UD 03 - DISEÑO Y REALIZACIÓN DE PRUEBAS

1. Definición de prueba. Cuándo hacer pruebas. La estrategia de pruebas

Prueba: Proceso que permite verificar y revelar la calidad de un producto software. Permite identificar los fallos de implementación, calidad o usabilidad de un programa.

Es necesario realizar pruebas en las diferentes etapas (diseño, implementación y una vez que la aplicación ha sido desarrollada) que permitan verificar que el software que se crea es correcto y cumple las especificaciones pedidas.

Pueden aparecer errores humanos como:

- incorrecta especificación de los objetivos
- errores del proceso de diseño
- errores de la fase de desarrollo.

Con las pruebas se realizan tareas de **VERIFICACION** y de **VALIDACION** :

- Verificación: Si sistema o sus partes cumplen las condiciones impuestas ¿Se construye correctamente?
- Validación: Si el sistema o sus partes satisfacen los requisitos ¿Es lo que se pide?

Debe implementarse una **ESTRATEGIA DE PRUEBAS** que, seguirían el modelo **Modelo en Espiral** (Modelo de ciclo de vida del software con actividades en espiral, cada iteración es un conjunto de actividades) :

- Empiezan con las **pruebas unitarias** (Se analiza el código implementado)
- Siguen con las **pruebas de integración** (Diseño y construcción de la arquitectura de software)
- Prosiguen las **pruebas de validación** (Se comprueba que cumpla el análisis de requisitos)
- Finalmente, **prueba de sistema** (Funcionamiento total del software y otros elementos)

2. Tipos de prueba y de casos de prueba

Se puede hablar de dos enfoques a la hora de pruebas y de casos de prueba:

- **Enfoque o Pruebas funcionales (De caja negra - Black Box Testing)**: Es la principal herramienta de VALIDACIÓN. Se analiza, para una concreta qué salida se espera recibir. Se utiliza la interfaz externa. No se indaga en la implementación de la misma. No es necesario conocer la estructura, ni el funcionamiento. **No se verifica el proceso, solo los resultados** . Se aplica con los valores límite y las clases de equivalencia.
- **Enfoque o Pruebas estructurales (De caja blanca - White Box Testing)**: Se prueba la aplicación desde dentro, usando su lógica de aplicación, sus distintos caminos de ejecución. Es necesario un conocimiento específico del código para poder analizar los resultados. **Se deberían probar todos los caminos que puede seguir la ejecución del programa** . Se aplica con el cubrimiento.
- **Pruebas aleatorias**: Modelos (muchas veces estadísticos) que representen las posibles entradas al programa para crear los casos de prueba

Pruebas de regresión: Probar un producto de software después de que se le hayan realizado modificaciones para garantizar que no se hayan introducido nuevos errores como resultado de los cambios. (errores, carencias de funcionalidad, divergencias funcionales con respecto al comportamiento esperado del software).

Detallamos un poco más los tipos de pruebas:

Tipo de prueba	Explicación
Pruebas de unidad	Funcionamiento de un módulo de código
Pruebas de carga	Observar el comportamiento bajo una cantidad de peticiones esperada. Podría mostrar los tiempos de respuesta de todas las transacciones importantes de la aplicación. Si se monitoriza la BBDD, el servidor... podría verse el cuello de botella. Es la prueba de rendimiento más sencilla.
Pruebas de estrés	Sirve para saber si la aplicación rendirá en casos en los que la carga real de peticiones supere a las esperadas. Se dobla el número de peticiones y se ejecuta una prueba de carga hasta que se rompe. Se usa para determinar la solidez de la aplicación en momentos de carga extrema.
Pruebas de estabilidad	Determinar si puede soportarse una carga esperada de forma continua. Usada para ver si hay fugas de memoria en la aplicación.
Pruebas de picos	Observar el comportamiento del sistema variando de forma drástica (bajadas y subidas) el número de usuarios.

2.1. El input de las pruebas funcionales (¿qué hace?)

Tendríamos varias opciones de inputs según la prueba que se quiera llevar a cabo:

- **Particiones equivalentes**: En ellas se considera el menor número posible de casos de pruebas, consiguiendo que cada caso abarque el mayor número posible de entradas diferentes. El objetivo es crear un conjunto de **clases de equivalencia** donde la prueba de un valor representativo de la misma, sea extrapolable al que se conseguiría probando cualquier valor de la clase.

