

TEMA 4

Contenido

1. - Planificación de las pruebas.	3
2. - Tipos de prueba.	4
2.1. - Funcionales.	5
2.2. - Estructurales.	6
2.3. - Regresión.	7
3. - Procedimientos y casos de prueba.	9
4. - Herramientas de depuración.	10
4.1. - Puntos de ruptura.	10
4.2. - Tipos de ejecución.	11
4.3. - Examinadores de variables.	11
INICIO DE LA DEPURACIÓN.	12
LA DEPURACIÓN.	12
5. - Validaciones.	15
6. - Pruebas de código.	16
6.1. - Cubrimiento.	16
6.2. - Valores límite.	17
6.3. - Clases de equivalencia.	18
7. - Normas de calidad.	19
8. - Pruebas unitarias.	21
8.1. - Herramientas para Java.	22
8.2. - Herramientas para otros lenguajes.	23
9. - Automatización de la prueba.	24
Uso de JUNIT en NetBeans.	24
INTRODUCCIÓN.	24
INICIO DE JUNIT.	24
CASOS DE PRUEBA.	25
10. - Documentación de la prueba.	28
Anexo I.- Norma ISO/IEC 29119.	29

[DISEÑO Y REALIZACIÓN DE PRUEBAS]

Blanca Calderón --- 2023/2024

Entornos de Desarrollo del curso de “Desarrollo de Aplicaciones Multimedia”

DISEÑO Y REALIZACIÓN DE PRUEBAS.

1.- Planificación de las pruebas.

Durante todo el proceso de desarrollo de software, desde la fase de diseño, en la implementación y una vez desarrollada la aplicación, es necesario realizar un conjunto de pruebas (*Proceso que permite verificar y revelar la calidad de un producto software. Se utilizan para identificar posibles fallos de implementación, calidad o usabilidad de un programa*), que permitan verificar que el software que se está creando, es correcto y cumple con las especificaciones impuesta por el usuario.

En el proceso de desarrollo de software, nos vamos a encontrar con un conjunto de actividades, donde es muy fácil que se produzca un error humano. Estos errores humanos pueden ser: una incorrecta especificación de los objetivos, errores producidos durante el proceso de diseño y errores que aparecen en la fase de desarrollo.

Mediante la realización de pruebas de software, se van a realizar las tareas de verificación (*Proceso por el que se comprueba que el software cumple los requisitos especificados*) y validación (*Proceso que comprueba si el software hace lo que el usuario deseaba. Tiene que estar verificado*) del software. La **verificación** es la comprobación que un sistema o parte de un sistema, cumple con las condiciones impuestas. Con la verificación se comprueba si la aplicación se está construyendo correctamente. La **validación** es el proceso de evaluación del sistema o de uno de sus componentes, para determinar si satisface los requisitos especificados.



Para llevar a cabo el proceso de pruebas, de manera eficiente, es necesario implementar una estrategia de pruebas. Siguiendo el Modelo en Espiral (*Modelo de ciclo de vida de ciclo de vida del software, en el que las actividades se conforman en una espiral. Cada bucle o iteración representa un conjunto de actividades*), las pruebas empezarían con la **prueba de unidad**, donde se analizaría el código implementado y seguiríamos en la **prueba de integración**, donde se prestan atención al diseño y la construcción de la arquitectura del software. El siguiente paso sería la **prueba de validación**, donde se comprueba que el sistema construido cumple con lo establecido en el análisis de requisitos de software. Finalmente se alcanza la **prueba de sistema** que verifica el funcionamiento total del software y otros elementos del sistema.

2.- Tipos de prueba.

No existe una clasificación oficial o formal, sobre los diversos tipos de pruebas de software. En la ingeniería del software, nos encontramos con dos enfoques fundamentales:

- **Prueba de la Caja Negra** (Black Box Testing): cuando una aplicación es probada usando su interfaz externa, sin preocuparnos de la implementación de la misma. Aquí lo fundamental es comprobar que los resultados de la ejecución de la aplicación, son los esperados, en función de las entradas que recibe.
- **Prueba de la Caja Blanca** (White Box Testing): en este caso, se prueba la aplicación desde dentro, usando su lógica de aplicación.



Una prueba de tipo **Caja Negra** se lleva a cabo sin tener que conocer ni la estructura, ni el funcionamiento interno del sistema. Cuando se realiza este tipo de pruebas, 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.

En contraposición a lo anterior, una prueba de **Caja Blanca**, va a analizar y probar directamente el código de la aplicación. Como se deriva de lo anterior, para llevar a cabo una prueba de Caja Blanca, es necesario un conocimiento específico del código, para poder analizar los resultados de las pruebas.

Tipos de pruebas de software	
Pruebas de unidad	Con ella se va a probar el correcto funcionamiento de un módulo de código
Pruebas de carga	<p>Este es el tipo más sencillo de pruebas de rendimiento. Una prueba de carga se realiza generalmente para observar el comportamiento de una aplicación bajo una cantidad de peticiones esperada.</p> <p>Esta carga puede ser el número esperado de usuarios concurrentes utilizando la aplicación y que realizan un número específico de transacciones durante el tiempo que dura la carga. Esta prueba puede mostrar los tiempos de respuesta de todas las transacciones importantes de la aplicación. Si la base de datos, el servidor de aplicaciones, etc también se monitorizan, entonces esta prueba puede mostrar el cuello de botella en la aplicación.</p>
Prueba de estrés	Esta prueba se utiliza normalmente para romper la aplicación. Se va doblando el número de usuarios que se agregan a la aplicación y se ejecuta una prueba de carga hasta que se rompe. Este tipo de prueba se realiza para determinar la solidez de la aplicación en los momentos de carga extrema y ayuda a los administradores para determinar si la aplicación rendirá lo suficiente en caso de que la carga real supere a la carga esperada.
Prueba de estabilidad	Esta prueba normalmente se hace para determinar si la aplicación puede aguantar una carga esperada continuada. Generalmente esta prueba se realiza para determinar si hay alguna fuga de memoria en la aplicación
Pruebas de picos	La prueba de picos, como el nombre sugiere, trata de observar el comportamiento del sistema variando el número de usuarios, tanto cuando bajan, como cuando tiene cambios drásticos en su carga.

