

UD 05. DISEÑO Y REALIZACIÓN DE PRUEBAS

Entornos de Desarrollo

Autor: Cristina Álvarez Villanueva

2023/2024

Licencia



Reconocimiento - NoComercial - CompartirIgual (by-nc-sa): No se permite un uso comercial de la obra original ni de las posibles obras derivadas, la distribución de las cuales se debe hacer con una licencia igual a la que regula la obra original.

NOTA: Este trabajo es derivado del trabajo original realizado por Cristina Álvarez Villanueva.

Nomenclatura

A lo largo de este tema se utilizarán distintos símbolos para distinguir elementos importantes dentro del contenido. Estos símbolos son:



Importante



Atención



Interesante

ÍNDICE DE CONTENIDO

1. DISEÑO Y REALIZACIÓN DE PRUEBAS.....	4
2. CONTROL DE CALIDAD Y PRUEBAS.....	5
3. TERMINOLOGÍA BÁSICA.....	6
4. PRINCIPIOS DE LA COMPROBACIÓN DEL SOFTWARE.....	7
5. TÉCNICAS MANUALES DE COMPROBACIÓN DE SOFTWARE.....	7
6. TÉCNICAS AUTOMÁTICAS DE COMPROBACIÓN DE SOFTWARE.....	9
6.1 Prueba de la Caja Blanca.....	9
6.2 Prueba de la Caja Negra.....	10
6.3 Comparativa Caja Blanca vs Caja Negra.....	11
6.4 Unidades de comprobación.....	12
7. PRUEBAS UNITARIAS.....	12
7.1 Uso de herramientas integradas en los entornos de desarrollo para realizar pruebas unitarias.....	13
7.2 Automatización de pruebas unitarias.....	13
8. PRUEBAS DE INTEGRACIÓN.....	13
9. DEPURACIÓN DE PROGRAMAS.....	14
10. PRÁCTICA EN NETBEANS.....	16
11. BIBLIOGRAFÍA.....	20

UD04. DISEÑO Y REALIZACIÓN DE PRUEBAS

1. DISEÑO Y REALIZACIÓN DE PRUEBAS

Cuando se implementa software, resulta recomendable comprobar que el código que hemos escrito funciona correctamente. Para ello, implementamos pruebas que verifican que nuestro programa genera los resultados que de el esperamos.

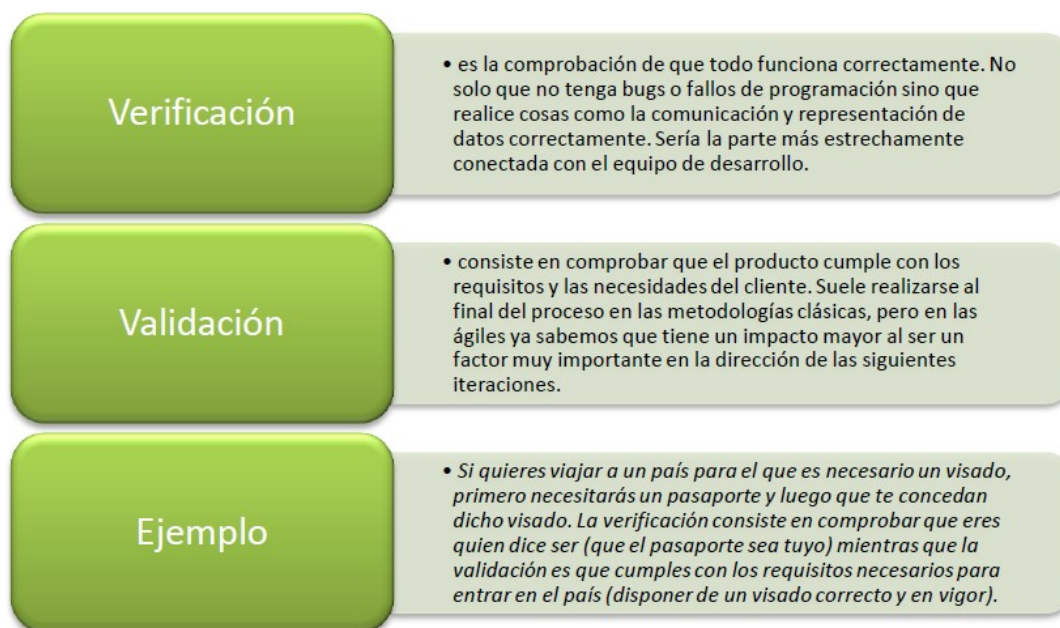
Conforme vamos añadiendo nueva funcionalidad a un programa, creamos nuevas pruebas con las que podemos medir nuestro progreso y comprobar que lo que antes funcionaba sigue funcionando tras haber realizado cambios en el código (test de regresión).

Las pruebas también son de vital importancia cuando **refactorizamos** (aunque no añadamos nueva funcionalidad, estamos modificando la estructura interna de nuestro programa y debemos comprobar que no introducimos errores al refactorizar).

Sin embargo, el concepto de pruebas va más allá de probar el código para eliminar y depurar errores de programación. La fase se conoce como **software testing** y afecta prácticamente a todo el ciclo de vida de una aplicación y a todo el personal involucrado en el desarrollo (incluso a gente “externa”).

Las pruebas del software deben ir dirigidas a **validar** y **verificar** el correcto funcionamiento de una aplicación. Para ello deben procurar que la aplicación:

- Cumpla con los requisitos acordados con el cliente.
- Funcione correctamente.
- Alcance la calidad esperada.
- Satisfaga las necesidades del cliente.