- **Análisis de valores límite**: Como entrada se introducen los que están en el límite de las clases de equivalencia.
- **Pruebas aleatorias**: Se usan generadores aleatorios de entradas para crear un volumen de casos al azar con los que alimentar la aplicación. Suelen usarse en aplicaciones no interactivas (es más difícil generarlas en este tipo de entornos)

2.2. Pruebas estructurales (¿cómo lo hace?)

Verificar la **estructura interna de cada componente de la aplicación**, **independientemente de la funcionalidad establecida** para el mismo.

No quiere comprobar la corrección de los resultados producidos, sino comprobar que:

- Se ejecutan todas las instrucciones del programa
- No hay código no usado
- Los caminos lógicos del programa se recorren
- ...

Para estas pruebas se realizan **CRITERIOS DE COBERTURA LÓGICA**, cuyo cumplimiento determina la mayor o menor seguridad en la detección de errores. Estos criterios pueden ser:

- **Cobertura de sentencias**: Generar casos de prueba suficientes para que cada instrucción del programa se ejecute al menos una vez.
- **Cobertura de decisiones**: Generar casos de prueba suficientes para que cada opción resultado de una prueba lógica del programa se evalúe al menos una vez a cierto y otra a falso.

- **Cobertura de condiciones**: Generar casos de prueba suficientes para que cada elemento de una condición se evalúe al menos una vez a cierto y otra a falso.
- **Cobertura de condiciones y decisiones**: Cumplir simultáneamente la cobertura de decisiones y la cobertura de condiciones.
- **Cobertura de caminos**: Criterio más importante. Se debe ejecutar al menos una vez cada secuencia de sentencias encadenadas, desde la sentencia inicial a la sentencia final. Como el número de caminos puede ser muy grande, se reduce el número a lo que se conoce como *camino prueba*.
- **Cobertura del camino de prueba**: Podría haber dos variantes:
  - Una indica que cada bucle debe ejecutarse solo una vez (hacerlo más veces no aumentaría la efectividad de la prueba)
  - Otra indica que se pruebe cada bucle tres veces (la primera sin entrar en su interior, la otra ejecutándolo una vez, la otra ejecutándolo dos veces)

2.3. Pruebas de regresión (¿se ha roto algo después de cambiar lo que fallaba?)

Las modificaciones de código pueden generar errores colaterales que no existían antes.
Deben repetirse las pruebas realizadas con anterioridad para comprobar que no se introduce comportamiento indeseado o errores adicionales.

Deben **llevarse a cabo cada vez que se hace un cambio en el sistema** (ya sean `fixes` o mejoras de la funcionalidad)

Pueden realizarse manualmente, volviendo a realizar un subconjunto de todos los casos de prueba o utilizando herramientas automáticas.

Las pruebas de regresión contiene tres clases diferentes de clases de prueba:

- Muestra representativa de pruebas que ejecute todas las funciones del software.
- Pruebas adicionales que se centran en las funciones del software que se van a ver probablemente afectadas por el cambio
- Pruebas que se centran en los componentes de software que han cambiado.

Las pruebas de regresión se deben diseñar para incluir solo aquellas pruebas que traten una o más clases de errores en cada una de las funciones del programa. **No es práctico ni eficiente volver a ejecutar cada prueba de cada función del programa después de un cambio.**

3. Tipos de errores en el proceso de desarrollo. Las herramientas de depuración ante los errores lógicos

En el proceso de desarrollo se producen:

- Errores de compilación: Olvidar un símbolo, referenciar a variable inexistente, utilizar sentencia incorrecta. El IDE aporta información de dónde se produce y el programa no puede compilarse hasta no ser corregido el erroores.
- Errores lógicos o bugs: Provoca que el programa devuelva resultados erróneos, termine antes de tiempo, se ejecute infinitamente... En estos casos debe recurrirse a un depurador.

Los IDE (Integrated Development Environment) suministran herramientas de depuración para verificar el código generado y realizar pruebas estructurales y funcionales.

El depurador (debugger) permite supervisar la ejecución de los programas para localizar y eliminar los errores lógicos. Con el depurador se puede suspender la ejecución del programa, examinar y establecer los valores de las variables, comprobar los valores devueltos por un determinado método, el resultado de una comparación lógica o relacional..

- **Puntos de ruptura (Breakpoints)**: Marcadores que pueden establecerse en cualquier línea de código ejecutable (no admite comentarios o líneas en blanco). El programa se ejecuta hasta la línea marcada, momento en el que se pueden examinar las variables, iniciar depuración paso a

paso comprobando el camino que toma el programa... Finalmente se puede abortar el programa o continuar la ejecución normal del mismo.
- **Examinadores de variables**: Permite ver el valor que van tomando las variables en el avance paso a paso.

