UT 3: Diseño y realización de pruebas

BIBLIOGRAFÍA:

UNIDAD 3: (TEMA DISPONIBLES EN MOODLE)

•Cabrera, G,Montoya, G : Análisis y Diseño Detallado de Aplicaciones Informáticas de Gestión. Mc Graw-Hill.
•Piattini, Calvo, Cervera, Fernández. Análisis y Diseño Detallado de Aplicaciones Informáticas de Gestión. Ra-Ma
•Pressman R.S. Ingeniería del Software. Un Enfoque Práctico. McGraw-Hill
•Sommerville,I. Ingeniería de Software.(6ª Edición),Addison Wesley.
•Larman C. UML y Patrones. (2ª Edición), Pearson Prentice Hall.
•Carlos Casado Iglesias: Entornos de Desarrollo. RA-MA
•Juan Carlos Moreno Pérez: Entornos de Desarrollo. Editorial SINTESIS
•RAMOS MARTÍN, Alicia (2014). Entornos de desarrollo. Madrid: Editorial Garceta.
•Alicia Ramos Martín, Mª Jesús Ramos Martín: Entornos de Desarrollo. Garceta grupo editorial
•Juan Carlos Moreno Pérez. Entornos de Desarrollo. Editorial SINTESIS
•Carlos Casado Iglesias. Entornos de Desarrollo. Ra-Ma

