3. Diseño y realización de pruebas

Autor	Ⓧ Xerach Casanova
Clase	Entornos de desarrollo
Fecha	@Dec 19, 2020 10:44 PM

Planificación de las pruebas

Desde la fase de análisis hasta la implantación del software en el cliente, es necesario realizar un conjunto de pruebas que permita comprobar que el producto que estamos creando, es correcto y cumple con las especificaciones solicitadas.

Las pruebas tratan de verificar y validar las aplicaciones:

- **La verificación** es la comprobación de que un sistema o parte de él, cumple con las condiciones impuestas, comprobando así que se está construyendo correctamente. **Es un ejercicio teórico para estar seguro que se cumplen los requerimientos en el diseño.**
- **La validación** es el proceso de evaluación del sistema o uno de sus componentes para determinar si satisface los requisitos especificados. **Es un ejercicio práctico que garantiza que el producto funciona a partir de esos requerimientos.**

El proceso de pruebas se lleva a cabo con estrategias de pruebas. En el **modelo en espiral,** se empiezan las pruebas en cada porción de código y una vez han sido pasadas con éxito, se sigue con la prueba de integración, donde se ponen todas las partes del código en común, comprobando que el ensamblado de todo atiende a lo establecido en la fase de diseño.

El siguiente paso es la prueba de validación y finalmente se alcanza la prueba de sistema, que verifica el funcionamiento total del software y otros elementos del sistema.

El objetivo de las pruebas es conseguir un software libre de errores, además el programador debe evitar probar sus propios programas para evitar que vuelva a pasar inadvertidos aspectos que no tuvo en consideración.

2. Tipos de prueba

Existen dos enfoques fundamentales:

- **Prueba de la caja negra (Black Box Testing):** La aplicación se prueba usando interfaz externa, no nos preocupamos de la implementación del código, solo si los resultados son correctos a partir de los datos introducidos. En el siguiente código, mediante pruebas de caja negra se detecta un error mediante caja negra, ya que si se saca un cinco se devuelve que has suspendido.

```
if(nota>=5)
      System.out.println("Has suspendido");
else
       System.out.println("Has aprobado");
```

- **Prueba de la caja blanca (White Box Testing):** se prueba la aplicación desde dentro, usando su lógica de aplicación. Más bien lo que se busca es encontrar estructuras ineficientes o incorrectas. En el siguiente código, mediante prueba de caja blanca detectamos un error en la estructura, ya que el segundo else nunca será ejecutado, aunque el resultado de caja negra haya salido correctamente

```
if(nota>=5)
      System.out.println("Has aprobado");
else
      if(nota<5)
             System.out.println("Has suspendido");
       else
             System.out.println("Esta instrucción nunca se ejecutará.");
```

2.1. Funcionales (pruebas de caja negra)

No nos interesa la implementación del software, solo si se realizan las funciones esperadas de él siguiendo el enfoque de las pruebas de Caja Negra. Comprenden actividades para verificar una acción específica o funcional del código de la aplicación.

Se responde a las preguntas: ¿Puede el usuario hacer esto? o ¿Funciona esta utilidad de la aplicación?

Para realizar este tipo de pruebas se analizan las entradas y las salidas de cada componente y se verifica el resultado. Este tipo de prueba no considera en ningún caso el código desarrollado, ni el algoritmo, ni la eficiencia ni las partes de código innecesarias. Existen tres tipos de pruebas:

- **Particiones equivalentes.** Se considera el menor número posible de casos de pruebas abarcando el mayor número posible de entradas distintas.
- **Análisis de valores límite:** Se eligen valores de entrada que se encuentran en el límite de las clases de equivalencias.
- **Pruebas aleatorias.** Generamos entradas aleatorias a la aplicación. Se suelen utilizar generadores de prueba, capaces de crear volúmenes de

casos de prueba al azar (suele utilizarse en aplicaciones no interactivas por su dificultad de implementación en entornos interactivos).

2.2. Pruebas estructurales (pruebas de caja blanca)

Las pruebas estructurales son el conjunto de pruebas de la caja blanca, con las cuales verificamos la estructura interna de cada componente de la aplicación, dejando de lado la funcionalidad establecida para el mismo.

