiende que la biblioteca se utilizará con una aplicación principal (sea web, escritorio o móvil), primero se creará una solución vacía donde se añadirán posteriormente los proyectos que se requieran. En la creación de proyecto, se elige la siguiente opción:



6. Solución en blanco

Por ejemplo, se puede estar creando una solución para gestión bancaria de una entidad llamada BancAvilés, por lo que se puede utilizar ese nombre para la solución, es decir **BancAviles** (sin tilde).

En ella se añade primero un proyecto de biblioteca de clases que contendrá la lógica de negocio. Por tanto, se le puede llamar **BancAvilesLogic**.

En ese proyecto van a existir dos clases:

- Cliente. Representa a un cliente de la entidad.
- Cuenta. Representa una cuenta bancaria de la entidad

Como se podría suponer, una cuenta va a tener un cliente como titular. Ambas clases se van a crear con los mínimos elementos para mostrar simplemente la operatividad justa y requerida para las pruebas.

Código de la clase Cliente:

```
public class Cliente
{
    public string Nombre { get; set; }
    public string Apellidos { get; set; }
    public Cliente(string nombre, string apellidos)
    {
        Nombre = nombre;
        Apellidos = apellidos;
    }
}
```

Código de la clase Cuenta:

```
public class Cuenta
{
    public Cliente Titular { get; set; }
    public double Saldo { get; set; }

    public Cuenta(Cliente titular)
    {
        Titular = titular;
        Saldo = 0;
    }

    public void Ingresar(double cantidad)
    {
        Saldo += cantidad;
    }

    public void Extraer(double cantidad)
    {
        Saldo -= cantidad;
    }
}
```

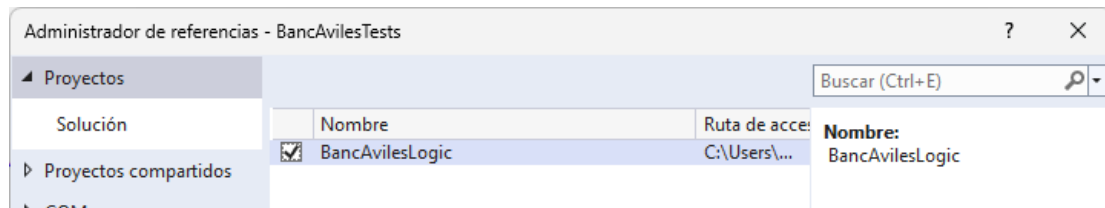
Como puede verse, esta última clase muestra cómo se inicializa una cuenta al construir un objeto de ese tipo y consta de dos métodos, uno para ingresar dinero y otro para extraerlo. Por tanto, aquí se puede ver de primeras que podría haber tres pruebas interesantes:

1. Comprobar que el saldo de una cuenta al inicializarla es 0
2. Comprobar que el saldo de una cuenta recién inicializada se incrementa al realizar un ingreso.
3. Comprobar que el saldo de una cuenta recién inicializada se decrementa al extraer una cantidad de dinero.

Bien, ahora hay que crear un nuevo proyecto de prueba con NUnit que permita realizar estas pruebas. Se le puede dar como nombre **BancAvilesTests**.

Cuando se crea la batería de pruebas de ejemplo, sigue llamando la atención un método llamado `Setup()` que aún no se ha utilizado. Con este método se pueden inicializar variables comunes a todas las pruebas. En este ejemplo se ha partido de que se creará una cuenta nueva, por lo que esa inicialización se puede delegar al `Setup`. Es muy importante tener en cuenta que `Setup()` se ejecutará de nuevo antes de cada método de prueba. Es decir, aunque como se verá en el código posterior, se va a utilizar una variable privada **cuenta** para el cuerpo de cada prueba, cada vez que se ejecute su método correspondiente, se llamará a ese método `Setup` creando una nueva cuenta cada vez.

Antes de codificar las pruebas, hay que añadir una dependencia al proyecto de pruebas para que sea capaz de reconocer al de lógica de negocio. Esta operación se realiza en el apartado Dependencias de BancAvilesTest > Agregar referencia del proyecto. Aparece un cuadro de diálogo y se selecciona el otro proyecto de la solución, tal como puede verse en la imagen.



7. Añadido de referencia de proyecto

Ahora se muestra el código pasando a explicarlo posteriormente.

