Pruebas Del Software

TEMA 2.1

Pruebas del Software.

Las **pruebas del software** se realizan con el objetivo principal de detectar los errores cometidos durante las fases de desarrollo, que provocarían defectos del software en alguno de los componentes desarrollados, que conducirían a fallos de los sistemas que hacen uso de dicho software. Las pruebas se realizan recreando la operativa de ejecución del software mediante **casos de prueba** con datos de ejemplo convenientemente escogidos según técnicas diversas, teniendo en cuenta como se ha desarrollado el código. Un buen caso de prueba es aquel que tiene una probabilidad alta de descubrir un nuevo defecto del software. El éxito de una prueba se produce cuando se descubre un defecto del software.

Las pruebas del software pueden usarse para demostrar la existencia de errores, nunca su ausencia. [Dijkstra]

La prueba (testing) del software es el proceso de ejecutar un programa con la intención de encontrar errores. [Glenford J. Myers]

La prueba (testing) es el proceso de operar un sistema o un componente bajo unas condiciones específicas, observando o registrando los resultados obtenidos con el fin de realizar a posteriori una evaluación de ciertos aspectos clave. [IEEE]

Un caso de prueba (test case) es una situación, contexto o escenario bajo el que se comprueba una funcionalidad de un programa para ver si se comporta de la forma en qué se espera.

Un caso de prueba (test case) es un conjunto de entradas, condiciones de ejecución y resultados esperados que son desarrollados con el fin de cumplir un determinado objetivo. [IEEE90]

Las pruebas se realizan utilizando casos de prueba con la intención de que afloren la mayor cantidad posible de disfunciones presentes en el producto software, intentando contemplar el escenario más exigente posible de ejecución que se pueda, considerando la mayor cantidad posible de situaciones representativas de toda la casuística que pueda presentarse durante la ejecución, de la forma más exhaustiva posible, tratando de alcanzar o acercarse lo máximo posible de posibilidades de ejecución (**prueba exhaustiva o prueba total**), cosa que es impracticable, no siendo posible contemplar en su totalidad, y podría llegar a ser excesivamente laborioso, dada la enorme cantidad de combinaciones posibles de premisas que se pueden dar, incluso en programas sencillos, teniendo en cuenta además que puede existir código inalcanzable, que puede ser sobrante sin más, o motivado por algún defecto en la lógica del algoritmo que impide alcanzar alguna funcionalidad necesaria.

Las pruebas de los programas sencillos son relativamente fáciles de realizar sin necesidad de demasiadas comprobaciones, pero las aplicaciones complejas requieren un procedimiento sistemático para comprobar su correcto funcionamiento en todas las posibles condiciones que se puedan dar..

Fases de Pruebas.

Fase de Pruebas

La **fase de pruebas** se lleva a cabo, antes de pasar a la fase de producción, verificando (durante la construcción del software) que se siguen los requisitos de las especificaciones funcionales y no funcionales, y validando (una vez confeccionado el software) que se cubren las expectativas del usuario.

La fase de pruebas supone una parte importante del esfuerzo de desarrollo del software, tanto mayor cuanto más alto es el nivel de críticidad de la aplicación (instrumental médico, centrales nucleares, control de tráfico aéreo, ...).

La **documentación** de las pruebas de software está estandarizada según normas (IEEE 829-2008 reemplazada por ISO/IEC/IEEE 29119-3:2013), que definen el tipo y formato de documentos y sus diferentes apartados y secciones (enfoque, alcance, recursos, calendario, responsables, riesgos)..

El **Desarrollo guiado o dirigido por pruebas** (TDD) consiste en plantear las pruebas antes de implementar el código a partir de los requisitos del producto software, de modo que la programación se debe centrar única y exclusivamente en pasar las pruebas.

Algunas de las recomendaciones a tener en cuenta en la fase de pruebas del software son:

- Dedicar los recursos necesarios a los planes de prueba, partiendo de la premisa de que siempre hay defectos en los programas.
- Planificar y diseñar las pruebas para poder detectar el máximo número y variedad de defectos con el mínimo consumo de tiempo y esfuerzo.
- Considerar las pruebas como una tarea necesaria, tanto o más creativa que el desarrollo de software, y no como una tarea destructiva y rutinaria (como se suele considerar).
- Inspeccionar a conciencia el resultado de cada prueba, y así poder descubrir posibles síntomas de defectos.
- Realizar, para cada requisito o funcionalidad, pruebas afirmativas, para evaluar que funciona correctamente, y pruebas negativas, para evaluar lo contrario, incluyendo casos de prueba con datos de entrada válidos o esperados por un lado y no válidos e inesperados por otro, desechando los casos arbitrarios que no responden a un diseño metódico y documentado.
- Prestar atención en las secciones de código en las que se detectan defectos, ya que la experiencia parece indicar que donde hay un defecto hay otros (la probabilidad de descubrir nuevos defectos en una parte del software es proporcional al número de defectos ya descubierto).
- Poner especial énfasis en probar cualquier módulo o sección de código sospechoso o susceptible de arrastrar algún error frente a las que no suelen albergar dudas (como los métodos getters y setters por ejemplo), a no ser que haya alguna razón para pensar que pueda no funcionen correctamente.
- Comprobar además de que el software hace lo que debe hacer, que lo hace sin provocar efectos secundarios adversos en otras partes (es habitual obviarlo).
- Evitar que el programador pruebe sus propios programas, ya que desea (consciente o inconscientemente) demostrar que funcionan sin problemas, además, es normal que las situaciones que olvidó considerar al crear el programa queden de nuevo olvidados al crear los casos de prueba.

Tipos de Pruebas.

Tipos de Pruebas

- Según el alcance (unitarias / de Integración / de Aceptación).
  - **Pruebas Unitarias**: orientadas a módulos independientes o sección concreta de código de las aplicaciones (clases, métodos, …). Son los tests más importantes en el desarrollo de software.
  - **Pruebas de Integración**: enfocadas en la interacción entre módulos a medida que se van desarrollando, en un planteamiento descendente (de los módulos generales a los particulares) o ascendente (de los módulos particulares a los generales).
  - **Pruebas de Aceptación o Validación o Usabilidad**: confirmación del correcto funcionamiento del producto completo, final o prototipo, en un entorno de trabajo real, y con usuarios finales para chequear el funcionamiento, contando con la aprobación de los clientes para la confección de las pruebas y la confirmación de resultados.
- Según el enfoque para el diseño de casos de prueba (de caja blanca / de caja negra).
  - **Pruebas de Caja Blanca o de Caja Transparente:** sobre aspectos estructurales de los programas, enfocadas en la operativa interna de los procedimientos, centradas no en lo que hacen, sino en como lo hacen.
  - **Pruebas de Caja Negra o de Caja Opaca**: sobre aspectos funcionales de los programas, enfocadas en la operativa externa, inducida por la entrada/salida de datos, centradas en lo que hacen y no como lo hacen.

Las **pruebas de regresión** son pruebas que se realizan para encontrar disfunciones causadas por los cambios realizados en los programas, con el fin de comprobar que las nuevas funcionalidades no alteran otras funcionalidades ya probadas y validadas anteriormente con sus resultados esperados. Las pruebas deberían implementarse de manera sistemática incrementalmente con cada nueva funcionalidad que se añada, teniendo de este modo las pruebas actualizadas en cada momento.

Las pruebas de caja blanca y de caja negra se pueden aplicar a cualquier nivel de testeo de software (pruebas unitarias, de integración, de aceptación.

Pruebas de Caja Blanca.

Las **pruebas de caja blanca** se realizan con la pretensión de buscar y analizar todos los posibles caminos de ejecución de código de programa, en base al diseño de casos de prueba, utilizando baterías de datos que sirvan para examinar la ejecución del código de programa, y comprobar si la reacción es la esperada, tratando de garantizar que se revisan todos los caminos de ejecución, sin dejar rincones sin rastrear, según diferentes criterios de cobertura lógica, que cubren distintos porcentajes de código de programa.

