P01- Diseño de pruebas de caja blanca

Diseño de pruebas de caja blanca (*structural testing*)

En esta sesión aplicaremos el método de **diseño** de casos de prueba visto en clase (método del camino básico) para obtener un conjunto de casos de prueba de unidad (recuerda que hemos definido una unidad como un método java). Usaremos la implementación (conjunto P) para determinar el conjunto de datos de entrada. Ojo! **Nunca uses la implementación para indicar el resultado esperado**, por razones obvias.

Cuando apliques el método, anota SIEMPRE los PASOS que vas siguiendo e indica, de forma explícita qué es lo que se pretende con dicho paso. La idea es que las prácticas te ayuden a entender bien lo que hemos explicado en clase de teoría.

El método del CAMINO BÁSICO tiene un objetivo que es común a todos los métodos de diseño de casos de prueba, pero también tiene un objetivo particular, que lo diferencia del resto de métodos. Cuando acabes la práctica te deben quedar muy claros ambos objetivos.

Tal y como hemos indicado en clase, cada sesión de prácticas está pensada para ayudarte a entender bien los conceptos explicados en la clase de teoría correspondiente. Por lo tanto, una vez acabada la práctica, tendrás que ser capaz de contestar y justificar razonadamente preguntas como: ¿por qué puedo obtener tablas diferentes aplicando el mismo método? ¿ambas tablas son igual de válidas? ¿por qué tienen que ser caminos independientes los caminos obtenidos a partir del grafo?, ¿qué implicaciones tiene el proporcionar menos caminos que los indicados por CC?,...

En esta sesión no utilizaremos ningún software específico, pero en las siguientes sesiones automatizaremos la ejecución de los casos de prueba que hemos diseñado en esta práctica.

GitHub

Crea la carpeta **P01-CaminoBasico**, en tu directorio de trabajo, (ppss-2024-Gx-apellido1-apellido2). Aquí estarán todos los ficheros con tus soluciones, que deberás subir a **GitHub**..

Tus soluciones de los ejercicios debes guardarlos en ficheros de **imagen (jpg, png)**, o en formato de **texto (.txt, .md)**.

Ejercicios

Puedes hacer los ejercicios en papel y subir luego una foto a GitHub (en formato png o jpg). Si prefieres usar alguna herramienta, puede serte útil ésta: https://www.draw.io. Desde aquí podrás hacer todo el ejercicio: tanto los grafos, como la tabla, texto, etc. También hay disponible una versión de escritorio. Si usas esta herramienta, cuando guardes tu trabajo, hazlo en el formato por defecto (es xml, pero la extensión es .drawio), por si quieres modificarlo en cualquier momento, y también en formato png o jpg (para poder visualizarlo directamente desde GitHub).

A continuación proporcionamos la especificación y el código de las unidades a probar.

➪ ➪ Ejercicio 1: *reservaButacas*

Crea la subcarpeta "**1_reservaButacas**", en donde guardarás tu solución de este ejercicio. Puedes crear uno o varios ficheros. En el caso de que la solución esté dividida en varios ficheros, todos ellos tendrán como nombre "**cine-<sufijo>.<extension>**", siendo <sufijo> la indicación del paso o pasos seguidos: por ejemplo cine-paso1.jpg, cine-pasos2-3.jpg,... o simplemente "**cine.<extension>**" si todo el ejercicio está resuelto en un fichero. <extension> denota el tipo de fichero: jpg, png, ...

Diseña los casos de prueba para un método que, a partir de una fila de butacas de cine (array de booleanos), y un número de asientos consecutivos a reservar (N), proceda a hacer efectiva la reserva de dichos asientos, y devuelva true o false, dependiendo de si ha sido posible hacer la reserva o no.

Cada una de las posiciones del array que representa la fila de butacas, tendrá un valor false, si la butaca correspondiente no está reservada, y true, en caso contrario.

La reserva será posible si en la fila hay N asientos consecutivos libres (siendo N>0). En otro caso, la reserva no será posible. Si el número de asientos consecutivos solicitados es negativo, se lanzará una excepción de tipo ***ButacasException*** con el mensaje "No se puede procesar la solicitud"

Si el número de asientos consecutivos solicitados excede el número de butacas de la fila, se lanzará una excepción de tipo ***ButacasException*** con el mensaje "No se puede procesar la solicitud"

A partir de la especificación anterior, hemos implementado el siguiente código:

```
1. public boolean reservaButacas(boolean[] asientos, int solicitados) {
1.   boolean reserva = false; int j=0;
2.   int sitiosLibres =0;  int primerLibre;
3.   while ( (j< asientos.length) && (sitiosLibres < solicitados) ) {
4.      if (!asientos[j]) {
5.         sitiosLibres++;
6.      } else {
7.          sitiosLibres=0;
8.      }
9.      j++;
10.  }
11.  if (sitiosLibres == solicitados) {
12.      primerLibre = (j-solicitados);
13.      reserva = true;
14.      for (int k=primerLibre; k<(primerLibre+solicitados); k++) {
15.          asientos[k] = true;
16.      }
17.   }
18.   return reserva;
19.}
```

Dada la especificación y código anteriores, diseña una tabla de casos de prueba aplicando el método de diseño del camino básico explicado en clase.

