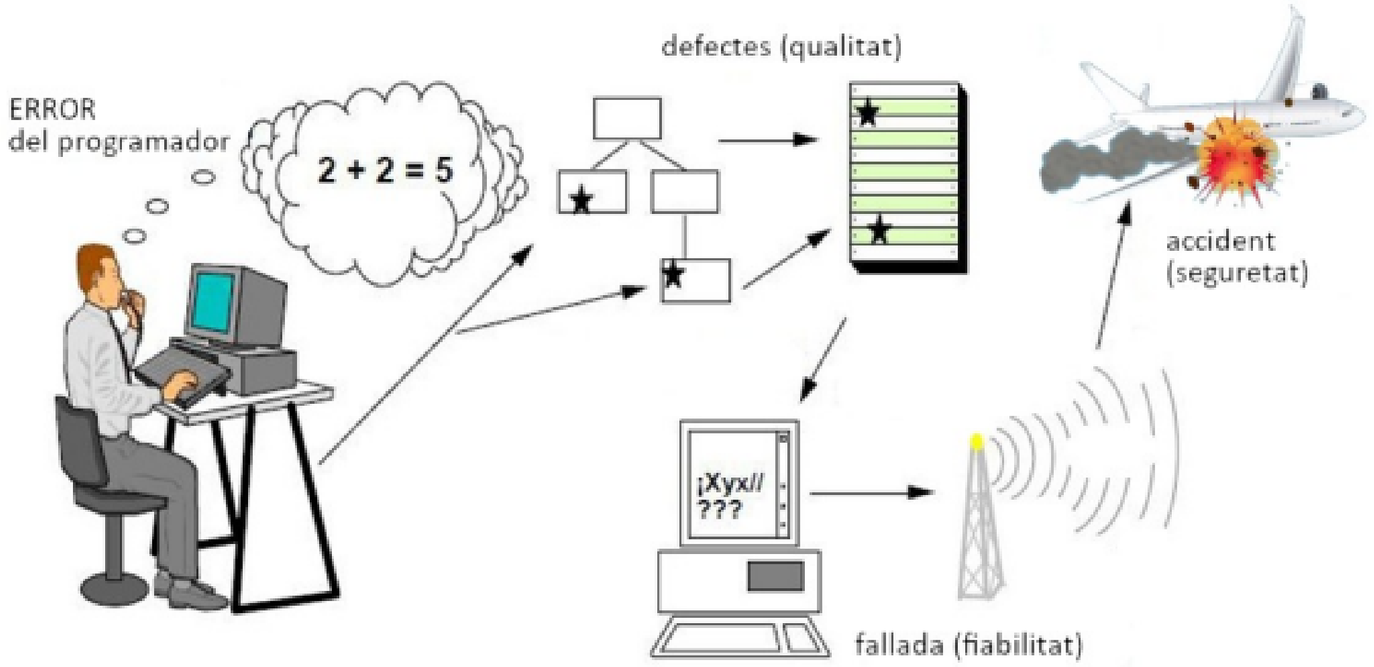
ENTORNS DE DESENRROTLLAMENT

1º DAM y DAW - CIP FP CHESTE

PRUEBAS

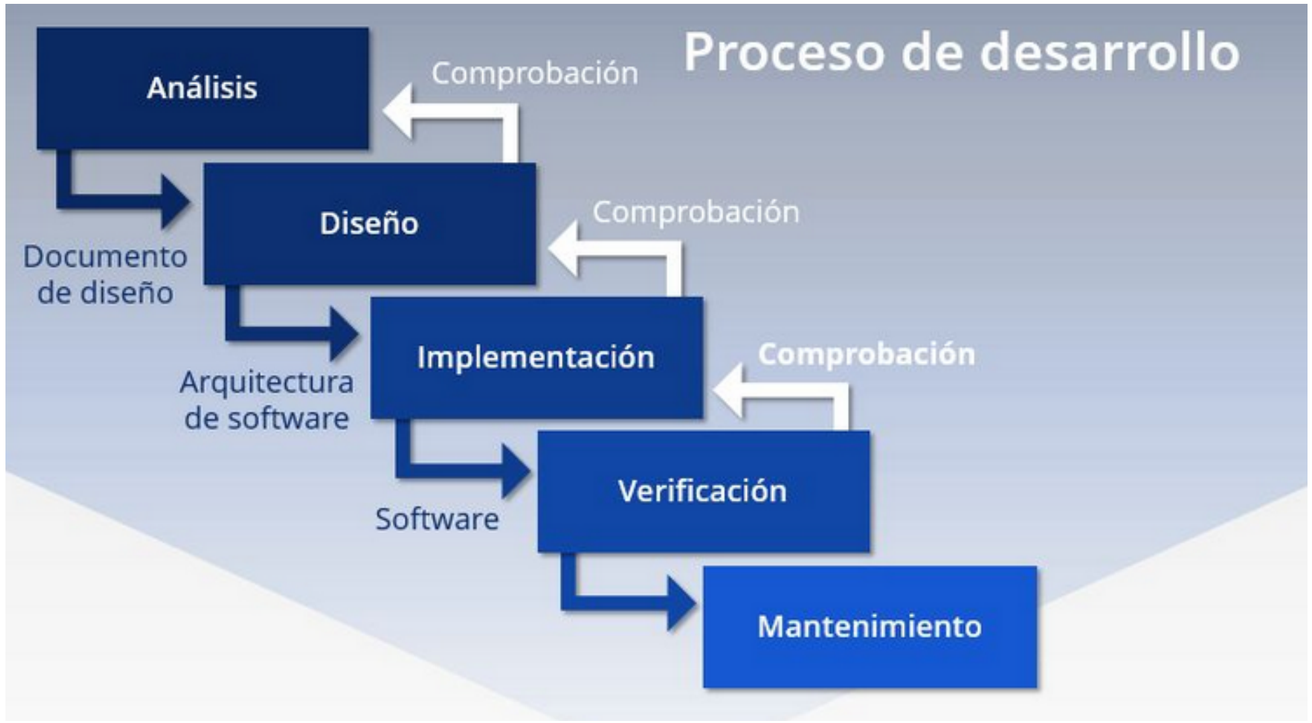
INTRODUCCIÓN

Un error del programador es un defecto que provoca un fallo en el sistema.



INTRODUCCIÓN

Los ciclos de desarrollo de software deben incorporar una fase de pruebas.



TERMINOLOGIA

- Prueba: actividad en la que un sistema se ejecuta en unas condiciones específicas.

- Caso de prueba: Conjunto de parámetros de entrada, condiciones y resultados obtenidos de un proceso en particular

TERMINOLOGIA

- Fallo: Incapacidad de un sistema de realizar las funciones para las que ha sido diseñado.

- Error: Diferencia entre el resultado del programa y el esperado. Resultado incorrecto.