	Esta prueba se recomienda que sea realizada con un software automatizado que permita realizar cambios en el número de usuarios mientras que los administradores llevan un registro de los valores a ser monitorizados
Prueba estructural	El enfoque estructural o de caja blanca se centra en la estructura interna del programa (analiza los caminos de ejecución).
Prueba funcional	El enfoque funcional o de caja negra se centra en las funciones, entradas y salidas que recibe y produce un módulo o función concreta.
Pruebas aleatorias	El enfoque aleatorio consiste en utilizar modelos (en muchas ocasiones estadísticos) que representen las posibles entradas al programa para crear a partir de ellos los casos de prueba.
Pruebas de regresión	Son cualquier tipo de pruebas de software que intentan descubrir las causas de nuevos errores, carencias de funcionalidad, o divergencias funcionales con respecto al comportamiento esperado del software, inducidos por cambios recientemente realizados en partes de la aplicación que anteriormente al citado cambio no eran propensas a este tipo de error. Esto implica que el error tratado se produce como consecuencia inesperada del citado cambio en el programa.

Reflexiona

Resulta habitual, que en una empresa de desarrollo de software se gaste el 40 por ciento del esfuerzo de desarrollo en la prueba ¿Por qué es tan importante la prueba? ¿Qué tipos de errores se intentan solucionar con las pruebas?

Las pruebas son muy importantes, ya que permiten descubrir errores en un programa, fallos en la implementación, calidad o usabilidad del software, ayudando a garantizar la calidad.

Con las pruebas se intenta verificar que cada componente que se ha diseñado, ya sea un método, función, módulo, etc. realiza la función para la que se ha diseñado. También se intenta comprobar, que existen condiciones en las que todos los caminos de una aplicación llegan a ejecutarse.

2.1. - Funcionales.

Estamos ante pruebas de la caja negra. Se trata de probar, si las salidas que devuelve la aplicación, o parte de ella, son las esperadas, en función de los parámetros de entrada que le pasemos. No nos interesa la implementación del software, solo si realiza las funciones que se esperan de él.

Las pruebas funcionales siguen el enfoque de las pruebas de Caja Negra. Comprenderían aquellas actividades cuyo objetivo sea verificar una acción específica o funcional dentro del código de una aplicación. Las pruebas funcionales intentarían responder a las preguntas ¿puede el usuario hacer esto? o ¿funciona esta utilidad de la aplicación?

Su principal cometido, va a consistir, en comprobar el correcto funcionamiento de los componentes de la aplicación informática. Para realizar este tipo de pruebas, se deben analizar las entradas y las salidas de cada componente, verificando que el resultado es el esperado. Solo se van a considerar las entradas y salidas del sistema, sin preocuparnos por la estructura interna del mismo.

Si, por ejemplo, estamos implementando una aplicación que realiza un determinado cálculo científico, en el enfoque de las pruebas funcionales, solo nos interesa verificar que ante una determinada entrada a ese programa el resultado de la ejecución del mismo devuelve como resultado los datos esperados. Este tipo de prueba, no consideraría, en ningún caso, el código desarrollado, ni el algoritmo (*Conjunto ordenado de pasos a seguir para la resolución de un problema*), ni la eficiencia, ni si hay partes del código innecesarias, etc.

Dentro de las pruebas funcionales, podemos indicar tres tipos de pruebas:

- **Particiones equivalentes:** La idea de este tipo de pruebas funcionales, es considerar el menor número posible de casos de pruebas, para ello, cada caso de prueba tiene que abarcar el mayor número posible de entradas diferentes. Lo que se pretende, es crear un conjunto de clases de equivalencia, donde la prueba de un valor representativo de la misma, en cuanto a la verificación de errores, sería extrapolable al que se conseguiría probando cualquier valor de la clase.
- **Análisis de valores límite:** En este caso, a la hora de implementar un caso de prueba, se van a elegir como valores de entrada, aquellos que se encuentra en el límite de las clases de equivalencia.
- **Pruebas aleatorias:** Consiste en generar entradas aleatorias para la aplicación que hay que probar. Se suelen utilizar generadores de prueba, que son capaces de crear un volumen de casos de prueba al azar, con los que será alimentada la aplicación. Este tipo de pruebas, se suelen utilizar en aplicaciones que no sean interactivas, ya que es muy difícil generar las secuencias de entrada adecuadas de prueba, para entornos interactivos.

Existen otros tipos de pruebas funcionales, aunque todas comparten un mismo objetivo, y es comprobar, solo actuando en la interfaz de la aplicación, que los resultados que produce son los correctos en función de las entradas que se le introducen para probarlos.

2.2. - Estructurales.

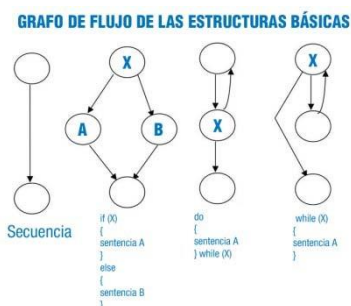
Ya hemos visto que las pruebas funcionales se centran en resultados, en lo que la aplicación hace, pero no en cómo lo hace.

