Tema 4: Testing

¿Qué aprenderás?

- Reconocer algunas herramientas del depurador.
- Diseñar y realizar pruebas.
- Conocer en qué consiste un plan de pruebas.
- Distinguir entre diferentes tipos de pruebas.
- Realizar pruebas unitarias.

¿Sabías que…?

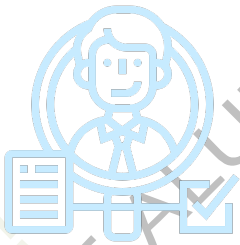
- La fase de testing debe ser paralela al proceso de desarrollo del software.
- Los testers o probadores de software son los que planifican y llevan a cabo pruebas de software para comprobar si funciona correctamente.
- Implementar el plan de pruebas desde cero es una tarea compleja y de bastante duración.

4.1. Diseño y realización de pruebas

4.1.1. Las pruebas

A pesar de que tengamos mucha experiencia en un lenguaje de programación y que pensemos que podemos programar cualquier funcionalidad por sencilla que sea, la tendremos que probar.

Los programadores no somos infalibles. De hecho, cuando nos ponemos a programar es poco probable que lo que codificamos salga bien a la primera (a no ser que sea un hello-world, un copy-paste o algo muy sencillo).



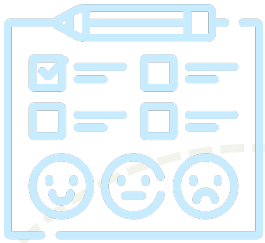
Por norma general, tendremos que probarlo y verificar que todo funciona. Necesitamos una serie de pruebas que nos garanticen que ejecutamos nuestro código y hace lo que debe hacer.

Las pruebas, ya sean unitarias, de integración o de cualquier otro tipo. Nos devolveran los resultados que esperamos si todo va bien. En el caso contrario, tendremos que corregirlo y probar de nuevo.

Pensad que somos humanos y que a pesar que hay mucho código que acabamos reutilizando, la programación tiene su vertiente artesanal. Por lo tanto, vamos a necesitar encontrar los fallos de nuestro código. ¿Cómo podemos corregir los errores mientras programamos? Mediante el depurador.

4.1.1.1. Planificar las pruebas

Las pruebas no son algo que se haga sin más, requieren de una planificación. Normalmente se realizará una checklist con los casos a probar.



¿Quién se encarga de hacer las pruebas?

Normalmente la misma persona qué se dedica codificar, es decir el programador.

La mejor práctica es ir probando a medida que codificamos y no esperar a una fase posterior para probar.

¿Cuánto tiempo se debe dedicar a las pruebas?

Como hemos visto antes podríamos dedicar un tiempo mucho mayor a realizar las pruebas que a codificar si nos planteamos probar cada caso posible que se pueda dar. Eso no es muy realista ya que en el desarrollo de software no tendremos tanto tiempo para dedicar a las pruebas. Es una buena práctica y muy conveniente dedicar el tiempo suficiente a las pruebas y acotarlo para evitar que se alargue demasiado el desarrollo del software.

4.1.1.2. Herramientas para hacer las pruebas y detectar errores

• El depurador

El depurador es una herramienta permite ver el proceso del programa paso a paso.

Podemos ver cómo funciona el código y controlar ciertos apartados del código fuente. Como por ejemplo ver qué valores guarda una variable en un momento dado o cómo se comporta una determinada función.

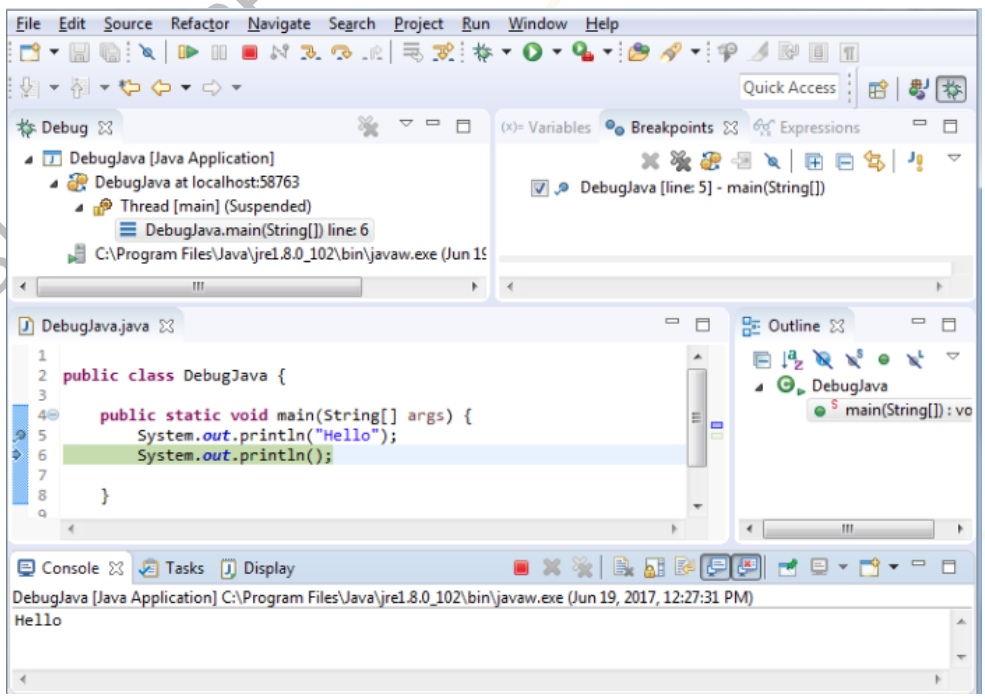
¿Qué hace el depurador?