Debes subir tu solución a GitHub.

NOTA: Recuerda que TODOS los datos de entrada y salida esperada deben ser siempre valores concretos.

NOTA: Independientemente del método de DISEÑO de casos de prueba que usemos, todos ellos proporcionan un conjunto de casos de prueba efectivo y eficiente (evidencian el máximo número posible de errores, con el mínimo número de pruebas), teniendo en cuenta un objetivo concreto.

En el caso del método del camino básico, el valor de complejidad ciclomática (CC) determina el número MÁXIMO de filas de la tabla. Para que el conjunto de casos de prueba sea eficiente, deberíamos generar el mínimo número de caminos con los que conseguimos recorrer todos los nodos y todas las aristas del grafo (ese número puede ser inferior al de CC).

En cualquier caso, en clase hemos dicho que vamos a considerar válida cualquier solución que contenga un número de caminos independientes menor o igual que el valor de CC. Un camino es independiente si añade al conjunto de caminos al menos un nodo o una arista que no se había recorrido ANTES.

➪ ➪ Ejercicio 2: *contarCaracteres*

Crea la subcarpeta "**2_contarCaracteres**", en donde guardarás tu solución de este ejercicio.

Igual que antes, puedes solucionar el problema con uno o varios ficheros. En el caso de que la solución esté dividida en varios ficheros, todos ellos tendrán como nombre "**caracteres-<sufijo>.<extension>**

Queremos diseñar los casos de prueba para un método que, dado el nombre de un fichero de texto, devuelve el número de caracteres que contiene, a menos que se produzca algún error en el tratamiento del fichero, en cuyo caso el método devolverá una excepción definida por el usuario de tipo *FicheroException* con un mensaje asociado que describe el error.

En concreto, los errores que se pueden detectar en la lectura del fichero y los mensajes de error que se devolverán asociados a la excepción son los siguientes:

Error	mensaje de error
Al abrir el fichero	*nombre_del_fichero* (No existe el archivo o el directorio)
Al realizar una lectura	*nombre_del_fichero* (Error al leer el archivo)
Al cerrar el fichero	*nombre_del_fichero* (Error al cerrar el archivo)

A partir de la especificación anterior, hemos implementado el siguiente código:

```
1.  public int contarCaracteres(String nombreFichero) throws FicheroException {
2.     int contador = 0;
3.     FileReader fichero = null;
4.     try {
5.        fichero = new FileReader(nombreFichero);
6.        int i=0;
7.        while (i != -1) { // comprobación de fin de fichero
8.           i = fichero.read();
9.           contador++;
10.       }
11.    } catch (FileNotFoundException e1) {
12.       throw new FicheroException(nombreFichero +
13.                                  " (No existe el archivo o el directorio)");
14.    } catch (IOException e2) {
15.             throw new FicheroException(nombreFichero +
16.                                  " (Error al leer el archivo)");
17.    }
18.    if (fichero != null) {
19.       try {
20.             fichero.close();
21.       } catch (IOException e) {
22.             throw new FicheroException(nombreFichero +
23.                                  " (Error al cerrar el archivo)");
24.       }
25.    }
26.    return contador;
27. }
```

Dada la especificación y código anteriores, diseña una tabla de casos de prueba aplicando el método de diseño del camino básico explicado en clase.

Nota: debes tener en cuenta que el constructor de FileReader puede lanzar una excepción de tipo *FileNotFoundException*, y que los métodos *close*() y *read*() pueden lanzar una excepción de tipo *IOException*.

Adicionalmente, para este ejercicio debes TENER EN CUENTA que puede haber varios conjuntos diferentes de caminos independientes en el grafo. Si alguno de los caminos independientes es imposible de recorrer con ningún dato de entrada, debes intentar cambiar dicho conjunto (porque ya no cumpliríamos el objetivo del método). Si detectamos que algún nodo y/o arista es imposible de

recorrer con ningún dato de entrada, entonces las sentencias de código implicadas en esos nodos/aristas imposibles de alcanzar NO deberían estar ahí.

En este último caso, debes modificar el código a probar de forma que no tenga sentencias inaccesibles. Y a continuación deberás volver a aplicar el método teniendo en cuenta el nuevo grafo resultante de los cambios realizados en dicho código.

Es indispensable que tengas claro cuántas y qué entradas y salidas tiene el método a probar. En este código tienes que darte cuenta de que NO siempre las entradas están todas ellas en forma de parámetros del método a probar.

Debes subir tu solución a GitHub.

➪ ➪ Ejercicio 3: método *reservaLibros()*

Crea la subcarpeta "**3_reservaLibros**", en donde guardarás tu solución de este ejercicio.