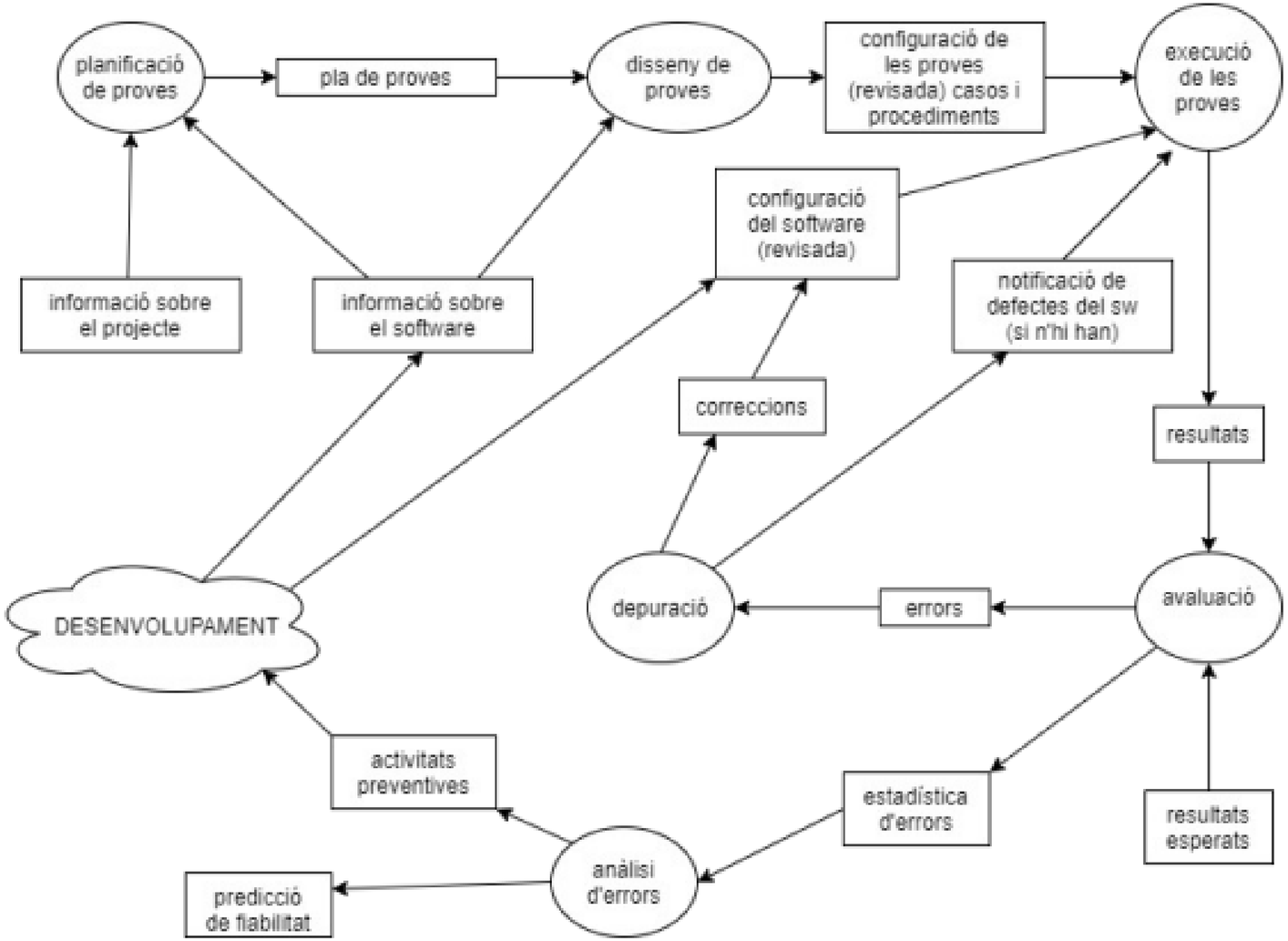
PRINCIPIOS BÁSICOS DE LA COMPROBACIÓN

- Las pruebas se harán en base a los requisitos del sistema.
- Los casos de prueba tienen que incluir tanto casos con datos válidos como inesperados.
- El diseño de un buen caso de prueba tiene altas probabilidades de descubrir un error.

PRINCIPIOS BÁSICOS DE LA COMPROBACIÓN

- Las pruebas comienzan desde lo básico a lo complejo.
- El diseño de pruebas debe copar una parte importante dentro del ciclo de vida.
- Las personas encargadas de diseñar las pruebas no deben ser las mismas que desarrollan el sistema.

EL PROCESO DE PRUEBA



DISEÑO DE CASOS DE PRUEBA

Tenemos un programa que calcula la suma de dos números enteros, del 0 al 99.

WHITE BOX TESTING

VS

BLACK BOX TESTING

DISEÑO DE CASOS DE PRUEBA

Tenemos un programa que calcula la suma de dos números enteros, del 0 al 99.

WHITE BOX TESTING

VS

BLACK BOX TESTING

WHITE BOX TESTING

En este tipo de enfoque probamos la estructura del programa. Es decir, que el programa recorra el flujo de trabajo esperado

Ejecución de sentencias y condiciones.

WHITE BOX TESTING

El diseño de los casos tiene que hacerse en base a los caminos que puede tomar el programa.

```java
public class Sumatoria {
    public static int sumatoria(int n) {
        int suma = 0;
        int i = 1;
        while (i <= n) {
            suma += i;
            i++;
        }
        return suma;
    }
}
```

WHITE BOX TESTING

```java
public class Sumatoria {
