Diseño y realización de pruebas.

Caso práctico

Situación

BK programación se encuentra desarrollando la primera versión de la aplicación de gestión hotelera.

Ada, la supervisora de proyectos de BK Programación, se reúne con Juan y María para empezar a planificar el diseño y realización de pruebas sobre la aplicación,

Ana se va a encargar, de ir probando los distintas partes de la aplicación que se van desarrollando. Para ello usará casos de prueba diseñados por Juan, y evaluará los resultados.

Juan evaluará los resultados de las pruebas realizadas por Ana, y en su caso, realizará las modificaciones necesarias en el proyecto.

GOBIERNO DE ESPAÑA
MINISTERIO DE EDUCACIÓN Y FORMACIÓN PROFESIONAL

Ministerio de Educación y Formación Profesional. (Dominio público)

Materiales formativos de FP Online propiedad del Ministerio de Educación y Formación Profesional.
Aviso Legal

1.- Planificación de las pruebas.

Caso práctico

Todos en la empresa están inmersos en el desarrollo de la aplicación de gestión hotelera. Para garantizar la corrección del desarrollo, Ada propone establecer la planificación de las pruebas. ¿Por qué hay que probar el software? ¿Es necesario seguir un orden concreto en la realización de pruebas? ¿Qué pruebas se realizan?

Durante todo el proceso de desarrollo de software, desde la fase de análisis hasta la implantación en el cliente, es habitual incurrir en errores de varios tipos: incorrecta especificación de los objetivos, errores producidos en el diseño o errores en la fase de desarrollo.

Por lo tanto, se hace necesario hacer un conjunto de **pruebas** que permita comprobar que el producto que se está creando, es correcto y cumple con las especificaciones solicitadas por el usuario.

Las pruebas tratan de **verificar** y **validar** las aplicaciones, entendiendo estos términos como:

- La **verificación** es la comprobación de 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 adecuada, se definen estrategias de pruebas.

Siguiendo el **modelo en espiral**, las pruebas empezarán con las **pruebas unitarias** de cada porción de código.

Una vez pasadas estas pruebas con éxito, se seguirá con las **pruebas de integración**, donde se ponen todas las partes del código en común, comprobando que el ensamblado de los bloques de código y sus pruebas atienden a los establecido durante la fase de diseño.

El siguiente paso será 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.

Notas:

- El objetivo de las pruebas es conseguir un software libre de errores, por lo tanto la detección de defectos en el software se considera un éxito en esta fase.
- El programador debe evitar probar sus propios programas, ya que aspectos no considerados durante la codificación podrán volver a pasar inadvertidos en las pruebas si son tratados por la misma persona.

2.- Tipos de prueba.

Caso práctico

Juan y María están implementando la mayor parte de la aplicación. ¿Es correcto lo realizado hasta ahora? ¿Cómo se prueba los valores devueltos por una función o método? ¿Es posible seguir la ejecución de un programa, y ver si toma los caminos diseñados?

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.

Una prueba de tipo caja negra se lleva a cabo sin tener necesidad de conocer la estructura interna del sistema. Cuando se realiza este tipo de pruebas, sólo 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 el siguiente ejemplo:

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

Se detecta un error mediante las pruebas de caja negra. Se entiende que tener una nota superior a 5 debería dar el mensaje de superado y viceversa. Los mensajes están intercambiados.

- **Prueba de la Caja Blanca** (White Box Testing): en este caso, se prueba la aplicación desde dentro, usando su lógica de aplicación.

En contraposición a lo anterior, una prueba de **Caja Blanca**, va a analizar y probar directamente el código de la aplicación, intentando localizar estructuras incorrectas o ineficientes en el código. 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.

A continuación se muestra un código que presenta un error detectable mediante las pruebas de caja blanca:

```
 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.");
```

Aunque el programa dará el resultado esperado para cualquier nota (comprobación que se lleva a cabo con las pruebas de caja negra-funcionales), presenta un error en su estructura, el **else** nunca es alcanzado. De este error nos daremos cuenta gracias a las pruebas de caja blanca.

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?

Mostrar retroalimentación

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 (pruebas de la caja negra)

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, 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. Esta 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.

Existe 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.- Pruebas estructurales (pruebas de la caja blanca)

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 la 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.

Autoevaluación

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.

Incorrecta, solo interactuamos con la interfaz, no con las instrucciones

No es correcta, esa es una prueba de caja blanca, ya que se comprueba la implementación de los métodos.

Muy bien. Esa es la idea...

No es correcta, ya que cada una tiene una utilidad diferente, y se complementan.

Solución

1. Incorrecto
2. Incorrecto
3. Opción correcta
4. Incorrecto

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

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.

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.

El diseño de las pruebas de regresión podrá ser una combinación de casos de uso obtenidos desde los enfoques de caja blanca, caja negra y aleatoria.

Autoevaluación

La prueba de regresión:

- ○ Se realiza una vez finalizado cada módulo 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.

Incorrecta, solo si se han producido modificaciones.

Incorrecta. Se trata de volver a realizar las pruebas, y estas pueden ser de cualquier tipo.

Muy bien. Esa es la idea...

No es correcta, ya que cada una tiene una utilidad diferente, y se complementan.

Solución

1. Incorrecto
2. Incorrecto
3. Opción correcta
4. Incorrecto

3.- Procedimientos y casos de prueba.

Caso práctico

En BK, ya tienen el proyecto bastante avanzado. Ahora llega una parte clave: definir las partes del sistema que se van a probar y establecer los casos de prueba para realizarla. Ana va a participar, pero cuando se habla de procedimientos y casos de prueba, se siente perdida. A ella le va a tocar ejecutar los casos de prueba.

Caso práctico

Juan y María prueban cada parte de código que están implementando. Algunos métodos requieren una comprobación de su estructura interna, en otros, valdría con probar los resultados que devuelven. Antonio se pregunta en que consiste cada prueba, y como se lleva a cabo en la práctica.

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.

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 la 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:

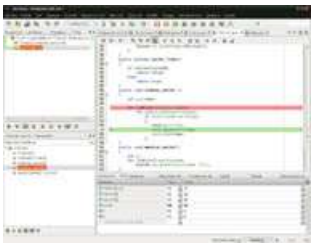
- **Enfoque funcional o de caja negra**. En este tipo de prueba, nos centramos en que el programa, o parte del programa que estamos probando, recibe un entrada de forma adecuada y se produce una salida correcta, así como que la integridad de la información externa se mantiene. La prueba no verifica el proceso, solo los resultados. Este enfoque se centra en la funciones, entradas y salidas. Se aplican los **valores límite** y las **clases de equivalencia.**.
- **Enfoque estructural o caja blanca**. En este tipo de pruebas, debemos centrar en la implementación interna del programa. 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. . Para ello se utilizan generadores automáticos de casos de prueba.

4.- Herramientas de depuración.

Caso práctico

Juan y María tiene muchas líneas de código implementadas. Como programadores de la aplicación, juegan un papel decisivo en la fase de pruebas. Al realizar este proyecto utilizando un entorno de desarrollo integrado (NetBeans), cuenta con una herramienta fundamental, el depurador.

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



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 codificación**. Cuando ocurre un error de codificación, 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.

Autoevaluación

¿Que concepto está relacionado con la prueba de caja negra?

- ○ Es la principal herramienta de validación.
- ○ Se pueden comprobar los valores que van tomando las variables
- ○ Se comprueba que todos los caminos del programa, se pueden recorrer, al menos una vez.
- ○ Es incompatible con la prueba de caja blanca.

Correcto. Con las pruebas de caja negra no se depura el programa, se comprueba que devuelve los valores esperados en función de las entradas introducidas.

Incorrecta, solo interactuamos con la interfaz, no con las instrucciones

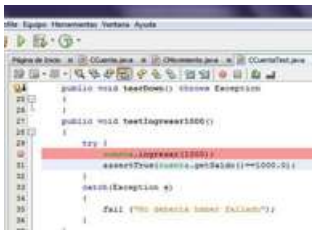
No es correcta, esa es una prueba de caja blanca, ya que se comprueba la implementación de los métodos.

No es correcta, ya que cada una tiene una utilidad diferente, y se complementan.

Solución

1. Opción correcta
2. Incorrecto
3. Incorrecto
4. Incorrecto

4.1.- Puntos de ruptura.



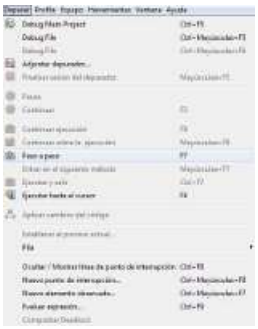
Dentro del menú de depuración, nos encontramos con la opción insertar punto de ruptura (breakpoint). Se selecciona la línea de código donde queremos que el programa se pare, para a partir de ella, inspeccionar variables, o realizar una ejecución paso a paso, para verificar la corrección del código

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 lo siguientes tipo 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 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.
- 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 probada 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 la partes que se consideren necesarias, de manera rápida, sencilla y los 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 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.

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.

Autoevaluación

¿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

Incorrecta. Podemos ir paso a paso por instrucción, o por procedimiento.

No es correcta. Podemos inspeccionar todas la variables del proyecto, cuando la depuración llegue al método en cuestión, podemos inspeccionar los valores tomados por las variables.

Muy bien. En depuración podemos insertar tantos puntos de ruptura como queremos, y que la depuración vaya saltando de uno a otro.

Solución

1. Incorrecto
2. Incorrecto
3. Opción correcta

5.- Validaciones.