A partir de las herramientas que tiene, nos permite ver los valores de retorno de funciones y métodos, los valores de variables, el estado de los objetos…

El depurador también nos deja establecer puntos de control. (*Breakpoints*) Incluso podemos interrumpir la ejecución del programa. Modificar valores y forzar situaciones para ver otros comportamientos del programa.

El modo depurador

El modo depuración nos permitirá controlar la la ejecución del programa paso a paso e incluso manipular ciertos aspectos y variables.



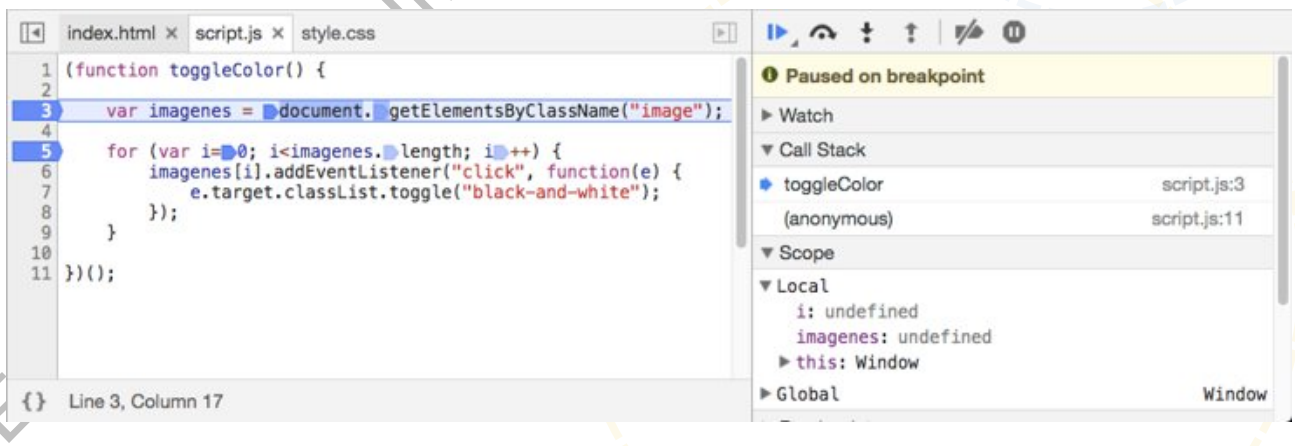
Podemos indicar que la ejecución siga o se detenga a nuestro gusto utilizando los botones de la vista Debug (o las opciones del menú "Run").

- **Resume.** Continúa la ejecución del programa hasta el próximo punto de ruptura o hasta que finaliza la ejecución.
- **Terminate.** Finaliza la ejecución del programa.
- **Step into**. Se ejecuta la línea actual y, en caso de ser una llamada a un método, la ejecución continúa dentro del método.
- **Step over**. Se ejecuta la línea actual y se pasa a la línea siguiente sin entrar en los métodos.
- **Step return.** Se sigue ejecutando hasta que se ejecute un return. Es decir, se ejecuta por completo el método en el que estemos y se pasa a la linea siguiente a la invocación del método.

¿Qué herramientas podemos usar para la depuración?

1. **Los puntos de ruptura o breakpoint:**

   Nos permiten detener la ejecución del programa cuando se alcanza uno de estos puntos. Cada punto se establece en una línea concreta del código.



Podemos hacer *step into* para ejecutar la línea actual y en caso de que sea una llamada un método la ejecución continuará dentro del método ejecutar a todas sus líneas una.

También tenemos el *step over* que ejecuta línea actual pero a diferencia del step into pasará a la siguiente línea sin entrar en el método eso si, lo ejecutará completamente pero sin ir línea a línea.

2. **Puntos de seguimiento:**

Son parecidos a los *breakpoints* pero no detienen el programa. Resultan muy útiles para el seguimiento de valores de variables.

Podemos situar los puntos de ruptura o de seguimiento en cualquier línea del código.

Una vez colocados nuestros puntos de ruptura seguimiento tendremos que iniciar el depurador. En Eclipse por ejemplo es un botón que aparece un bicho.

El analizador de código

Esta herramienta de análisis del código nos ayudar a identificar los errores en tiempo real a medida que estamos programando.

Del mismo modo que en Gmail nos aparece una serie de sugerencias de corrección del texto a medida que estamos redactando. Dispondremos de una serie de herramientas que nos permitiran analizar en código en tiempo real.

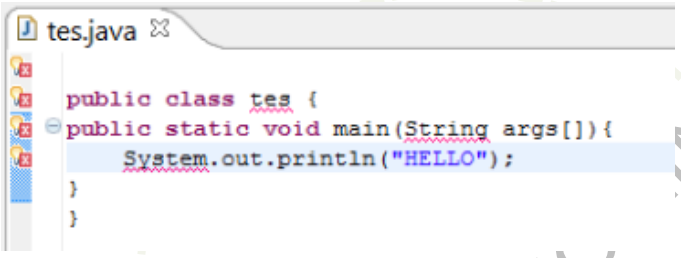
Por ejemplo, cuando codificamos en el IDE eclipse una clase de Java el analizador de código nos puede advertir cosas como:

- Avisar que falta un punto y coma.
- Que estamos cambiando una variable de tipo.
- Hemos escrito algo mal y el editor de código no lo entiende.