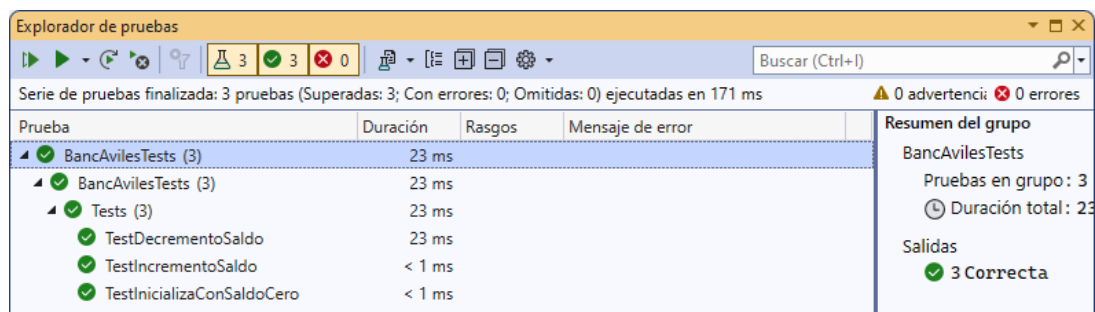
```
using BancAvilesLogic;
namespace BancAvilesTests
{
    public class Tests
    {
        private Cuenta cuenta;
        [SetUp]
        public void Setup()
        {
            Cliente titular = new Cliente("Juan", "Nadie");
            this.cuenta = new Cuenta(titular);
        }

        [Test]
        public void TestInicializaConSaldoCero()
        {
            Assert.That(cuenta.Saldo, Is.EqualTo(0));
        }

        [Test]
        public void TestIncrementoSaldo()
        {
            double cantidad = 10;
            cuenta.Ingresar(cantidad);
            Assert.That(cuenta.Saldo, Is.EqualTo(cantidad));
        }

        [Test]
        public void TestDecrementoSaldo()
        {
            double cantidad = 10;
            cuenta.Extraer(cantidad);
            Assert.That(cuenta.Saldo, Is.EqualTo(-cantidad));
        }
    }
}
```

En primer lugar, en Setup se puede ver cómo se inicializa una cuenta a nombre de Juan Nadie. El comportamiento esperado al dar de alta una cuenta es que su saldo sea cero, por lo que la primera prueba comprueba que eso es así. La siguiente prueba se encarga de verificar si cuando se produce un ingreso se refleja en el saldo. De modo similar, la última prueba chequea que cuando se extrae una cantidad, el saldo también lo refleja, siendo en este caso negativo porque conviene recordar que una cuenta se ha inicializado con 0. Se pueden realizar modificaciones en las cuales incluir un saldo inicial, aunque hay que recordar que se debe tener en cuenta en los Assert. Si se ejecutan, debería mostrar que se han pasado las tres:



8. Ejecución de pruebas de BancAvilés

¿Cuál es la reflexión sobre el sentido de este tipo de pruebas? Si en algún momento se modifica la clase y se cambia el comportamiento (por ejemplo, que al inicializar la clase el saldo sea de 100 como deferencia del banco), es posible que las pruebas ya no pasen correctamente y si se produce esta incidencia, es bastante probable que la aplicación ya no funcione correctamente. Por tanto, en cada compilación de la aplicación que esté encaminada a un despliegue, es imprescindible lanzar la batería completa de pruebas para verificar el buen funcionamiento del sistema.

8.5.3. HERRAMIENTAS SOFTWARE PARA REALIZACIÓN DE PRUEBAS

Entre las herramientas que existen para las pruebas automatizadas de software, además de las ya citadas para pruebas unitarias, se encuentran las siguientes:

- [Bugzilla](#). Es un sistema de propósito general basado en web de seguimiento de bugs desarrollado y utilizado por el proyecto Mozilla. Realmente consiste en una base de datos que va almacenando los defectos encontrados en las diferentes versiones.
- [Apache JMeter](#). Se trata de una herramienta de código abierto desarrollada en Java y diseñada para realizar pruebas de carga del comportamiento funcional y medir rendimiento. Originalmente fue diseñado para probar aplicaciones web, pero desde entonces se ha expandido a otras funciones de prueba.

8.6. PRUEBAS DE REGRESIÓN

Se denominan pruebas de regresión a aquellas que intentan descubrir las causas de nuevos errores o bugs, carencias de funcionalidad o divergencias funcionales con respecto al comportamiento esperado del software. Estos errores pueden ser inducidos por cambios recientemente realizados en parte de la aplicación que anteriormente no provocaban ningún fallo. Esto implica que el error tratado deriva de los cambios o modificaciones realizadas para una nueva versión de la aplicación. Este tipo de cambio se puede deber a prácticas no adecuadas de control de versiones, no considerar algunos aspectos del ámbito o contexto de producción final o simplemente como consecuencia del rediseño de la aplicación. El objetivo de estas pruebas es eliminar el efecto onda, es decir, comprobar que los cambios sobre un componente de un sistema no introducen un comportamiento no deseado o errores adicionales en componentes no modificados.