Tipos de ejecución del depurador

- **Avance paso a paso**: Se ejecuta línea por línea
- **Paso a paso por procedimientos**: Se introduce el parámetro que se quiere a un método o función y nos devuelve el resultado. (Hemos comprobado que el método funciona correctamente y solo nos interesa el valor que devuelve)
- **Ejecución hasta instrucción**: El depurador se ejecuta hasta la instrucción en la que se encuentra el cursor y, en ese punto, hacer depuración paso a paso o por procedimientos.
- **Ejecución hasta el final** sin detenerse en instrucciones intermedias.

Los modos de ejecución deben ajustarse a las necesidades de depuración que tengamos en cada momento. Lo importante es que se puedan examinar todas las partes que se consideren necesarias de forma rápida, sencilla y clara.

Cómo debuggear en el prehistórico IDE Netbeans....

En NetBeans hay un panel llamado ventana de inspección en la que se pueden añadir todas las variables de las que se tenga interés en inspeccionar su valor.

En el menú de depuración de Netbeans se pueden seleccionar los modos de ejecución especificados antes y algunos más.

¿Cómo debuggear?
Pinchamos en **Depurar** .
Vamos analizando el código "Paso a paso", "Ejecutar hasta el cursor".
Se puede detener la ejecución, continuarla o ejecutar el programa hasta el final.
Se puede elegir "Entrar en el siguiente método" o seguir con la ejecución paso a paso con F7.
Si no necesita depuración podemos "Continuar ejecución" pulsando F8.
Si no nos interesa seguir con la depuración podemos "Finalizar sesión del depurador"
Si queremos pausar la depuración "Pausa"
Si queremos continuar la ejecución normal o seguir hasta el siguiente punto de ruptura "Continuar"

Línea roja indica punto de ruptura. Línea verde indica por donde está pasando el depurador.

El depurador también tiene:

- Consulta de la pila (stack) para ver qué funciones se llaman
- Inserción de puntos de ruptura
- Evaluar expresión

Una vez depurado, si consideramos que no hace lo que debe lo cambiamos.
Si una vez depurado hace lo que debe, puede pasarse a probar diseñando casos de prueba en JUnit y aplicándolos.

4. La validación

Se intenta descubrir errores desde el punto de vista de los requisitos (Comportamiento y casos de uso que se esperan que cumpla el software que se está diseñando). Interviene de forma decisiva el cliente.

Se realizan una serie de pruebas de caja negra que demuestran la conformidad con los requisitos.

Plan de prueba: Clases de pruebas que se han de llevar a cabo
Procedimiento de prueba: Define los casos de pruebas específicos para descubrir errores según los requisitos

Se busca que se satisfacen todos los requisitos funcionales, que se alcanzan todos los requisitos de rendimiento, que las documentaciones son correctas e inteligibles, que se alcanzan requisitos de portabilidad, compatibilidad, recuperación de errores, facilidad de mantenimiento...

Finalmente;

- Puede ser que las características están de acuerdo con las especificaciones o puede ser que se descubra una desviación se elabore una lista de deficiencias.
- Las desviaciones o errores descubiertos en esta fase del proyecto, raramente se pueden corregir antes de la terminación planificada.

5. Pruebas de código

5.1. Cubrimiento

Comprobar que todas las funciones, sentencias, decisiones y condiciones se ejecutan.

```
int prueba (int x, int y)
{
        int z=0;
        if ((x>0) && (y>0))
        {
                z=x;
        }
        return z;
}
```

Cubrimiento de la función: Durante la ejecución del programa la función debe ser llamada al menos una vez.

- Cubrimiento de sentencias: `prueba(1,1)` lo cumpliría
- Cubrimiento de decisión: `prueba(1,1)` Y `prueba(0,1)` lo cumplirían
- Cubrimiento de condición: `prueba(1,1)` , `prueba(1,0)`, `prueba(0,0)`.

Hay otros criterios para comprobar el cubrimiento:

- Secuencia lineal de código y salto
- JJ-Path Cubrimiento (Cantidad de rutas de control ejecutadas en las pruebas)
- Cubrimiento de entrada y salida (Evaluar si se han probado todas las combinaciones posibles de valores de entrada y de salida)
  Herramientas comerciales como Clover o Jacoco.