Profesora: Elena Fernández Chirino

DEPARTAMENTO DE INFORMÁTICA

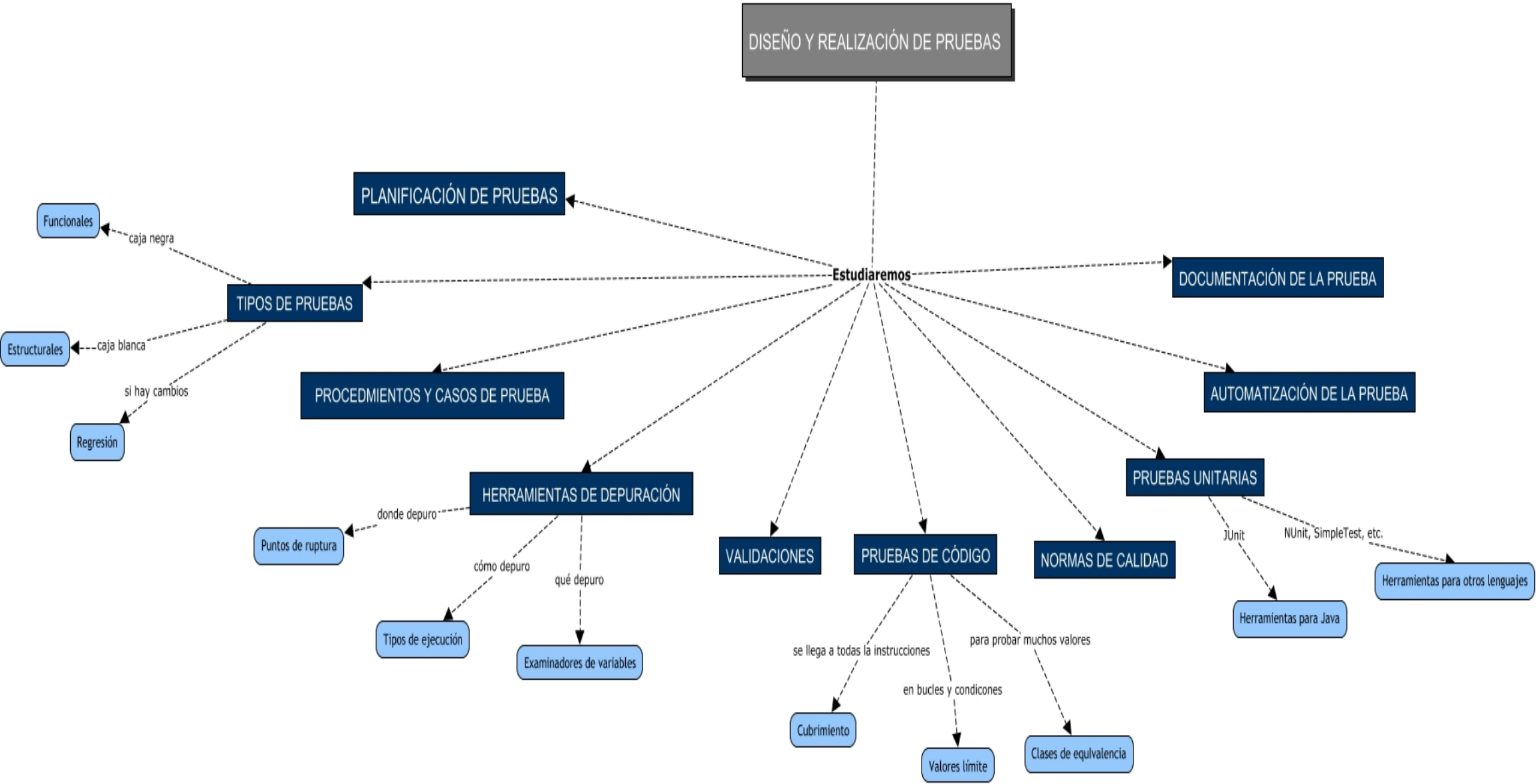
http://www.iestrassierra.com

RESULTADOS DE APRENDIZAJE

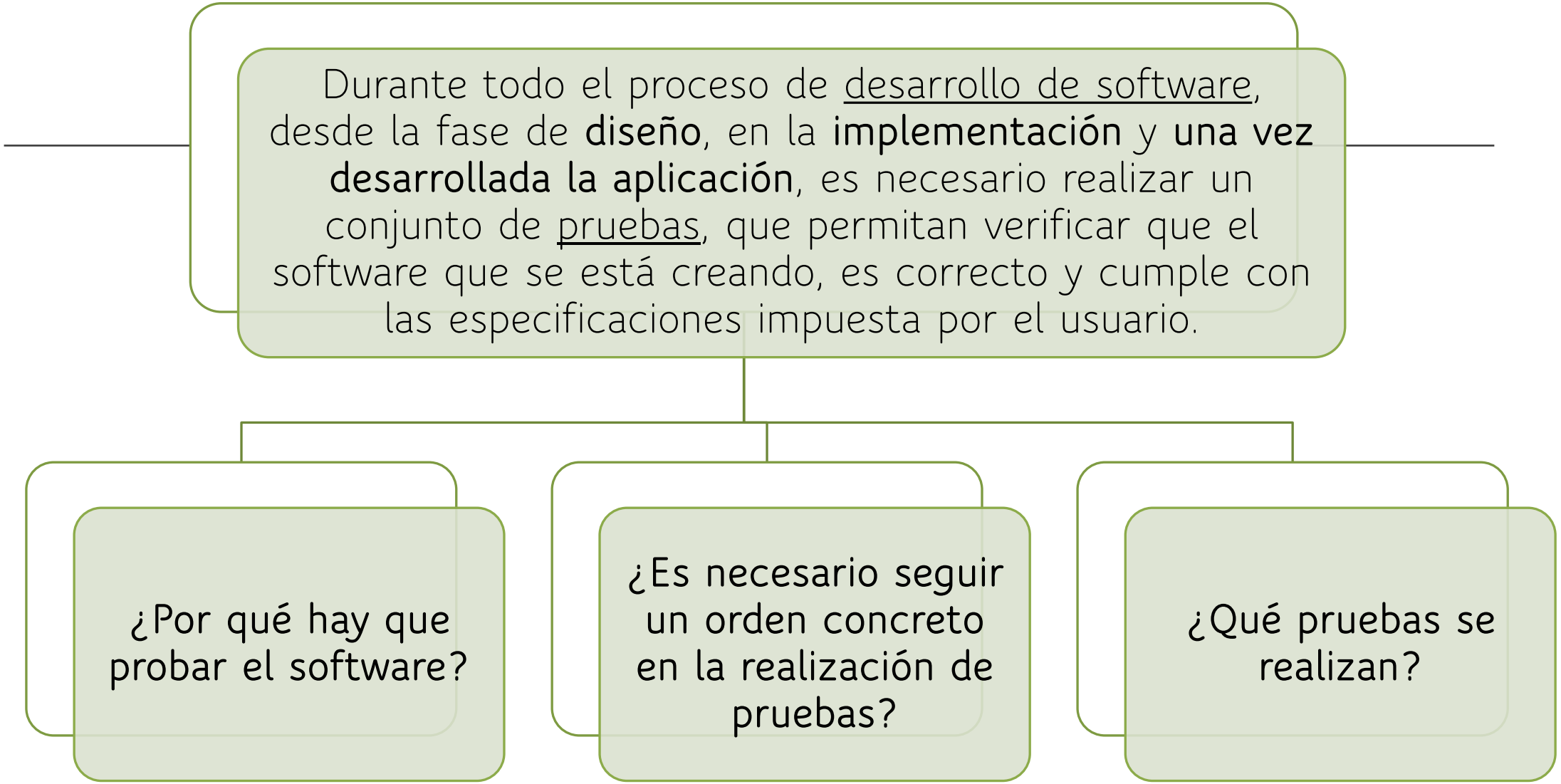
Al finalizar esta unidad valoraremos que el alumno:

- R.A.3: Verifica el funcionamiento de programas diseñando y realizando pruebas.
- Diseño y realización de pruebas.

0.- ¿Qué aprenderemos en esta unidad?



1.- Planificación de las pruebas



1.- Planificación de las pruebas

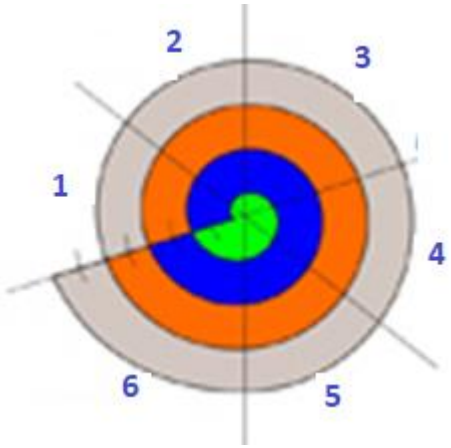
Mediante la realización de **pruebas** de software, se van a realizar las tareas de verificación y validación del software

La verificación

- **Comprobación** de que un sistema o parte de un sistema, **cumple con las condiciones impuestas**.
- **Comprobación** de que la aplicación se está **construyendo correctamente**.

La validación

- Proceso de **evaluación** del sistema o de uno de sus componentes, para determinar si **satisface los requisitos** especificados.



ESTRATEGIA DE PRUEBAS: la prueba, en el contexto de la ingeniería del software, realmente es una serie de cuatro pasos que se llevan a cabo secuencialmente.

La prueba nunca termina, ya que el responsable del desarrollo del software carga o pasa el problema al cliente.

- **Prueba de unidad**: se analiza el código implementado en la mínima unidad de diseño del software (el módulo)
- **Prueba de integración**: El objetivo es coger los módulos probados en unidad y construir una estructura de programa que esté de acuerdo con el diseño
- **Prueba de validación**: Se comprueba que el sistema construido cumple la ERS.
- **Prueba de alto nivel o de sistema**: Se verifica el funcionamiento total del software y otros elementos del sistema.

2.- Enfoque de pruebas

ENFOQUES FUNDAMENTALES DE LAS PRUEBAS DEL SOFTWARE

PRUEBA DE CAJA NEGRA

Lo fundamental es comprobar que los **resultados de la ejecución** de la aplicación, son los esperados, **en función de las entradas que recib**e.