No se comprueba la corrección de los resultados producidos, sino que se ejecutan todas las instrucciones del programa, que no hay código no usado, o que los caminos lógicos del programa se van a recorrer. Los criterios de cobertura que se siguen son:

- **Cobertura de sentencias**. Se generan casos de pruebas suficientes para que cada instrucción del programa se ejecute al menos una vez.
- **Cobertura de decisiones.** Se generan casos de prueba para que cada opción resultado de una prueba lógica, se evalúe al menos una vez en cierto o falso.
- **Cobertura de condiciones:** Se generan casos de prueba para que cada elemento de una condición se ejecute al menos una vez a falso y otra a verdadero.
- **Cobertura de condiciones y decisiones:** que se cumplen simultáneamente las dos anteriores.
- **Cobertura de caminos:** es el criterio más importante. Se debe ejecutar al menos una vez cada secuencia de sentencias encadenadas desde la sentencia inicial del programa hasta la sentencia final. Como el número de caminos que puede tener una aplicación es muy grande, se reduce el número a lo que se conoce como camino prueba.
- **Cobertura del camino de prueba:** Se realizan dos variantes. Una indica que cada bucle se debe ejecutar sólo una vez y otra recomienda que se pruebe el bucle tres veces, la primera sin entrar, una ejecutándolo una vez y otra ejecutándolo más veces.

Cubrimiento

La tarea la realiza el programador y consiste en comprobar los caminos definidos por el código y que se recorren.

Por ejemplo:

Considerando que esta función forma parte de un programa mayor, se considera lo siguiente:

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

- Si durante la ejecución del programa, la función es llamada, al menos una vez, el cubrimiento de la función es satisfecho.
- El cubrimiento de sentencias para esta función, será satisfecho si es invocada, por ejemplo como prueba(1,1), ya que en esta caso, cada línea de la función se ejecuta, incluida z=x;
- Si invocamos a la función con prueba(1,1) y prueba(0,1), se satisfará el cubrimiento de decisión. En el primer caso, la if condición va a ser verdadera, se va a ejecutar z=x, pero en el segundo caso, no.
- El cubrimiento de condición puede satisfacerse si probamos con prueba(1,1), prueba(1,0) y prueba(0,0). En los dos primeros casos x<0 se evalúa a verdad mientras que en el tercero, se evalúa a falso. Al mismo tiempo, el primer caso hace (y>0) verdad, mientras el tercero lo hace falso.

Existen otra serie de criterios, para comprobar el cubrimiento.

- Secuencia lineal de código y salto.
- JJ-Path Cubrimiento.
- Cubrimiento de entrada y salida.

Existen herramientas comerciales y también de software libre, que permiten realizar la pruebas de cubrimiento, entre ellas, para Java, nos encontramos con Clover.

2.3. Pruebas de regresión

Las modificaciones en el programa por detección de fallo pueden generar errores colaterales que no existían antes, lo cual nos obliga a repetir pruebas que hemos realizado antes.

Las pruebas de regresión se hacen, en definitiva, cada vez que se hace un cambio en el sistema, ya sea para corregir errores o para realizar mejoras. No solo es suficiente probar componentes modificados o añadidos, sino controlar

que esas modificaciones no producen efectos negativos en otros componentes.

Las pruebas de software con éxito, son aquellas que dan como resultado el descubrimiento de errores y, en consecuencia, se produce corrección y modificación de algún componente del software, documentación y datos que lo soportan. La prueba de regresión nos ayuda a asegurar que estos cambios no introducen comportamiento no deseado y errores adicionales.

Las pruebas de regresión contienen tres clases distintas:

- Una muestra representativa de pruebas que ejercite todas las funciones.
- Pruebas adicionales centradas en funciones del software que van a ser afectadas por el cambio.
- Pruebas que se centran en componentes que han cambiado.

Al no ser práctico ejecutar cada una de las pruebas después de cada cambio, se deben diseñar estas pruebas para incluir solo aquellas que traten una o más clases de errores en cada una de las funciones principales del programa.

3. Procedimientos y casos de prueba

La prueba consiste en la ejecución de un programa con el objetivo de encontrar errores.

Según el IEEE, un caso de prueba es un conjunto de entradas, condiciones de ejecución y resultados esperados, desarrollados para un objetivo particular (ejercitar un camino concreto de un programa o verificar el cumplimiento de un determinado requisito, incluyendo toda la documentación asociada).