Criterios de Cobertura

- **Cobertura de sentencias o segmentos**: establece un número de casos de prueba suficiente para que se ejecute cada sentencia o instrucción al menos una vez, sin que se requiera para ello evaluar todas las vertientes de las estructuras condicionales de decisión, sino solo las que contienen segmentos de código.
- **Cobertura de decisiones o ramas**: refinamiento de la cobertura de segmentos, establece un número de casos de prueba suficiente para que se ejecute cada decisión al menos una vez, recorriendo todas las bifurcaciones y evaluando (verdadero / falso) todas las decisiones
- **Cobertura de condiciones**: refinamiento de la cobertura de decisiones, establece un número de casos de prueba suficiente para que se ejecute cada condición al menos una vez, recorriendo todas las bifurcaciones y evaluando (verdadero / falso) todas las condiciones.
- **Cobertura múltiple o de condiciones/decisiones**: refinamiento de la cobertura de condiciones, establece un número de casos de prueba suficiente para que se ejecute cada combinación distinta de condición al menos una vez, recorriendo todas las bifurcaciones y contemplando todas las combinaciones posibles de evaluación (verdadero / falso) de todas las condiciones dentro de cada decisión.
- **Cobertura de bucles**: establece casos de prueba que abarcan situaciones suficientes para que se ejecuten las iteraciones un número de veces cercano a los límites inferior y superior, para comprobar que se ejecutan el número adecuado de veces, ni una más, ni una menos, ya que un numero distinto de ejecuciones, produce normalmente consecuencias indeseadas, variando el número de casos pruebas necesarios según el tipo de iteración:
  - **Iteración MIENTRAS**: 5 casos de prueba (0, 1, m < n, n-1, n iteraciones). / (n=nº máximo de iteraciones)
  - **Iteración HASTA**: 4 casos de pruebas (1, m < n, n-1, n iteraciones). / (n=nº máximo de iteraciones)
  - **Iteración PARA**: 1 caso de prueba (1 iteración) sería suficiente, al no depender de condiciones.

Consideramos que todo punto de decisión de un programa o simplemente decisión es una expresión condicional completa, es decir, una lista de condiciones conectadas por medio de operadores, donde cada condición atómica parcial es cada uno de los factores de decisión independientes. **<decisión> = <condición> <operador> <condición> <operador> ...**

La clasificación de coberturas abarcan criterios de menor a mayor fortaleza (garantía de los resultados), alcance (porcentaje de código que cubre), rigurosidad (amplitud de valores que se contempla), y coste (esfuerzo de aplicación), siendo la cobertura de sentencias o segmentos el criterio más débil, aunque necesario, pero no suficiente, aumentando progresivamente la fortaleza de cada criterio, y proporcionalmente, el alcance, rigurosidad y coste.

Pruebas de Caja Blanca.

Criterios de Cobertura

En resumen, las pruebas se plantean con diversas intencionalidades de cobertura:

- **Cobertura de sentencias o segmentos**: ejecutar al menos una vez cada sentencia o instrucción.
- **Cobertura de decisiones o ramas**: ejecutar al menos una vez cada decisión para cada valor (verdadero y falso).
- **Cobertura de condiciones**: ejecutar al menos una vez cada condición para cada valor (verdadero y falso).
- **Cobertura múltiple o de condiciones/decisiones**: ejecutar al menos una vez cada condición para cada valor (verdadero y falso), abarcando al menos una vez cada decisión para cada valor (verdadero y falso).
- **Cobertura de bucles**: ejecutar al menos una vez los casos centrales y casos próximos a los límites de las iteraciones.

En la práctica, se suele procurar alcanzar una cobertura de segmentos completa (100%), se recomienda conseguir una buena cobertura de decisiones (cercana al 100%), no siendo estrictamente necesario alcanzar una cobertura de condiciones, aunque sí aconsejable, y necesario en ciertos casos, pudiendo variar la cobertura deseable según diversos factores como el nivel de criticidad y el coste de distribución. Es necesario alcanzar una cobertura máxima (100%), exprimiendo al máximo la fase de pruebas para encontrar prácticamente todos los errores sin pasar nada por alto, si el nivel de criticidad del software es elevado, siendo aceptable una cobertura relativamente alta (entre el 60% y el 80% al menos) para el resto del software. Es conveniente alcanzar una cobertura alta, si el coste de distribución del software es elevado, como en el caso de que los clientes estén muy dispersos geográficamente en un área extensa, pues el coste de un posible defecto detectado tardíamente puede llegar a ser desmesurado.

El resultado de las pruebas puede ser engañoso, ya que, aún sin detectarse ninguna disfunción en el 100% de código probado, podría haber código defectuoso enmascarado en algún caso, en función del criterio de cobertura:

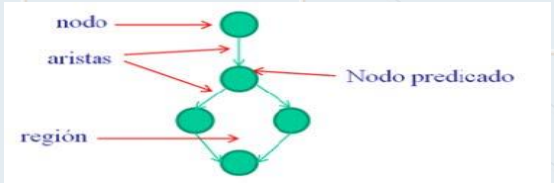
- **Cobertura de sentencias o segmentos**: posible código defectuoso en el caso de las alternativas con ramas sin código, que no se prueban.
- **Cobertura de decisiones o ramas**: posible código defectuoso en alguna condición que no se necesite verificar para probar todas las ramas.
- **Cobertura de condiciones**: posible código defectuoso en alguna de las combinaciones de condiciones que no se necesite verificar para probar todas las ramas.
- **Cobertura múltiple o de condiciones/decisiones**: posible código defectuoso si algún operador es erróneo, aunque se contemplen todas las combinaciones de condiciones.
- **Cobertura de bucles**: posible código defectuoso si dentro del ciclo se altera el valor de la variable de paso, o si hay rupturas de secuencia, que aumentan mucho el riesgo de mal funcionamiento, sin técnicas sistemáticas para abordarlo, siendo deseable algún tipo de cobertura que contemple decisiones y condiciones.

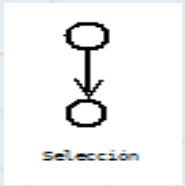
Las pruebas de caja blanca suelen ser de gran utilidad, y casi imprescindibles, siendo deseable alcanzar una buena cobertura de caja blanca, aunque ello no es suficiente para asegurar la ausencia de disfunciones en un producto software, pues sirven para probar que el programa hace bien lo que hace, pero no que haga lo que tiene que hacer (ordenar ascendentemente y no descendentemente por ejemplo), por lo que se requieren otro tipo de pruebas adicionales (pruebas de caja negra).

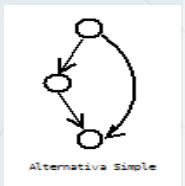
Pruebas de Caja Blanca. Camino Básico.

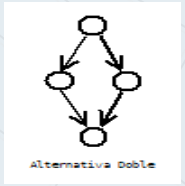
La **técnica del camino básico** (propuesta por el matemático Thomas J. McCabe) es un método consistente en la representación del **grafo de flujo** y el cálculo de la **complejidad ciclomática** de un algoritmo o programa, para determinar los caminos independientes que contiene, que son los que deben cubrir los casos de prueba para asegurar que se prueban todos los fragmentos de código. Los caminos independientes son los que incluyen instrucciones no incluidas en otros caminos, siendo un camino una secuencia de instrucciones de principio a fin de programa.

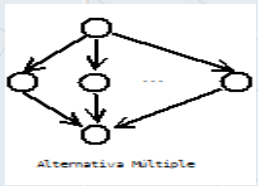
El **Grafo de flujo** de un algoritmo o programa es la representación gráfica de todos los segmentos y flujos de código del algoritmo o programa, compuesto por nodos (porciones de código) y aristas (rutas de flujo).

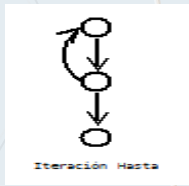












La **complejidad ciclomática** de un algoritmo o programa es un parámetro que da una idea de la complejidad de la lógica del algoritmo o programa, proporcional al nivel de riesgo o probabilidad de código deficiente. La complejidad ciclomática coincide con el número de caminos independientes que contiene el algoritmo o programa, y por tanto, de los casos de prueba que se deben diseñar para realizar las pruebas. Las fórmulas de cálculo son cualquiera de las siguientes:

- **V(G) = número de aristas – número de nodos + 2**
- **V(G) = número de nodos predicado (con decisiones / bifurcaciones) + 1**
- **V(G) = número de regiones delimitadas por aristas (incluida la exterior)**

Complejidad ciclomática	Tipo de programa - evaluación del riesgo
1-10	programa sencillo - riesgo bajo
11-20	programa medio - riesgo moderado
21-50	programa complejo - riesgo alto
>50	programa arriesgado - riesgo muy alto

Los pasos para realizar las pruebas aplicando el método del camino básico, según el criterio de cobertura que se pretenda alcanzar, son los siguientes:

1. Representar el **grafo de flujo** de un algoritmo o programa.
2. Calcular la **complejidad ciclomática** del grafo.
3. Determinar el conjunto de **caminos independientes**.
4. Diseñar **casos de prueba** para ejecutar todos los caminos independientes.
5. Recoger en una **tabla de pruebas** los caminos independientes, los casos de prueba y los resultados esperados.

La tabla de pruebas servirá de punto de partida para aplicar los casos y comprobar si los resultados obtenidos concuerdan con los resultados esperados, y por tanto el éxito de la prueba.

La **mutación** es una técnica consistente en alterar el comportamiento y funcionamiento de los programas con ligeras modificaciones convenientemente elegidas, que provoquen resultados distintos (eliminando sentencias, modificando operadores de decisiones alternativas o iterativas, etc), para verificar la fiabilidad de la estrategia de prueba utilizada comprobando si es capaz de detectar esos cambios.