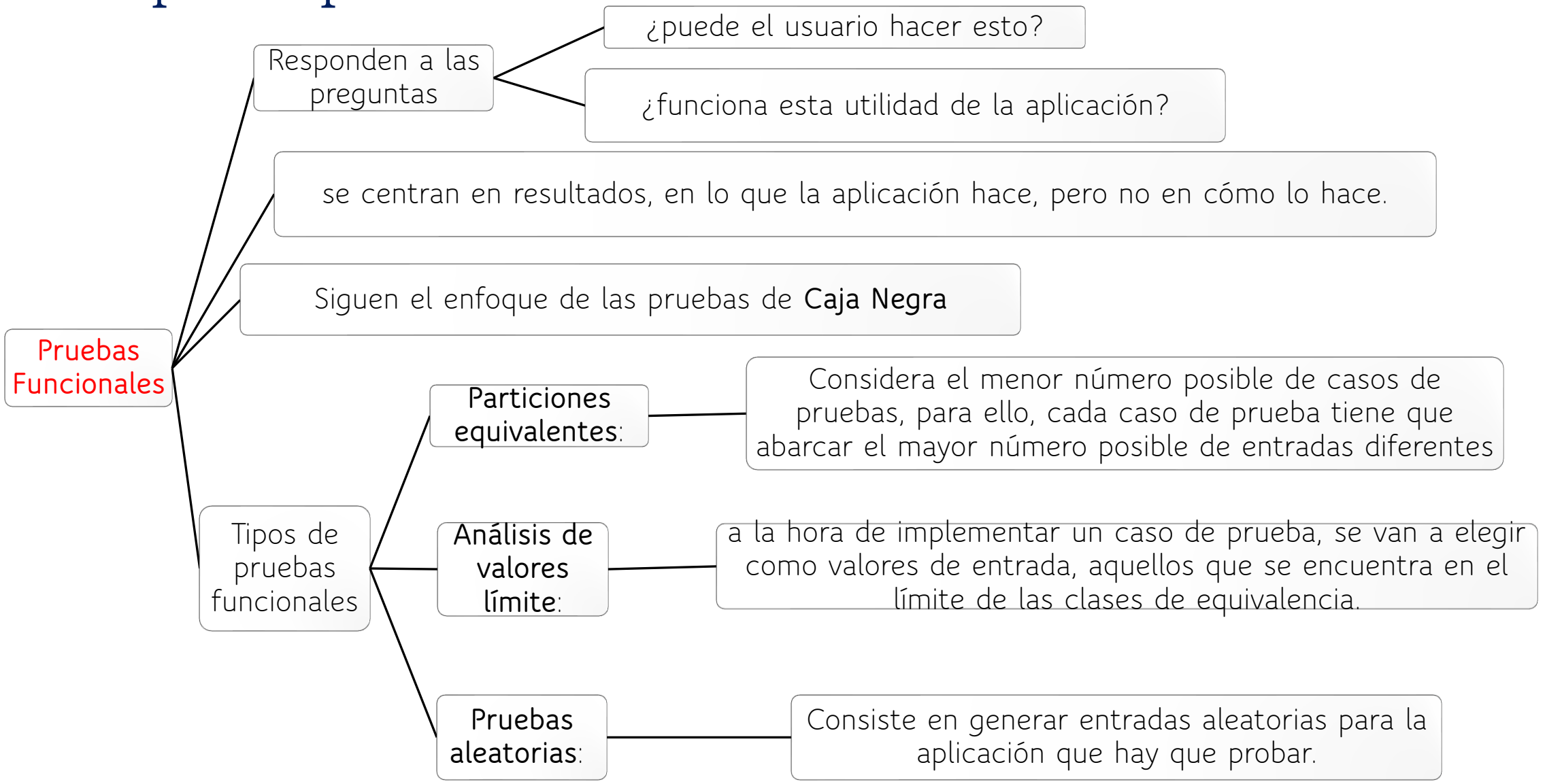
Solo se conocen las **entradas** adecuadas que deberá recibir la aplicación, así como las **salidas** que les correspondan, pero **no se conoce el proceso** mediante el cual la aplicación obtiene esos resultados.

PRUEBA DE CAJA BLANCA

En este caso, se prueba la aplicación desde dentro, usando su lógica de aplicación.

Se analiza y prueba directamente el código de la aplicación. Es necesario un **conocimiento específico del código**.

3.- Tipos de pruebas



3.- Tipos de pruebas

Pruebas Estructurales

- Se utilizan para
  - Ver cómo el programa se va ejecutando
  - Y así comprobar su corrección
- No pretenden comprobar la corrección de los resultados producidos por los distintos componentes, su función es comprobar que se van a ejecutar todas la instrucciones del programa, que no hay código no usado, comprobar que los caminos lógicos del programa se van a recorrer
- Siguen el enfoque de las pruebas de **Caja Blanca**
- se basan en unos **criterios de cobertura lógica**, cuyo cumplimiento determina la mayor o menor seguridad en la detección de errores
  - **Cobertura de sentencias**: se han de generar casos de pruebas suficientes para que cada instrucción del programa sea ejecutada, al menos, una vez.
  - **Cobertura de decisiones**: se trata de crear los suficientes casos de prueba 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**: se trata de crear los suficientes casos de prueba para que cada elemento de una condición, se evalúe al menos una vez a falso y otra a verdadero.
  - **Cobertura de condiciones y decisiones**: consiste en cumplir simultáneamente las dos anteriores.
  - **Cobertura de caminos**: es el criterio más importante. Establece que se debe ejecutar al menos una vez cada secuencia de sentencias encadenadas, desde la sentencia inicial del programa, hasta su sentencia final.
  - **Cobertura del camino de prueba**: Se pueden realizar dos variantes, una indica que cada bucle se debe ejecutar sólo una vez y otra que recomienda que se pruebe cada bucle tres veces: la primera sin entrar en su interior, otra ejecutándolo una vez y otra más ejecutándolo dos veces.