5.2. Valores límite

Los casos de prueba con mayor probabilidad de éxito son los de los valores límite.
Es una técnica complementaria a las particiones equivalentes pero se generan no un conjunto de valores sino unos pocos en el límite del rango de valores aceptado por el componente a probar.

```
public double funcion1 (double x)
{
   if (x > 5)
      return x;
   else
      return -1;
}

public int funcion2 (int x)
{
   if (x > 5)
      return x;
   else
      return -1;
}
```

Los valores límite son diferentes porque un método admite como parámetro un `double` y el otro un `int`. Como el parámetro de entrada tiene que ser mayor que 5. En uno valores límite 4,99 y 5,01. En otro valores límite 4 y 6.

Si el parámetro de entrada está entre -4 y+4, los valores límite son -5, -4, -3, 3, 4 y 5

Si el parámetro de entrada es entre 5 y 10 siendo valor `single` (precisión simple, `float`), los valores límite pueden ser 4,99; 5; 5,01; 9,99; 10; 10,01)

5.3. Clases de equivalencia

Cada caso de prueba quiere cubrir el mayor número de entradas posibles.

El dominio de valores de entrada se divide en número finito de clases de equivalencia.
La prueba de un valor representativo de cada clase, permite suponer que el resultado será el mismo que con cualquier otro valor de equivalencia.

Ejemplos:

- Cuando un parámetro de entrada debe estar en un determinado rango hay tres clases de equivalencia: Por debajo, en y por encima.
- Si una entrada requiere un valor en un conjunto hay dos clases de equivalencia: En el conjunto o fuera de él.
- Si una entrada es booleana hay dos clases de equivalencia: Sí y no.

```
public double funcion1 (double x)
{
    if (x > 0 && x < 100)
        return x+2;
    else
        return x-2;
}
```

Clases de equivalencia:
1- Por debajo x <= 0
2- En x > 0 y x < 100
3- Encima x >= 100

Casos de prueba
1 - Por debajo x = 0
2.- En x = 50
3.- Por encima x = 100

6. Normas de calidad

Estándar usado ahora: ISO/IEC 29119 (O hace mil años, cuando se hizo el temario)

Pretende:

- Unificar en una única norma todos los estándares dando vocabulario, procesos, documentación y técnicas para cubrir todo el ciclo de vida de software. Desde estrategias de prueba para la organización y políticas de prueba, prueba de proyecto al análisis de casos de prueba, diseño, ejecución e informe.
- Corregir el hecho de que estándares anteriores no cubren facetas de la fase de pruebas como la organización del proceso y gestión de las pruebas y tienen pocas pruebas funcionales y no-funcionales, etc.

Contenido ISO/EIC 29119:

- Parte 1. Conceptos y vocabulario:
  - Introducción a la prueba.
  - Pruebas basadas en riesgo.
  - Fases de prueba (unidad, integración, sistema, validación) y tipos de prueba (estática, dinámica, no funcional, …).
  - Prueba en diferentes ciclos de vida del software.
  - Roles y responsabilidades en la prueba.
  - Métricas y medidas.
- Parte 2. Procesos de prueba:
  - Política de la organización.
  - Gestión del proyecto de prueba.
  - Procesos de prueba estática.

    - Procesos de prueba dinámica.
- Parte 3. Documentación
    - Contenido.
    - Plantilla.
- Parte 4. Técnicas de prueba:
    - Descripción y ejemplos.
    - Estáticas: revisiones, inspecciones, etc.
    - Dinámicas: Caja negra, caja blanca, Técnicas de prueba no funcional (Seguridad, rendimiento, usabilidad, etc) .

ChatGPT dice que...

Algunas organizaciones y profesionales de pruebas de software pueden optar por seguir las pautas y estándares establecidos por la ISO/IEC 29119, mientras que otros pueden utilizar metodologías y prácticas diferentes, como las propuestas por el ISTQB (International Software Testing Qualifications Board) u otras.

ISTQB: Es una organización que proporciona un esquema de certificación profesional en el campo de las pruebas de software.

Estándares usados anteriormente

Estándares BSI
-> BS 7925-1 Pruebas de software Parte 1. Vocabulario.
-> _BS 7925-2 Pruebas de software. Parte 2. Pruebas de los componentes software.

Estándares IEEE
-> IEEE estándar 829 Documentación de la prueba de software
-> IEEE estándar 1008 Pruebas de unidad

Estándar *ISO/EIC 12207, 15289*

7. Pruebas unitarias

Las **pruebas unitarias** o **pruebas de unidad** buscan probar el funcionamiento de un módulo de código por separado.
En las pruebas de integración se asegurará el funcionamiento del sistema.