Para ver cómo el programa se va ejecutando, y así comprobar su corrección, se utilizan las pruebas estructurales, que se fijan en los caminos que se pueden recorrer:

Las pruebas estructurales son el conjunto de pruebas de la Caja Blanca. Con este tipo de pruebas, se pretende verificar la estructura interna de cada componente de la aplicación, independientemente de la funcionalidad establecida para el mismo. Este tipo de pruebas, no pretenden comprobar la corrección de los resultados producidos por los distintos componentes, su función es comprobar que se van a ejecutar todas las instrucciones del programa, que no hay código no usado, comprobar que los caminos lógicos del programa se van a recorrer, etc.

Este tipo de pruebas, se basan en unos criterios de cobertura lógica, cuyo cumplimiento determina la mayor o menor seguridad en la detección de errores. Los criterios de cobertura que se siguen son:

- **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. La ejecución de este conjunto de sentencias, se conoce como camino. Como el número de caminos que puede tener una aplicación, puede ser muy



grande, para realizar esta prueba, se reduce el número a lo que se conoce como camino prueba.

- **Cobertura del camino de prueba:** Se pueden realizar dos variantes, una indica que cada bucle se debe ejecutar sólo una vez, ya que hacerlo más veces no aumenta la efectividad de la prueba 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.

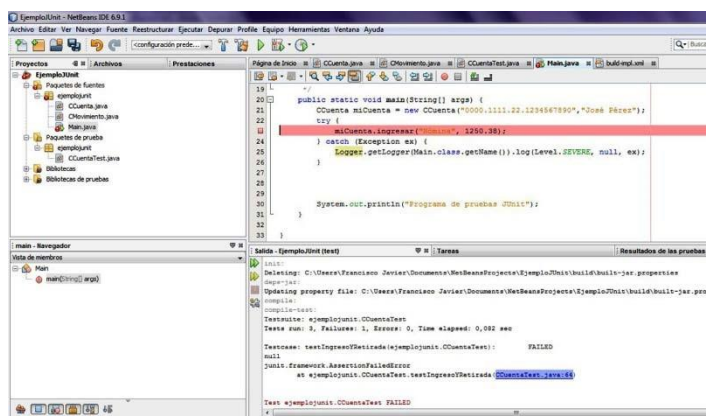
**PREGUNTA

En las pruebas de caja negra:

- Es necesario conocer el código fuente del programa para realizar las pruebas.
- Se comprueba que todos los caminos del programa se pueden recorrer al menos una vez.
- Se comprueba que los resultados de una aplicación son los esperados para las entradas que se le han proporcionado.
- Es incompatible con la prueba de caja blanca.

2.3.- Regresión.

Durante el proceso de prueba, tendremos éxito si detectamos un posible fallo o error. La consecuencia directa de ese descubrimiento, supone la modificación del componente donde se ha detectado. Esta modificación, puede generar errores colaterales, que no existían antes. Como consecuencia, la modificación realizada nos obliga a repetir pruebas que hemos realizado con anterioridad.



El objetivo de las pruebas de regresión, es comprobar que los cambios sobre un componente de una aplicación, no introduce un comportamiento no deseado o errores adicionales en otros componentes no modificados.

- ✓ Las pruebas de regresión se deben llevar a cabo cada vez que se hace un cambio en el sistema, tanto para corregir un error, como para realizar una mejora. No es suficiente probar sólo los componentes modificados o añadidos, o las funciones que en ellos se realizan, sino que también es necesario controlar que las modificaciones no produzcan efectos negativos sobre el mismo u otros componentes.

Normalmente, 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.

En un contexto más amplio, las pruebas de software con éxito, son aquellas que dan como resultado el descubrimiento de errores. Como consecuencia del descubrimiento de errores, se procede a su corrección, lo que implica la modificación de algún componente del software que se está desarrollando, tanto del programa, de la documentación y de los datos que lo soportan. La prueba de regresión es la que nos ayuda a asegurar que estos cambios no introducen un comportamiento no

deseado o errores adicionales. La prueba de regresión se puede hacer manualmente, volviendo a realizar un subconjunto de todos los casos de prueba o utilizando herramientas automáticas.

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.

****PREGUNTA**

La prueba de regresión:

- Se realiza una vez finalizado cada módulo (cada parte, con una funcionalidad concreta, en que se divide una aplicación) del sistema a desarrollar.
- Solo utiliza el enfoque de la caja negra.
- Se realiza cuando se produce una modificación, debido a la detección de algún error, en la fase de prueba.
- Es incompatible con la prueba de caja blanca.
-

3.- Procedimientos y casos de prueba.

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

Dada la complejidad de las aplicaciones informáticas, que se desarrollan en la actualidad, es prácticamente imposible, probar todas las combinaciones que se pueden dar dentro de un programa o entre un programa y las aplicaciones que pueden interactuar con él. Por este motivo, en el diseño de los casos de prueba, siempre es necesario asegurar que con ellos se obtiene un nivel aceptable de probabilidad de que se detectarán los errores existentes.

Las pruebas deben buscar un compromiso entre la cantidad de recursos que se consumirán en el proceso de prueba, y la probabilidad obtenida de que se detecten los errores existentes.

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

- ✓ **Enfoque funcional o de caja negra.** En este tipo de prueba, nos centramos en que el programa, o parte del programa que estamos probando, recibe una entrada de forma adecuada y se produce una salida correcta, así como que la integridad de la información externa se mantiene. La prueba no verifica el proceso, solo los resultados.

- ✓ **Enfoque estructural o caja blanca.** En este tipo de pruebas, debemos centrar en la implementación interna del programa. Se trata de comprobar que la operación interna se ajusta a las especificaciones. En esta prueba, se deberían de probar todos los caminos que puede seguir la ejecución del programa.
- ✓ **Enfoque aleatorio.** A partir de modelos obtenidos estadísticamente, se elaboran casos de prueba que prueben las entradas del programa.