Por tanto, en la mayor parte de las situaciones se considera una buena práctica que, cuando se localiza y corrige un bug, grabar una prueba que lo exponga y volver a probar regularmente después de los cambios que experimente el programa. Existen herramientas de software que permiten detectar este tipo de errores de forma parcial o totalmente automatizada. Lo más habitual es que este tipo de pruebas se ejecute en cada uno de los pasos del ciclo de vida del desarrollo de software.

Como tipos de regresión, se pueden distinguir los siguientes:

- Local. Los cambios introducen nuevos errores.
- Desenmascarada. Los cambios revelan errores propios.
- Remota. Los cambios vinculan algunas partes del programa (módulo) e introducen errores en ella.

En una clasificación temporal se tienen en cuenta dos tipos de características:

- Nueva características. Los cambios realizados con respecto a nuevas funcionalidades en la versión introducen errores en otras novedades en la misma versión del software.
- Características preexistentes. Los cambios realizados con respecto a nuevas funcionalidades introducen errores en la funcionalidad existente de previas versiones.

Para mitigar los riesgos en las modificaciones realizadas en los módulos del programa ya desarrollados existen unas operaciones que se enumeran a continuación:

- Repetición completa y habitual de la batería de pruebas, manual o automatizada.
- Repetición parcial basada en trazabilidad y análisis de los posibles riesgos.
- Pruebas de cliente o usuario.
 - Beta. Distribución a clientes potenciales y actuales de las versiones Beta.
 - Pilot. Distribución a un subconjunto bien definido y localizado.

- Paralela. Simultaneando uno de ambos sistemas. Usar release candidates (software candidato a ser definitivo). Probar nuevas funciones a menudo cubre las funciones existentes. Cuantas más nuevas características haya en un release, habrá mayor nivel de pruebas de regresión “accidental”.
- Pruebas de emergencia. Estos parches se publican inmediatamente y serán incluidos en versiones de mantenimiento futuras.

Las pruebas de regresión pueden usarse no solo para probar la corrección de un programa, sino a menudo para rastrear la calidad de su salida. Por ejemplo, en el diseño de un compilador, las pruebas de regresión deben rastrear el tamaño del código, tiempo de simulación y tiempo de compilación de las suites de prueba.

8.7. PRUEBAS DE CAPACIDAD Y RENDIMIENTO. PRUEBAS DE USO DE RECURSOS

Las pruebas de rendimiento sirven para diferentes propósitos. Pueden demostrar que el sistema cumple los criterios de rendimiento, comparar dos sistemas para encontrar cuál funciona mejor o medir qué partes del sistema o de carga de trabajo provocan que el conjunto rinda mal. Para su diagnóstico, los ingenieros de software usan herramientas que miden qué partes de un dispositivo o software contribuyen más a un mal rendimiento o para establecer umbrales que mantengan un tiempo de respuesta aceptable.

Es fundamental que los esfuerzos de estas pruebas comiencen en el inicio del proyecto de desarrollo y se amplíe durante su construcción con la finalidad de alcanzar un buen nivel de rendimiento si se construye un nuevo sistema. Cuanto más se tarde en detectar un defecto de rendimiento, mayor puede llegar a ser el coste de la solución. Esto, que se aplica en las pruebas funcionales, también tiene una gran importancia en las de rendimiento debido a que su ámbito de aplicación es de principio a fin.

Los nuevos conceptos en la implementación de una arquitectura de software añaden complejidad adicional a las pruebas de rendimiento. Los servicios o recursos de una empresa (los cuales comparten infraestructura o plataforma) requieren pruebas de rendimiento coordinadas para reproducir realmente el estado del entorno de producción. Debido a la complejidad, coste y tiempo necesario en torno a esta actividad, algunas organizaciones emplean herramientas que pueden mostrar y crear condiciones de ruido en los entornos de pruebas de rendimiento para comprender la capacidad y necesidades de recursos y verificar/validar los niveles de calidad.

Dependiendo del objetivo perseguido se pueden diseñar distintos tipos de pruebas:

- Pruebas de carga. Intentarán validar que se alcanzan los objetivos de prestaciones a los que se verá sometido el sistema en un entorno de producción. Ejemplos:
 - Un sistema web debe soportar 3.500 reservas de viajes por hora en un tiempo de respuesta no superior a 6 segundos por página.
 - Se deben alcanzar 25 llamadas al servicio de localización geográfica con un tiempo de respuesta máximo de 5 segundos.
- Pruebas de capacidad. Su objetivo es encontrar los límites de funcionamiento del sistema y detectar el cuello de botella o elemento limitante para poder actuar en caso de ampliación del servicio. Por ejemplo, el consumo de CPU en los servidores de datos alcanza el 100% con un nivel de servicio de 3.950 operaciones por hora.
- Pruebas de estrés. Someten al sistema a una carga por encima de los límites requeridos de funcionamiento. Ejemplos pueden ser una puesta a la venta de un producto estrella en un canal de venta o visitas masivas a una web de noticias ante un evento relevante.
- Pruebas de estabilidad. Comprueban que no existe degradación del servicio por un uso prolongado del sistema. Un ejemplo sería: el sistema debe funcionar sin incidencia ni degradación del sistema durante 24 horas.