A parte, el analizador estático de código nos indicará si hay Errores o Warnings a medida que codificamos:

Los errores

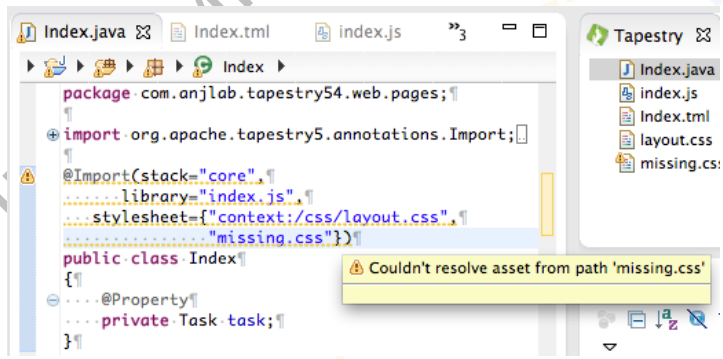
Normalmente nos lo marcará con una aspa de color rojo en el margen. Si no corregimos este error no podremos seguir con el proceso de compilación.



Los Warnings

Por otra parte el analizador también nos marcará los warnings. Son avisos de que algo podría ir mal. Algunos ejemplos podrian ser:

- Posibles valores de variables que se pueden desbordar
- Perdidas de valores en variables
- Alguna variable sin inicializar.



No es necesario que solucionemos todos los Warnings. A pesar de tener un código con estos avisos podremos seguir con el proceso de compilación. No son errores fatales que nos impidan compilar.

Es muy probable que si nuestro código es muy largo tengamos bastantes warnings, aún así nuestro programa funcionará.

Vemos ahora algunos de los errores más típicos que se suelen detectar gracias al analizador de código:

Error de declaración de una variable:

Se nos mostrará cuando se usa una variable que no sea declarado previamente. Por ejemplo en Java es obligatorio declarar una variable antes de su uso en cambio en JavaScript no es necesario.

```
tipo_var nombre_var = VALOR_INICIAL;
```

```
num = 0;  NO!
```

```
int num = 0;  SI!
```

```
mensaje = "Hola";  NO!
```

```
String mensaje = "Hola";  SI!
```

```
int num = 2;
int num2 = 3;
num3 = num1 + num2  NO!
```

```
int num3;  SI!
int num = 2;
int num2 = 3;
num3 = num1 + num2  SI!
```

Error de tipo de variable

Se muestra cuando se mezcla el uso de variables con tipos diferentes. El tipo de variable siempre decidirá los valores de la variable que puede contener.

```
int num1 = 2;
int num2 = "Hola";  NO!
```

```
int num1 = 2;
int num2 = 5;  SI!
```

```
String entrada = br.readLine();
int num1 = entrada;  NO!
```

```
String entrada = br.readLine();
int num1 = Integer.valueOf(entrada);  SI!
```

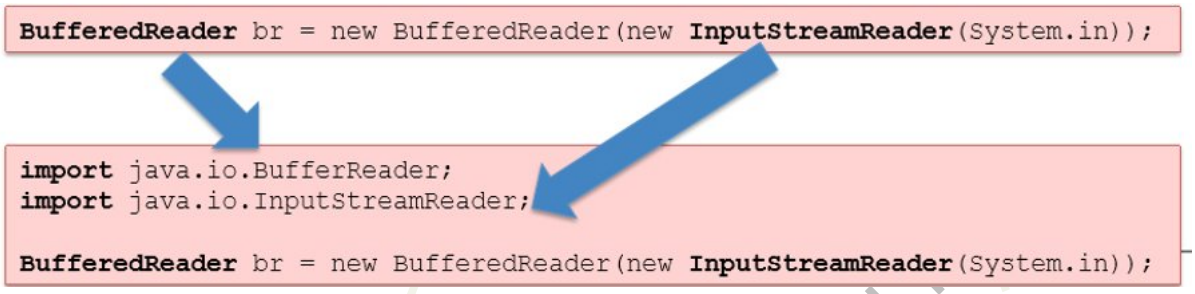
```
int num1, num3;
String entrada = br.readLine();
String num2 = entrada;
num3 = num1 + num2;  NO!
```

```
int num1, num3;
String entrada = br.readLine();
int num2 = Integer.valueOf(entrada);
num3 = num1 + num2;  SI!
```

Java es un lenguaje fuertemente tipado es decir qué es declarar la variable y además establecer de un tipo determinado que no cambiará a lo largo del programa. Es decir si declaramos una variable de tipo entera no podremos asignarle un string después. En cambio en Javascript sí que podríamos hacer este tipo de cambio de tipo ya que es un lenguaje débilmente tipado y permite que una variable cambie de tipo al largo del programa.

Error de importación:

Muchas veces hacemos servir funciones que nos pensamos que aparecen por arte de magia, pero en realidad las hemos de importar porque son de una librería externa. Son funcionalidades como por ejemplo el *Scan* de Java que permite leer por teclado.



Nosotros no tenemos porque codificar una funcionalidad que sea capaz de leer por teclado. Para eso usamos una función que ya está previamente codificada, eso sí que hemos de importar esta función de la librería. Si no lo hacemos nos aparecerá un error de importación.