Existen varios procedimientos para el diseño de los casos de prueba:

- **Enfoque funcional o de caja negra.** Se centra en probar que el programa o parte de él, recibe una entrada de forma adecuada y se produce una salida correcta. Se centra en funciones, entradas y salidas. Se aplican valores límite y clases de equivalencia.
- **Enfoque estructural o caja blanca.** Nos centramos en la implementación interna del programa, se deben probar todos los caminos que puede seguir la ejecución del programa.
- **Enfoque aleatorio.** A partir de modelos obtenidos estadísticamente, se elaboran casos de prueba de entrada en el programa.

4. Herramientas de depuración

Cada IDE y lenguaje de programación, incluye herramientas de depuración como: inclusión de puntos de ruptura, ejecución paso a paso de cada instrucción, ejecución del procedimiento, inspección de variables, etc.

Se pueden producir errores de compilación y errores lógicos.

Cuando cometemos errores de codificación, el entorno nos proporciona información donde se produce y como solucionarlo. Estos errores no permiten compilar hasta que se corrijan.

Los errores lógicos también se llaman bugs, permiten compilar, sin embargo, puede provocar que el programa devuelva resultados erróneos y que el programa no pueda terminar o no termine nunca.

Los entornos de desarrollo incorporan el depurador para solventar estos problemas. El cual solo se puede utilizar cuando el programa puede compilarse. Permite análisis en ejecución, suspender, examinar y establecer valores de las variables, etc.

4.1. Puntos de ruptura

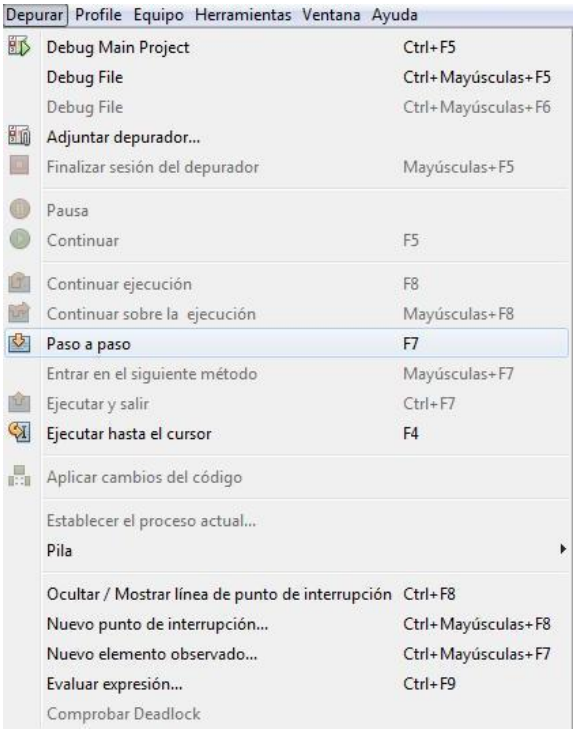
Dentro del menú de depuración existe la opción de insertar puntos de ruptura (breakpoints).

Los puntos de ruptura son marcadores que se pueden establecer en cualquier línea de código ejecutable. El programa se ejecuta hasta el punto de ruptura y a partir de ahí se pueden examinar variables y comprobar que los valores asignados son correctos, o se puede iniciar la depuración paso a paso. Una vez realizada la comprobación podemos abortar o continuar la ejecución normal del mismo. Se insertan y se quitan puntos de ruptura con la misma facilidad.

4.3. Tipos de ejecución

Se puede ejecutar el programa de diferentes formas en función del problema que queramos solucionar. Los distintos métodos de ejecución se utilizarán según las necesidades de depuración en cada momento:

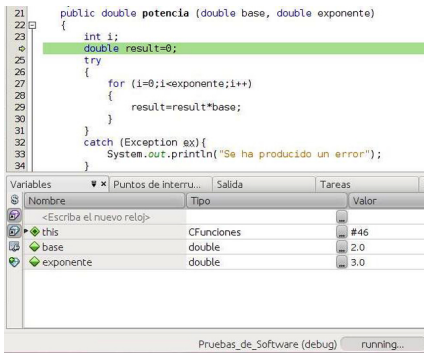
- **Paso a paso**. Ayuda a verificar que el código de un método se ejecuta de manera correcta.
- **Paso a paso por procedimientos,** nos permite introducir los parámetros que queramos a un método o función de nuestro programa, pero no ejecuta el método paso a paso, sino devuelve su resultado. Útil si ya hemos comprobado ese procedimiento y solo nos interesa su valor.
- **Ejecución hasta una instrucción.** Se ejecuta el programa y se detiene la instrucción en donde se encuentra el cursor y, a partir de ahí podemos depurar paso a paso o por procedimiento.