Fuente: adaptado de Cuesta (2014:50)

2. CONTROL DE CALIDAD Y PRUEBAS

Las pruebas suelen ubicarse en los ciclos de vida **después de la fase de desarrollo**, pero realmente debe ser de esas actividades que se aplican durante (casi) todo el proceso.

Un primer objetivo de las pruebas debe ser encontrar errores y corregirlos. Cualquier usuario o desarrollador sabe, por experiencia propia, que los programas tienen errores. Cuanto mejor sea el proceso de ingeniería del software y el proceso de control de calidad y pruebas que se utilice, menos errores tendrá.

Un factor fundamental para comprender todo el resto del tema es saber que las pruebas se deben diseñar antes de empezar a programar nada. Para ello se escriben casos de pruebas unitarios compuestos por varios datos pero principalmente con los factores, datos o pasos a seguir para realizar la prueba y el resultado esperado. Tras realizar la prueba se registra el resultado real. Si una prueba se ejecuta con resultado insatisfactorio, debemos buscar el germen del error en la primera fase que aparezca (por ejemplo análisis o diseño) y empezar a solucionarlo desde ahí.

Muchas veces las empresas tienen gente especializada en probar software. No es necesario que sea siempre el equipo de desarrollo. Aún así lo aconsejable es que el equipo de desarrollo al menos se encargue de detectar (y por supuesto eliminar) la mayor parte de los errores de programación; sin embargo se considera muy útil que las pruebas de ejecución de una aplicación las lleve a cabo gente que no esté directamente involucrada en el desarrollo de la aplicación.

Las metodologías ágiles no son ajenas a este sistema pero al hacer ciclos más cortos y muchas veces con la pretensión de cubrir solo parte de los objetivos las pruebas en cada ciclo estarán más acotadas, sin embargo un cambio puede producir errores o fallos en partes del sistema que considerábamos probadas.

Cuando se trata de una aplicación real, hay que entender que no vamos a corregir absolutamente todos los problemas, pero tomar esta afirmación como un salvoconducto para hacer lo que queramos nos llevará inevitablemente al fracaso profesional. El objetivo final es que el cliente esté satisfecho con el producto que ha adquirido, pero ello implica pruebas a muchos niveles, desde **pruebas unitarias de pequeñas partes de código** (por ejemplo un método o una clase) hasta **pruebas del producto final** desde el punto de vista del usuario, pasando por pruebas de integración de diferentes componentes.

Es por ello que cualquier proyecto de software integra dentro de su ciclo de desarrollo procesos de control de calidad y pruebas. Cada proyecto de software utiliza la combinación de métodos que mejor se adapta a sus necesidades.

Los proyectos que no integran un control de calidad como parte de su ciclo de desarrollo a la larga tienen un coste más alto. Esto es debido a que cuanto más avanzado está el ciclo de desarrollo de un programa, más costoso resulta solucionar sus errores. Por esto, siempre que iniciemos el desarrollo de un nuevo proyecto deberíamos incluir un plan de gestión de calidad.

Para poder llevar a cabo un control de calidad, es imprescindible haber definido los requisitos del sistema de software que vamos a implementar, ya que son necesarios para disponer de una especificación del propósito del software. Los procesos de calidad verifican que el software cumple con los requisitos que definimos originalmente.

A grandes rasgos hay dos categorías de pruebas: **las pruebas estáticas y las dinámicas**. Las primeras se refieren a observar el código o el comportamiento del programa para encontrar fallos o errores; puede hacerse de manera individual o en grupo. No profundizaremos más en este aspecto ya que a partir de ahora nos centraremos en técnicas que nos ayuden con las pruebas dinámicas en las que se diseñan casos de prueba y se ejecutan, etc.

Las pruebas responden a varios objetivos pero típicamente se encuadran en el aseguramiento de la calidad.

3. TERMINOLOGÍA BÁSICA

bug / error	<ul style="list-style-type: none">• Fallo en la codificación o diseño de un sistema que causa que el programa no funcione correctamente o falle.
code coverage / alcance del código	<ul style="list-style-type: none">• Proceso para determinar qué partes de un programa nunca son utilizadas o que nunca son probadas como parte del proceso de pruebas.
stress testing / prueba de estrés	<ul style="list-style-type: none">• Conjunto de pruebas que tienen como objetivo medir el rendimiento de un sistema bajo cargas elevadas de trabajo. Por ejemplo, si una determinada aplicación web es capaz de satisfacer con unos estándares de servicio aceptables un número concreto de usuarios simultáneos.
regression testing / prueba de regresión	<ul style="list-style-type: none">• Conjunto de pruebas que tienen como objetivo comprobar que la funcionalidad de la aplicación no ha sido dañada por cambios recientes que hemos realizado. Por ejemplo, si hemos añadido una nueva funcionalidad a un sistema o corregido un error, comprobar que estas modificaciones no han dañado al resto de funcionalidad existente del sistema.
usability testing / prueba de usabilidad	<ul style="list-style-type: none">• Conjunto de pruebas que tienen como objetivo medir la usabilidad de un sistema por parte de sus usuarios. En general, se centra en determinar si los usuarios de un sistema son capaces de conseguir sus objetivos con el sistema que es objeto de la prueba.
tester / testeador	<ul style="list-style-type: none">• Persona encargada de realizar un proceso de pruebas, bien manuales o automáticas, de un sistema y reportar sus posibles errores.