3.- Tipos de pruebas

Regresión

Si en las pruebas detectamos un error, debemos hacer una corrección. La modificación realizada nos obliga a repetir pruebas que hemos realizado con anterioridad.

Este tipo de pruebas implica la **repetición de las pruebas que ya se hayan realizado previamente**, con el fin de asegurar que no se introducen errores que puedan comprometer el funcionamiento de otros componentes que no han sido modificados y confirmar que el sistema funciona correctamente una vez realizados los cambios.

El conjunto de pruebas de regresión contiene tres clases diferentes de clases de prueba:

- Una muestra representativa de pruebas que ejercite 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 del software que han cambiado.

Para evitar que el número de pruebas de regresión crezca demasiado, se deben de diseñar para incluir sólo aquellas pruebas que traten una o más clases de errores en cada una de las funciones principales del programa. No es práctico ni eficiente volver a ejecutar cada prueba de cada función del programa después de un cambio.

4.- Herramientas de depuración

Durante el proceso de desarrollo de software, se pueden producir **dos tipos de errores:**

Errores de compilación

Cuando se desarrolla una aplicación en un IDE, ya sea, Eclipse o Netbeans, si al escribir una sentencia, olvidamos un ";", hacemos referencia a una variable inexistente o utilizamos una sentencia incorrecta, se produce un **error de compilación**

Cuando ocurre un error de compilación, el entorno nos proporciona información de donde se produce y como poder solucionarlo.

Errores lógicos (bugs)

Estos no evitan que el programa se pueda compilar con éxito. Sin embargo **pueden provocar que el programa devuelva resultados erróneos**, que no sean los esperados o pueden provocar que el programa termine antes de tiempo o no termine nunca.

Para solucionar este tipo de problemas, los entornos de desarrollo incorporan una herramienta conocida como DEPURADOR

- Permite supervisar la ejecución de los programas, para localizar y eliminar los errores lógicos
- Permite analizar todo el programa, mientras éste se ejecuta
- Permite suspender la ejecución de un 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, etc.

4.1.- Puntos de ruptura

Dentro del menú de depuración, nos encontramos con la **opción insertar punto de ruptura (breakpoint)**. Se selecciona la línea de código donde queremos que el programa se pare, para a partir de ella, inspeccionar variables, o realizar una ejecución paso a paso, para verificar la corrección del código

- Los **puntos de ruptura son marcadores** que pueden establecerse en cualquier línea de código ejecutable (no sería válido un comentario, o una línea en blanco).

- Una vez insertado el punto de ruptura, e iniciada la depuración, el programa a evaluar se ejecutaría hasta la línea marcada con el punto de ruptura.

- En ese momento, se pueden realizar **diferentes labores**,
  - por un lado, se pueden examinar las variables, y comprobar que los valores que tienen asignados son correctos,
  - o se pueden iniciar una depuración paso a paso, e ir comprobando el camino que toma el programa a partir del punto de ruptura.

- Una vez realiza la comprobación, podemos abortar el programa, o continuar la ejecución normal del mismo.

- Dentro de una aplicación, se pueden insertar varios puntos de ruptura, y se pueden eliminar con la misma facilidad con la que se insertan.

4.2.- Tipos de Ejecución

Para poder depurar un programa, podemos ejecutar el programa de diferentes formas, de manera que en función del problema que queramos solucionar, nos resulte más sencillo un método u otro. Nos encontramos con lo siguientes tipo de ejecución:

- **Paso a paso por instrucción:** Algunas veces es necesario ejecutar un programa línea por línea, para buscar y corregir errores lógicos

- **Paso a paso por procedimiento**: nos permite introducir los parámetro que queremos a un método o función de nuestro programa, pero en vez de ejecutar instrucción por instrucción ese método, nos devuelve su resultado. Es útil, cuando hemos comprobado que un procedimiento funciona correctamente, y no nos interese volver a depurarlo, sólo nos interesa el valor que devuelve.

- **Ejecución hasta una instrucción**: el depurador ejecuta el programa, y se detiene en la instrucción donde se encuentra el cursor, a partir de ese punto, podemos hacer una depuración paso a paso o por procedimiento.