- **Ejecución de un programa hasta el final.** No se detiene en instrucciones intermedias.

4.3. Examinador de variables

Una de las maneras más comunes de comprobar que la aplicación funciona adecuadamente es comprobar que las variables van tomando los valores adecuados. Los entornos de desarrollo suelen traer examinadores de variables (En NetBeans es Ventana de inspección).



Los examinadores de variables sirven para comprobar los distintos valores que adquieren las variables, así como su tipo, en el proceso de depuración, por ejemplo en el paso a paso.

5. Validaciones.

En el proceso de validación interviene de manera decisiva el cliente. Ellos son quienes deciden si la aplicación se ajusta a sus requerimientos.

La validación de software se consigue mediante una serie de pruebas de caja negra, que de muestran la conformidad de los requisitos.

Un **plan de prueba** traza la clase de pruebas que se han de llevar a cabo y un **procedimiento de prueba** define los casos de prueba específicos en un intento por descubrir errores de acuerdo con los requisitos. Ambos estarán diseñados para asegurar que se satisfacen los requisitos funcionales, que se alcanzan los requisitos de rendimiento, documentaciones correctas e inteligibles y otros requisitos como portabilidad, recuperación de errores, facilidad de mantenimiento, etc...

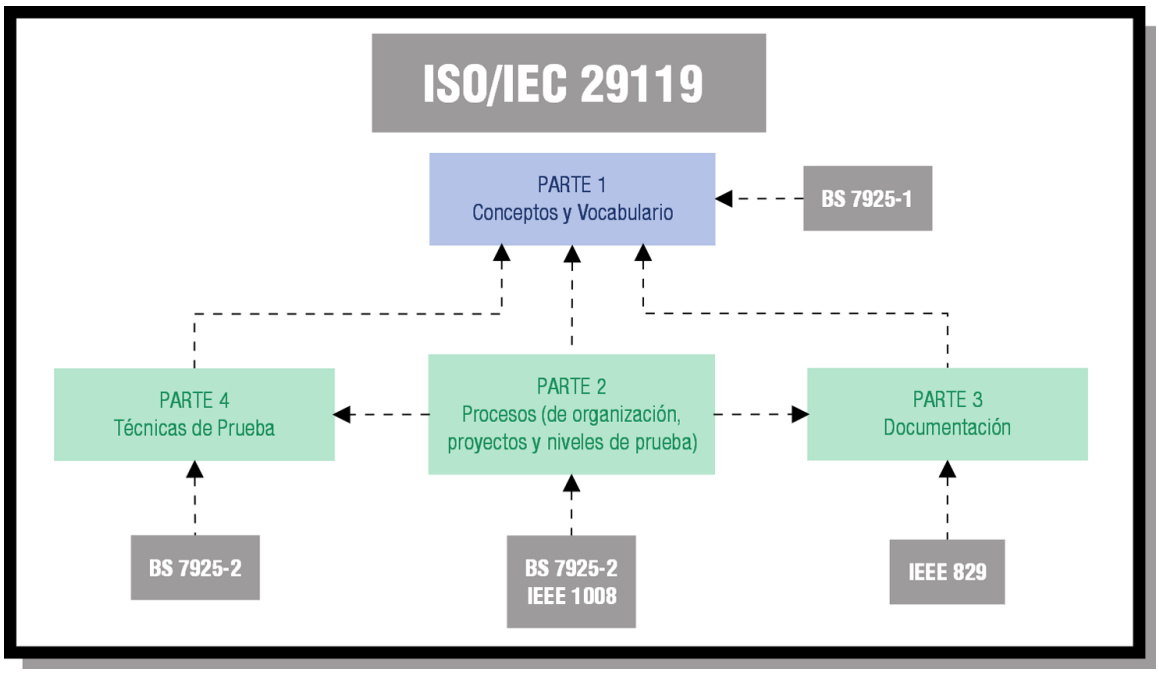
6. Normas de calidad

Los estándares que se han venido utilizando en la fase de prueba de software son:

- Metodología Métrica V3
- Etándares BSI
- Metedología **Métrica v3.**
- 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 de pruebas de software.:

  IEEE
  - IEEE estándar 829, Documentación de la prueba de software.
  - IEEE estándar 1008, Pruebas de unidad
  - Otros estándares **ISO /** 12207, 15289

    IEC
  - Otros estándares sectoriales



Pero estos estándares no cubren determinadas facetas de la fase de pruebas, como la organización, el proceso y la gestión de las pruebas. Ante ello la industria ha desarrollado la norma ISO/IEC 29119, que pretende unificar en una única forma todos los estándares, de forma que proporcione vocabulario, procesos, documentación y técnicas para cubrir todo el ciclo de vida del software (estrategias de prueba para organización y políticas de prueba, prueba de proyecto, análisis de casos de prueba, diseño, ejecución e informe).

ISO/IEC 29119

se compone de las siguientes partes:

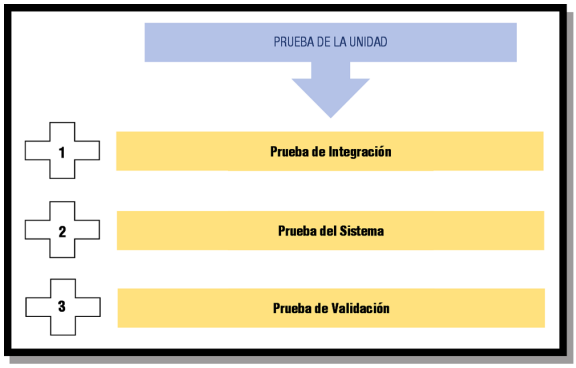
- **Parte 1**. Conceptos y vocabulario.
  - Introducción a la prueba.
  - Pruebas basadas en riesgo.
  - Fases de prueba (unidad, integración, sistema, validación) y tipos de prueba (estática, dinámica, no funcional, ...).
  - Prueba en diferentes ciclos de vida del software.
  - Roles y responsabilidades 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) .

7. Pruebas unitarias

Una unidad es la parte de la parte de la aplicación más pequeña para probar.

Una unidad puede ser una función o un procedimiento en programación procedural, en POO, normalmente es un método.



Las pruebas unitarias o de unidad prueban el correcto funcionamiento de un módulo o código. Cada prueba es independiente al resto.

Los entornos de desarrollo integran frameworks que permiten automatizar pruebas.

En el diseño de casos de pruebas unitarias hay que tener en cuenta los siguientes requisitos:

- Automatizable: sin intervención manual.
- Completas: cubrir la mayor cantidad de código.

- Repetibles o reutilizables: que puedan ejecutarse más de una vez.
- Independientes: la ejecución de una prueba no debe afectar a la ejecución de otra
- Profesionales: las pruebas deben ser consideradas con la misma profesionalidad que el código o la documentación.

Las ventajas que proporcionan las pruebas unitarias son:

- Fomentar el cambio. Facilitan que el programador cambie el código para mejorar su estructura.
- Simplifica la integración: permite llegar a la fase de integración con un grado alto de seguridad.
- Documenta el código. Ya que gracias a ellas se puede ver como se utiliza.
- Separación de interfaz e implementación. Dado que la única interacción entre los casos de prueba y las unidades bajo prueba son las interfaces de estas últimas, se puede cambiar cualquiera de los dos sin afectar al otro.
- Errores acotados y fáciles de localizar.

7.1. Herramientas para gráfica

Las más destacadas son:

- **Jtiger:**
  - Pruebas unitarias.
  - Código abierto.
  - Exporta informes en HTML, XML o texto plano.
  - Ejecuta casos de prueba de Junit mediante plugin.
  - Gran variedad de aserciones como la comprobación de cumplimiento del contrato de un método.
  - Los metadatos en los casos de prueba son especificados como anotaciones del lenguaje Java.
  - Incluye una tarea de Ant para automatizar las pruebas.
  - Documentación muy completa en JavaDOc y una web con la info necesaria para compender su uso y utilizarlo en IDE como Eclipse.
  - Incluye pruebas unitarias sobre sí mismo.