4. PRINCIPIOS DE LA COMPROBACIÓN DEL SOFTWARE

Cualquiera que sea la técnica o conjunto de técnicas que utilicemos para asegurar la calidad de nuestro software, existen un conjunto de principios que debemos tener siempre presentes. A continuación enumeramos los principales:

- **Es imperativo disponer de unos requisitos que detallen el sistema:**
 - Los procesos de calidad se basan en verificar que el software cumple con los requisitos del sistema. Sin unos requisitos que describan el sistema de forma clara y detallada, es imposible crear un proceso de calidad con unas mínimas garantías.
- **Los procesos de calidad deben ser integrados en las primeras fases del proyecto:**
 - Los procesos de control de calidad del sistema deben ser parte integral del mismo desde sus inicios. Realizar los procesos de pruebas cuando el proyecto se encuentra en una fase avanzada de su desarrollo o cerca de su fin es una mala práctica en ingeniería del software. Por ejemplo, en el ciclo final de desarrollo es muy costoso corregir errores de diseño. Cuanto antes detectemos un error en el ciclo, más económico será corregirlo.
- **Quien desarrolle un sistema no debe ser quien prueba su funcionalidad:**
 - La persona o grupo de personas que realizan el desarrollo de un sistema no deben ser en ningún caso las mismas que son responsables de realizar el control de pruebas. De la misma manera que sucede en otras disciplinas, como por ejemplo la escritura, donde los escritores no corrigen sus propios textos. Es importante recordar que a menudo se producen errores en la interpretación de la especificación del sistema y una persona que no ha estado involucrada con el desarrollo del sistema puede más fácilmente evaluar si su interpretación ha sido o no correcta.

Es importante tomar en consideración todos estos principios básicos porque el incumplimiento de alguno de ellos se traduce en la imposibilidad de poder garantizar la corrección de los sistemas de control de calidad que aplicamos.

5. TÉCNICAS MANUALES DE COMPROBACIÓN DE SOFTWARE

Las técnicas manuales de comprobación de software son un conjunto de métodos ampliamente usados en los procesos de control de pruebas.

Se utilizan **procedimientos de pruebas**, que son pequeñas guías de acciones que un testeador debe efectuar y los resultados que debe obtener si el sistema está funcionando correctamente. Es un método que se aplicará para realizar las pruebas. Debe incluir aspectos como la preparación necesaria (estado de los diferentes componentes, entradas de datos, etc) y salida o resultados esperados.

Suele incluir una secuencia de pasos a realizar. Aunque no existe un estándar podríamos considerar los siguientes básicos:

1. Estrategia de pruebas: definimos el objetivo.
2. Plan de pruebas.
3. Diseño de las pruebas: especificar los distintos casos de prueba.
4. Ejecución de las pruebas: caso por caso.
5. Registro de los resultados.
6. Repetir hasta eliminar los errores (desde el punto 3 o 4).
7. Cierre de las pruebas cuando se cumple con los requisitos.

Veamos otro ejemplo: guión de pruebas del proyecto OpenOffice.org que ilustra su uso (fuente: <http://www.openoffice.org/qa/>)

1. Área de la prueba: Solaris - Linux - Windows
2. Objetivo de la prueba: Verificar que OpenOffice.org abre un fichero .jpg correctamente
3. Requerimientos: Un fichero .jpg es necesario para poder efectuar esta prueba
4. Descripción de las acciones:
 1. Desde la barra de menú, seleccione File - Open.
 2. La caja de diálogo de abertura de ficheros debe mostrarse.
 3. Introduzca el nombre del fichero .jpg y seleccione el botón Open.
 4. Cierre el fichero gráfico.
5. Resultado esperado: OpenOffice.org debe mostrar una nueva ventana mostrando el fichero gráfico.

Como vemos, se trata de una **descripción de acciones** que el testeador debe seguir, junto con el resultado esperado para dichas acciones.

Es habitual en muchos proyectos que antes de liberar una nueva versión del proyecto deba superarse con satisfacción un conjunto de estos guiones de pruebas.

La mayoría de guiones de pruebas tienen como objetivo asegurar que la funcionalidad más común del programa no se haya visto afectada por las mejoras introducidas desde la última versión y que un usuario medio no encuentre ningún error grave.

Muchas veces se intenta automatizar las pruebas minimizando la interacción del usuario para agilizarlas y poder cubrir un espectro más amplio. Los casos de prueba son un aspecto fundamental del proceso.

6. TÉCNICAS AUTOMÁTICAS DE COMPROBACIÓN DE SOFTWARE

Existen muchos tipos de pruebas que generalmente son complementarios. Históricamente las pruebas han ido evolucionando desde una perspectiva de corrección de “**bugs**” hasta implicarse en la satisfacción del cliente, sin abandonar los otros aspectos.

En su enfoque más básico están dos tipos de pruebas completamente orientados a la corrección de errores (**caja negra**, **caja blanca**), pero existen muchos más tipos de pruebas, según el enfoque o el objetivo a alcanzar.

6.1 Prueba de la Caja Blanca

La Prueba de la caja blanca o **white-box testing**, es un conjunto de técnicas que tienen como objetivo validar la lógica de la aplicación. Las pruebas se centran en **verificar la estructura interna del sistema** sin tener en cuenta los requisitos del mismo.