Pruebas de Caja Blanca. Informe de Pruebas.

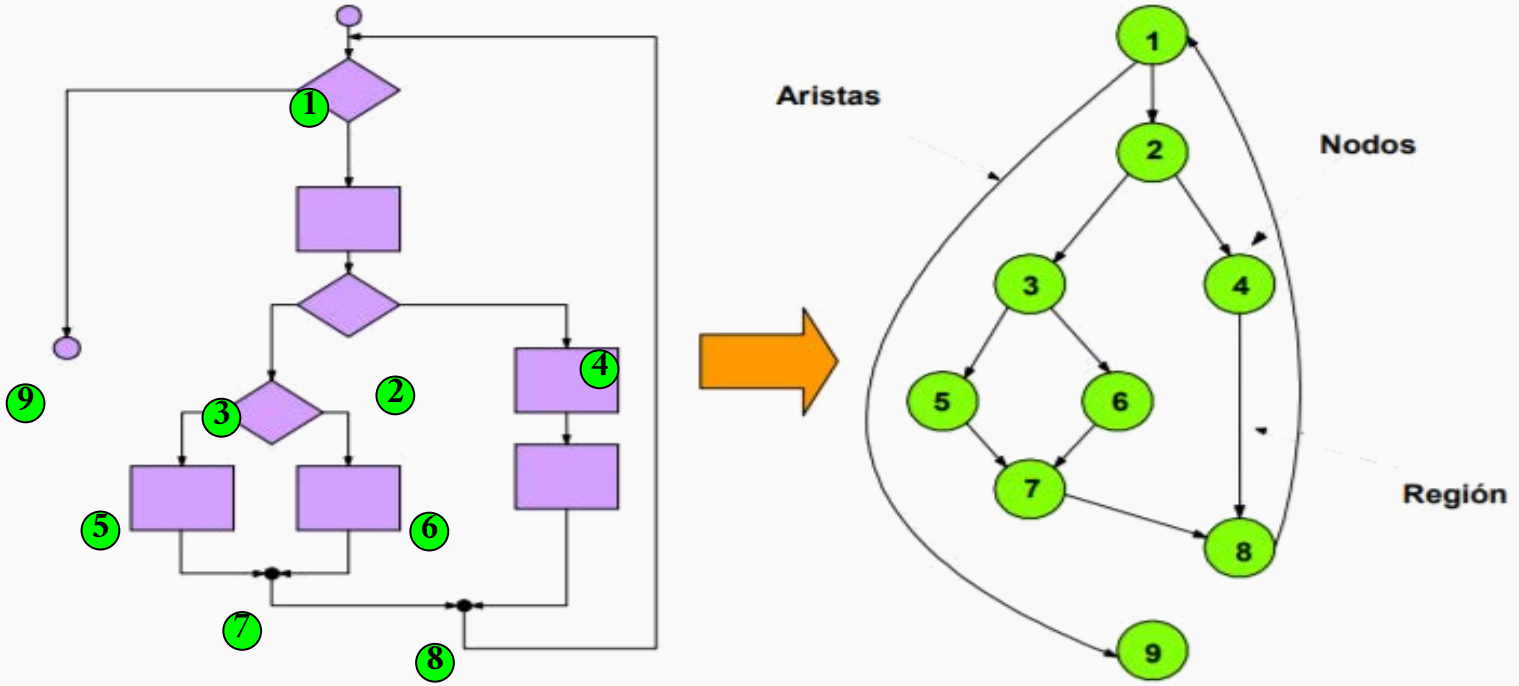
	FORMATO CASOS DE PRUEBA DE SOFTWARE	Código:	130,13,15-10
		Versión:	1
	PROCESO DE GESTION DE TECNOLOGIAS DE LA INFORMACION	Fecha de Aprobación:	31/08/2015
	PROCEDIMIENTO DE DESARROLLO DE NUEVOS SISTEMAS DE INFORMACION Y/O FUNCIONALIDADES EN SISTEMAS EXISTENTES	Pág:	1 de 1

Proyecto	
Desarrollador	
Analista de Pruebas	
Version del Producto	
Fecha de Ejecucion	

Id	Modulo a Probar	Objetivo	Datos de Prueba	Pre-Requisitos	Pasos Para Ejecución	Resultado Esperado	Resultado Obtenido	Estado Prueba	Incidencia Reportada

Pruebas de Caja Blanca.