En el proceso de compilación se enlazarán estas librerías estamos importando con tal de complementar nuestro código que se puede ejecutar.

4.2. El desarrollo y la realización de pruebas

Tal como deciamos, los programadores no somos infalibles y vemos obligados a probar nuestro código.

Para eso debemos tener claro qué es una prueba, saber qué tipos de pruebas podemos hacer y cómo definir diferentes casos de prueba.

4.2.1. El objetivo de una prueba

El objetivo de una prueba es el de detectar errores.

Eso si, una prueba no puede asegurar la ausencia de errores, aunque nos puede ayudar es detectar si existen defectos en el software.

Aunque una prueba salga bien, no quiere decir que todos los casos funcionen correctamente.

Veamos por ejemplo una suma de una calculadora:

Pongamos el caso en que implementamos una calculadora. Diseñamos una bateria de pruebas sólo para la función SUMA con la finalidad de demostrar que funciona correctamente. Esta bateria de pruebas tiene una combinación de valores de entrada y resultados esperados para toda la casuística (que a priori puede parecer algo demasiado simple, pero no lo es). Una vez que ejecutamos nuestras pruebas verificamos que obtenemos los resultados esperados.

¿Qué ocurre si cada caso de esta función suma tarda 2 segundos en ejecutarse y devolver el resultado? A pesar de que la prueba ha devuelto un resultado correcto seguramente tengamos otro tipo de errores relativos al rendimiento de la función.

4.2.2. ¿Qué son los casos de pruebas?

Los casos de prueba son las condiciones que se establecen con el objetivo de determinar si la aplicación funciona correctamente.

Tendremos que hacer pruebas a nivel atómico es decir de cada función o bucle y posteriormente tendremos que abstraernos más y hacer casos de pruebas de la integración de estas funciones. Mientras más funciones y líneas de código tengamos más casos de pruebas tendremos que realizar.

Veamos por ejemplo una suma de una calculadora:

En el programa entran los valores A y B y como salida o tenemos un único resultado con el valor de la suma. A priori podemos pensar que esta función es muy sencilla y no es necesario probarla. Pero no es así hemos de tener en cuenta una serie de casos.

Sumar números enteros positivos:

Empezamos con un caso básico. Por ejemplo sumamos A con valor 3 y B con valor 2 como resultado de salida esperamos un 5. Este caso de prueba nos puede indicar que en el caso de valores positivos enteros funciona correctamente.

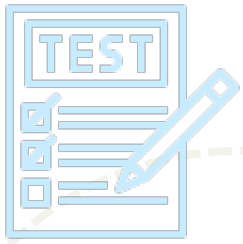
Sumar un número entero negativo con uno postivio:

También es posible sumar números negativos, para eso establecemos A con valor -3 y B con valor 2. Sí hemos implementado bien la función suma, nos debería aparecer un resultado negativo con valor de -1. Piensa que hubiera pasado si nuestras variables solo soportará números enteros (es decir solo números positivos del 0 en adelante). Nuestra función podría dar error.

Sumar números con el valor máximo que permite la variable de entrada

¿Qué pasaría si sumamos dos números positivos que se correspondan al valor mayor que puede almacenar la variable?

Por ejemplo en Java una variable de tipo integer puede guardar un valor mínimo de -2,147,483,648 y un máximo 2,147,483,647 (inclusive) esto sería otro caso de prueba. Imaginate qué puede ocurrir si sumamos los dos valores máximos…



Con una sencilla suma hemos visto que hemos de tener en cuenta unos cuantos casos de prueba. De hecho, aun nos queda pendiente los casos de prueba siguientes:

- Sumar dos números negativos.
- Sumar cualquier número con el elemento neutro de la suma (el cero).
- La combinación de los casos anteriores en diferente orden.
- Los casos anteriores pero añadiendo parte decimal.
- Sumar números con el valor mínimo que permite la variable de entrada.
- Una suma que de como resultado un valor que sobrepase el valor máximo o mínimo del valor de retorno.

Mientras más casos de prueba tengamos cubiertos estaremos más cerca de tener un programa robusto y que cumpla con todos los requisitos.

4.2.3. Tipos de pruebas

Existen muchos tipos de prueba que se pueden realizar en un proyecto de software. Antes de empezar a desarrollar conceptualmente los tipos de prueba vamos a ordenarlos de más bajo nivel a mayor nivel.

1. **Pruebas unitarias**: para verificar el correcto funcionamiento de una unidad de código.
    a. Pruebas de caja blanca
        i. El método del camino básico
        ii. Complejidad ciclomática
        iii. Caminos independientes
    b. Pruebas de caja negra
        i. Particiones equivalentes
        ii. Valores limites
2. **Pruebas funcionales**: para detectar errores del código teniendo en cuenta los requerimientos del cliente.
3. **Pruebas de integración**: pruebas sobre las relaciones entre las diferentes partes del software.
4. **Pruebas de sistema**: para detectar errores relacionados con los requisitos del programa.
    a. Pruebas de carga.
    b. Pruebas de estrés.
    c. Pruebas de seguridad.
5. **Pruebas de aceptación**: las pruebas que realizan los usuarios finales.
    a. Pruebas alfa
    b. Pruebas beta

Antes de empezar, ten en cuenta que según el proyecto será necesario aplicar una serie de pruebas o otras o tal vez todas, es más, tal vez sea necesario hacer un tipo de pruebas mucho más específicas que las que hemos enumerado anteriormente.