Caso práctico

Durante todo el proceso de desarrollo, existe una permanente comunicación entre Ada y su equipo, con representantes de la empresa a la que va destinado el proyecto. Ana y Juan van a asistir a la siguiente reunión, donde se va a mostrar a los representantes de la empresa, las fases de proyectos ya implementadas. Será la primera reunión de validación del proyecto.

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.

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, compatibilidad, recuperación de errores, facilidad de mantenimiento etc.

Autoevaluación

Durante la validación:

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

Incorrecta.

No es correcta.

En esta prueba, es el cliente, junto con el equipo de desarrollo, quienes comprueban que lo desarrollado cumple las especificaciones establecidas.

Solución

1. Incorrecto
2. Incorrecto
3. Opción correcta

6.- Normas de calidad.

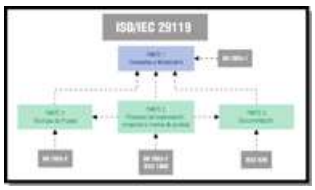
Caso práctico

Las aplicaciones que desarrolla BK Programación, se caracterizan por cumplir todos los estándares de calidad de la industria. Como no podía ser de otro modo, a la hora de realizar las pruebas, también van a seguir losestándares más actuales del mercado. Ada se va a encargar de supervisar el cumplimiento de los estándares más actuales.

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

- 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 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.

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) .

Autoevaluación

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

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

Incorrecta.

No es correcta.

Muy bien.

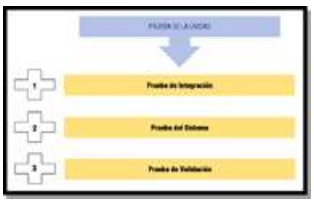
Solución

1. Incorrecto
2. Incorrecto
3. Opción correcta

7.- Pruebas unitarias.

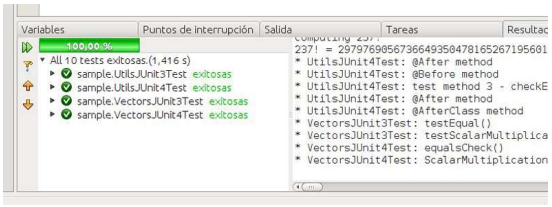
Caso práctico

Antonio está un poco confuso. La aplicación que están diseñando Juan y María es ya de cierta envergadura y se pregunta por la labor ingente que queda, solo para probar todos los componentes de la aplicación. María le tranquiliza y le comenta que los Entornos de Desarrollo actuales, incorporan herramientas que realizan la pruebas de cada método, de forma automática.



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

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, es decir, que cada caso de prueba sea independiente del resto.



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

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

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.

7.1..- Herramientas para Java.

Entre la 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** 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 pagina web con toda la información necesaria para comprender su uso, y utilizarlo con IDE como Eclipse.
  - El **Framework** incluye pruebas unitarias sobre sí mismo.
- TestNG:
  - Esta 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 maquinas esclavas.
  - Soportado por gran variedad de plug-ins (Eclipse, NetBeans, IDEA ...)
  - Los clases de pruebas no necesitan implementar ninguna interfaz ni extender ninguna otra clase.
  - Una vez compiladas la pruebas, estas se pueden invocar desde la linea 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:

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.

Sus características principales son:

- 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.

Caso práctico

Juan está realizando pruebas de la unidad, es decir, comprueba el correcto funcionamiento de los métodos que ha implantado. Para ello, utiliza las herramienta de prueba incorporadas en el entorno de

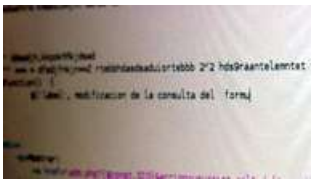
desarrollo. En sucaso, ya que está utilizando NetBeans, se decanta por Junit. Ana está muy interesada en conocer esta herramienta, que ayuda notablemente en el proceso de pruebas.

Para saber más

En el siguiente enlace nos encontramos con un ejempo completo de prueba de la unidad con NetBeans

Creación de Casos de Prueba en NetBeans con Junit

7.2.- Herramientas para otros lenguajes.



En la actualidad, nos encontramos con un amplio conjunto de herramientas destinadas a la automatización del 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 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. Junit
- **SimpleTes**t: 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).

Autoevaluación

Las herramientas de automatización de pruebas más extendida para Java es:

- ○ Junit.
- ○ FoxUnit.
- ○ Simple Test.

Muy bien. Hay otras herramientas para Java, pero esta es la más popular.

No es correcta. Esta diseñada para Microsoft Visual FoxProFoxUnit

No es correcta. Esta diseñada para PHP

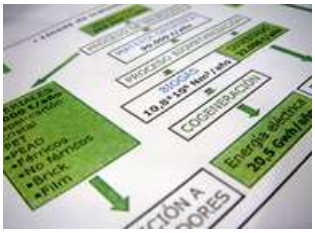
Solución

1. Opción correcta
2. Incorrecto
3. Incorrecto

8.- Documentación de la prueba.

Caso práctico

BK Programación, al igual que en todas la fase de diseño de un sistema, en la fase de prueba realiza la documentación. Ada, como coordinadora, le pide a Ana que ayuda a realizar la documentación de la prueba, y le pide que se repase la metodología Métrica v.3 y ayude a María y a Juan en la labor de documentación.



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**, proponen que la documentación de la fase de pruebas se basen 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.

Para saber más

En el siguiente enlace podrás visitar la página de Ministerio de Política Territorial y Administración Pública, dedicada a Métrica v.3

Métrica v.3

Autoevaluación

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

Incorrecta. Hay que documentar cada paso que se dé en el proceso de pruebas.

No es correcta. Hay que seguir alguna metodología de la industria.

Muy bien. Es la metodología más extendida en la actualidad.

Solución

1. Incorrecto
2. Incorrecto
3. Opción correcta

Anexo I.- Pruebas de la caja blanca

Las **pruebas estructurales** son el conjunto de **pruebas de la caja blanca**. Consiste en hacer diferentes pruebas del software con el fin de pasar por todas las líneas de ejecución del programa.

Se trata de plantear casos de prueba viendo el código interno.

Las pruebas estructurales no pretenden asegurar la corrección de los resultados producidos, 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, 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 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 comprobación lógica del programa, se evalúe al menos una vez a cierto y otra a falso. En la decisión **MIENTRAS (A and B)**, habrá casos de prueba donde (A and B) sea verdadero y donde (A and B) sea 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. En la decisión **MIENTRAS (A and B)**, habrá casos de prueba donde A sea falso, A sea verdadero, B sea falso y B sea verdadero.
- **Cobertura de condiciones y decisiones**: consiste en cumplir simultáneamente las dos anteriores.
- **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 al menos dos veces.

◁

a.- Pasos a seguir

La técnica para determinar los **casos de prueba de caja blanca** que garantiza cobertura de sentencias, decisiones/condiciones y de caminos.

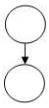
Se realiza completando los siguientes pasos:

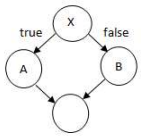
- **Crear un grafo** que represente el código a probar.
- Calcular la **complejidad ciclomática o de McCABE** del grafo obtenido.
- Determinar tantos **caminos** (recorridos del grafo) como la complejidad ciclomática calculada.
- Generar un **caso de uso** por cada camino, determinando sus datos de entrada y los resultados esperados.
- Lanzar una **ejecución del programa** por cada caso de uso y **comparar los resultados obtenidos con los esperados** para comprobar la corrección del código.

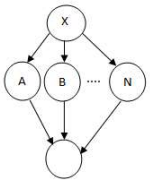
a.1.- Creación del grafo

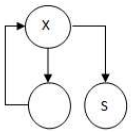
Se trata de **crear un grafo** en base al tipo de instrucciones que vayamos encontrando en nuestro código. Los tipos de estructuras principales que aparecen en los programas son secuencias de instrucciones, condiciones e iteraciones. Éstas se representan como sigue en el grafo.

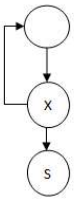
Estructuras básicas	

Secuencia	
	 `String sNota = "10";`
	`System.out.println("Tu nota es: " + sNota);`

Condición	
	 `int iNota = 3;`
	`if( iNota>= 5)`
	`{ System.out.println("Enhorabuena. Superado."); }`
	`else if ( iNota < 5)`
	`{ System.out.println("La proxima vez sera"); }`

Selección múltiple	
	 `switch (iNota) {`
	`case 1: case 2: case 3: case 4:`
	`{ System.out.println("La proxima vez sera");`
	`break;}`
	`default:`
	`{ System.out.println("Enhorabuena. Superado."); }`
	`}`

Iteración	
	 `int iNumSal = 2;`
	`while(iNumSal > 0)`
	`{`
	`System.out.println("Hola !!!!!");`
	`iNumSal--;`
	`}`

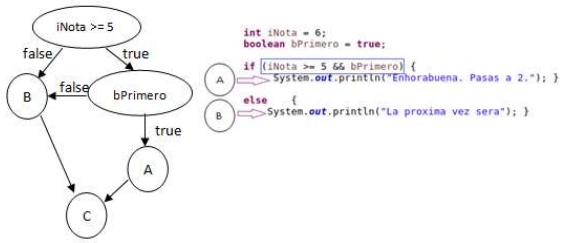
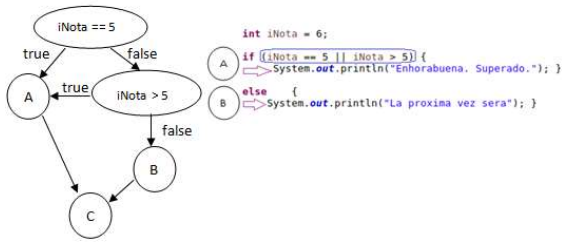
Do Iteración	
	 `int iNumSal = 2;`
	`do`
	`{`
	`System.out.println("Hola !!!!!");`
	`iNumSal--;`
	`}`
	`while(iNumSal > 0);`

Los **grafos se construyen a partir de nodos y aristas**. Los nodos representan secuencias de instrucciones consecutivas donde no hay alternativas en la ejecución o condiciones a evaluar, que en función del resultado hará que la ejecución siga una dirección u otra.