Uno se centra en el código del componente, por ejemplo un método y se prueba en función a él. Como norma general se intenta que cada instrucción se ejecute al menos una vez y que cada decisión se evalúe al menos una vez a cierto y otra a falso.


También puede usarse en pruebas de integración, y no únicamente a nivel de método. En este caso se deben haber probado los componentes que se incluyan y se usarán como si fueran instrucciones sueltas.

Existen varios métodos en este conjunto de técnicas, los más habituales usan la inspección del código de la aplicación por parte de otros desarrolladores para **encontrar errores en la lógica del programa**.

Así, algunas herramientas usadas:

- Control de flujo: realización de un diagrama de flujo del algoritmo a probar, centrándose en que se recorran todos los posibles caminos al menos una vez, implicando ejecutar todas las instrucciones. Para facilitar el diseño de estas pruebas podemos considerar: cada bifurcación, cada camino, cubrir todas las instrucciones y cubrir todas las decisiones a cierto y a falso.
- Flujo de datos: seguir el flujo de los datos durante todo el proceso. La *ejecución paso a paso* tendrían una gran utilidad aquí.

Existen también aplicaciones propietarias de uso extendido que permiten automatizar todo este tipo de pruebas, como *PureCoverge* de Rational Software.

 **Para saber más...** Una prueba que usa el control de flujo en caja blanca se llama “Prueba del camino básico”. Consiste en emplear una representación del flujo de control (grafo del programa) para darle al diseñador de casos de prueba una medida de la complejidad lógica de un diseño procedimental, y usarla así para definir un conjunto básico de caminos de ejecución.

6.2 Prueba de la Caja Negra

La prueba de la caja negra o **black-box testing** se basa en comprobar la funcionalidad de los componentes de una aplicación. Determinados **datos de entrada** a una aplicación o componente **deben producir unos resultados determinados**.

Este tipo de prueba está dirigida a **comprobar los resultados de un componente** de software, no a validar como internamente ha sido estructurado.

Se llama *black-box* (caja negra) porque el proceso de pruebas asume que se desconoce la estructura interna del sistema, solo se comprueba que los datos de salida producidos por una entrada determinada son correctos.

Imaginemos que tenemos un sistema orientado a objetos donde existe una clase de manipulación de cadenas de texto que permite concatenar cadenas, obtener su longitud, y copiar fragmentos a otras cadenas.

Podemos crear una prueba que se base en realizar varias operaciones con cadenas predeterminadas: concatenación, medir la longitud, fragmentarlas, etc. Sabemos cuál es el resultado correcto de estas operaciones y podemos comprobar la salida de esta rutina.

Cada vez que ejecutamos la prueba, la clase obtiene como entradas estas cadenas conocidas y comprueba que las operaciones realizadas han sido correctas.

Este tipo de **tests** son muy útiles para asegurar que a medida que el software va evolucionando no se rompe la funcionalidad básica del mismo.

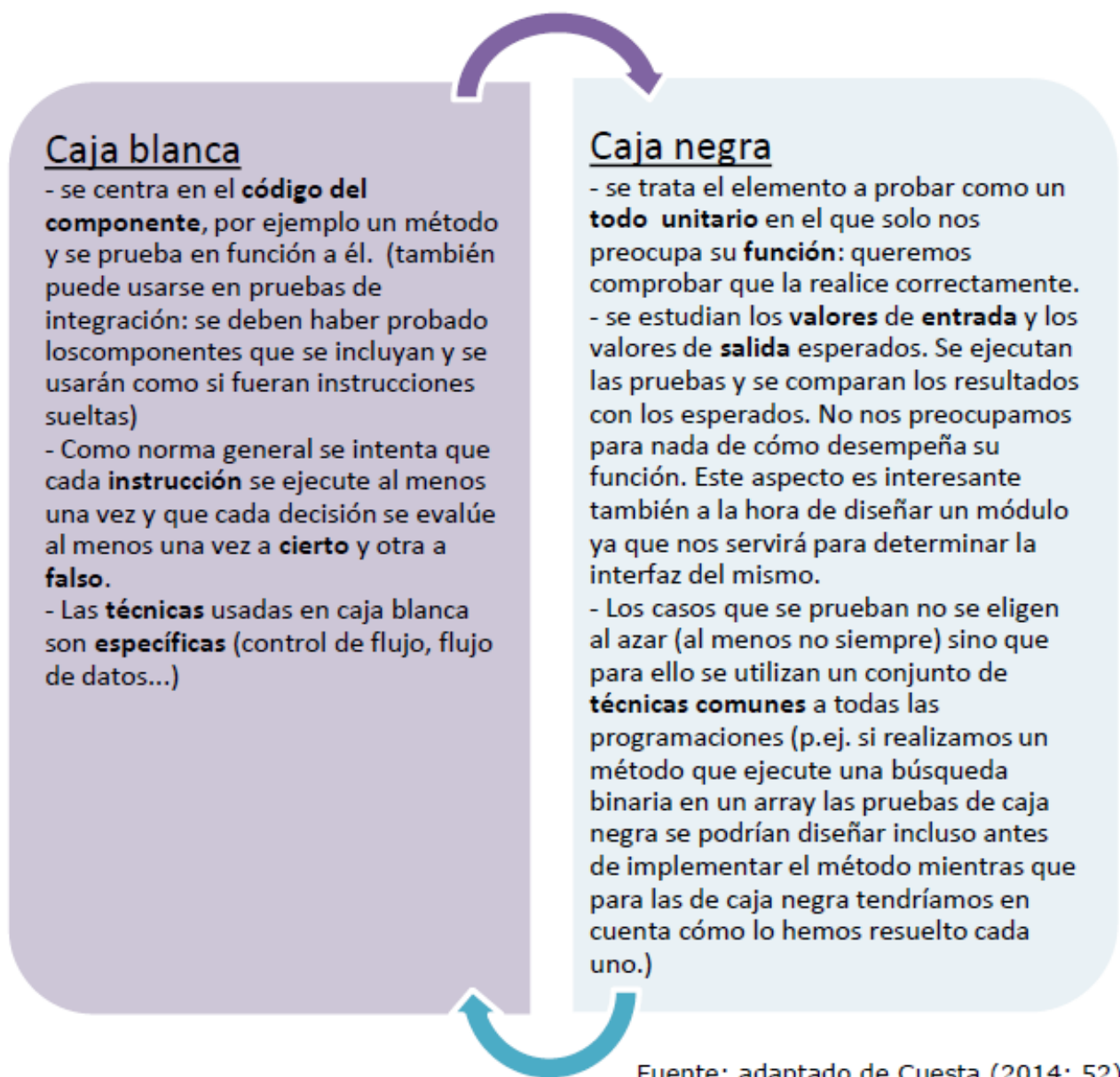
Ejemplos de método son las “particiones” o “clases de equivalencia” y el “análisis de los valores límite”.