Por ejemplo, no van a ser las mismas pruebas que se van a realizar para provar un videojuego (completar el juego en todos los niveles) que para un programa de facturación que está conectado a la nube (volumen altos de compras) o para una plataforma de vídeo en streaming a través de subscripción (visualizar la pelicula hasta el final).

4.2.3.1. **Pruebas unitarias**

Las pruebas unitarias son pruebas individuales de las cuáles conocemos los datos de entrada y sabemos cuál debería ser el resultado esperado. Pertenecen a los casos de prueba de caja blanca.

En cada nueva funcionalidad que se añada en el código tendremos que implementar sus pruebas unitarias correspondiente. Lo normal es realizar una checklist con todas las pruebas que tenemos pensaras realizar sobre el código antes de codificarlo y una vez codificado pasar esa checklist. Siempre que testemos un error en las pruebas tenemos que corregir el código fuente y a continuación volver a aplicar todas las checklist desde el inicio, nunca sabemos si al corregir el fallo hemos podido modificar alguna otra cosa.

Es muy importante y una buena práctica realizar las pruebas y ejecutarlas de forma incremental dejarlas para el final seguramente nos conllevará perder mucho más tiempo ya que cualquier error que detectemos al final implicará modificar el código fuente y nos veremos obligados pasar la checklist de pruebas desde el principio. Así que lo más recomendable es realizar las pruebas mientras se va codificando.

Pruebas de caja blanca:

Son las pruebas estructurales. Partimos de que una función siempre tiene unos valores de entrada y otros valores de salida. A parte de la entrada y salida, hay una serie de sentencias que se deben ir ejecutando linea a linea.

Para entender el concepto de las pruebas de caja blanca, hacemos un simil comparando la una función con una caja, para este caso, la caja blanca vamos a considerar que es transparente, es decir, podemos ver todo lo que ocurre dentro.

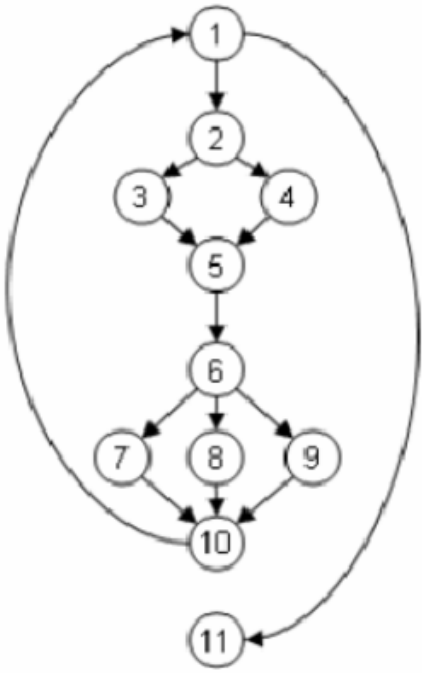
Este tipo de pruebas de caja blanca se encargan de ir probando instrucción a instrucción y pretenden asegurar el paso por todas las líneas de código para verificar que todo funciona correctamente. En definitiva, se centren en el funcionamiento interno.

Para resolver las pruebas de caja blanca haremos servir el método del camino básico.

El método del camino básico

El método del camino básico consiste en trazar todos los caminos de instrucciones que nos permite realizar el código y **construir un grafo** que lo represente. Como por ejemplo el siguiente:

```
(1)   while(x<100){
(2)     if (a[x] % 2 == 0) {
(3)         parity =  0;
        }
        else {
(4)         parity =  1;
(5)     }
(6)     switch(parity){
            case 0:
(7)           println( "a[" + i + "] is even");
            case 1:
(8)           println( "a[" + i + "] is odd");
            default:
(9)           println( "Unexpected error");
         }
(10)     x++;
      }
(11)  p = true;
```