Las aristas son las encargadas de unir los nodos.

En el caso de que las decisiones tengan múltiples condiciones, habrá que separar cada condición en un nodo como sigue:

Estructuras de decisión compuestas	
And (&&)	

	
Or (||)	

Algunos **consejos útiles** al crear grafos son:

- Separar todas las condiciones.
- Agrupar sentencias ‘simples’ en bloques.
- Numerar todos los bloques de sentencias y también las condiciones.

a.2.- Complejidad de McCabe o ciclomática

A partir del grafo se determina su **complejidad ciclomática**. Es posible hacerlo por tres métodos diferentes, pero todos ellos han de dar el mismo resultado.

- **V (G) = a - n + 2**, siendo a el número de arcos o aristas del grafo y n el número de nodos.
- **V (G) = r**, siendo r el número de regiones cerradas del grafo (incluida la externa).
- **V(G) = c + 1**, siendo c el número de nodos de condición.

a.3.- Caminos de prueba

El número de **caminos de prueba** debe ser igual a la complejidad calculada. Consiste en hacer recorridos desde el inicio hasta el final del método con el que se esté trabajando. Se irán registrando los nodos por los que va pasando la ejecución del camino.

Cada nuevo camino deberá aportar el paso por nuevas aristas/nodos del grafo. La definición de caminos se hará desde los más sencillos a los más complicados.

a.4.- Casos de uso, resultados esperados y análisis

Ahora toca definir los datos de entrada al programa que nos permitan recorrer los caminos definidos en el apartado anterior. El conjunto de entradas al programa utilizados en la ejecución de cada camino se denomina **caso de uso**.

Además habrá que definir los **resultados previstos.** Cuando posteriormente se lance la ejecución del programa para cada caso de uso, los resultados obtenidos serán comparados con los esperados y así determinar la corrección del código.

b.- Ejemplo 1. Fibonnaci

El siguiente programa **java:**

```java
package seriefibonaccicajablanca;
import java.util.Scanner;
public class SerieFibonacciCajaBlanca {
    public static void main(String[] args) {
        SerieFibonacciCajaBlanca misCal = new SerieFibonacciCajaBlanca();
        misCal.Fibonacci();
    }
    public void Fibonacci() {
        Scanner miScan = new Scanner(System.in);
        System.out.print("¿Quiere salir del programa?: ");
        String sSalir = miScan.nextLine();
        int iValor = 0;
        String sResultado;
        String sAux;
        while (!(sSalir.equals("S") || sSalir.equals("s"))) {
            sResultado = "";
            System.out.print("\n\t¿Cuantos numeros de la serie deseas mostrar?: ");
            sAux = miScan.nextLine();
            iValor = Integer.parseInt(sAux);
            switch (iValor) {
                case 3:
                    sResultado = " 1";
                case 2:
                    sResultado = " 1" + sResultado;
                case 1:
                    sResultado = " 0" + sResultado;
                    System.out.println("\t\tLos " + iValor + " primeros numeros de la serie de Fibonacci son: " + sI
                    break;
                default:
                    System.out.println("\t\tNúmero no permitido. Tiene que estar entre 1 y 3.");
            }
            System.out.print("\n¿Quiere salir del programa?: ");
            sSalir = miScan.nextLine();
        }
    }
}
```

Hace el cálculo de la **serie Fibonacci** y muestra el resultado por pantalla. El programa visualizará tantos dígitos de la serie como se indique por el teclado, siendo tres el número más alto que se puede indicar.

La serie de Fibonacci, comienza por el cero, sigue por el uno, y los siguientes números se van calculando como la suma de los dos anteriores, es decir: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55.

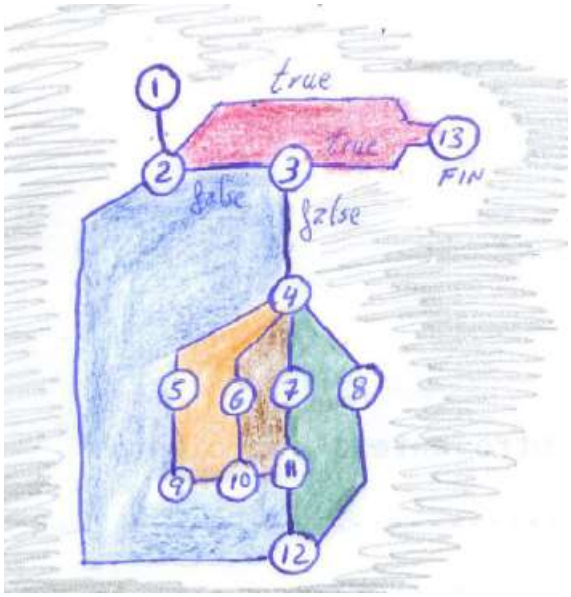
Con lo cual:

- Si el usuario inserta un 1, se visualizará: "0"
- Si el usuario inserta un 2, se visualizará: "0 1"
- Si el usuario inserta un 3, se visualizará: "0 1 1"

Una vez visualizada la serie, podrá insertar otro número el usuario hasta que inserte una "S" o una "s" indicando que quiere salir del programa.

Pulsa aquí para descargarte este proyecto y poder hacer pruebas ("SerieFibonacciCajaBlanca.zip")..

b.1.- Creación del grafo

Grafo	Nodo	Línea - Condición
	1	Lin: 9-14.
	2	Lin: 15. Cond. "== S"
	3	Lin: 15. Cond. "== s"
	4	Lin 16-20. Sentencia Switch
	5	Lin 21. iValor == 3
	6	Lin 23. iValor == 2
	7	Lin 25. iValor == 1
	8	Lin 29-30. iValor != anteriores. **"Número no permitido. Tiene que estar entre 1 y 3."**
	9	Lin 22.
	10	Lin 24.
	11	Lin 26.
	12	Lin 32-33.
	13	Lin 36.

Nota: los nodos 5 y 9 podrían haberse agrupado en uno sólo. Tanto juntos como por separado, el número de caminos, casos de uso y complejidad ciclomática es el mismo.

Lo mismo sucede con los nodos 6-10 y 7-11.

b.2.- Complejidad de McCabe o ciclomática

Calculada por los tres métodos posibles. Todos ellos deben dar el mismo resultado.

Método de cálculo	**Complejidad**	**Comentarios**
Nº de regiones	6	Hay que considerar la región exterior.
Nº de aristas - Nº nodos + 2	16 - 12 + 2 = 6	
Nº de condiciones + 1	5 + 1 = 6	Nodos 2, 5, 6, 7 y 8

b.3.- Caminos de prueba

El número de caminos de prueba debe ser igual a la complejidad calculada. Cada nuevo camino deberá aportar el paso por nuevas aristas/nodos del grafo. La definición de caminos se hará desde los más sencillos a los más complicados.

- Camino 1: **1 - 2 - 13.**
- Camino 2: **1 - 2 - 3 - 13.**
- Camino 3: **1 - 2 - 3 - 4 - 5 - 9 - 10 - 11 - 12 - 2 - 13.**
- Camino 4: **1 - 2 - 3 - 4 - 6 - 10 - 11 - 12 - 2 - 13.**
- Camino 5: **1 - 2 - 3 - 4 - 7 - 11 - 12 - 2 - 13.**
- Camino 6: **1 - 2 - 3 - 4 - 8 - 12 - 2 - 13.**

b.4.- Casos de uso, resultados esperados y análisis

Caminos/ Casos de uso	Datos		Salidas
	sSalir	sAux	
1	“s”	Indistinto	Fin del programa.
2	“S”	Indistinto	Fin del programa.
3	Cualquiera diferente de 'S' y 's	3	“Los 3 primeros número de la serie de Fibonacci son: 0 1 1”
4	Cualquiera diferente de 'S' y 's	2	“Los 2 primeros número de la serie de Fibonacci son: 0 1”
5	Cualquiera diferente de 'S' y 's	1	“Los 1 primeros número de la serie de Fibonacci son: 0”
6	Cualquiera diferente de 'S' y 's	4	Número no permitido. Tiene que estar entre 1 y 3.

c.- Ejemplo 2. "Tablero para jugar al bingo"

El siguiente programa java:

```
public class Tablero
{
	String tab[][];
	String sNombre;

	public String PintTab(char cTipo, int iFil, int iCols, String sNomb)
	 {
		tab = new String[iFil][iCols];
		sNombre = "";
		int iNumPos = 0;

		if (cTipo == 'T') {
			sNombre = sNomb;
		} else if (cTipo == 'D') {
			sNombre = "CARTON";
		} else if (cTipo == 'D') {
			sNombre = "";
		}
		for (int i=0; i < iFil; i++)
		{
			for (int j=0; j<iCols; j++)
			{
				iNumPos++;
				tab[i][j] = "";
			}
		 }

		return sNombre + " tiene " + iNumPos + " posiciones";
	}
}
```

Para poder probar el método PintTab, será necesario escribir un método main que haga uso de él. Se proporciona el siguiente código de ejemplo para poder probar el método PintTab:

```
public class Test {

    public static void main(String[] args) {

        Tablero tablero=new Tablero();

         System.out.println(tablero.PintTab('T', 4, 5, "Mario"));

    }

}
```

Se crearía una clase llamada Test en el mismo paquete y se usaría esta clase para probar el método PintTab. En este ejemplo se prueba con los valores 'T',4,5 y "Mario"