- **Ejecución de un programa hasta el final del programa**: ejecutamos las instrucciones de un programa hasta el final, sin detenernos en las instrucciones intermedias.

.

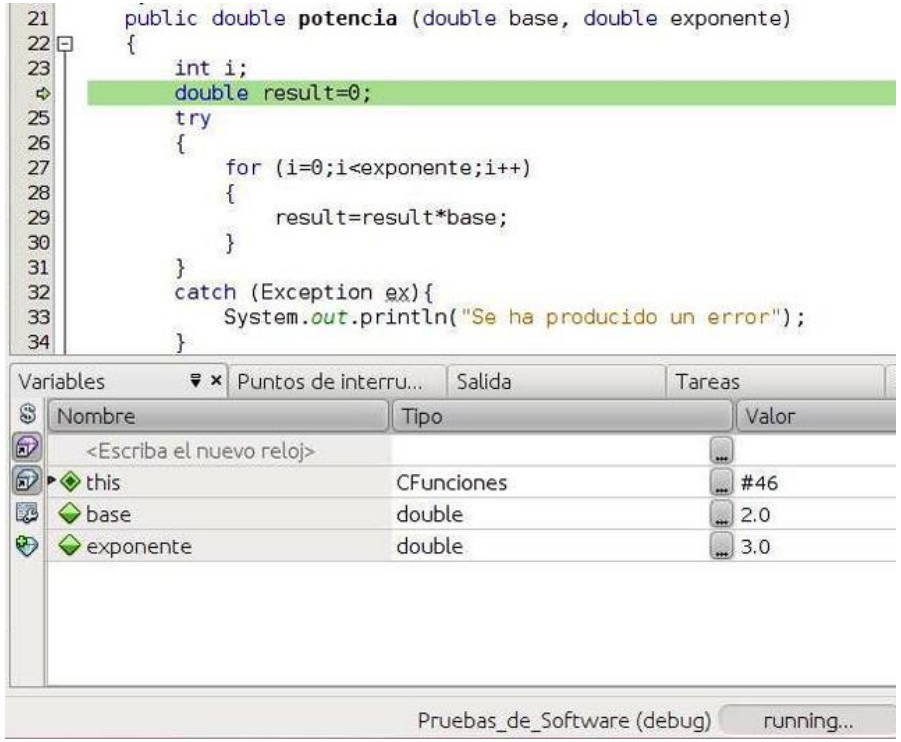
En el IDE NetBeans, dentro del menú de depuración, podemos seleccionar los modos de ejecución especificados, y algunos más. El objetivo es poder examinar todas la partes que se consideren necesarias, de manera rápida, sencilla y los más clara posible.