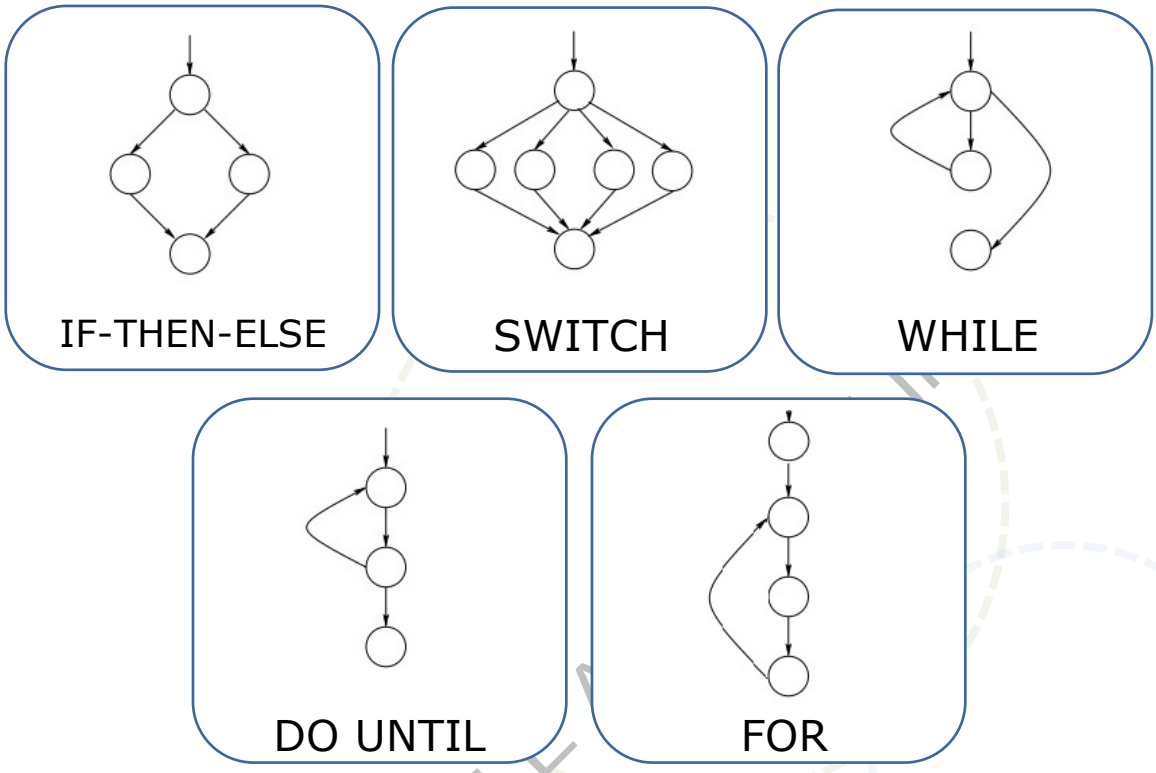


No trates de entender lo que hace el código de la imagen, no tiene importancia ahora. Trata de identificar cada número de linea con el nodo que ocupa en el grafo.

Por ejemplo si tenemos una sentencia *IF* con una condición y una serie de instrucciones para el *THEN* y otras para el *ELSE*. Tendremos que asegurarnos de hacer un camino para cada una de ellas. En la imagen anterior serian los nodos 2-3-4-5 (aunque el nodo 5 podría ser la siguiente sentencia que se ejecute una vez acaba cualquiera de los caminos *THEN* o *ELSE*.

Ya sea por cada sentencia, condicional o bucle tendremos qué asegurarnos de pasar por todos los caminos que nos ofrece cada estructura y combinarlos entre ellos. Mientrás más complejo sea el código más caminos básicos tendremos que implementar.

Las estructuras más comunes que nos podemos encontrar las podemos encontrar de las siguientes formas:



Pruebas de caja negra

Son las pruebas funcionales. Para entender el concepto de las pruebas tenemos que pensar en que nuestro programa o función es como una caja totalmente opaca, con lo cuál no podemos ver lo que hay dentro. Esta caja solo tiene unos agujeritos por donde entraran unos valores (input) y otros agujeritos por donde saldrá alguna cosa (output). Lo que pasa dentro, no lo vamos a ver, por lo tanto nos centramos en lo que entra y lo que sale.

Son las pruebas que se centran en los métodos de entrada y salida con tal de validar y controlar los datos que entran para evitar errores en estos y la finalidad es obtener los resultados esperados de los valores de salida. En ningún momento este tipo de pruebas se centra en lo que ocurre en las líneas de código internas tan solo tiene en cuenta el input y el output.

Por ejemplo en el caso de una suma cuando realicemos tus pruebas de caja negra tan sólo deberemos tener en cuenta los valores de entrada qué pueden ser dos y el valor de salida qué es el resultado. endremos que pensar en posibles combinaciones de los valores de entrada y pensar en qué valor de salida se obtendrá con estos. lo que pasa internamente en el código, como por ejemplo que venga instrucciones que no se estén ejecutando o que no sean del todo óptimas no lo tendremos en cuenta. Este tipo de pruebas se centra en la funcionalidad es decir si sale lo que esperamos con la entrada de los valores que le hemos dado.

Para este tipo de pruebas vamos a usar la técnica de la partición equivalente.

El método de la partición equivalente

El método de la partición equivalente consiste en dividir y separar los campos de entrada según el tipo de datos y las restricciones que el tipo impone a estos datos.

Un ejemplo muy sencillo sería una función para introducir las notas de un alumno. Si consideramos que un alumno puede tener notas del 0 al 10 ambos incluidos establecer y amos una partición de valores válidos de entrada desde el 0 hasta el 10. Para representar está partición escogeremos un único valor como por ejemplo el seis. Con esta prueba daríamos por validado este rango de valores. En cambio los números negativos o superiores a 10 serán considerados como no válidos. Sería conveniente probar también estos valores y verificar que nuestra función está preparada para recibir valores fuera del rango de los válidos y ver si devuelve un mensaje de aviso al usuario para que vuelva a introducir una nota correcta.

Otro ejemplo con otro tipo de datos. Si tenemos una función te los devuelve si una persona es mayor de edad, podríamos tener por ejemplo como valor de entrada una fecha y como valor de salida los valores cierto o falso, refiriéndonos a si es mayor de edad o no lo es. Para este caso podemos aplicar el método de la partición equivalente estableciendo una serie de fechas válidas por ejemplo fechas desde el 1 de enero del 1900 hasta la fecha actual (a día de hoy en el momento que estás leyendo esto). Este rango de valores sería una partición equivalente con valores válidos de entrada, es decir si escogemos el valor 2 de marzo de 1994 que está situado entre estas dos fechas válidas lo consideraremos como que hemos probado este rango de valores compartición equivalente. En cambio si escogiéramos una fecha anterior como por ejemplo el 31 de diciembre del 1899 nos estaríamos refiriendo a valores de entrada no válidos. Cualquier fecha anterior a esta última pertenecería a este rango de valores. Pasaría lo mismo con fechas futuras es decir la fecha de mañana también sería un rango no válido de valores para probar.