Grafo de flujo, complejidad ciclomática, y caminos independientes, del algoritmo correspondiente al siguiente diagrama de flujo.



V(G) = número de aristas – número de nodos + 2 = 11 – 9 + 2 = 4
V(G) = número de nodos predicado (número de bifurcaciones) + 1 = 3 + 1 = 4
V(G) = número de regiones delimitadas por aristas (incluida la exterior) = 4

CAMINOS
1-9
1-2-4-8-1-9
1-2-3-5-7-8-1-9
1-2-3-6-7-8-1-9

Pruebas de Caja Blanca. Herramientas de Pruebas.

Las pruebas de caja blanca se pueden realizar, una vez que se han diseñado los casos de pruebas, bien manualmente, resultando un proceso tedioso, ya que requieren seguir paso a paso el flujo de código, controlando las líneas por las que se va pasando y los valores de los datos, o bien de forma automatizada, con ayuda de herramientas que permiten sistematizar el proceso, entre las que están los **depuradores** (**debuggers**) y los **marcos de trabajo** (**frameworks**) específicos.

El **depurador** es una herramienta para automatizar la traza de un programa mediante rastreo controlado de código fuente en tiempo de ejecución, que permite la observación de los estados intermedios de los datos durante la ejecución de casos de prueba que ayuden a descubrir la causa de posibles funcionamientos indeseados, con opciones de gran ayuda para el seguimiento de la lógica de los programas (puntos de ruptura, inspección y modificación de datos, ejecución de línea y de bloque, ...).

La utilización habitual del depurador consiste en ejecutar el código de manera normal hasta una línea determinada, donde se sospecha que puede estar el origen o la evidencia de un desajuste de los valores de los datos, estableciendo un punto de ruptura (breakpoint) en dicha línea, a partir del cual se puede rastrear paso a paso, hacia adelante o hacia atrás. la ejecución del código hasta encontrar la causa de dicho desajuste.

Un **framework** específico para pruebas unitarias es un conjunto de bibliotecas de funciones, integrables en los entornos de desarrollo, que proporciona código preescrito que permite probar módulos de código, generando informes de los resultados de comparar los valores esperados y los valores obtenidos a partir de determinados datos de entrada.

Existen diversidad de frameworks para realizar pruebas unitarias prácticamente en cualquier lenguaje de programación (**PHPUnit** para PHP, **CPPUnit** para C++, **CUnit** para C, **Nunit** para .Net, **PyUnit** para Python), y en concreto para Java (**Junit**, **TestNG**, **Jbehave**, **Serenity**, **seleniuros**, **Unidad JWeb, Calibre**, **Geb**, **Spock**, **Mockito**, **Mock de poder**).

JUNIT es un framework para hacer pruebas unitarias de funcionamiento de los métodos de las clases de aplicaciones java de cada proyecto, que suele estar disponible en los entornos de desarrollo, pudiéndose también añadir las librerías jar de la versión que se desee (JUNIT5 es la versión actual).

La utilización habitual de JUNIT consiste en crear, desde el entorno de desarrollo, para un proyecto determinado, un test para una clase existente. Cuando se crea el test de prueba, JUNIT genera en un paquete aparte (Test Package), el esqueleto de los tests, con diversos métodos de prueba (testXXX) para los métodos (XXX) de la clase a testear. Las pruebas están dirigidas por medio de anotaciones (annotations) y aserciones o afirmaciones (assertions). Las anotaciones son etiquetas (precedidas del carácter @) que identifican el tipo de acción asociada a cada método de prueba (@Test, @Disabled, @BeforeEach, @BeforeAll, @AfterEach, @ AfterAll, …). Las aserciones son métodos estáticos (assertEquals, assertNotEquals, assertTrue, assertFalse, assertNull, assertNotNull, assertSame, assertNotSame, …) encargados de comparar los valores esperados con los valores obtenidos según distintos criterios (valores iguales, distintos, verdadero, falso, nulo, no nulo, …). Las pruebas se realizan ejecutándolas mediante la opción Test del proyecto, mostrándose en la ventana de resultados las pruebas que han sido exitosas y las que no, indicándose en este caso la discrepancia entre el valor esperado y el valor real obtenido. Los métodos de prueba se crean con un método provisional de fallo (fail), que hace fallar el test intencionadamente mientras no se haya terminado de programar, y que debe eliminarse o comentarse cuando ya se ha codificado.