4.3.- Examinador de Variables

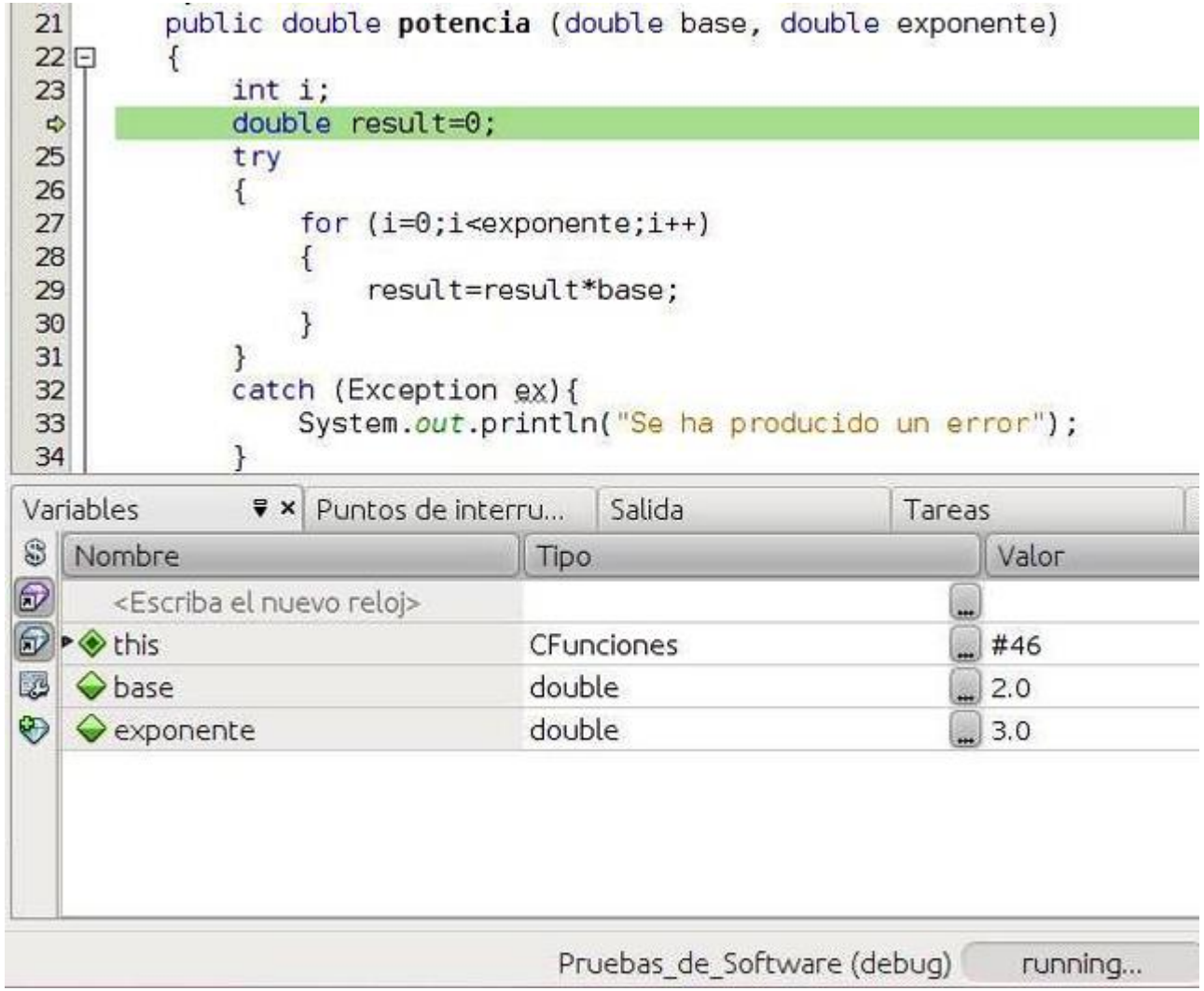
Durante el proceso de implementación y prueba de software, una de las maneras más comunes de comprobar que la aplicación funciona de manera adecuada, es comprobar que las variables vayan tomando los valores adecuados en cada momento.

Con los **examinadores de variables**, podemos comprobar los distintos valores que adquiere las variables, así como su tipo. Esta herramienta es de gran utilidad para la detección de errores.

En el caso del entorno de desarrollo NetBeans, nos encontramos con un panel llamado **Ventana de Inspección**. En la ventana de inspección, se pueden ir agregando todas aquellas variables de las que tengamos interés en inspeccionar su valor. Conforme el programa se vaya ejecutando, NetBeans irá mostrando los valores que toman las variables en al ventana de inspección.



4.3.- Ejemplo de examinador de Variables de Netbeans



- Como podemos apreciar, en una ejecución paso a paso, el programa llega a una **función de nombre potencia**.
- Esta función tiene definida tres variables: base, exponente y result
- A lo largo de la ejecución del bucle, vemos como la variable **result**, va cambiando de valor.
- Si con valores de entrada para los que conocemos el resultado, la función no devuelve el valor esperado, "Examinando las variables" podremos encontrar la instrucción incorrecta.

5.- Validaciones

En el proceso de validación, interviene de manera decisiva el cliente.

En la validación intentan descubrir errores, pero desde el punto de vista de los requisitos.

La validación del software se consigue mediante una serie de **pruebas de caja negra** que demuestran la **conformidad con los requisitos**.

- Un **plan de prueba** traza la clase de pruebas que se han de llevar a cabo.
- Un **procedimiento de prueba** define los casos de prueba específicos en un intento por descubrir errores de acuerdo con los requisitos.

Tanto el plan como el procedimiento estarán diseñados para asegurar que:

- se **satisfacen todos los requisitos funcionales**
- se alcanzan todos los **requisitos de rendimiento**
- las documentaciones son correctas e inteligible
- se alcanzan otros requisitos, como portabilidad, compatibilidad, recuperación de errores, facilidad de mantenimiento etc.