- Pruebas de aislamiento. Provocan concurrencia sobre componentes aislados del sistema para tratar de detectar posibles errores en ellos.
- Pruebas de regresión de rendimiento. Su objetivo es comprobar si se mantienen los niveles de rendimiento tras un cambio en el sistema, comparando el nivel de rendimiento (tiempo de respuesta, operaciones/hora, etc.) con el que ofrecía anteriormente.

Las pruebas de uso de recursos se pueden englobar entre las de capacidad y rendimiento, ya que se encargan de determinar qué elementos son críticos en el sistema, habitualmente la memoria RAM, espacio en disco y uso de procesador.

8.8. PRUEBAS DE USUARIO

En este apartado se encuentran las pruebas de usabilidad y accesibilidad. Las primeras ya se trataron en la unidad correspondiente. En cuanto a las de accesibilidad, se consideran un subgrupo de las de usabilidad en las que los usuarios que se tienen en cuenta tienen algún tipo de diversidad funcional que afecta a su manera de utilizarlo. El objetivo final, tanto con respecto a usabilidad como a accesibilidad, es descubrir la facilidad con la que se puede utilizar un software y aprovechar esta información para futuros diseños e implementaciones.

La evaluación de la accesibilidad está más formalizada, en general, que las pruebas de usabilidad. Las leyes y la opinión pública desaprueban totalmente la discriminación de las personas con diversidad funcional, por lo que los gobiernos y otras organizaciones intentan cumplir con los estándares de accesibilidad.

8.9. PRUEBAS DE ACEPTACIÓN

El objetivo de las pruebas de aceptación es validar que un sistema cumple con el funcionamiento esperado y permitir al usuario de dicho sistema que determine su aceptación desde el punto de vista de su funcionalidad y rendimiento.

Las pruebas de aceptación se definen por el usuario del sistema y preparadas por el equipo de desarrollo, aunque la ejecución y aprobación final corresponden al usuario.

8.9.1. DESARROLLO DEL PLAN DE PRUEBAS

La validación del sistema se consigue mediante la realización de pruebas de caja negra, que demuestran la conformidad con los requisitos y que se recogen en el plan de pruebas, el cual define las verificaciones a realizar y los casos de prueba asociados. Dicho plan se diseña para asegurar que se satisfacen todos los requisitos funcionales especificados por el usuario teniendo en cuenta también los requisitos no funcionales relacionados con el rendimiento, seguridad del acceso al sistema, a los datos y procesos, así como a los distintos recursos del sistema.

8.10. VERSIONES ALFA Y BETA.

Cuando se construye software a medida para un cliente, se llevan a cabo una serie de pruebas de aceptación para permitir que el cliente valide los requisitos, tal como se vio en el apartado anterior. La mayoría de los desarrolladores de productos de software llevan a cabo un proceso denominado pruebas alfa y beta para descubrir errores que parece que solo el usuario final puede descubrir.

- Pruebas alfa (versiones alfa). Se llevan a cabo por un cliente en el lugar del desarrollo. Se usa el software de forma natural con el desarrollador como observador registrando errores y problemas de uso. Las pruebas alfa se realizan en un entorno controlado.
- Pruebas beta (versiones beta). Se llevan a cabo por los usuarios finales del software en el lugar de trabajo de los clientes. A diferencia de las pruebas alfa, el desarrollador no está presente normalmente. Así, este tipo de pruebas es una aplicación en vivo del software en un entorno que no puede ser controlado por el equipo de desarrollo. El cliente registra todos los problemas encontrados durante las pruebas beta e informa a intervalos regulares al desarrollador.

ÍNDICE DE FIGURAS

1. Prueba unitaria de comprobación de longitud	12
2. NUnit en Nuget	14
3. Proyecto de prueba NUnit	14
4. Ejecución de pruebas	15
5. Ejecución de pruebas de cadena.....	16
6. Solución en blanco	17
7. Añadido de referencia de proyecto.....	19
8. Ejecución de pruebas de BancAvilés	20

BIBLIOGRAFÍA - WEBGRAFÍA

Ferrer Martínez, J. (2015). *Desarrollo de interfaces*. Editorial Ra-Ma

Nicolás Páez (2021). *Unit Testing en C#*. <https://www.udemy.com/course/unit-testing-nicopaez/>