^x
x^o **Para saber más...** Las particiones equivalentes son un método de prueba de caja negra que divide los valores de los campos de entrada de un programa en clases de equivalencia, que pueden ser válidas o no (sus valores de entrada).

El análisis de los valores límite se basa en que todos los errores tienden a producirse con más probabilidad en los límites o extremos de los campos de entrada. Se centra en las condiciones de entrada y también en las condiciones de salida, definiendo las clases de equivalencia de salida.

6.3 Comparativa Caja Blanca vs Caja Negra

Veamos en el siguiente gráfico la comparativa entre los dos tipos de prueba:



Fuente: adaptado de Cuesta (2014: 52)

6.4 Unidades de comprobación

Cuando trabajamos en proyectos de una cierta dimensión, necesitamos tener procedimientos que aseguren la correcta funcionalidad de los diferentes componentes del software, y muy especialmente, de los que forman la base de nuestro sistema.

La técnica de las unidades de comprobación, **unit testing**, se basa en la **comprobación sistemática de clases o rutinas de un programa** utilizando unos datos de entrada y comprobando que los resultados generados son los esperados. Hoy en día, las unidades de comprobación son la técnica black-boxing más extendida.

Una unidad de prueba bien diseñada debe cumplir los siguientes requisitos:

- **Debe ejecutarse sin atención del usuario (desatendida):**
 - Una unidad de pruebas debe poder ser ejecutada sin ninguna intervención del usuario: ni en la introducción de los datos ni en la comprobación de los resultados que tiene como objetivo determinar si la prueba se ha ejecutado correctamente.
- **Debe ser universal:**
 - Una unidad de pruebas no puede asumir configuraciones particulares o basar la comprobación de resultados en datos que pueden variar de una configuración a otra. Debe ser posible ejecutar la prueba en cualquier sistema que tenga el software que es objeto de la prueba.
- **Debe ser atómica:**
 - Una unidad de prueba debe ser atómica y tener como objetivo comprobar la funcionalidad concreta de un componente, rutina o clase.

Dos de los entornos de comprobación más populares en entornos de software libre son **JUnit** y **NUnit**. Ambos entornos proporcionan la infraestructura necesaria para poder integrar las unidades de comprobación como parte de nuestro proceso de pruebas. **JUnit** está pensado para aplicaciones desarrolladas en entorno Java y **NUnit** para aplicaciones desarrolladas en entorno .Net.

Es habitual el uso del término **xUnit** para referirse al sistema de comprobación independientemente del lenguaje o entorno que utilizamos.

7. PRUEBAS UNITARIAS

Como ya comentamos las pruebas se realizan a diferentes niveles. El más bajo de ellos son las pruebas unitarias en las que se prueban componentes de forma autónoma.

Una prueba unitaria es un trozo de código que tiene por finalidad ejercitar una funcionalidad específica del programa a probar. Las pruebas unitarias aseguran que el código funciona como se espera. Un buen conjunto de pruebas unitarias automatizadas permiten añadir o modificar código asegurando que el programa sigue cumpliendo sus requisitos.

Una prueba unitaria es una forma de probar el correcto funcionamiento de un módulo de código. Esto sirve para asegurar que cada uno de los módulos funcione correctamente por separado. Luego, con las Pruebas de Integración, se podrá asegurar el correcto funcionamiento del sistema o subsistema en cuestión.

La idea es escribir casos de prueba para cada método en el módulo de forma que cada caso sea independiente del resto.

7.1 Uso de herramientas integradas en los entornos de desarrollo para realizar pruebas unitarias

Existen muchas herramientas que pueden ayudarnos en el diseño y ejecución de pruebas unitarias.

Nosotros veremos **JUnit**. Es un **entorno basado en Java** que utiliza anotaciones para identificar métodos de prueba.

Asume que todos los métodos de prueba se pueden ejecutar en un orden arbitrario, así que unos tests no deberían depender de otros. Se utiliza alguno de los métodos de **JUnit** para comprobar si el resultado de la ejecución coincide con el resultado esperado.

En las prácticas de este tema se detalla cómo trabajar con esta herramienta.