Herramientas de Pruebas. JUNIT.

Resumen de la clase de tests Junit5 que crea el entorno de desarrollo, con diversas anotaciones y aserciones posibles, descripción de algunas de ellas.

```java
package com.jos.mates;
import org.junit.jupiter.api.AfterEach;
import org.junit.jupiter.api.AfterAll;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.BeforeAll;
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.*;

public class CaseTest {

  public ClaseTest () {
  }

  @BeforeAll
  public static void setUpClass () {
  }

  @AfterAll
  public static void tearDownClass () {
  }

  @BeforeEach
  public void setUp () {
  }

  @AfterEach
  public void tearDown () {
  }

  @Test
  public void testMain() {
    System.out.println ("main");
    String[] args = null;
    Mates.main (args);
    }

  @Test
  public void testMetodo () {
    assertEquals (valor_esperado, valor_calculado, mensaje_error);
    }

  @Test
  public void testMetodo () {
    assertNotEquals (valor_esperado, valor_calculado, mensaje_error);
    }

  @Test
  public void testMetodo () {
    assertNull (valor_esperado, valor_calculado, mensaje_error);
    }

  @Test
  public void testMetodo () {
    assertNotNull (valor_esperado, valor_calculado, mensaje_error);
    }

  @Test
  public void testMetodo () {
    assertTrue (valor_esperado, valor_calculado, mensaje_error);
    }

  @Test
  public void testMetodo () {
    assertFalse (valor_esperado, valor_calculado, mensaje_error);
    }
}
```

Anotación	Descripción
@Test	Indica que el método es un test
@DisplayName	Indica un nombre para el *test class* o el *test method*
@Tag	Define etiquetas para filtrar por tests
@BeforeEach	Se aplica a un método para indicar que se ejecute antes de cada método de prueba. (JUnit4 @Before)
@AfterEach	Se aplica a un método para indicar que se ejecuta después de cada método de prueba. (JUnit4 @After)
@BeforeAll	Se aplica a un método `static` para indicar que se ejecuta antes que todos lo métodos de prueba de la clase. (JUnit4 @BeforeClass)
@AfterAll	Se aplica a un método `static` para indicar que se ejecuta antes que todos lo métodos de prueba de la clase. (JUnit4 @AfterClass)
@Disable	Ese aplica a un método de prueba para evitar esa prueba (JUnit4 @Ignore)

Metodo	Descripción
assertTrue(boolean valor)	Falla si valor no es `true`
assertFalse(boolean valor)	Falla si valor no es `false`
assertFalse(boolean valor, String mensaje)	Falla si valor no es `false` y muestra el mensaje
assertEquals(int esperado, int actual)	Falla si `esperado` es distinto de `actual`
assertEquals(int esperado, int actual, String mensaje)	Falla si `esperado` es distinto de `actual`
assertEquals(double esperado, double actual, double delta)	Falla si la diferencia entre `esperado` y `actual` es mayor a `delta`
assertNull(Object obj)	Falla si `obj` es distinto de `null`
assertNotNull(Object obj)	Falla si `obj` es `null`
assertEquals(Object esperado, Object actual)	Falla si los objetos son distintos, evaluando su método `equals()`
assertNotEquals(Object esperado, Object actual)	Falla si los objetos no son distintos, evaluando su método `equals()`
assertSame(Object esperado, Object actual)	Falla si las referencias de los objetos sin distintas

Herramientas de Pruebas. JUNIT.

Ejemplo 1 de clase de tests JUnit5 para clase con métodos matemáticos.

```
// Programa Principal
package com.jos.mates;
public class Mates {
  public static void main (String[] args) {
    int num1 = 0; int num2 = 0;
    int res = suma (num1, num2);
    System.out.println (num1 + " + " + num2 + " = " + res);
    }
  public static int suma(int num1, int num2) {
    return num1 + num2;
    }
// resto de métodos
  }
```

```
// Varios tests con casos de prueba diferentes
package com.jos.mates;
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.*;
public class MatesTest {
  @Test
  public void testSumaIgualSI () {
    assertEquals (3, Mates.suma (1, 2) , "La suma de 1+2 debería dar 3");
    }
  @Test
  public void testSumaIgualNO () {
    assertEquals (3, Mates.suma (1, 1), "La suma de 1+1 debería dar 2");
    }
  @Test
  public void testSumaNoIgualSI () {
    assertNotEquals (3, Mates.suma (1, 1), "La suma de 1+1 debería ser 2");
    } @Test
  public void testSumaNoIgualNO () {
    assertNotEquals (3, Mates.suma (1, 2), "La suma de 1+2 debería ser 3");
    }
  }
```