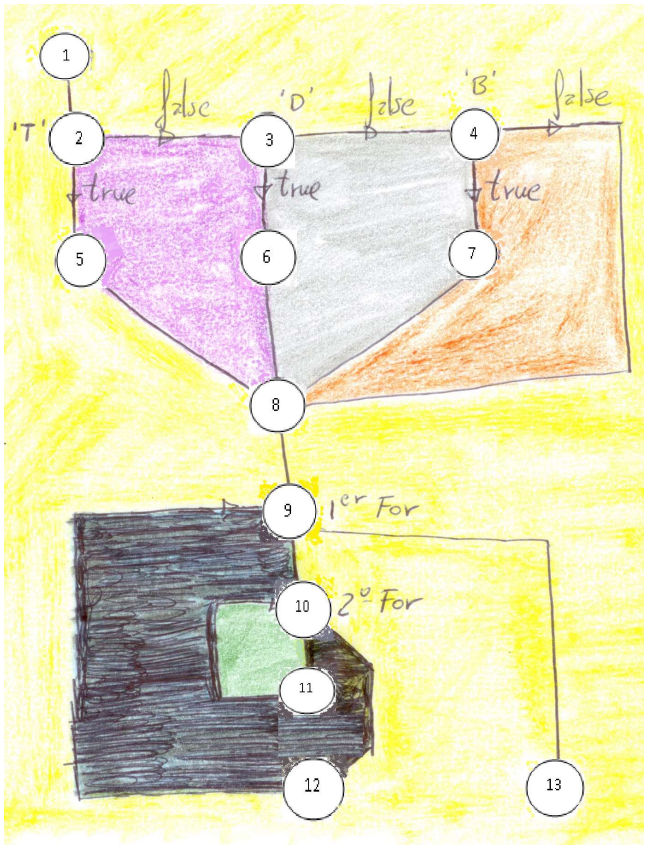
Este programa crea un tablero/cartón para jugar al bingo. Además cada cartón tendrá un nombre que dependiendo del valor cTipo será:

- 'D' - "CARTON".
- 'T' - El nombre será el valor del parámetro sNomb,
- 'B' - "" (vacío).

Y devuelve el número de posiciones que tiene el tablero y su nombre.

Pulsa aquí para descargarte dicho proyecto ("TableroBingoCajaBlanca.zip")..

c.1.- Creación del grafo

Grafo	Nodo	Línea - Condición
	1	Lin. 3 - 11.
	2	Lin. 12. Cond. cTipo == "T"
	3	Lin. 14. Cond. cTipo == "D"
	4	Lin. 16. Cond. cTipo == "D"
	5	Lin. 13
	6	Lin. 15
	7	Lin. 17
	8	Lin. 18. Toda condición if termina ahí.
	9	Lin. 19. Cond. i<iFil
	10	Lin. 21. Cond. <iCols
	11	Lin. 23-24
	12	Lin. 25
	13	Lin. 28-29

Notas:

- La vuelta a los for (nodos 9 y 10), se hace desde los nodos 12 y 11 respectivamente. Por otra parte, cuando la condición de los bucles no se cumple, la salida se realiza por las arista 9-13 y 10-12 en cada caso.
- El paso de los nodos 4 a 8 no aparece reflejado de forma explícita en el código

c.2.- Complejidad de McCabe o ciclomática

Calculada por los tres métodos posibles. Todos ellos deben dar el mismo resultado.

Método de cálculo	**Complejidad**	**Comentarios**
Nº de regiones	6	Hay que considerar la región exterior.
Nº de aristas - Nº nodos + 2	17 - 13 + 2 = 6	
Nº de condiciones + 1	5 + 1 = 6	Nodos 2, 3, 4, 9 y 10.

c.3.- Caminos de prueba

El número de caminos de prueba debe ser igual a la complejidad calculada. Cada nuevo camino deberá aportar el paso por nuevas aristas/nodos del grafo. La definición de caminos se hará desde los más sencillos a los más complicados.

- Camino 1: **1 - 2 - 5 - 8 - 9 - 13**.
- Camino 2: **1 - 2 - 3 - 6 - 8 - 9 - 13.**
- Camino 3: **1 - 2 - 3 - 4 - 7 - 8 - 9 - 13.**
- Camino 4: **1 - 2 - 3 - 4 - 8 - 9 - 13. Pasa por arista 4-8.**
- Camino 5: **1 - 2 - 5 - 8 - 9 - 10 - 12 - 9 - 13.**
- Camino 6: **1 - 2 - 5 - 8 - 9 - 10 - 11 - 10 - 12 - 9 - 13.**

c.4.- Casos de uso, resultados esperados y análisis

Caminos/ Casos de uso	Datos				Salidas
	cTipo	iFilas	iColum	sNombre	
1	T	0	0	Valeria	Valeria tiene 0 posiciones
2	D	0	0	Noa	CARTON tiene 0 posiciones
3	B	0	0	Mila	tiene 0 posiciones.
4	S	0	0	Miguel	tiene 0 posiciones.
5	T	1	0	Agus	Agus tiene 0 posiciones.
6	T	1	1	Raquel	Raquel tiene 1 posiciones

Anexo II.- Pruebas de la caja negra

El propósito de las **pruebas de caja negra o funcionales** es comprobar si las salidas que devuelve la aplicación son las esperadas en función de los parámetros de entrada.

Este tipo de prueba, no consideraría, en ningún caso, el código desarrollado, ni el algoritmo, ni la eficiencia, ni si hay partes del código innecesarias, etc. Estos aspectos son comprobados en las pruebas de caja blanca.

Cómo ya se explicó anteriormente, dentro de las pruebas funcionales, podemos indicar los siguientes **tipos**:

- **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.
- **Conjetura de errores**: trata de generar casos de prueba que la experiencia ha demostrado generan típicamente errores. En valores numéricos, un buen ejemplo es comprobar si funciona correctamente con el valor 0, ya que si es utilizado como denominador en alguna división podría generar un error en nuestro programa.

a.- Pasos a seguir

La técnica para determinar los casos de prueba de caja negra se realiza completando los siguientes pasos:

- Determinar **las clases de equivalencia**.
- Determinar un **análisis de valores límite**.
- **Conjetura de errores**.
- Generar los **caso de uso** necesarios para probar las clases válidas y no válidas. Establecer los datos de entrada y los resultados esperados.
- Lanzar una **ejecución del programa** por cada caso de uso y **comparar los resultados obtenidos con los esperados** para determinar la corrección del código.

a.1.- Determinar las clases de equivalencia

La técnica de **clases de equivalencia** es un tipo de prueba funcional, donde en cada caso de prueba se agrupa el mayor número de entradas posibles. A partir de aquí, se asume que 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.

Los **pasos** a seguir para identificar las clases de equivalencia son:

- **Identificar las condiciones de las entradas del programa**, es decir, restricciones de formato o contenido de los datos de entrada.
- A partir de ellas, **identificar clases de equivalencia** que pueden ser:
  - De datos válidos.
  - De datos no válidos o erróneos.

Existen algunas **reglas** que ayudan a identificar las clases:

Tipo de dato	Ejemplo	Clases equivalencia
Rango de valores de entrada. Crear una clase válida y dos clases no válidas.	La edad de acceso a un evento está comprendida entre 18 y 100 años.	**Clase válida:**
		Valor entre 18 - 100
		Clases no válidas:
		Menor de 18. Mayor de 100.
Número finito y consecutivo de valores. Creará una clase válida y dos no válidas.	Una encuesta puede ser valorada con los valores 0, 1, 2, 3.	**Clase válida:**
		Cualquiera de los valores 0,1,2,3
		Clases no válidas:
		Menor de 0. Mayor de 3.
Condición **verdadero/falso**.	Una persona tiene la condición de ser mayor de edad.	**Clase válida:**
		Edad >=18
		Clase no válida:
		Edad<18
Conjunto de valores admitidos. Se identifica una clase válida por cada valor y una no válida.	Una opción de menú puede aceptar los valores 'A' para altas, 'B' para bajas y 'S' para salir del programa.	**Clases validas:**
		Opción 'A' Opción 'B' Opción 'S'
		Clase no válida:
		Opción 'J'

En cualquier caso, si se sospecha que ciertos elementos de una clase no se tratan igual que el resto de la misma, deben dividirse en clases menores.

a.2.- Análisis de valores límite

La experiencia indica que los casos de prueba que exploran las condiciones límite de un programa producen un mejor resultado para detectar defectos.

El **AVL (Análisis de valores límite)** es una técnica de diseño de casos de prueba que complementa a la de particiones de equivalencia.

La principal diferencia se encuentra en el tratamiento que tienen las clases de equivalencia de rango de valores y de número finito y consecutivo de valores. Ahora la prueba se realizará sobre los valores límite de los rangos.