 **Para saber más...** Aquí tienes un listado de herramientas integradas en los IDEs para pruebas unitarias de Java: https://en.wikipedia.org/wiki/List_of_unit_testing_frameworks#Java

7.2 Automatización de pruebas unitarias

Las pruebas unitarias muchas veces tienden a automatizarse aunque no siempre sea el único enfoque.

 **Para saber más...** Échale un vistazo a la automatización del código aquí explicada: https://en.wikipedia.org/wiki/Test_automation

8. PRUEBAS DE INTEGRACIÓN

Después de probar los componentes por separado hay que realizar unas pruebas conjuntas. Éstas suelen realizarse en distintos niveles o pasos especialmente en programas complejos y de gran tamaño. Por ello no está muy claro dónde acaban unas pruebas unitarias y dónde empiezan las de integración (método, clase, paquete, etc).

9. DEPURACIÓN DE PROGRAMAS

Cuando encontramos un fallo debemos localizar el error para solucionarlo. **La depuración es el conjunto de técnicas** que nos ayudan a ello:

- Herramientas de depuración integradas en los entornos de desarrollo
- Puntos de ruptura y seguimiento en tiempo de ejecución
- Examinadores de variables

El proceso de depuración comienza con la ejecución de un caso de prueba. Se evalúan los resultados de dicha ejecución y se comprueba si hay una falta de correspondencia entre los resultados obtenidos y los esperados. El proceso de depuración siempre tiene uno de estos resultados:

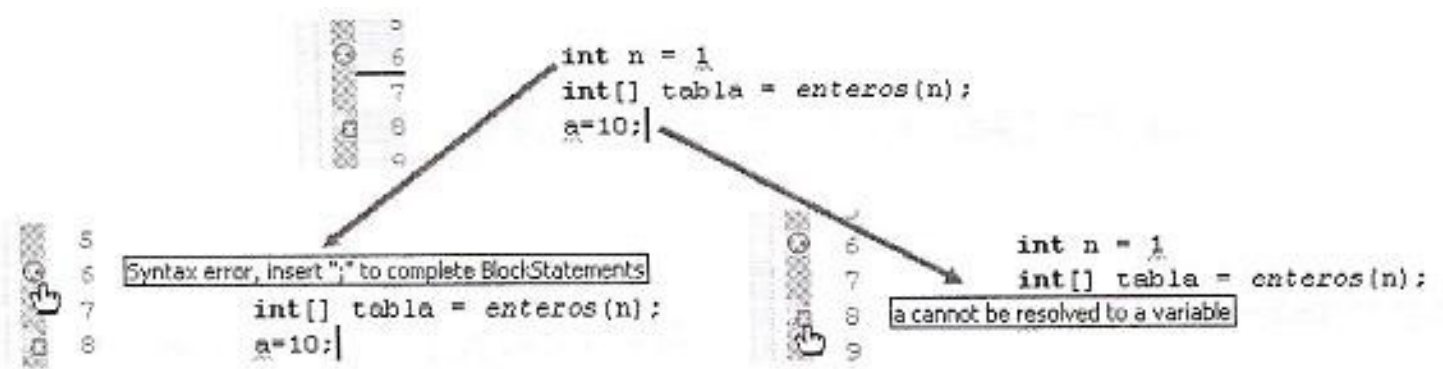
- Si se encuentra la causa del error: se corrige y se elimina
- Si no se encuentra la causa del error: se deben diseñar casos de prueba para localizar el error y repetir el proceso (pruebas de regresión, repetición selectiva de pruebas para detectar fallos introducidos durante la modificación...)



Proceso de depuración (Ramos, 2014:122)

Al escribir programas podemos cometer dos tipos de fallos:

- **Errores de compilación:** fáciles de corregir gracias al IDE (ej. falta un “;”)
- **Errores lógicos o “bugs”:** difíciles de detectar (se compila con éxito pero la ejecución no devuelve lo deseado). Se utilizan los **debuggers** para solucionarlos.



Errores devueltos por el IDE al escribir el código (Ramos, 2014: 22)

10. PRÁCTICA EN JUNIT

En la práctica anterior vimos **JUnit con Eclipse**. Aquí se muestra su funcionamiento con **NetBeans** (es casi idéntico) con otro ejemplo distinto, de manera que sirva de repaso.

Usaremos **APACHE NETBEANS 12**. Si tienes una versión inferior a la 8, puede que algún comando esté cambiado de sitio. Busca en Google la solución o consulta en los foros y te ayudaremos.

10.1. CREACIÓN DE UNA CLASE DE PRUEBA

Desmarcar la opción de crear la clase Main.

Crea un proyecto nuevo en NetBeans por ejemplo (**CalculadoraNB**). Vamos a hacer un par de clases, una clase **Suma.java** y otra clase **Resta.java**. Ambas tienen dos métodos, la primera **getSuma()** e **incrementa()**, la segunda **getDiferencia()** y **decrementa()**.