4.- Herramientas de depuración.

Todo entorno de desarrollo, independientemente de la plataforma, así como del lenguaje de programación utilizado, suministra una serie de herramientas de depuración, que nos permiten verificar el código generado, ayudándonos a realizar pruebas tanto estructurales como funcionales.

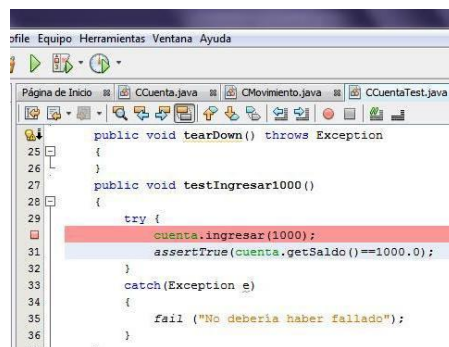
Durante el proceso de desarrollo de software, se pueden producir dos tipos de errores: errores de compilación o errores lógicos. Cuando se desarrolla una aplicación en un IDE, ya sea Visual Studio, Eclipse o Netbeans, si al escribir una sentencia, olvidamos un ";", hacemos referencia a una variable inexistente 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. El programa no puede compilarse hasta que el programador o programadora no corrija ese error.

El otro tipo de errores son lógicos, comúnmente llamados bugs, estos no evitan que el programa se pueda compilar con éxito, ya que no hay errores sintácticos, ni se utilizan variables no declaradas, etc. Sin embargo, los errores lógicos, 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**. El depurador permite supervisar la ejecución de los programas, para localizar y eliminar los errores lógicos. Un programa debe compilarse con éxito para poder utilizarlo en el depurador. El depurador nos permita 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

Durante la prueba de un programa, puede ser interesante la verificación de determinadas partes del código. No nos interesa probar todo el programa, ya que hemos delimitado el punto concreto donde inspeccionar. Para ello, utilizamos los puntos de ruptura.

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 los siguientes tipos de ejecución: paso a paso por instrucción, paso a paso por procedimiento, ejecución hasta una instrucción, ejecución de un programa hasta el final del programa,

- ✓ Algunas veces es necesario ejecutar un programa línea por línea, para buscar y corregir errores lógicos. El **avance paso a paso** a lo largo de una parte del programa puede ayudarnos a verificar que el código de un método se ejecute en forma correcta.
- ✓ El **paso a paso por procedimientos**, nos permite introducir los parámetros 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 interesa volver a depurarlo, sólo nos interesa el valor que devuelve.
- ✓ En la **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.
- ✓ En la **ejecución de un programa hasta el final** del programa, ejecutamos las instrucciones de un programa hasta el final, sin detenernos en las instrucciones intermedias.

Los distintos modos de ejecución, se van a ajustar a las necesidades de depuración que tengamos en cada momento. Si hemos probado un método, y sabemos que funciona correctamente, no es necesario realizar una ejecución paso a paso en él.

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 las partes que se consideren necesarias, de manera rápida, sencilla y lo más clara posible.

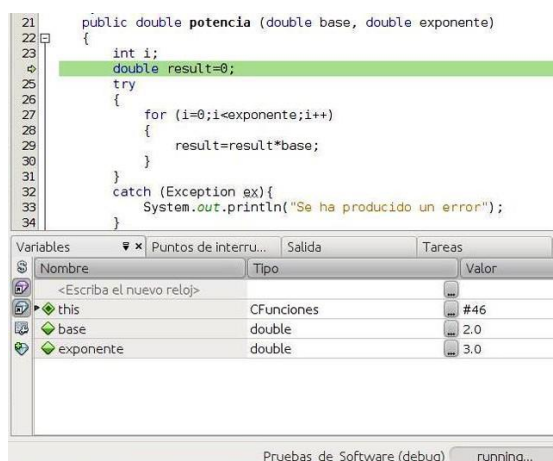
4.3. - Examinadores de variables.

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.

Los examinadores de variables, forman uno de los elementos más importantes del proceso de depuración de un programa. Iniciado el proceso de depuración, normalmente con la ejecución paso a paso, el programa avanza instrucción por instrucción. Al mismo tiempo, las distintas variables, van tomando diferentes valores. Con los examinadores de variables, podemos comprobar los distintos valores que adquieren 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 la ventana de inspección.

Como podemos apreciar, en una ejecución paso a paso, el programa llega a una función de nombre potencia. Esta función tiene definida tres variables. A lo largo de la ejecución del bucle, vemos como la variable **result**, van 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.



Los depuradores son programas que permiten analizar de manera exhaustiva y en paso a paso, lo que pasa dentro del código de un programa. Gracias a los depuradores es fácil probar las aplicaciones para encontrar los posibles errores, analizando sus causas y posibles soluciones.

En el caso del IDE NetBeans, que es el que utilizamos en el ejemplo, estas utilidades están integradas en el entorno de desarrollo

INICIO DE LA DEPURACIÓN

Para conocer las opciones más habituales de depuración en NetBeans, vamos a depurar un pequeño programa. Este programa dispone de una clase Vector, que implementa algunos métodos, como insertar y ordenar.

Para iniciar la depuración del programa, en el menú principal, seleccionamos **Depurar**.

Como hemos indicado, el objetivo de la depuración es la detección de errores en la aplicación. Para encontrar los posibles errores, o comprobar el buen diseño de una aplicación, debemos navegar por el código diseñado. Podemos analizar el código de varias formas:

Paso a paso: en cuyo caso vamos a ir ejecutando instrucción por instrucción, todo el código.

Ejecutar hasta el cursor: El cursor tendrá el foco en una determinada instrucción. El depurador ejecutará el programa y se parará cuando llegue a esa instrucción. Esto es útil, si ya hemos analizado parte del código y no nos interesa volver a depurarlo, ganando tiempo.