Ejemplo 2 de clase de tests JUnit5 para clase con métodos matemáticos.

```
// Varios tests con casos de prueba y anotaciones diferentes
package com.jos.mates;
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.*;
public class MatesTest {
  @Test
  public void testSumaIgual () {
    int num1 = 1; int num2 = 2;
    int esperado = 3;
    int resultado = Mates.suma (num1, num2);
    assertEquals (esperado, resultado);
    }
  }
```

```
// Varios tests con casos de prueba y anotaciones diferentes
package com.jos.mates;
import org.junit.jupiter.api.AfterEach;
import org.junit.jupiter.api.AfterAll;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.BeforeAll;
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.*;
public class MatesTest {
  private int num1, num2, esperado, resultado;
  @BeforeEach
  public void setUp() {
    num1 = 1; num2 = 2;
    esperado = 3;
    resultado = Mates.suma (num1, num2);
    }
  @Test
  public void testSumaIgual () {
    assertEquals (esperado, resultado, "El resultado de " + num1 + " + " + num2 + " debería ser: " + resultado);
//  fail ("El caso de prueba es un prototipo"); // línea de fallo provocado, eliminada o comentada tras escribir test.
    }
  }
```

Pruebas de Caja Negra.

Las **pruebas de caja negra** se realizan para evaluar el valor de las salidas de un sistema a partir de unas entradas concretas, sin tener en cuenta el funcionamiento interno del sistema, **con la intención de encontrar incongruencias entre las entradas y las salidas de los módulos**, empleándose para ellos distintas técnicas para escoger los casos de prueba, intentando reducir su número sin mermar la efectividad de las pruebas.

Técnicas de Caja Negra

- **Clases de equivalencia**: escogiendo valores representativos de los distintos grupos en los que se puedan dividir los datos de entrada/salida (clases de equivalencia), en función de la casuística que se pueda dar, según las condiciones de decisión, para abarcar la mayor funcionalidad posible del programa.
- **Análisis de los valores límite**: escogiendo como casos de prueba representativos de las clases de equivalencia los valores iniciales y finales, sus valores inferiores y superiores, y si se considera oportuno algún valor intermedio.
- **Estudio de errores típicos**: escogiendo como casos de prueba los que provocan errores que suelen repetirse con frecuencia en los programas (valor 0, valores pequeños, valores grandes, lista de valores vacía o con un solo elemento, errores tipográficos).
- **Datos aleatorios**: escogiendo casos de prueba al azar, no optimizados, pero en abundancia, generalmente generados por alguna herramienta automática.
- **Manejo de interfaz gráfica**: escogiendo casos de prueba que permitan descubrir errores en el manejo de sus elementos (menús, ventanas, iconos, botones, desplegables, cuadros de texto, etc).

Los pasos para realizar las pruebas de caja negra según la técnica de clases de equivalencia son: 1. Identificar las condiciones de las entradas y las salidas, 2. Identificar las clases de equivalencia válidas y no válidas, 3. Identificar un elemento representativo de cada clase de equivalencia como casos de prueba.

Los pasos para realizar las pruebas de caja negra según la técnica de análisis de valores límite son los mismos que según la técnica de clases de equivalencia, pero identificando como elementos representativos de las clases de equivalencia los valores límite, por encima y por debajo, de cada clase de equivalencia de valores de entrada, intentando provocar también valores límite de los valores de salida.

Pruebas de Caja Negra.

1. Clases de equivalencia para pruebas de caja negra de función que devuelve el nombre del mes a partir de su valor numérico.

- Clase 1: **1<=X<=12**
- Clase 2: **X<1**
- Clase 3: **X>12**.

- Caso de prueba clase 1: **X=**.6
- Caso de prueba clase 2: **X=0**
- Caso de prueba clase 3: **X=13**.

2. Análisis de valores límite para pruebas de caja negra de función que cuenta valores enteros entre 1 y 100 (ambos inclusive).

- Rango de valores: **0<=X<=100**

- Valores límite: **{0,100,1,99,-1,101}**

Nº	Caso de prueba		Representante
1	X <0	No valido	-1
2	0<= X <=100	Válido	0, 1, 99, 100
3	X >100	No valido	101
4	X decimal	No valido	1,5
5	X alfanumérico	No valido	a