La finalidad de la partición equivalente es escoger un valor de este rango de valores que lo represente. De este modo daríamos por probado estos valores. Imaginaos si tuviéramos que probar todas las fechas desde el 1900 hasta día de hoy…

Cuando usamos el método de la partición equivalente es muy importante tener en cuenta los valores límite.

Valores límite

Es un tipo de prueba complementaria a la partición equivalente en la cuál indicaremos los valores o números qué delimitan los rangos de valores que hemos definido con la partición equivalente.

Por ejemplo para el caso de la función de introducir las notas del 0 al 10, aparte de probar la nota con valor 6 deberíamos añadir los límites de este rango de valores que son el 0 y el 10 propiamente. Cuando se trata de valores numéricos es importante probar siempre el primer número desde el que empieza el rango y el último así como los extremos positivos y negativos y los valores máximos de almacenamiento de la variable.

Para los rangos de los valores no válidos sería conveniente probar también los valores límite, para el mismo caso de las notas deberíamos probar el siguiente valor al 10 que podríamos considerar qué es el 10,01 y el valor anterior al 0 que sería el menos 0,01.

En resumen podríamos decir que las pruebas de caja blanca van ligadas al procedimiento en si realizando una correcta evaluación de las condiciones y puzzles y instrucciones.
En cambio las pruebas de caja negra se realizan desde el punto de vista del usuario final es decir que tiene en cuenta lo que entra y lo que sale.

4.2.3.2. **Pruebas de integración**

Son las pruebas que se encargan de detectar errores entre la comunicación entre diferentes funciones o componentes. Se hacen una vez que se han hecho las pruebas unitarias y se hacen probando diferentes funciones en conjunto.

Por ejemplo, para el caso de la calculadora, una vez se han probado las funciones suma, resta, multiplicación y división de forma individual, se deben hacer las pruebas de integración entre todas estas funciones (con una operación compuesta por ejemplo 3+2-6*3/2). Tal vez por si solas funcionan correctamente, pero una vez que las funciones estan integradas y se deben pasar parametros de unas a otras será necesario hacer estas pruebas de integración.

4.2.3.3. **Pruebas de sistema**

Son las pruebas que se deberían hacer después de las pruebas de integración. Estas pruebas pretenden verificar que el sistema completo funciona (desde el punto de vista de la aplicación).

Por ejemplo, en el caso de la calculadora, deberiamos ejecutar el programa de la misma forma que lo haria un usuario y verificar que realiza todas las operaciones correctamente.
Las pruebas de sistema abarcan aspectos como:

- **Rendimiento**: sobre todo teniendo en cuenta los tiempos de respuesta.
- **Seguridad**: verificar permisos de diferentes roles de usuario y proteger el acceso al sistema.
- **Usabilidad**: verificar que la experiencia de usuario sea óptima.
- **Instalación**: comprobar que no hay problemas al instalarse en diferentes sistemas operativos o compatibilidades.

4.2.3.4. **Pruebas de carga**

Son pruebas de rendimiento del software. Se trata de provar nuestro software en un escenario en el que se le hace una demanda elevada de peticiones para verificar que no se altera el funcionamiento.

4.2.3.5. **Pruebas de estrés**

Son pruebas en las que se simula un escenario de situaciones extremas con tal de ver el comportamiento ante estos escenarios. Por ejemplo, provocar que el sistema caiga con un gran número de peticiones y ver si es capaz de recuperarse.

4.2.3.6. **Pruebas de seguridad**

Son las pruebas que se hacen cuando hay diferentes niveles de permisos en la aplicación. Por ejemplo, para una aplicación académica en la que pueden acceder, invitados, alumnos, profesores y directores se puede verificar que cada perfil pueda acceder a sus datos pero no a los de un perfil superior.

4.2.3.7. **Pruebas de aceptación**

Son las pruebas que se encargan de validar si el software cumple con las expectativas del cliente y de los usuarios. Son pruebas que hacen normalmente los usuarios finales o los betatesters. Podemos clasificarlas en pruebas alfa y beta.

> **Pruebas alfa:** Estas pruebas se hacen al final del desarrollo con el usuario final acompañado de los programadores o de alguien relacionado en el desarrollo. El cliente y el desarrollador toman nota juntos de los aspectos a corregir o mejorar.
>
> **Pruebas beta:** Estas pruebas se hacen al final del desarrollo con usuarios finales con el software en producción o bien en una fase beta previa al lanzamiento. A diferencia de las pruebas alfa, estas pruebas pretenden que las haga el usuario final en un entorno no controlado. El cliente prueba el software solo y toma nota de los aspectos a corregir o mejorar, después se los enviará al equipo de desarrollo para que lo mejore.

Recursos y enlaces

- Información sobre el JDK de Java

Conceptos clave

- **Casos de prueba:** son las condiciones que se establecen con el objetivo de determinar si la aplicación funciona correctamente.
- **El depurador:** es una herramienta permite ver el proceso del programa paso a paso.
- **Analizador el código**: nos ayuda a identificar los errores en tiempo real.
- **Casos de pruebas:** son las condiciones que se establecen con el objetivo de determinar si la aplicación funciona correctamente.