Ejemplo	Clase de equivalencia	Valores límite
La edad de acceso a un evento está comprendida entre 18 y 100 años.	**Clase valida:**	**Clases válidas:**
	Valor entre 18 - 100. Caso único. P.e 30	Caso 1: 18 Caso 2: 100
	Clases no válidas:	**Clases no válidas:**
	Menor de 18. P.e 15 Mayor de 100. P.e. 110	Menor de 18. 17 Mayor de 100. 101
Una encuesta puede ser valorada con los valores 0, 1, 2, 3.	**Clase valida:**	**Clases válidas:**
	Cualquier de los valores 0,1,2,3 Caso único. P.e 2	Caso 1: 0 Caso 2: 3
	Clases no válidas:	**Clases no válidas:**
	Menor de 0. P.e -10 Mayor de 3. P.e 7	Valor -1 Valor 4

En las pruebas AVL también habría que generar casos de prueba atendiendo a clases de equivalencia de los datos de salida.

```
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 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.

Autoevaluación

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

- ○ 4 y 11
- ○ 4,99 y 11
- ○ 4,99 y 9,99.

Incorrecta. El valor de la variable es real, el 4 y el 11 no son valores límite

No es correcta. El 4,99 si es valor límite para la primera parte de la condición, pero 11 no.

Muy bien. En este caso, estos dos valores, pertenecen al conjunto de valores límite.

Solución

1. Incorrecto
2. Incorrecto
3. Opción correcta

a.3.- Conjetura de errores

La experiencia en la fase de pruebas indica que existen ciertos valores de entrada que típicamente son generadores de errores, y que en ocasiones, pasan desapercibidos en las técnicas de clases de equivalencia y de valores límite.

La **conjetura de errores** considera esos datos y define nuevos casos de prueba a los que someter a los programas. Se trata de una técnica menos métodica que las anteriores, tiene mucho más que ver con la **intuición y experiencia** del programador.

Una prueba típica en conjetura de errores es probar el valor de entrada 0 para datos numéricos por si pudiera participar como denominador en alguna división durante la ejecución del programa.

a.4.- Casos de uso, resultados esperados y análisis.

Ahora toca definir los datos de entrada al programa. Al conjunto de entradas al programa utilizados para cada ejecución se le denomina **caso de uso**. Los casos de uso se generarán a partir de las clases de equivalencia, valores límite y conjeturas de errores obtenidos en los apartados anteriores.

Este proceso consta de las siguientes **fases**:

- **Numerar las clase de equivalencia**.
- **Crear casos de uso que cubran todas las clases de equivalencia válidas**. Se intentará agrupar en cada caso de uso tantas clases de equivalencia como sea posible.
- **Crear un caso de uso para cada clase de equivalencia no válida**.

Además toca definir los **resultados previstos** en cada ejecución. Cuando posteriormente se lance la ejecución del programa para cada caso de uso, los resultados obtenidos serán comparados con los esperados y así determinar la corrección del código.

b.- Ejemplo práctico

Indica:

- Las clases válidas y las clases no válidas.
- Todos los casos de prueba.

Siguiendo los criterios de las pruebas funcionales o caja negra.

De un programa que pide que indique una de estas dos palabras: “amigos”, “visDivisores”. Si no inserta ninguna de estas dos cosas, se cerrará el programa, visualizando este mensaje:

“Lo siento, no se ha indicado la operación adecuada. Cerramos el programa”. (Mensaje 1)

Si indica una de estas dos cosas (“amigos” ´o “visDivisores”) se pedirá un primer número.

Dicho número tiene que ser par y positivo y, por supuesto, que no inserte letras.

Si dicho número:

- No es par y positivo, se visualizará un mensaje como este:

“El primer número no es par y positivo. Cerramos el programa”. (Mensaje 2)

Y se cerrará el programa.

- Si se inserta letras, se visualizará un mensaje como este:

“Ha insertado letras en la indicación del primer número. Cerramos el programa”. (Mensaje 3)

Y se cerrará el programa.

Si el número es correcto, se pedirá un segundo número que tiene que estar entre 3000 y 5000 (incluido el 3000 pero no el 5000) y, por supuesto, que no inserte letras (en la inserción de este segundo número).

Si este segundo número:

- No está entre 3000 y 5000, se visualizará un mensaje como este:

“El segundo número no está entre 3000 y 5000. Cerramos el programa”. (Mensaje 4)

Y se cerrará el programa.

- Si inserta letras, en la indicación del segundo número, se visualizará un mensaje como este;

“Ha insertado letras en la indicación del segundo número. Cerramos el programa”. (Mensaje 5)

Y se cerrará el programa.

Si indica bien el segundo número se visualizará:

- Si estos dos número son amigos

Dos números son amigos si tienen el mismo número de divisores, sin contar el 1 y el propio número.

O

- Los divisores de cada número.

Según se haya indicado al principio

b.1.- Determinar las clases de equivalencia y analizar los valores límite

Condición de entrada	Clases de equivalencia	Clases válidas	COD	Clases no válidas	COD
Operación	Conjunto de valores admitidos. Se identifica una clase válida por cada valor y una no válida.	“Amigos”	C1B1	Algo distinto de “Amigos” o “Divisores”	C1E
		“Divisores”	C1B2		
Primer número	Condición verdadero/falso	Un dato que sea par y positivo	C2B	Un dato que no sea par y positivo	C2E1
				Inserte letras	C2E2
Segundo número	Rango de valores de entrada.	Un dato entre 3000 y 5000	C3B1	Un dato menor de 3000	C3E1
		Valor límite de 3000(es válido)	C3B2	Valor límite de 5000(no es válido)	C3E2
				Un dato mayor de 5000	C3E3
				Inserte letras	C3E4

b.2.- Casos de uso, resultados esperados y análisis

Casos de prueba	Clases de equivalencia	Condiciones de entrada			Resultado esperado
		Operación	Primer número	Segundo número	
CP1	C1B1,C2B,C3B1	“Amigos”	18	3000	Indicar si son amigos el 18 y el 3000
CP2	C1B1,C2B,C3B2	“Amigos”	18	**4000**	Indicar si son amigos el 18 y el 4000
CP3	C1B2,C2B,C3B1	“Divisores”	18	3000	Visual. Los divis.de 18 y 3000
CP4	C1B2,C2B,C3B2	“Divisores”	18	**4000**	Visual. Los divis.de 18 y 4000
CP5	C1E	“Paula”	----	----	Mensaje 1
CP6	C1B1,C2E1	“Amigos”	-15	----	Mensaje 2
CP7	C1B1,C2E2	“Amigos”	“Hola”	----	Mensaje 3
CP8	C1B2,C2E1	“Divisores”	-15	----	Mensaje 2
CP9	C1B2,C2E2	“Divisores”	“Hola”	----	Mensaje 3
CP10	C1B1,C2B,C3E1	“Amigos”	18	2000	Mensaje 4
CP11	C1B1,C2B,C3E2	“Amigos”	18	5000	Mensaje 4
CP12	C1B1,C2B,C3E3	“Amigos”	18	6000	Mensaje 4
CP13	C1B1,C2B,C3E4	“Amigos”	18	“Adiós”	Mensaje 5
CP14	C1B2,C2B,C3E1	“Divisores”	18	2000	Mensaje 4
CP15	C1B2,C2B,C3E2	“Divisores”	18	5000	Mensaje 4
CP16	C1B2,C2B,C3E3	“Divisores”	18	6000	Mensaje 4
CP17	C1B2,C2B,C3E4	“Divisores”	18	“Adiós”	Mensaje 5

Habría que insertar números que no son amigos entre si o que no tengan divisores, pero eso no lo establece la caja negra. Siempre se puede dar cuenta el testeador y tenerlo en cuenta,

Anexo III. JUnit

En los anexo anteriores se veía que entre los diferentes tipos de pruebas a los que someter los desarrollos software están las **pruebas unitarias**.

JUnit es una herramienta que ayuda a pasar de forma automática estas pruebas a los métodos y clases de los programas **Java**.

Para ello, habrá que identificar los casos de uso, datos de entrada y resultados esperados con las técnicas de caja blanca y negra ya conocidos. Una vez esté disponible esta información, toca lanzar con **JUnit** las ejecuciones programadas y valorar si los resultados obtenidos son exitosos (de acuerdo a lo previsto).

a.- Probando JUnit

Este manual describe de forma introductoria conceptos relacionados con **JUnit** en su **versión 5**, tales como:

- Etiquetas disponibles para configurar pruebas en Java (**anotaciones**).
- Llamadas a métodos de prueba para comprobar el funcionamiento del código (**assertions**).
- Aprender a utilizar el **plugging JUnit** disponible en Eclipse.

En las siguientes **urls** podrás encontrar información útil referida a **JUnit** y su uso en el **IDE Eclipse**:

Propósito	**URL**
Guía de uso de JUnit 5	https://junit.org/junit5/docs/current/user-guide/
Uso de JUnit 5 en Eclipse	https://www.eclipse.org/eclipse/news/4.7.1a/#junit-5-support

a.1.- Código propuesto

Vamos a usar JUnit con NetBeans. Desde la versión 7, ya lo tiene incorporado.

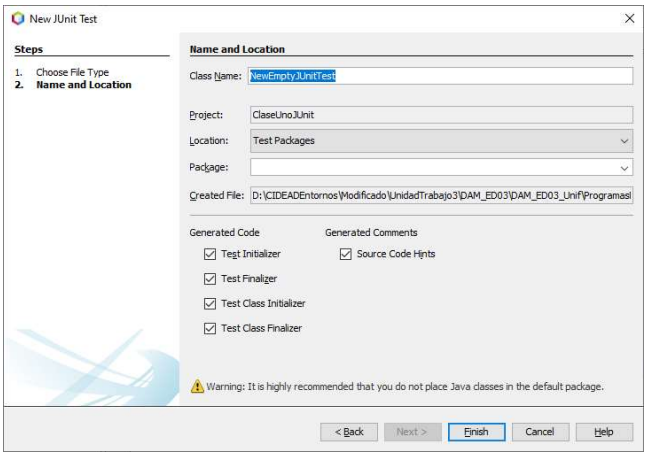
Para probar JUnit, vamos a crear un nuevo proyecto (en NetBeans) con este código:

```
package claseunojunit_;


public class ClaseUnoJunit_ {
int dato1,dato2;
public ClaseUnoJunit_(int dato1, int dato2) {
this.dato1=dato1;
this.dato2=dato2;
}
public int suma()
{
return dato1+dato2;
}
public int resta()
{
return dato1-dato2;
}

public int dividir()
{
 return dato1%dato2;
}



}
```

Pulsa aquí para bajarte dicho proyecto ("ClaseUnoJunit_.zip");.

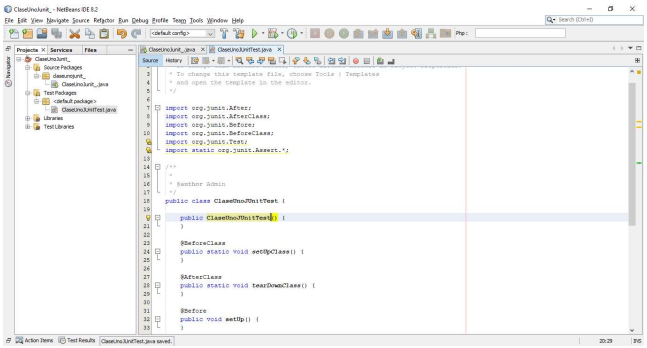
Posteriormente, sobre el nombre del proyecto. En el menú contextual, que aparece, elegir la opción new/JUnit Test.

Se visualizará esta ventana:



Os aconsejo poner el nombre de la clase (a probar) acabado en Test.

Vais a ver que os creado un fichero java para hacer las pruebas, en el paquete “Test packages”:


ClaseUnoJunit_ - NetBeans IDE 8.2
File Edit View Navigate Source Refactor Run Debug Profile Team Tools Window Help
Projects Services Files
ClaseUnoJunit_
Source Packages
claseunojunit_
ClaseUnoJunit_.java
Test Packages
<default package>
ClaseUnoJUnitTest.java
Libraries
Test Libraries
ClaseUnoJunit_.java
ClaseUnoJUnitTest.java
Source History
* To change this template file, choose Tools | Templates
* and open the template in the editor.
*/
import org.junit.After;
import org.junit.AfterClass;
import org.junit.Before;
import org.junit.BeforeClass;
import org.junit.Test;
import static org.junit.Assert.*;
/**
*
* @author Admin
*/
public class ClaseUnoJUnitTest {
public ClaseUnoJUnitTest() {
}
@BeforeClass
public static void setUpClass() {
}
@AfterClass
public static void tearDownClass() {
}
@Before
public void setUp() {
}
Action Items
Test Results
ClaseUnoJUnitTest.java saved.
20:29
INS

a.2.- Etiquetas JUnit

.En esta sección se describen las utilizadas con mayor frecuencia.

Estas serían las etiquetas más utilizadas:

- **@BeforeClass:** Sólo puede haber un método con este marcador, es invocado una vez al principio de todas los test. Se suele usar para inicializar atributos.
- **@AfterClass**: Sólo puede haber un método con este marcador y se invoca al finalizar todas los test.
- **@Before**: Se ejecuta antes de de cada test.
- **@After**: Se ejecuta después de cada test.
- **@Ignore**: Los métodos marcados con esta anotación no serán ejecutados.
- **@Test**: Identifica los métodos test que deben ser probados

La que mas usaremos sera **BeforeClass y Test**, pero siempre nos puede venir venir bien usar alguna de las anotaciones anteriores.

Unas anotaciones a destacar son @beforeclass y @afterclass, tienen algunas diferencias respecto a las anteriores:

- @beforeclass: solo puede haber un método con esta etiqueta. El método marcado con esta anotación es invocado una vez al principio del lanzamiento de todas las pruebas. Se suele utilizar para inicializar atributos comunes a todas las pruebas o para realizar acciones que tardan un tiempo considerable en ejecutarse.
- @afterclass: solo puede haber un método con esta anotación. Este método será invocado una sóla vez cuando finalicen todas las pruebas.

Este es el código que te ha generado NetBeans:

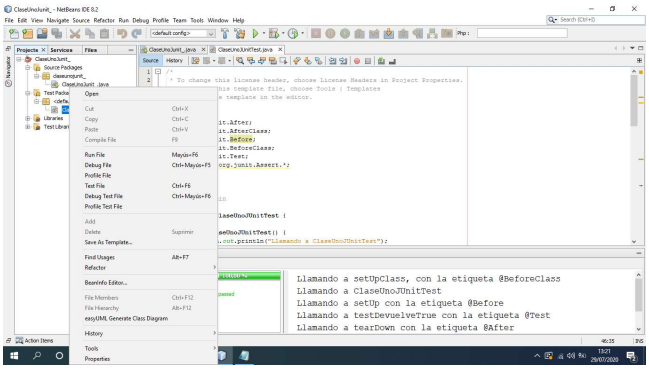
```
/*
* To change this license header, choose License Headers in Project Properties.
* To change this template file, choose Tools | Templates
* and open the template in the editor.
*/

import org.junit.After;
import org.junit.AfterClass;
import org.junit.Before;
import org.junit.BeforeClass;
import org.junit.Test;
import static org.junit.Assert.*;

/**
*
* @author Admin
*/

public class ClaseUnoJUnitTest_ {

public ClaseUnoJUnitTest_() {


}

@BeforeClass
public static void setUpClass() {
}

@AfterClass
public static void tearDownClass() {
}

@Before
public void setUp() {
}

@After
public void tearDown() {
}
```

```
// TODO add test methods here.
// The methods must be annotated with annotation @Test. For example:
//
// @Test
// public void hello() {}
}
```

En dicho código he incluido una serie de mensajes para ver en que orden se ejecutan los métodos. Este sería el código:

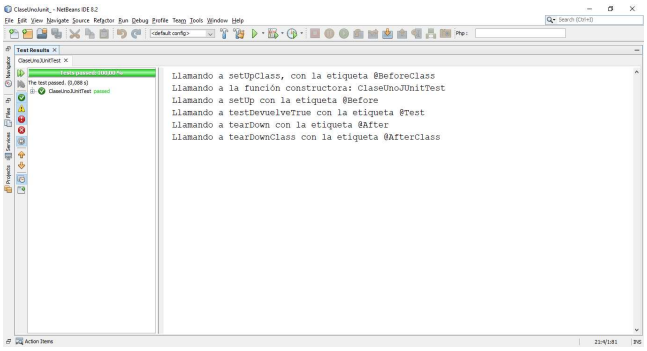
```
/*
* To change this license header, choose License Headers in Project Properties.
* To change this template file, choose Tools | Templates
* and open the template in the editor.
*/

import org.junit.After;
import org.junit.AfterClass;
import org.junit.Before;
import org.junit.BeforeClass;
import org.junit.Test;
import static org.junit.Assert.*;

/**
*
* @author Admin
*/
public class ClaseUnoJUnitTest_ {

public ClaseUnoJUnitTest_() {
  System.out.println("Llamando a la función constructora: ClaseUnoJUnitTest");
}

@BeforeClass
public static void setUpClass() {
System.out.println("Llamando a setUpClass, con la etiqueta @BeforeClass");

}

@AfterClass
public static void tearDownClass() {
System.out.println("Llamando a tearDownClass con la etiqueta @AfterClass");
}

@Before
public void setUp() {

System.out.println("Llamando a setUp con la etiqueta @Before");
}

@After
public void tearDown() {
System.out.println("Llamando a tearDown con la etiqueta @After");
}

// TODO add test methods here.
// The methods must be annotated with annotation @Test. For example:
//
// @Test
// public void hello() {}
@Test
public void testDevuelveTrue() {

System.out.println("Llamando a testDevuelveTrue con la etiqueta @Test");
}


}
```

Para ejecutarlo, tienes que dar al botón derecho del ratón:



Y eliges la opción "Run File".

La salida sería esta:



Así puedes ver en que orden se ejecutan los métodos.

a.3.- Assertions (afirmaciones).

En el apartado anterior se indica que las llamadas a los métodos de prueba son configurables en diferentes aspectos, tales como: qué hacer antes/después de cada prueba, cómo pasar parámetros a las llamadas o cómo repetir de forma automática un determinado test. Pero en realidad, no hemos lanzado prueba alguna a los métodos del código del proyecto implementado en la clase ClaseUnoJUnit.

JUnit utiliza la **clase Assertions** para lanzar los **test**, que básicamente está compuesta por una serie de métodos, que una vez llamados ejecutan los métodos a probar y analizan su comportamiento comparándolos con los resultados que se espera de ellos.

Así, hay métodos que nos permiten comprobar si dos valores son o no iguales, si el valor del parámetro pasado se puede resolver como true o false, si el tiempo consumido en ejecutar un método supera el previsto, etc.

Además, los métodos están sobrecargados, permitiendo en algunos casos indicar el mensaje que ha de devolver si la comprobación no resulta exitosa, o definir un margen que valide dos números como iguales si su diferencia es inferior a dicha tolerancia .

Ahora vamos a ver los métodos que usaremos, disponibles en JUnit, todos se invocan con la clase **Assert** de forma estático. Los mas usados son:

- **assertEquals(resultado esperado, resultado actual):** le pasamos el resultado que nosotros esperamos y invocamos la función que estamos testando.
- **assertNull(objeto):** si un objeto es null el test sera exitoso.
- **assertNotNull(objeto):** al contrario que el anterior.
- **assertTrue(condición):** si la condición pasada (puede ser una función que devuelva un booleano) es verdadera el test sera exitoso.
- **assertFalse(condición):** si la condición pasada (puede ser una función que devuelva un booleano) es falsa el test sera exitoso.
- **assertSame(Objeto1, objeto2):** compara las referencias de los objetos.
- **assertNotSame(Objeto1, objeto2):** al contrario que el anterior.

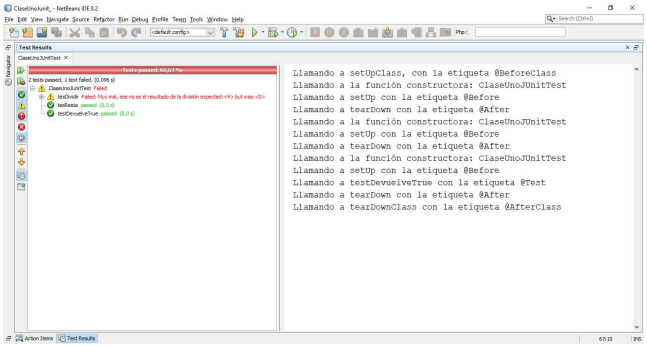
Para entender mejor los **métodos assert**, se va a modificar la clase ClaseUnoJUnitTest nuevamente, sustituyendo los mensajes por pantalla que se mostraban en la sección anterior por llamadas assert que permitan probar los métodos de la clase ClaseUnoJUnit_.

Nuevo código de la ClaseUnoJUnitTest.java:

```
import org.junit.After;
import org.junit.AfterClass;
import org.junit.Before;
import org.junit.BeforeClass;
import org.junit.Test;
import static org.junit.Assert.*;
import claseunojunit_.ClaseUnoJunit_;

/**
*
* @author Admin
*/
public class ClaseUnoJUnitTest {

ClaseUnoJunit_ calc;

public ClaseUnoJUnitTest() {
System.out.println("Llamando a la función constructora: ClaseUnoJUnitTest");
}

@BeforeClass
public static void setUpClass() {
System.out.println("Llamando a setUpClass, con la etiqueta @BeforeClass");

}

@AfterClass
public static void tearDownClass() {
System.out.println("Llamando a tearDownClass con la etiqueta @AfterClass");
}
```

```
@Before
public void setUp() {

System.out.println("Llamando a setUp con la etiqueta @Before");

calc=new ClaseUnoJunit_(4, 67);
}

@After
public void tearDown() {
System.out.println("Llamando a tearDown con la etiqueta @After");
}

// TODO add test methods here.
// The methods must be annotated with annotation @Test. For example:
//
// @Test
// public void hello() {}
@Test
public void testDevuelveTrue() {

System.out.println("Llamando a testDevuelveTrue con la etiqueta @Test");
}

public void testSuma() {

assertEquals(71, calc.suma());

}

@Test
public void tesResta()
{

assertEquals(-63, calc.resta());
}
@Test
public void texDividir()
{
ClaseUnoJunit_ calc=new ClaseUnoJunit_(100, 25);
// assertTrue(calc.dividir()==1);
assertEquals("Muy mal, ese no es el resultado de la división",4, calc.dividir());//Lanza ese mensaje si no cuadra
}

}
```

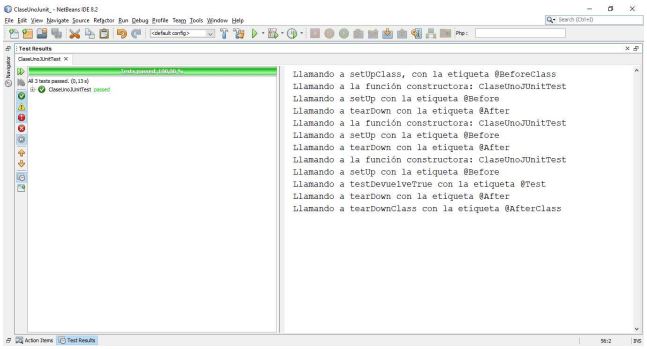
Ejecutamos dicha clase y vemos que el resultado de dicha ejecución es:



Donde nos da información de que ha ejecutado los métodos que se le ha pedido. Y te confirma que el resultado era el que esperabas menos el de la división. Pusimos que tenia que devolver un 4 y devuelve un 0.

Al ver ese error, vamos a dicho método y comprobamos que efectivamente el método dividir está mal. Pusimos el operador "%" que da el resto de una división y había que haber puesto "/" que es el operador en Java que devuelve el cociente de una división.

Si corregimos ese fallo y volvemos a ejecutar el Test veremos que el resultado es este:


Test Results
ClaseUnoJUnitTest
All 3 tests passed. (0,13 s)
ClaseUnoJUnitTest passed
Llamando a setUpClass, con la etiqueta @BeforeClass
Llamando a la función constructora: ClaseUnoJUnitTest
Llamando a setUp con la etiqueta @Before
Llamando a tearDown con la etiqueta @After
Llamando a la función constructora: ClaseUnoJUnitTest
Llamando a setUp con la etiqueta @Before
Llamando a tearDown con la etiqueta @After
Llamando a la función constructora: ClaseUnoJUnitTest
Llamando a setUp con la etiqueta @Before
Llamando a testDevuelveTrue con la etiqueta @Test
Llamando a tearDown con la etiqueta @After
Llamando a tearDownClass con la etiqueta @AfterClass

Anexo IV.- Depuración con Eclipse

Para poder comprobar algunas de las **funciones de depuración** que nos ofrece **Eclipse**, vamos a crear un proyecto en **Java** con el siguiente código:

Clase Test

```
public class Test {
	public static void main(String[] args) {
		Contador contador = new Contador();
		contador.contar();
		System.out.println("Cuenta: " + contador.getResultado());
	 }
}
```

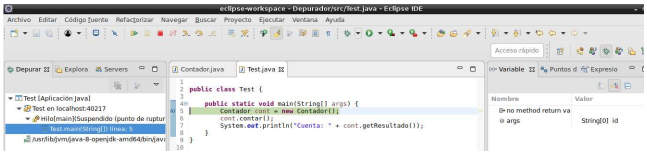
Clase Contador

```
public class Contador {
	private int resultado = 0;
	public int getResultado()
	{
		return resultado;
	}