LA DEPURACIÓN

Una vez iniciada la depuración (**paso a paso o ejecutar hasta el cursor**), podemos decidir detener la depuración, continuarla, o ejecutar el programa hasta el final.

Podemos elegir **Entrar en el siguiente método**, si queremos analizarlo, o seguir con la ejecución **paso a paso** con la tecla **F7**.

Si el método en cuestión, no necesita depuración podemos **Continuar Ejecución** o pulsar **F8**, con esto el método se ejecuta, pero entramos en su código. Si no nos interesa seguir con la depuración, podemos seleccionar **Finalizar sesión del depurador**, si queremos pausar la depuración **Pausa** y si queremos seguir la ejecución normal, o seguir hasta el siguiente punto de ruptura **Continuar**.

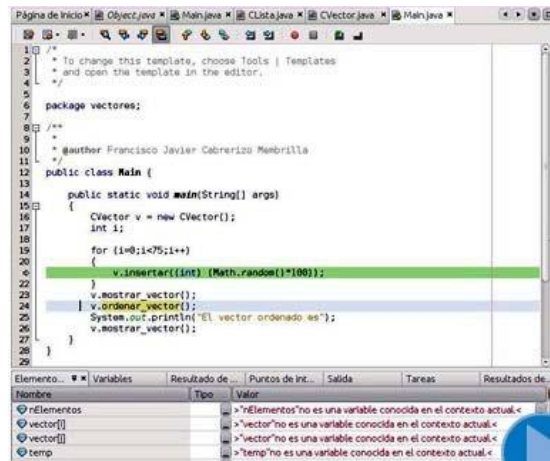
Otras opciones del depurador son:

Consulta de la pila (stack) para comprobar las funciones que están siendo llamadas.

Inserción de puntos de interrupción, para indicar la línea de código que queremos analizar con el depurador

Evaluar expresión, que nos sirve para comprobar los valores que toma dicha expresión.

Si hacemos una depuración **paso a paso**, vamos a ir instrucción por instrucción. Esto nos ayuda a comprobar que los caminos que hemos diseñado en el código se pueden recorrer.



```
public static void main(String[] args)
{
    CVector v = new CVector();
    int i;
    for (i=0; i<75; i++)
    {
        v.insertar((int) (Math.random()*100));
    }
    v.mostrar_vector();
    v.ordenar_vector();
    System.out.println("El vector ordenado es");
    v.mostrar_vector();
}
```

Como vemos en el ejemplo, hay un bucle que se repite 75 veces, si depuramos paso a paso, la ejecución del método **v.insertar()**, deberíamos realizarla 75 veces. En general, se entra una vez en el método, y se realiza la depuración.

Para pasar a otra parte de código que sí queramos depurar, podemos colocar el cursor en la instrucción en cuestión, y pulsamos **Ejecutar hasta el cursor**, o bien, podemos ejecutar y para en el siguiente punto de ruptura.

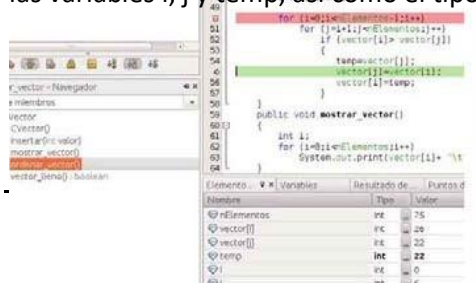
Para insertar un **punto de ruptura**, colocamos el cursor en la instrucción que nos interese, o pulsamos en la parte izquierda de la ventana.

```
public boolean vector_lleno()
{
    if (nElementos<100)
        return false;
    else
        return true;
}
public void ordenar_vector ()
{
    int i,j,temp;
    for (i=0; i<Elementos-1; i++)
        for (j=i+1; j<Elementos; j++)
            if (vector[i]> vector[j])
            {
                temp=vector[j];
                vector[j]=vector[i];
                vector[i]=temp;
            }
}
```

Nos aparecerá la instrucción señalada en rojo. Esto significa, que cuando depuremos, independientemente del tipo de depuración, vamos a parar en esa instrucción. Esto es útil para depurar una parte de código a partir de ese punto de ruptura.

La línea verde, me indica la instrucción por la que va analizando el depurador. En la imagen, también podemos observar la ventana de variables. Esta ventana se utiliza en depuración para inspeccionar el valor que van tomando a lo largo de la ejecución.

En nuestro ejemplo depuramos un método de ordenación de un array, vemos los valores que toman las variables **i**, **j** y **temp**, así como el tipo al que están definidas.



En la imagen, vemos cómo podemos evaluar las expresiones que queremos. En este caso estamos evaluando los valores

los valores almacenados en el vector. Si durante la depuración comprobamos que la aplicación no hace aquello que debiera, podemos cambiarla. Si hemos depurado, y consideramos que la aplicación funciona, podemos pasar a probarla. Para ello, diseñamos casos de prueba en Junit y los aplicamos

****PREGUNTA**

¿Qué afirmación sobre depuración es incorrecta?

- En la depuración podemos inspeccionar las instrucciones que va ejecutando el programa.
- No es posible conocer los valores que toman las variables definidas dentro de un método.
- Solo podemos insertar un punto de ruptura en la depuración.

5.- Validaciones.

En el proceso de validación, interviene de manera decisiva el cliente. Hay que tener en cuenta, que estamos desarrollando una aplicación para terceros, y que son estos los que deciden si la aplicación se ajusta a los requerimientos establecidos en el análisis.

En la validación intentan descubrir errores, pero desde el punto de vista de los requisitos
(Comportamiento y casos de uso que se esperan que cumpla el software que se está diseñando).