Test de autoevaluación

1. Las pruebas…
   a) Nos ayudan a identificar los errores en tiempo real.
   b) Son una herramienta permite ver el proceso del programa paso a paso.
   c) Nos permiten verificar el correcto funcionamiento y detectar errores.
   d) Solo verifican permisos de diferentes roles de usuario y proteger el acceso al sistema.

2. Un breakpoint:
   a) No detiene el programa. Resultan muy útiles para el seguimiento de valores de variables.
   b) Permite detener la ejecución del programa cuando se alcanza uno de estos puntos. Cada punto se establece en una línea concreta del código.

3. Normalmente se marcan con un aspa de color rojo en el margen. Si no corregimos no podremos seguir con el proceso de compilación.
   a) Error
   b) Warning

4. Normalmente se marcan con triángulo amarillo en el margen. Son avisos de que algo podría ir mal, aun así, podremos seguir con el proceso de compilación.
   a) Error
   b) Warning

Ponlo en práctica

Actividad 1

Dibuja el grafo de flujo, calcula la complejidad ciclomática e identifica los caminos independientes del siguiente código:

```
Import java.io.*;

Public class calculaMaximo

{

      public static void main (String args[]) throws IOException

      {

            BufferedReader entrada = new BufferedReader (new
            InputStreamReader(System.in));

            Int a,b,c,max;

            System.out.println("Introduce a,b,c: ");

            a = Integer.parseInt (entrada.readLine());

            b = Integer.parseInt (entrada.readLine());

            c = Integer.parseInt (entrada.readLine());

            if (a>b && a>c) max = a;

              else

                if (c>b)

                  max = c;

                else

                  max = b;

            System.out.println ("El número más grande es "+ max);

      }

}
```

SOLUCIONARIOS

Test de autoevaluación

1- Las pruebas…

a) Nos ayudan a identificar los errores en tiempo real.

b) Son una herramienta permite ver el proceso del programa paso a paso.

c) Nos permiten verificar el correcto funcionamiento y detectar errores.

d) Solo verifican permisos de diferentes roles de usuario y proteger el acceso al sistema.

2- Un breakpoint:

a) No detiene el programa. Resultan muy útiles para el seguimiento de valores de variables.

b) Permite detener la ejecución del programa cuando se alcanza uno de estos puntos. Cada punto se establece en una línea concreta del código.

3- Normalmente se marcan con un aspa de color rojo en el margen. Si no corregimos no podremos seguir con el proceso de compilación.

a) Error

b) Warning

4- Normalmente se marcan con triángulo amarillo en el margen. Son avisos de que algo podría ir mal, aun así podremos seguir con el proceso de compilación.

a) Error

b) Warning

Ponlo en práctica

Actividad 1

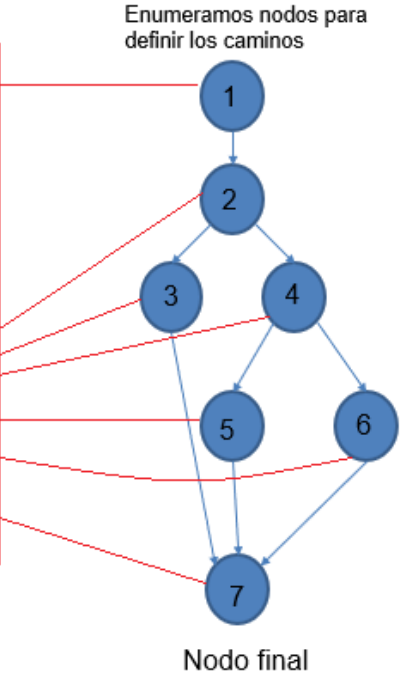
Dibuja el grafo de flujo, calcula la complejidad ciclomática e identifica los caminos independientes del siguiente código:

```
Import java.io.*;

Public class calculaMaximo

{

      public static void main (String args[]) throws IOException

      {

            BufferedReader entrada = new BufferedReader (new
            InputStreamReader(System.in));

            Int a,b,c,max;

            System.out.println("Introduce a,b,c: ");

            a = Integer.parseInt (entrada.readLine());

            b = Integer.parseInt (entrada.readLine());

            c = Integer.parseInt (entrada.readLine());

            if (a>b && a>c) max = a;

              else

                if (c>b)

                  max = c;

                else

                  max = b;

            System.out.println ("El número más grande es "+ max);

      }

}
```

Solución:

Vamos leyendo línea a línea y vamos extrayendo nodos:

```
Import java.io.*;
Public class calculaMaximo
{
     public static void main (String args[]) throws IOException
     {
          BufferedReader entrada = new BufferedReader (new
          InputStreamReader(System.in));
          Int a,b,c,max;
          System.out.println("Introduce a,b,c: ");
          a = Integer.parseInt (entrada.readLine());
          b = Integer.parseInt (entrada.readLine());
          c = Integer.parseInt (entrada.readLine());
          if (a>b && a>c)
            max = a;
          else
            if (c>b)
              max = c;
            else
              max = b;

          System.out.println ("El número más grande es "+ max);
     }
}
```



Calcular complejidad ciclomática:

V(G) = aristas – nodos + 2

V(G) = 8 – 7 + 2 = 3

Identifica los caminos independientes:

C1= 1,2,3,7

C2= 1,2,4,5,7

C3= 1,2,4,6,7