Cuando se procede con cada caso de prueba de validación, puede darse una de las dos condiciones siguientes:

- Las características de funcionamiento o rendimiento están de acuerdo con las especificaciones y son aceptables o
- Se descubre una desviación de las especificaciones y se crea una lista de deficiencias. Las desviaciones o errores descubiertos en esta fase del proyecto raramente se pueden corregir antes de la terminación planificada.

6.- Pruebas de Código

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

Para llevar a cabo las pruebas, es necesario definir una serie de **casos de prueba**, que van a ser un conjunto de entradas, de condiciones de ejecución y resultados esperados, desarrollados para un objetivo particular.

Para el diseño de casos de prueba, se suelen utilizar **tres enfoques principales**:

Enfoque estructural o de caja blanca. Este enfoque se centra en la estructura interna del programa, analizando los caminos de ejecución. Dentro de nuestro proceso de prueba, lo aplicamos con el cubrimiento.

Enfoque funcional o de caja negra. Este enfoque se centra en la funciones, entradas y salidas. Se aplican los valores límite y las clases de equivalencia..

Enfoque aleatorio, que consiste en utilizar modelos que represente las posibles entradas al programa, para crear a partir de ellos los casos de prueba. En esta pruebas se intenta simular la entrada habitual que va a recibir el programa, para ello se crean datos entrada en la secuencia y con la frecuencia en que podrían aparecer. Para ello se utilizan generadores automáticos de casos de prueba.

6.1.- Cubrimiento

Por ejemplo: Imagen que muestra el código de una **función**, de nombre **prueb**a, implementada en Java y que recibe dos argumentos. Considerando que esta función forma parte de un programa mayor, se considera lo siguiente:

- 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 este 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.

Esta tarea la realiza el programador o programadora y consiste en comprobar que los caminos definidos en el código, se pueden llegar a recorrer. (CAJA BLANCA)

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

6.2.- Valor límite

```
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;
}
```

En el código Java adjunto, aparecen dos funciones que reciben el parámetro x. En la función1, el parámetro es de tipo real y en la función2, el parámetro es de tipo entero. Como se aprecia, el código de las dos funciones es el mismo, sin embargo, los casos de prueba con valores límite va a ser diferente.

Esta técnica, se suele utilizar como complementaria de las particiones equivalentes, pero se diferencia, en que se suelen seleccionar, no un conjunto de valores, sino unos pocos, en el límite del rango de valores aceptado por el componente a probar.

Cuando hay que seleccionar una valor para realizar una prueba, se escoge aquellos que están situados justo en el límite de los valores admitidos.

Por ejemplo, supongamos que queremos probar el resultado de la ejecución de una función, que recibe un parámetro x:

Si el parámetro x de entrada tiene que ser mayor estricto que 5, y el valor es real, los valores límite pueden ser 4,99 y 5,01.
Si el parámetro de entrada x está comprendido entre -4 y +4, suponiendo que son valores enteros, los valores límite serán -5, -4 ,-3,3, 4 y 5.

6.3.- Clases de equivalencia

El dominio de valores de entrada, se divide en número finito de clases de equivalencia. Como la entrada está dividida en un conjunto de clases de equivalencia, la prueba de un valor representativo de cada clase, permite suponer que el resultado que se obtiene con él, será el mismo que con cualquier otro valor de la clase.

Cada clase de equivalencia debe cumplir:

- Si un parámetro de entrada debe estar comprendido entre un determinado rango, hay tres clases de equivalencia: por debajo, en y por encima.
- Si una entrada requiere un valor entre los de un conjunto, aparecen dos clase de equivalencia: en el conjunto o fuera de él.
- Si una entrada es booleana, hay dos clases: sí o no.

Los mismos criterios se aplican a las salidas esperadas: hay que intentar generar resultados en todas y cada una de las clases.

Las clases de equivalencia, es un tipo de prueba funcional, en donde cada caso de prueba, pretende cubrir el mayor número de entradas posible.

En este ejemplo, las clase de equivalencia serían:

Captura de una función Java, que nos sirve para explicar las clases de equivalencia.

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

Por debajo: x=0
En: x=50
Por encima: x=100

Por debajo: x<0
En: x>0 y x< 100
Por encima: x>100
y los respectivos casos de prueba, podrían ser:

7.- Normas de Calidad

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

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

- Otros estándares sectoriales

Anexo I.- Norma ISO/IEC 29119.

La norma ISO/IEC 29119 se compone la 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) .