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, y 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, que se alcanzan todos los requisitos de rendimiento, que las documentaciones son correctas e inteligible y que se alcanzan otros requisitos, como **portabilidad** (*Capacidad de un programa para ser ejecutado en cualquier arquitectura física de un equipo*), 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. El programa o parte de él, se va a ejecutar bajo unas condiciones previamente especificadas, para una vez observados los resultados, estos sean registrados y evaluados.

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 las 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 prueba 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.

****PREGUNTA**

Durante la validación:

- Procedemos a depurar el programa.
- Sometemos el código a pruebas de cubrimiento.
- Comprobamos que la aplicación cumple los requisitos del cliente.

6.1.- Cubrimiento.

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

Este tipo de prueba, es de caja blanca, ya que nos vamos a centrar en el código de nuestra aplicación.

Con este tipo de prueba, lo que se pretende, es comprobar que todas las funciones, sentencias, decisiones, y condiciones, se van a ejecutar.

Por ejemplo

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


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.

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 las pruebas de cubrimiento, entre ellas, para Java, nos encontramos con Clover.

6.2. - Valores 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.

La experiencia ha demostrado que los casos de prueba que obtienen una mayor probabilidad de éxito, son aquellos que trabajan con valores límite.

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 un 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.

****PREGUNTA**

Si en un bucle while la condición es `while(z>5 && x<10)`, siendo x un valor single, sería valores límite:

- 4 y 11
- 4,99 y 11
- 4,99 y 9,99

6.3. - Clases de equivalencia.

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

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 clases 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.

En este ejemplo, las clases de equivalencia serían:

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

1. Por debajo: $x \leq 0$
2. En: $x > 0$ y $x < 100$
3. Por encima: $x \geq 100$

y los respectivos casos de prueba, podrían ser:

1. Por debajo: $x=0$
2. En: $x=50$
3. Por encima: $x=100$

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

Sin embargo, estos estándares no cubren determinadas facetas de la fase de pruebas, como son la organización el proceso y gestión de las pruebas, presentan pocas pruebas funcionales y no funcionales etc. Ante esta problemática, la industria ha desarrollado la norma ISO/IEC 29119.

La norma ISO/IEC 29119 de prueba de software, pretende unificar en una única norma, todos los estándares, de forma que proporcione vocabulario, procesos, documentación y técnicas para cubrir todo el ciclo de vida del 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. Con este estándar, se podrá realizar cualquier prueba para cualquier proyecto de desarrollo o mantenimiento de software.

Debes conocer

Norma ISO/IEC 29119.

La norma ISO/IEC 29119 la compone 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) .

Para saber más

En el siguiente enlace podrás visitar la página internacional, donde se detallan las normas a seguir, por las pruebas de software:

Normas para la prueba de software <http://softwaretestingstandard.org/>

**PREGUNTA

¿Qué norma de calidad intenta unificar los estándares para pruebas de software?

- BS 7925-1
- -IEEE 1008
- ISO/IEC 29119

8.- Pruebas unitarias.

Las pruebas unitarias, o prueba de la unidad, tienen por objetivo probar el correcto funcionamiento de un módulo de código. El fin que se persigue, es que cada módulo funciona correctamente por separado.

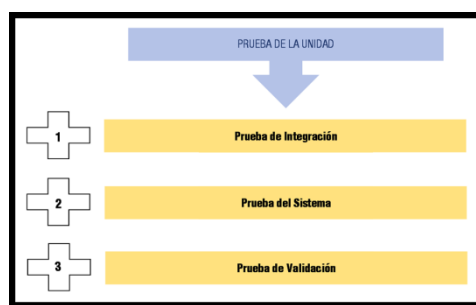
Posteriormente, con la prueba de integración, se podrá asegurar el correcto funcionamiento del sistema.

Una unidad es la parte de la aplicación más pequeña que se puede probar. En programación procedural, una unidad puede ser una función o procedimiento, En programación orientada a objetos, una unidad es normalmente un método.

Con las pruebas unitarias se debe probar todas las funciones o métodos no triviales de forma que cada caso de prueba sea independiente del resto.

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

- ✓ **Automatizable:** no debería requerirse una intervención manual.
- ✓ **Completas:** deben cubrir la mayor cantidad de código.
- ✓ **Repetibles o Reutilizables:** no se deben crear pruebas que sólo puedan ser ejecutadas una sola vez.
- ✓ **Independientes:** la ejecución de una prueba no debe afectar a la ejecución de otra.
- ✓ **Profesionales:** las pruebas deben ser consideradas igual que el código, con la misma profesionalidad, documentación, etc.



El objetivo de las pruebas unitarias es aislar cada parte del programa y demostrar que las partes individuales son correctas. Las pruebas individuales nos proporcionan cinco ventajas básicas:

1. **Fomentan el cambio:** Las pruebas unitarias facilitan que el programador cambie el código para mejorar su estructura, puesto que permiten hacer pruebas sobre los cambios y así asegurarse de que los nuevos cambios no han introducido errores.
2. **Simplifica la integración:** Puesto que permiten llegar a la fase de integración con un grado alto de seguridad de que el código está funcionando correctamente.
3. **Documenta el código:** Las propias pruebas son documentación del código puesto que ahí se puede ver cómo utilizarlo.
4. **Separación de la interfaz y la 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.
5. **Los errores están más acotados y son más fáciles de localizar:** dado que tenemos pruebas unitarias que pueden desenmascararlos.

8.1. - Herramientas para Java.

Entre las herramientas que nos podemos encontrar en el mercado, para poder realizar las pruebas, las más destacadas serían:

✓ **Jtiger:**

- Framework de pruebas unitarias para Java (1.5).
- Es de código abierto.
- Capacidad para exportar informes en HTML, XML o texto plano.
- Es posible ejecutar casos de prueba de Junit mediante un plugin.
- Posee una completa variedad de aserciones como la comprobación de cumplimiento del contrato en un método.
- Los metadatos (*Conjunto de datos que se utilizan para describir otros datos*) de 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 página web con toda la información necesaria para comprender su uso, y utilizarlo con IDE como Eclipse.
- El 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*) incluye pruebas unitarias sobre sí mismo.