Unidad: Parte de la aplicación más pequeña que se puede probar (Función, procedimiento método).

Las pruebas unitarias deben ser:

- Automatizables (sin intervención manual)
- Completas (elevado cubrimiento)
- Repetibles (ejecutar más de una vez)
- Independientes entre sí
- Profesionales (con profesionalidad, documentación, ..)

Las pruebas unitarias tienen las siguientes ventajas:

- **Fomentan el cambio**: Facilitan que el programador cambie el código para mejorar su estructura al poder hacer pruebas sobre los cambios y asegurarse de que no se han introducido errores
- **Simplifica la integración**: Fase de integración con grado alto de seguridad de que el código funciona correctamente
- **Documenta el código**: Las propias pruebas actúan de documentación del código
- **Separación entre interfaz e implementación**: La interacción entre los casos de prueba y las unidades probadas son las interfaces de estas últimas, se puede cambiar cualquiera de los dos sin afectar al otyro
- **Errores más acotados y son fáciles de localizar** gracias a las pruebas unitarias.

7.1. Herramientas para Java

Jtiger

- Pruebas unitarias Java 5
- Código abierto
- Exportar informes en HTML, XML, Texto plano
- Casos de prueba de JUnit mediante plugin.
- Aserciones para comprobar el cumplimiento del método
- Metadatos (datos que describen otros datos) de los casos de prueba se especifican con anotaciones
- Tarea de Ant para automatizar pruebas
- Documentación completa
- El framework incluye pruebas sobre sí mismo.

TestNG

- Inspirado en JUnit y NUnit
- Diseñado para cubrir todo tipo de pruebas (unitarias, funcionales, integración)
- Anotaciones de Java 5 (WTF... una novedad, claro)
- Compatible con JUnit
- Soportado por Eclipse, Netbeans, IntelliJ
- Las pruebas pueden invocarse desde línea de comandos con tarea de ANT o fichero XML
- Métodos de prueba se organizan en grupos (método puede pertenecer a uno o varios grupos)

JUnit

- Framework (Estructura conceptual y tecnológica de soporte definida, normalmente con módulos de software concretos con base en la cual otro proyecto de software puede ser organizado y desarrollado) de pruebas unitarias creado por Gamma y Beck
- Código abierto
- Implementación de la arquitectura xUnit
- Muy documentado
- Es el que se usa de verdad en Java (Estándar)
- Usa desde JUnit 4 las anotaciones JDK 1.5 (qué novedad....)
- Se pueden crear informes en HTML
- Organización de pruebas en Suites de pruebas

7.2. Herramientas para otros lenguajes

(Basados en xUnit)

CppUnit

- Pruebas unitarias para C++

Nunit

- Framework para .NET

SimpleTest, PHP Unit

- PHP

FoxUnit

- Microsoft visual FoxPro

MOQ

- Para mockeo

8. Automatización de la prueba

Para IDEs como NetBeans, Eclipse, IntelliJ tenemos JUnit.
Permite diseñar clases de prueba para cada clase diseñada en la aplicación.
Se establecen los métodos que se quieren probar y para ello se diseñan casos de prueba.
JUnit presenta un informe con los resultados de la prueba. En función de ellos se debe (o no) modificar el código.

`assertTrue()`: Evalúa expresión booleana. Pasa la prueba si es cierta.
`assertFalse()`: Evalúa expresión booleana. Pasa la prueba si es falsa.
`assertNull()`: Verifica que sea nulo.
`assertNotNull()`: Verifica que no sea nulo.
`assertSame()`: Asegura que los objetos de las dos referencias tienen misma dirección de memoria.
`assertNotSame()`: Asegura que los objetos de las dos referencias NO tienen misma dirección de memoria.
`assertEquals()`: Asegura que son iguales en contenidos (primitivos, objetos con `equals()`
`assertThrows()`: Éxito si se lanza una excepción
`assertAll()`: Agrupar conjunto de asserts
`fails()`: Provoca que la prueba falle inmediatamente.