    public static int sumatoria(int n) {
        int suma = 0;
        int i = 1;
        while (i <= n) {
            suma += i;
            i++;
        }
        return suma;
    }
}
```

No pasar por el bucle

Pasar solo una vez

Pasar n veces

WHITE BOX TESTING

```java
public class Sumatoria {
    public static int sumatoria(int n) {
        int suma = 0;
        int i = 1;
        while (i <= n) {
            suma += i;
            i++;
        }
        return suma;
    }
}
```

No pasar por el bucle

n = 0

WHITE BOX TESTING

```java
public class Sumatoria {
    public static int sumatoria(int n) {
        int suma = 0;
        int i = 1;
        while (i <= n) {
            suma += i;
            i++;
        }
        return suma;
    }
}
```

Pasar solo una vez

n = 1

WHITE BOX TESTING

```java
public class Sumatoria {
    public static int sumatoria(int n) {
        int suma = 0;
        int i = 1;
        while (i <= n) {
            suma += i;
            i++;
        }
        return suma;
    }
}
```

Pasar n veces

n > 1

BLACK BOX TESTING

El diseño de las pruebas se centra en el diseño de las especificaciones, es decir, en los resultados obtenidos en los métodos y los parámetros que se pasan a estos.

BLACK BOX TESTING

Tengo que saber cuales son los parámetros de entrada que se le pueden pasar a un método para poder agruparlos en:

CLASES DE EQUIVALENCIA

CLASES DE EQUIVALENCIA

Son utilizadas para identificar conjuntos de valores de entrada que produzcan resultados similares.