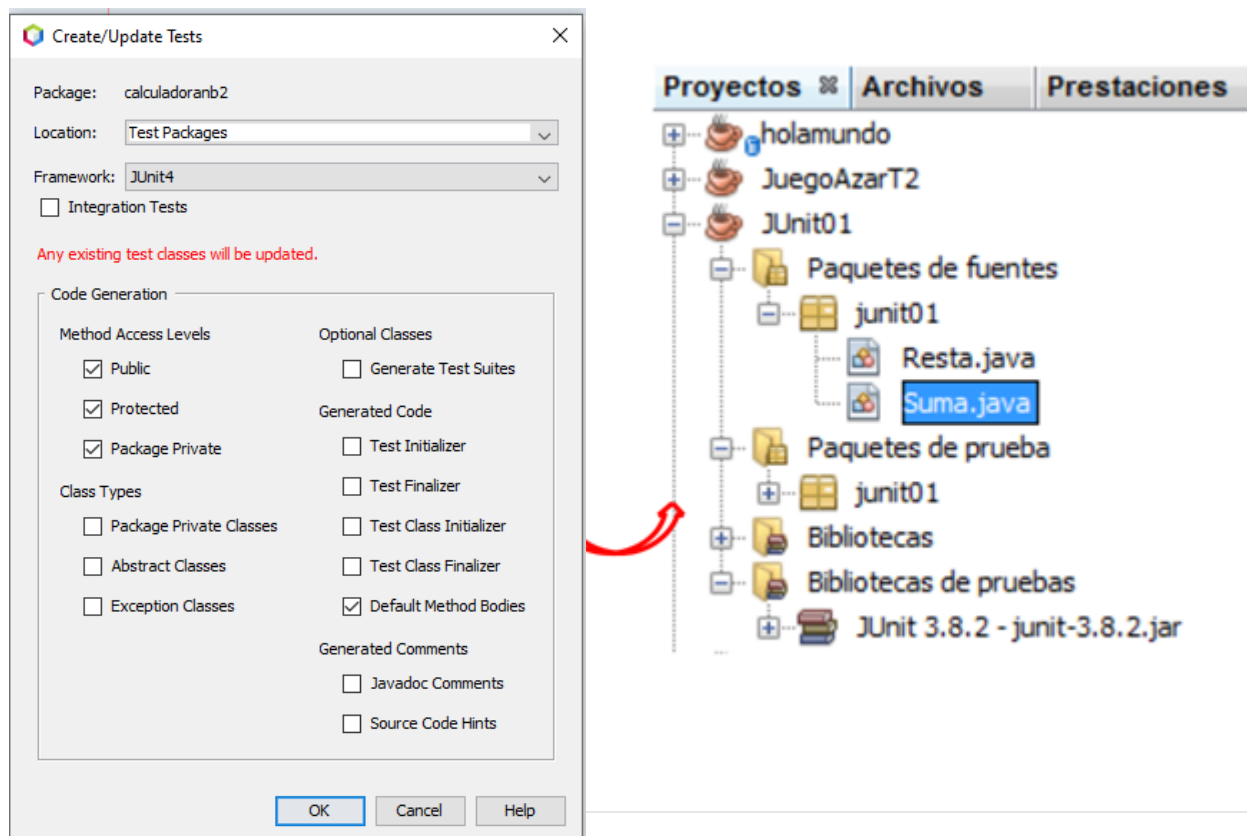
Los códigos son los siguientes:

<pre>package calculadoranb; public class Suma { public double getSuma(double a, double b) { return a + b; } public double incrementa(double a) { return a + 1; } }</pre>	<pre>package calculadoranb; public class Resta { public double getResta(double a, double b) { return a - b; } public double decrementa(double a) { return a - 1; } }</pre>
--	--

Vamos a usar JUnit para hacer los test de estas clases y de sus métodos. Para nuestros programas de prueba podemos hacer una o más clases de prueba, pero lo normal es hacer una clase de prueba por cada clase a probar o bien, una clase de prueba por cada conjunto de pruebas que esté relacionado de alguna manera.

Nosotros tenemos dos clases; **Suma** y **Resta**, así que hacer dos clases de prueba **SumaTest** y **RestaTest**.

Comenzamos por el método Suma(). Hay varias maneras de abrir JUnit en Netbeans: con el **menú contextual > Tools > Create/Update Tests** sobre la clase a probar o **Menú Herramientas > Crear Actualizar Test**. Aparece un menú contextual, donde nos da a elegir el tipo de pruebas a hacer (lo dejamos de momento por defecto y le damos a Aceptar).



Nos crea automáticamente **SumaTest.java**. En el Tab Proyectos debajo del nombre **CalculadoraNB**, nos aparecerán dos cosas nuevas: en **Paquetes de pruebas** nos saldrá la clase creada **SumaTest.java** que acabamos de crear, y en la librería **de pruebas** nos saldrá la **librería de Junit** y su versión **4**. Ten en cuenta que **Apache NetBeans 12 no es compatible con JUnit 5**.

10.2. PERSONALIZAR CLASES TEST

Realizar lo mismo con **Resta**, ahora tenemos **dos clases; Suma y Resta**, y **dos de prueba SumaTest y RestaTest**.

Ahora tenemos que hacer los métodos de test. Cada método probará alguna cosa de la clase. En el caso de **SumaTest**, vamos a hacer dos métodos de test, de forma que cada uno pruebe uno de los métodos de la clase *Suma*, que son **getSuma e Incrementa**.

Por ejemplo, para probar el método **getSuma()** de la clase *Suma*, vamos a hacer un método de prueba **testGetSuma()**. Este método solo tiene que instanciar la clase *Suma*, llamar al método **getSuma()** con algunos parámetros y comprobar que el resultado es el esperado.