Asunciones
`assumeTrue(suposicion, mensaje)`: Continua ejecutando la prueba soplo si la condición es verdadera. Si es falsa se considera omitida (no se ejecutan las aserciones siguientes)
`assumeFalse(suposicion, mensaje)`
`assumingThat(suposicion, ejecutable)`

Anotaciones...
`@DisplayName("")` Da un nombre al test (Clase o método)
`@Disabled("")` Deshabilita el test y da motivos
`@BeforeEach`: Lo ejecuta antes de cada
`@AfterEach`: Lo ejecuta después de cada
`@BeforeAll`: Lo ejecuta antes de correr el primero de los test
`@AfterAll`: Lo ejecuta al final de los test
`@EnabledOnOs("")`: Solo activa el test en un SO (o sistemas operativos) concretos
`@EnabledOnJre`: Solo en una versión de JRE indicada
`@EnabledIfSystemProperty(named:"", matches:"")` En función de la propiedad
`@EnabledIfenvironmentProperty(named:"", matches:"")` En función de la variable de entorno

En el viejo Netbeans...

Tengamos una clase `Car` que tiene un constructor `Car(String name, double prize, int stock)`

Queremos testear:

- El método `buyCars(int cars)` que sumará estos coches al stock (y si son negativos o 0 arrojará excepción)
- El método `sellCars(int cars)` que restará estos coches del stock y que si son negativos o 0 arrojará excepción y que si el stock es menor que estos también arrojará excepción.

VALORES CORRECTOS

VALORES NO VÁLIDOS

VALORES LÍMITE. En nuestro caso el valor límite es 0.

File -> New File -> Test for Existing Class (seleccionando la clase).

```
@Test
public void testBuyCarsValid() throws Exception {
        try {
                Car car = new ("Patata", 12000, 300);
                car.buyCars(100);
                assertTrue(car.getStock()==400);
                //assertEquals(400, car.getStock);  // Tercer parámetro de offset
        } catch (Exception e) {
                fail("Excepción no esperada"+e);
        }
}

@Test
public void testBuyCarsNegativeThrowException() throws Exception {
        try {
                Car car = new ("Patata", 12000, 300);
                Exception exception = assertThrow(Exception.class, () -> {car.buyCars(-100)};
                assertEquals("Intento de comprar un número negativo de coches",
exception.getMessage());
        } catch (Exception e) {
                fail("Excepción no esperada"+e);
        }
}

// Comprar 0
// Vender válido
// Vender 0
// Vender negativo
// Vender menor que el stock
```

Suite de tests

Es un contenedor que agrupa un conjunto de pruebas unitarias relacionadas para que se ejecuten juntas. Útil para proyectos grandes con muchas pruebas. La Suite en sí no debería contener ningún test, solo agrupa las clases de prueba.

```
import org.junit.runner.RunWith;
import org.junit.runners.Suite;

@RunWith(Suite.class)
@Suite.SuiteClasses({
    TestClass1.class,
    TestClass2.class,
    TestClass3.class
})
public class TestSuite {
    // Esta clase no necesita contener ningún método de prueba, solo sirve para agrupar
pruebas
}
```

Pueden incluirse y excluirse clases con `@SelectClasses`, `@SelectPackages`, `@IncludePackages`, `@ExcludePackages`, `@IncludeClassNamePatters`, `@ExcludeClassNamePatters`, `@IncludeTags`, `@ExcludeTags`, `@SelectMethod`..

TDD

Se llama Test Driven Development (TDD) a la metodología de desarrollo de software que consiste en escribir primero los test unitarios y después el código.
Pasos:

- Escribir prueba para el requisito
- Comprobar si la prueba falla
- Escribir el código que haga pasar la prueba
- Correr la prueba y comprobar qué pasa
- Refactorizar si es necesario

9. Documentación de la prueba

Metodologías actuales como `Métrica v.3` (Metodología de Planificación, Desarrollo y Mantenimiento de Sistemas de Información. Se puede usar libremente solo citando la propiedad intelectual que es el Ministerio de Presidencia) proponen que la documentación de fase de pruebas se base en estándares ANSI / IEEE de verificación y validación de software.

Se busca describir un conjunto de documentos para las pruebas de software (Facilita un marco común).

Documentos a generar (Proceso de Análisis del Sistema):

- Plan de pruebas: Planificación general
- Especificación del diseño de pruebas: Ampliación y detalle del plan de pruebas
- Especificación de un caso de prueba: A partir de la especificación del diseño de pruebas
- Especificación de un procedimiento de prueba: Detallar el modo en que serán ejecutados cada uno de los casos de prueba
- Registro de pruebas: Registro de sucesos durante las pruebas
- Informe de incidente de pruebas: Para cada incidente y defecto detectado. solicitud de mejoras, etc.
- Informe sumario de pruebas: Resumen de las actividades de prueba vinculadas a una o más especificaciones de diseño de pruebas.