- **TestNG**
  - Inspirado en JUnit y NUnit.
  - Diseñado para cubir todo tipo de pruebas: unitarias, funcionales, integración
  - Anotaciones de Java 1.5.
  - Compatible con pruebas Junit.
  - Soporte para el paso de parámetros a los métodos de pruebas.
  - Permite distribución de pruebas en máquinas esclavas
  - Soportado por gran variedad de plugins (Eclipse, NetBeans, IDEA...)
  - Las clases de pruebas no necesitan implementar ninguna inferfaz ni extender ninguna otra clase.
  - Una vez compiladas las pruebas estas se pueden invocar desde la línea de comandos con una tarea de Ant o con un fichero XML.
  - Los métodos de prueba se organizan en grupos.
- **Junit.**

Sirve para NetBeans y Eclipse, es una herramienta de automatización de pruebas, para elaborarlas de manera rápida y sencilla. Nos permite diseñar clases de prueba para cada clase diseñada en nuestra aplicación. Una vez creadas, establecemos los métodos que queremos probar y diseñamos casos de prueba. Los criterios de creación de casos de prueba pueden ser muy diversos y dependen de lo que se quiera probar.

Una vez diseñados los casos de prueba, probamos la aplicación, en este caso, nos presenta un informe con los resultados de la prueba y en función de los resultados modificamos o no el código.

- Framework de pruebas unitarias creado por Erich Gamma y Kent Beck.
- Es una herramienta de código abierto.
- Multitud de documentación y ejemplos en la web.
- Se ha convertido en el estándar de hecho para las pruebas unitarias en Java.
- Soportado por la mayoría de los IDE como eclipse o Netbeans.

- Es una implementación de la arquitectura xUnit para los frameworks de pruebas unitarias.
- Posee una comunidad mucho mayor que el resto de los frameworks de pruebas en Java.
- Soporta múltiples tipos de aserciones.
- Desde la versión 4 utiliza las anotaciones del JDK 1.5 de Java.
- Posibilidad de crear informes en HTML.
- Organización de las pruebas en Suites de pruebas.
- Es la herramienta de pruebas más extendida para el lenguaje Java.
- Los entornos de desarrollo para Java, NetBeans y Eclipse, incorporan un plugin para Junit.

7.2. Herramientas para otros lenguajes

CppUnit

- Framework de pruebas unitarias para el lenguaje C++.
- Es una herramienta libre.
- Existe diversos entornos gráficos para la ejecución de pruebas como QtTestRunner.
- Es posible integrarlo con múltiples entornos de desarrollo como Eclipse.
- Basado en el diseño de xUnit.

Nunit

- Framework de pruebas unitarias para la plataforma .NET
- Es una herramienta de código abierto.
- También está basado en xUnit.
- Dispone de diversas expansiones como Nunit.Forms o Nunit.ASP

SimpleTest: Entorno de pruebas para aplicaciones realizadas en PHP.

PHPUnit: framework para realizar pruebas unitarias en PHP.

FoxUnit: framework OpenSource de pruebas unitarias para Microsoft Visual FoxPro

MOQ: Framework para la creación dinámica de objetos simuladores (mocks).

8. Documentación de la prueba

La documentación de las pruebas es un requisito indispensable para su correcta realización. Las metodologías actuales como Métrica V.3 proponen que la documentación se base en estándares ANSI/IEEE sobre verificación y validación de software.

El propósito de estos estándares es describir un conjunto de documentos para las pruebas de software. Este documento puede facilitar la comunicación entre desarrolladores al suministrar un marco de referencia común.

Los documentos a generar son:

- **Plan de Pruebas**: Al principio se desarrollará una planificación general, que quedará reflejada en el "Plan de Pruebas". El plan de pruebas se inicia el proceso de Análisis del Sistema.
- **Especificación del diseño de pruebas.** De la ampliación y detalle del plan de pruebas, surge el documento "Especificación del diseño de pruebas".
- **Especificación de un caso de prueba**. Los casos de prueba se concretan a partir de la especificación del diseño de pruebas.
- **Especificación de procedimiento de prueba**. Una vez especificado un caso de prueba, será preciso detallar el modo en que van a ser ejecutados cada uno de los casos de prueba, siendo recogido en el documento "Especificación del procedimiento de prueba".
- **Registro de pruebas**. En el "Registro de pruebas" se registrarán los sucesos que tengan lugar durante las pruebas.
- **Informe de incidente de pruebas**. Para cada incidente, defecto detectado, solicitud de mejora, etc, se elaborará un "informe de incidente de pruebas".
- **Informe sumario de pruebas**. Finalmente un "Informe sumario de pruebas" resumirá las actividades de prueba vinculadas a uno o más especificaciones de diseño de pruebas.