✓ **TestNG:**

- Está inspirado en JUnit y NUnit.
- Está diseñado para cubrir todo tipo de pruebas, no solo las unitarias, sino también las funcionales, las de integración ...
- Utiliza las anotaciones de Java 1.5 (desde mucho antes que Junit).
- Es compatible con pruebas de Junit.
- Soporte para el paso de parámetros a los métodos de pruebas.
- Permite la distribución de pruebas en máquinas esclavas.
- Soportado por gran variedad de plug-ins (Eclipse, NetBeans, IDEA ...)
- Las clases de pruebas no necesitan implementar ninguna interfaz ni extender ninguna otra clase.
- Una vez compiladas la 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 (un método puede pertenecer a uno o varios grupos).

✓ **Junit:**

- 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.

8.2. - Herramientas para otros lenguajes.

En la actualidad, nos encontramos con un amplio conjunto de herramientas destinadas a la automatización de la prueba, para la mayoría de los lenguajes de programación más extendidos en la actualidad. Existen herramientas para C++, para PHP, FoxPro, etc.

Cabe destacar las siguientes herramientas:

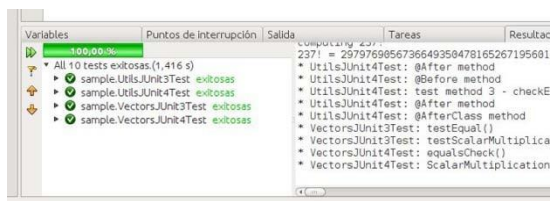
- ✓ **CppUnit:**
 - Framework de pruebas unitarias para el lenguaje C++.
 - Es una herramienta libre.
 - Existe diversos entornos gráficos para la ejecución de pruebas como QTestRunner.
 - Es posible integrarlo con múltiples entornos de desarrollo como Eclipse.
 - Basado en el diseño de xUnit.
- ✓ **Nunit:**
 - 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. Junit
- ✓ **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).

****PREGUNTA**

La herramienta de automatización de pruebas más extendida para Java es:

- Junit
- FoxUnit
- Simple Test

9.- Automatización de la prueba.



Las pruebas de software son parte esencial del ciclo de desarrollo. La elaboración y mantenimiento de unidad, pueden ayudarnos a asegurar que los métodos individuales de nuestro código, funcionan correctamente. Los entornos de desarrollo, integran frameworks, que permiten automatizar las pruebas.

En el caso de entornos de desarrollo para Java, como NetBeans y Eclipse, nos encontramos con el framework JUnit. JUnit es una herramienta de automatización de pruebas que nos permite de manera rápida y sencilla, elaborar pruebas. La herramienta nos permite diseñar clases de prueba, para cada clase diseñada en nuestra aplicación. Una vez creada las clases de prueba, establecemos los métodos que queremos probar, y para ello diseñamos casos de prueba. Los criterios de creación de casos de prueba, pueden ser muy diversos, y dependerán de lo que queramos probar.

Una vez diseñados los casos de prueba, pasamos a probar la aplicación. La herramienta de automatización, en este caso Junit, nos presentará un informe con los resultados de la prueba (imagen anterior). En función de los resultados, deberemos o no, modificar el código.

Uso de JUNIT en NetBeans

INTRODUCCIÓN

Los entornos de desarrollo más extendidos, que se utilizan para implementar aplicaciones Java, tales como NetBeans o Eclipse, incorporan un plugin para trabajar con Junit

JUnit nos va a servir para realizar pruebas unitarias de clases escritas en Java, dentro de un entorno de pruebas. Es un framework con muy pocas clases fácil de aprender y de utilizar.

Una vez que hemos diseñado nuestra aplicación, y la hemos depurado, procedemos a probarla. En el caso del ejemplo, disponemos de una clase, de nombre **CVector**, donde se han definido una serie de métodos.

El objetivo va a ser el diseño y ejecución de algunos casos de prueba.

INICIO DE JUNIT

Para iniciar Junit, seleccionada en la ventana de proyectos la clase a probar, abrimos el menú contextual y seleccionamos **Herramientas > Crear pruebas Junit**.

Nos aparece un formulario donde nos da a elegir entre JUnit 4.x. Es la versión de JUnit disponible en NetBeans 19.

Puesto que vamos a probar la clase CVector, por convenio es recomendable llamar a la clase de prueba CVectorTest. Esta clase se va a insertar en un nuevo paquete de nuestro proyecto, denominado **Paquete de prueba**.

Como se aprecia en el formulario, JUnit va a generar los métodos que aparecen seleccionados. En nuestro caso lo vamos a dejar tal cual, aunque luego van a ser modificados en el código.

Al pulsar el botón *Aceptar* nos aparecen una nueva clase de nombre *CVectorTest*, que contiene los métodos que estaban seleccionados en el formulario anterior, con un código prototipo. Es en ese código en el que el programador creará sus casos de prueba.

```
public class CVectorTest extends TestCase {
    public CVectorTest(String testName) {
        super(testName);
    }

    @Override
    protected void setUp() throws Exception {
        super.setUp();
    }

    @Override
    protected void tearDown() throws Exception {
        super.tearDown();
    }

    /**
     * Test of insertar method, of class CVect
     */
}
```

El diseño de los casos de prueba, requiere que se establezcan criterios que garanticen que esa prueba tiene muchas probabilidades de encontrar algún error no detectado hasta el momento.