EJEMPLO

Programa que calcula la suma de dos números enteros positivos que se pasan por parámetro.

```java
public class SumaEnteros {
    public static int suma(int num1, int num2) {
        if (num1 <= 0 || num2 <= 0) {
            throw new IllegalArgumentException("Los números deben ser mayores
        }
        return num1 + num2;
    }
```

EJEMPLO

```java
public class SumaEnteros {
    public static int suma(int num1, int num2) {
        if (num1 <= 0 || num2 <= 0) {
            throw new IllegalArgumentException("Los números deben ser mayores
        }
        return num1 + num2;
    }
```

EJEMPLO

C.E. 1 -> Números enteros positivos (1, 2, 3, 4 ...)

C.E. 2 -> Números enteros negativos (-1, -2, -3, ...)

C.E. 3-> Cero (0)

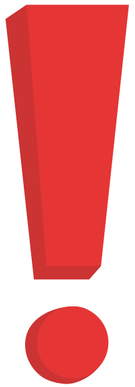
C.E. 4-> Valores no enteros (2'5, "coche")

C.E. 5-> Valores que superan el rango de int (2147483648)

EJEMPLO

C.E. 1 -> Números enteros positivos (1, 2, 3, 4 ...)

REVISAR LAS CONDICIONES LÍMITE



TIPOS DE PRUEBAS

Pruebas unitarias

Evalúan cada uno de los métodos por separado. "Comprobar que cada unidad funciona bien"

Pruebas de integración

Se encargan de comprobar el correcto funcionamiento de todas las partes en su conjunto.

TIPOS DE PRUEBAS

Pruebas de sistema

Verifican el comportamiento del software completo.

Pruebas de aceptación

El cliente comprueba que se cumplen todas las especificaciones requeridas.

TIPOS DE PRUEBAS

- Pruebas de usabilidad
- Pruebas de rendimeinto
- Pruebas de seguridad
- Pruebas de regresión

AUTOMATIZACIÓN DE PRUEBAS UNITARIAS

CPPUNIT

PRUEBAS UNITARIAS

El objetivo es aislar a cada parte del programa y mostrar que su funcionamiento es correcto.

- Casos de prueba (test case)

  Son clases con métodos para probar los métodos de una clase en concreto.

  Para cada clase que tengamos en nuestro proyecto, habrá un caso de prueba.

CREAR UNA CLASE DE PRUEBAS UNITARIAS

- TOOLS->CREATE/UPDATE TEST

ANOTACIONES JUnit

JUnit 4.x	JUnit 5.x	mètode
@Test	@Test	És el mètode de prova a realitzar
@BeforeClass	@BeforeAll	setUpClass s'executa una sola vegada abans de l'execució de les proves
@AfterClass	@AfterAll	tearDownClass s'executa una sola vegada abans de l'execució de les proves
@Before	@BeforeEach	setUp s'executa abans de cada mètode de prova
@After	@AfterEach	tearDown s'executa després de cada mètode de prova
@Ignore	@Ignore	Ignora el mètode de prova

ANOTACIONES TestNG

TestNG	mètode
@Test	És el mètode de prova a realitzar
@BeforeClass	setUpClass s'executa una sola vegada abans de l'execució de les proves
@AfterClass	tearDownClass s'executa una sola vegada abans de l'execució de les proves
@BeforeMethod	setUpMethod s'executa abans de cada mètode de prova
@AfterMethod	tearDownMethod s'executa després de cada mètode de prova
@Ignore	Ignora el mètode de prova

ASERCIONES

Comparaciones del valor esperado con el valor obtenido.

Si la aserción no es correcta se lanza una AssertionError.

Cada caso de prueba puede contener varias aserciones pero lo recomendable es que solo contenga una.

ASERCIONES en TestNG

- assertsEquals (actual, esperado, [mensaje])
- assertsEquals (actual, esperado, tolerancia [mensaje])
- assertsEqualsNoOrder (actual, esperado, [mensaje])
- assertTrue (condición, [mensaje])
- assertFalse (condición, [mensaje])
- assertNull (objeto, [mensaje])

https://www.javadoc.io/doc/org.testng/testng/6.8.17/index.html

EJEMPLO CALCULADORA

EJEMPLO MATRICES

TEST CON EXCEPCIONES

A veces no se espera como salida un valor, sino una EXCEPCIÓN.

Para comprobar que se lanza la excepción tenemos dos formas:

- Indicar la excepción en @Test

```
@Test(expectedExceptions = {java.lang.IllegalArgumentException.class})
```

- Utilizar el método fail

MÉTODO FAIL

Nos permite indicar que hay un fallo en la prueba. Cuando se ejecuta se finaliza la prueba.

```
@Test
public void testCalculaHoresExtras() {
 try {
   Empleat.calculaHoresExtras(-1.0, 20.0);
   fail("S'esperava l'excepció HoresException ");
 } catch(HoresException e) {   }
 try {
   Empleat.calculaHoresExtras(10, -20.0);
   fail("S'esperava l'excepció PreuException");
 } catch(PreuException e) {   }
}
```

EJEMPLO Matricad