Modificaremos los métodos autogenerados por **NetBeans**:

```
@Test
public void testGetSuma() {
    System.out.println(" getSuma ");
    double a = 1.0;
    double b = 1.0;
    Suma instance = new Suma();
    double expectedResult = 2.0;
    double result = instance.getSuma(a, b);
    assertEquals(expResult, result, 0);
}
```

```
@Test
public void testIncrementa() {
    System.out.println("incrementa");
    double a = 1.0;
    Suma instance = new Suma();
    double expectedResult = 2.0;
    double result = instance.incrementa(a);
    assertEquals(expResult, result, 0);
}
```

```
@Test
public void testGetResta() {
    System.out.println(" getResta ");
    double a = 3.0;
    double b = 2.0;
    Resta instance = new Resta();
    double expectedResult = 1.0;
    double result = instance.getResta(a, b);
    assertEquals(expResult, result, 0);
}
```

```
@Test
public void testDecrementa() {
    System.out.println("decrementa");
    double a = 3.0;
    Resta instance = new Resta();
    double expectedResult = 2.0;
    double result = instance.decrementa(a);
    assertEquals(expResult, result, 0);
}
```

Vamos a sumar 1 más 1, que ya sabemos que devuelve 2. ¿Cómo comprobamos ahora que el resultado es el esperado?.

Uno de los métodos más usados es **assertEquals(expResult, result, sensibilidad)** , que en general admite dos parámetros: el primero es el valor esperado y el segundo el valor que hemos obtenido. **El tercer parámetro es la sensibilidad de la comparación y solo te lo pedirá si estás usando Apache NetBeans (versión > 8). Dejalo siempre a cero.**

El método **testGetSuma** está indicando que tome como parámetros a y b los valores 1.0 y 1.0 el método **getSuma**, y debe dar como resultado 2.0.

El método **assertEquals(expResult, result, sensibilidad)** indica el valor esperado que es 2.0 y el valor devuelto por la función **getSuma** que deberá calcularse.

10.3. TIPOS DE ASSERT

Existen varios tipos de assert (afirmaciones), se resumen en:

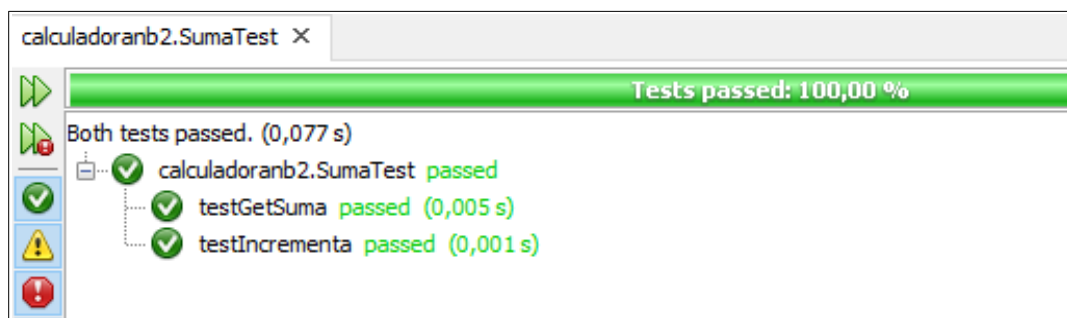
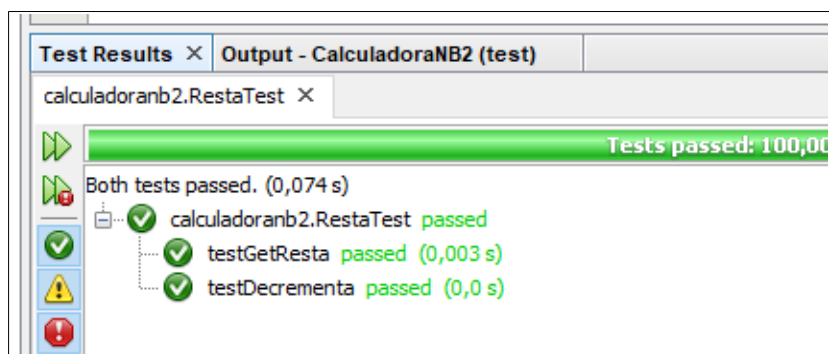
<code>static void assertEquals(int expected, int actual);</code>	Afirma que son iguales dos enteros.
<code>static void assertEquals(String message , int expected, int actual);</code>	Afirma que son iguales dos enteros con mensaje si no falla.
<code>static void assertEquals(java.lang.String message, double expected, double actual, double delta)</code>	Afirma si son iguales dos doubles. Siendo: <i>expected</i> es el primer tipo Double que se va a comparar. Es el tipo Double que la prueba unitaria espera. <i>actual</i> es el segundo tipo Double que se va a comparar. Es el tipo Double producido por la prueba unitaria. <i>delta</i> es la precisión necesaria. Se producirá un error en la aserción sólo si <i>expected</i> es diferente de <i>actual</i> en más de <i>delta</i> .
<code>static void assertNotNull(Object object)</code>	Afirma que un objeto no es null
<code>static void fail (String message)</code>	Falla un test con un mensaje dado.

Hay más tipos que se pueden consultar en el API:

<https://junit.org/junit5/docs/5.3.1/api/org/junit/platform/runner/JUnitPlatform.html>

10.4. EJECUCIÓN DE LAS PRUEBAS

Si ejecutamos cada una de las clases de prueba obtendremos los resultados similares a los que vimos con Eclipse:



11. BIBLIOGRAFÍA Y ENLACES

Álvarez, C. (2014): "Entornos de desarrollo", CEEDCV