CASOS DE PRUEBA

En primer lugar, vamos a ver la función de los Inicializadores y Finalizadores. El método *setUp* y el método *tearDown*, se utilizan para inicializar y finalizar las condiciones de prueba, como puede ser la creación de un objeto, inicialización de variables, etc. En algunos casos, no es necesario utilizar estos métodos, pero siempre se suelen incluir.

El **método *setUp*** es un método de inicialización de la prueba y se ejecutan antes de cada caso de prueba, en la clase de prueba. Este método no es necesario para ejecutar pruebas, pero si es necesario para inicializar algunas variables antes de iniciar la prueba.

El **método *tearDown*** es un método finalizador de prueba, y se ejecutará después de cada test en la clase prueba. Un método finalizador no es necesario para ejecutar las pruebas, pero si necesitamos un finalizador para limpiar algún dato que fue requerido en la ejecución de los casos de prueba.

En segundo lugar, es necesario conocer las aserciones. Los método *assertXXX()*, se utilizan para hacer las pruebas. Estos métodos, permiten comprobar si la salida del método que se está probando, concuerda con los valores esperados. Las principales son:

AssertTrue() evalúa una expresión booleana. La prueba pasa si el valor de la expresión es true.

assertFalse() evalúa una expresión booleana. La prueba pasa si el valor de la expresión es false.

AssertNull() verifica que la referencia a un objeto es nula.

assertNotNull() verifica que la referencia a un objeto es no nula.

AssertSame() compara dos referencias y asegura que los objetos referenciados tienen la misma dirección de memoria. La prueba pasa si los dos argumentos son el mismo objeto o pertenecen al mismo objeto.

assertNotSame() Compara dos referencias a objetos y asegura que ambas apuntan a diferentes direcciones de memoria. La prueba pasa si los dos argumentos suplidos son objetos diferentes o pertenecen a objetos distintos.

assertEquals() Se usa para comprobar igualdad a nivel de contenidos. La igual de tipos primitivos se compara usando "==" , la igual entre objetos se compara con el método equals(). La prueba pasa si los valores de los argumentos son iguales.

fails() causa que la prueba falle inmediatamente. Se puede usar cuando la prueba indica un error o cuando se espera que el método que se está probando llame a una excepción.

En este punto, nos disponemos a diseñar los métodos que necesitamos para los casos de prueba. Ejemplos de casos de prueba pueden ser:

```
public void test_posicion (int pos){
    vectorTest.insertar(3);
    vectorTest.insertar(5);
    try{
        assertTrue (vectorTest.posicion(7)==1);
    }catch (Exception e){
        fail("El método test_posicion no funciona");
    }
}

/**
 * Test del método ordenar_vector, de la clase CVector. Si ordena bien, la
 * comparación entre el
 * vector vOrdenado y prueba, debe ser cierta y la prueba un éxito
 */
public void test_ordenar_vector(){
    int [] vOrdenado = {7,36,45,52,85};
    int [] prueba = new int[5];
    try{
        vectorTest.insertar(36);
        vectorTest.insertar(85);
        vectorTest.insertar(45);
        vectorTest.insertar(52);
        vectorTest.insertar(7);
        vectorTest.ordenar_vector();
        for (int i=0;i<5;i++)
            prueba[i]=vectorTest.posicion(i);
        assertEquals(vOrdenado,prueba);
    }catch (Exception e){
        fail("El método vector_lleno no funciona");
    }
}

/**
 * Test del método vector_lleno(), de la clase cVector. Debería devolver true, al
 * insertar 100 elementos
 */
public void test_vector_lleno(){
    try{
        for(int i=0;i<100;i++)
            vectorTest.insertar(i);
        assertTrue(vectorTest.vector_lleno());
    }
    catch(Exception e){
        fail("El método vector_lleno no funciona");
    }
}
```

Estos tres métodos intentan probar algunos métodos de la clase CVector. Para ello, teniendo seleccionado el proyecto, accedemos al menú contextual y pulsamos la opción Probar.

Como se puede comprobar, la prueba sobre el método vector_lleno ha sido un éxito, pero ha fallado la prueba sobre ordenar_vector. Con esta información, debemos comprobar que el caso de prueba está diseñado, en cuyo caso, lo que se ha encontrado es un error en el diseño del método ordenar_vector, y hay que rediseñarlo. La ventaja de utilizar herramientas automatizadas, es que se facilita la regresión, ya que tenemos diseñado el caso de prueba para el método, así que una vez



rediseñado, podemos volver a probarlo con el mismo caso de prueba.

10. - Documentación de la prueba.

Como en otras etapas y tareas del desarrollo de aplicaciones, la documentación de las pruebas es un requisito indispensable para su correcta realización. Unas pruebas bien documentadas podrán también servir como base de conocimiento para futuras tareas de comprobación.

Las metodologías actuales, como Métrica v.3 (*Metodología de Planificación, Desarrollo y Mantenimiento de sistemas de información. Métrica v3 se puede usar libremente, con la única restricción de citar la fuente de su propiedad intelectual, que es el Ministerio de Presidencia*), proponen que la documentación de la fase de pruebas se base en los estándares ANSI / IEEE sobre verificación y validación de software.

En propósito de los estándares ANSI/IEEE es describir un conjunto de documentos para las pruebas de software. Un documento de pruebas estándar puede facilitar la comunicación entre desarrolladores al suministrar un marco de referencia común. La definición de un documento estándar de prueba puede servir para comprobar que se ha desarrollado todo el proceso de prueba de software.

Los documentos que se van 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.

**PREGUNTA

La documentación de la prueba:

- Es una labor voluntaria que se puede realizar al final del proceso de pruebas
- Cada equipo de pruebas decide qué documenta y cómo.
- En España se usa Métrica v.3

Anexo I.- Norma ISO/IEC 29119.

La norma 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).