	public void contar()
	{
		for(int i=0; i < 100 ; i++)
		{
			resultado = resultado + i + 1;
		}
	}
}
```

Pulse aquí para descargarte dicho proyecto ("Depuracion-sw.rar").

Existen varias alternativas para lanzar la ejecución de un programa en modo debug. Una de ellas es pulsando con el botón derecho del ratón sobre la clase de inicio del proyecto (implementa el método main), y seleccionar en el menú contextual que aparece "**Depurar como => Aplicación Java**".

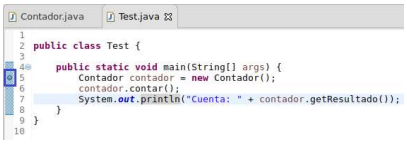
En **modo depuración**, **Eclipse** da la opción de trabajar con la perspectiva depurar, que ofrece una serie de vistas muy interesantes para este tipo de ejecución, tales como: la vista de visualización y cambio de variables, la vista de puntos de parada establecidos o la pila de llamadas entre otras.



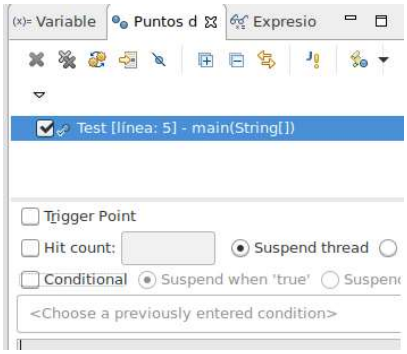
a.- Puntos de ruptura.

Al solicitar una ejecución en modo debug, si no hemos establecido puntos de parada en el código, éste ejecutará hasta el final del mismo modo que lo haría en una ejecución normal.

Para establecer un **punto de parada**, basta con hacer doble clic en el margen izquierdo de la línea de código donde se va a establecer.



El punto de parada es dado de alta y ya aparece en la **vista de puntos de parada**.



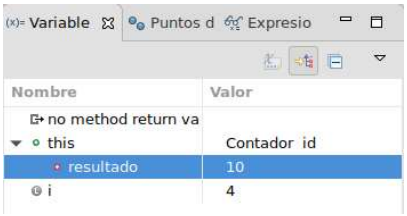
Eclipse permite definir puntos de parada condicionales para personalizar cuándo o porqué se para el programa. Si por ejemplo, seleccionamos en un punto de parada la **opción Hit count** y determinamos un valor, el programa parará cuando haya pasado por este punto el número de veces indicado.

Es posible crear condiciones más elaboradas dependientes de otras variables disponibles en el contexto de la ejecución. El programa sólo parará cuando la condición definida se cumpla y la ejecución pase por esa línea de código.

En el momento que tenemos detenido el programa, 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 realizada la comprobación, podemos abortar el programa, o continuar la ejecución normal del mismo.

b.- 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 chequear que las **variables** vayan tomando los valores adecuados en cada momento. **Eclipse** nos proporciona la **vista variables** donde podemos ir comprobando el valor que van tomando las variables activas en la zona de código donde está el programa parado.



Además, pulsando sobre el valor de cualquiera de estas variables es posible modificarlas. Esto nos permite evaluar nuevos escenarios de prueba con datos diferentes.

c.- Botones de depuración

Cuando estamos en el proceso de depuración de un programa con la **perspectiva Depurar**, **Eclipse** nos ofrece una serie de botones en su **barra de herramientas** que pasamos a describir a continuación.

	Ctrl + Alt + B	Desactiva temporalmente todos los puntos de parada del código.
	F8	Continua la ejecución del programa. Se detendrá en el siguiente punto de parada.
		Pausa/detiene la ejecución del programa en el punto de código donde se encuentre al ser pulsado.
	Ctrl + F2	Finaliza la ejecución del programa.
		Función no considerada en este manual. Consultar manual de Eclipse.
	F5	Si el programa se encuentra detenido en la llamada a un método, al pulsar este botón la ejecución pasa a la primera línea del mismo.
	F6	Si el programa se encuentra detenido en la llamada a un método, al pulsar este botón la ejecución del método se hace por completo, sin depurar su implementación.
	F7	Avanza la ejecución del programa hasta que nos salimos del método actual y vamos hasta el sitio donde fue llamado.

Anexo V.- Enlaces de interés

Algunos enlaces de interés.

Metodología métrica.

https://administracionelectronica.gob.es/pae_Home/pae_Documentacion/pae_Metodolog/pae_Metrica_v3.html#.W3u0j7i-nIU
Portal de Administración Electrónica Ministerio de Política Territorial y Función Pública Secretaría General de Administración Digital.

Metodologías ágiles o DevOps (Desarrollo y operación).

Se tratan de metodologías más actuales que buscan minimizar los tiempos desde que se adquiere el compromiso en un desarrollo hasta que se encuentra disponible en operación.

https://es.wikipedia.org/wiki/DevOps
Esta obra contiene una traducción parcial derivada de *DevOps* de Wikipedia en inglés, publicada por sus editores bajo la Licencia de documentación libre de GNU y la Licencia Creative Commons Atribución-CompartirIgual 3.0 Unported.

Caja blanca.

https://www.youtube.com/watch?v=GVegCwwfBZ0
Canal de Youtube de Juan V. Carrillo publicado con Licencia Atribución de Creative Commons.

https://www.youtube.com/watch?v=9N5vPeSWRfQ
Canal de Youtube de Juan V. Carrillo publicado con Licencia Atribución de Creative Commons.

https://www.youtube.com/watch?v=iILI-n57IEs
Canal de Youtube de Juan V. Carrillo publicado con Licencia Atribución de Creative Commons.

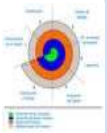
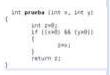
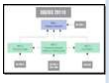
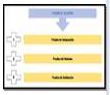
Caja negra.

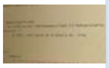
https://www.youtube.com/watch?v=PmdFMDZVmmM

Canal de Youtube de Juan V. Carrillo publicado con Licencia Atribución de Creative Commons.

Anexo VI.- Licencias de recursos.

Licencias de recursos utilizados en la Unidad de Trabajo.

Recurso (1)	Datos del recurso (1)	Recurso (2)	Datos del recurso (2)
	Autoría:Ebnz Licencia:Creative Commons. Genérica de Atribución/Compartir-Igual 3,0 Procedencia:Montaje sobre http://es.wikipedia.org/		Autoría: Oracle Corporation Licencia:Copyright cita Procedencia:Captura de pantalla de Netbeans
	Autoría:Oracle Corporation Licencia: Copyright cita Procedencia: Captura de pantalla de Netbeans		Autoría:Oracle Corporation Licencia:Copyright cita Procedencia:Captura de pantalla de Netbeans
	Autoría: Oracle Corporation. Licencia: Copyright cita Procedencia: Captura de pantalla de Netbeans		Autoría: Francisco Javier Cabrerizo Membrilla. Licencia: Uso educativo no comercial. Procedencia: Elaboración propia.
	Autoría: Francisco Javier Cabrerizo Membrilla. Licencia: Uso educativo no comercial. Procedencia: Elaboración propia.		Autoría: Francisco Javier Cabrerizo Membrilla. Licencia: Uso educativo no comercial. Procedencia: Elaboración propia.
	Autoría:Oracle Corporation Licencia:Copyright cita Procedencia:Captura de pantalla de Netbeans		Autoría:Scott Schram. Licencia:CC by dominio público. Procedencia:www.flickr.com
	Autoría:Oracle Corporation Licencia:Copyright cita Procedencia:Captura de pantalla de Netbeans		Autoría:JaulaDeArdilla Licencia:CC by-nc-nd Procedencia:http://www.flickr.com/photos/jauladeardilla/2285620559/
	Autoría: Ministerio de Educación. Licencia: Uso Educativo no comercial. Procedencia: Elaboración propia.		Autoría: Ministerio de Educación. Licencia: Uso Educativo no comercial. Procedencia: Elaboración propia.
	Autoría: Ministerio de		Autoría: Ministerio de Educación.

	Educación. Licencia: Uso Educativo no comercial. Procedencia: Elaboración propia.		Licencia: Uso Educativo no comercial. Procedencia: Elaboración propia.
	Autoría: Ministerio de Educación. Licencia: Uso Educativo no comercial. Procedencia: Elaboración propia.		Autoría: Ministerio de Educación. Licencia: Uso Educativo no comercial. Procedencia: Elaboración propia.
	Autoría: Ministerio de Educación. Licencia: Uso Educativo no comercial. Procedencia: Elaboración propia.		Autoría: Ministerio de Educación. Licencia: Uso Educativo no comercial. Procedencia: Elaboración propia.
	Autoría: Ministerio de Educación. Licencia: Uso Educativo no comercial. Procedencia: Elaboración propia.		Autoría: Ministerio de Educación. Licencia: Uso Educativo no comercial. Procedencia: Elaboración propia.
	Autoría: Ministerio de Educación. Licencia: Uso Educativo no comercial. Procedencia: Elaboración propia.		Autoría: Ministerio de Educación. Licencia: Uso Educativo no comercial. Procedencia: Elaboración propia.
	Autoría: Ministerio de Educación. Licencia: Uso Educativo no comercial. Procedencia: Elaboración propia		Autoría: Ministerio de Educación. Licencia: Uso Educativo no comercial. Procedencia: Elaboración propia
	Autoría: Ministerio de Educación. Licencia: Uso Educativo no comercial. Procedencia: Elaboración propia		Autoría: Ministerio de Educación. Licencia: Uso Educativo no comercial. Procedencia: Elaboración propia
	Autoría: Ministerio de Educación. Licencia: Uso Educativo no comercial. Procedencia: Elaboración propia		Autoría: Ministerio de Educación. Licencia: Uso Educativo no comercial. Procedencia: Elaboración propia
	Autoría: Ministerio de Educación. Licencia: Uso Educativo no comercial. Procedencia: Elaboración propia		Autoría: Ministerio de Educación. Licencia: Uso Educativo no comercial. Procedencia: Elaboración propia

	Autoría: Ministerio de Educación. Licencia: Uso Educativo no comercial. Procedencia: Elaboración propia		Autoría: Ministerio de Educación. Licencia: Uso Educativo no comercial. Procedencia: Elaboración propia
	Autoría: Ministerio de Educación. Licencia: Uso Educativo no comercial. Procedencia: Elaboración propia		Autoría: Ministerio de Educación. Licencia: Uso Educativo no comercial. Procedencia: Elaboración propia
	Autoría: Ministerio de Educación. Licencia: Uso Educativo no comercial. Procedencia: Elaboración propia		Autoría: Ministerio de Educación. Licencia: Uso Educativo no comercial. Procedencia: Elaboración propia
	Autoría: Ministerio de Educación. Licencia: Uso Educativo no comercial. Procedencia: Elaboración propia		Autoría: Ministerio de Educación. Licencia: Uso Educativo no comercial. Procedencia: Elaboración propia
	Autoría: Ministerio de Educación. Licencia: Uso Educativo no comercial. Procedencia: Elaboración propia		Autoría: Ministerio de Educación. Licencia: Uso Educativo no comercial. Procedencia: Elaboración propia
	Autoría: Ministerio de Educación. Licencia: Uso Educativo no comercial. Procedencia: Elaboración propia		Autoría: Ministerio de Educación. Licencia: Uso Educativo no comercial. Procedencia: Elaboración propia
	Autoría: Ministerio de Educación. Licencia: Uso Educativo no comercial. Procedencia: Elaboración propia		Autoría: Ministerio de Educación. Licencia: Uso Educativo no comercial. Procedencia: Elaboración propia
	Autoría: Ministerio de Educación. Licencia: Uso Educativo no comercial. Procedencia: Elaboración propia		Autoría: Ministerio de Educación. Licencia: Uso Educativo no comercial. Procedencia: Elaboración propia
	Autoría: Ministerio de Educación. Licencia: Uso Educativo no comercial. Procedencia: Elaboración propia		Autoría: Ministerio de Educación. Licencia: Uso Educativo no comercial. Procedencia: Elaboración propia

	Autoría: Ministerio de Educación. Licencia: Uso Educativo no comercial. Procedencia: Elaboración propia		Autoría: Ministerio de Educación. Licencia: Uso Educativo no comercial. Procedencia: Elaboración propia
	Autoría: Ministerio de Educación. Licencia: Uso Educativo no comercial. Procedencia: Elaboración propia		Autoría: Ministerio de Educación. Licencia: Uso Educativo no comercial. Procedencia: Elaboración propia
	Autoría: Ministerio de Educación. Licencia: Uso Educativo no comercial. Procedencia: Elaboración propia		Autoría: Ministerio de Educación. Licencia: Uso Educativo no comercial. Procedencia: Elaboración propia
	Autoría: Oracle Corporation. Licencia: Copyright cita. Procedencia: Captura de pantalla de Netbeans.		Autoría: Oracle Corporation. Licencia: Copyright cita. Procedencia: Captura de pantalla de Netbeans.
	Autoría: Oracle Corporation. Licencia: Copyright cita. Procedencia: Captura de pantalla de Netbeans.		Autoría: Oracle Corporation. Licencia: Copyright cita. Procedencia: Captura de pantalla de Netbeans.
	Autoría: Oracle Corporation. Licencia: Copyright cita. Procedencia: Captura de pantalla de Netbeans.		Autoría: Oracle Corporation. Licencia: Copyright cita. Procedencia: Captura de pantalla de Netbeans.
	Autoría: Oracle Corporation. Licencia: Copyright cita. Procedencia: Captura de pantalla de Netbeans.		