Puedes crear uno o varios ficheros. En el caso de que la solución esté dividida en varios ficheros, todos ellos tendrán como nombre "**reserva-<sufijo>.<extension>**

Se proporciona la siguiente **especificación** para el método ***reservaLibros()*:**

Se quiere llevar a cabo la reserva de una serie de libros por parte de un socio de una biblioteca, el método recibe por parámetro el login y password del empleado de la biblioteca (que será el que realice la reserva), un identificador de un socio de la misma (que es la persona que quiere reservar los libros), y una colección de isbns de los libros que quiere reservar. Solamente un empleado de la biblioteca con rol de bibliotecario puede realizar la reserva.

La reserva propiamente dicha (para cada uno de los libros) se hace efectiva en **otro método** (invocado desde reservaLibros), el cual puede lanzar varias excepciones, de forma que devolverá:

- la excepción `IsbnInvalidoException, con el mensaje "ISBN invalido:<isbn>", si el` isbn del libro que se quiere reservar no existe en la base de datos de la biblioteca (siendo <isbn> el isbn de dicho libro).
- la excepción `SocioInvalidoException`, con el mensaje `"SOCIO invalido", si` el identificador del socio no existe en la base de datos. En ese caso, no se podrá hacer efectiva la reserva para ninguno de los libros de la lista.
- la excepción `JDBCException`, con el mensaje `"CONEXION invalida"`, si no se puede acceder a la BD.
- si todo va bien y se puede hacer la reserva, el método invocado termina normalmente (no devuelve nada)

El método ***reservaLibros*** termina normalmente (sin devolver nada) si todo va bien y se realiza la reserva de todos los libros de la lista pasada como parámetro. En el caso de que no se pueda hacer efectiva la reserva de algún libro, el método *reservaLibros*() devolverá una excepción de tipo *ReservaException*, con un mensaje formado por todos los mensajes de las excepciones generadas durante el proceso de reserva de cada libro, separados por ";".

Por ejemplo: suponiendo que el login y password del bibliotecario son "biblio", "1234", que el identificador de socio proporcionado existe en la base de datos, que la lista de isbns a reservar es (12345, 23456, 34567), y que el segundo y tercer isbns no están en la base de datos, el método reservaLibros devolverá como resultado una excepción de tipo *ReservaException* con el mensaje: "ISBN invalido: 23456; ISBN invalido: 34567;"

A continuación mostramos una **implementación** del método *reservaLibros()*.

```
1.public void reservaLibros(String login, String password,
2.                          String socio, String [] isbns) throws Exception {
3.  ArrayList<String> errores = new ArrayList<String>();
4.    //El método compruebaPermisos() devuelve cierto si la persona que hace
5.    //la reserva es el bibliotecario y falso en caso contrario
6.  if(!compruebaPermisos(login, password, Usuario.BIBLIOTECARIO)) {
7.    errores.add("ERROR de permisos");
8.  } else {
9.    FactoriaBOs fd = FactoriaBOs.getInstance();
10.    //El método getOperacionBO() devuelve un objeto de tipo IOperacionBO
11.    //a partir del cual podemos hacer efectiva la reserva
12.    IOperacionBO io = fd.getOperacionBO();
13.    try {
14.      for(String isbn: isbns) {
15.        try {
16.          //El método reserva() registra la reserva de un libro (para un socio)
17.          //dados el identificador del socio e isbn del libro a reservar
18.          io.reserva(socio, isbn);
19.        } catch (IsbnInvalidoException iie) {
20.          errores.add("ISBN invalido" + ":" +  isbn);
21.        }
22.      }
23.    } catch (SocioInvalidoException sie) {
24.      errores.add("SOCIO invalido");
25.    } catch (JDBCException je) {
26.      errores.add("CONEXION invalida");
27.    }
28.  }
29.  if (errores.size() > 0) {
30.    String mensajeError = "";
31.    for(String error: errores) {
32.      mensajeError += error + "; ";
33.    }
34.    throw new ReservaException(mensajeError);
35.  }
36.}
```

A partir del código y la especificación proporcionadas, diseña una tabla de casos de prueba para el método *`reservaLibros()`* usando el método del camino básico.

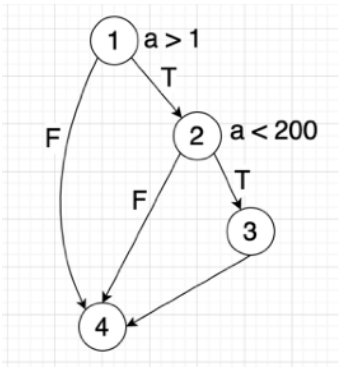
NOTA: En este ejercicio hay comportamientos programados que NO hemos especificado. Cuando esto ocurre, se usa un interrogante (?) como valor del comportamiento esperado. Con ello estamos indicando que no es responsabilidad del tester el completar la especificación o modificarla en modo alguno. Ante esta situación, el diseño de ese caso de prueba quedará pendiente hasta que complete la especificación por quien corresponda,

Recuerda subir tu solución a GitHub.

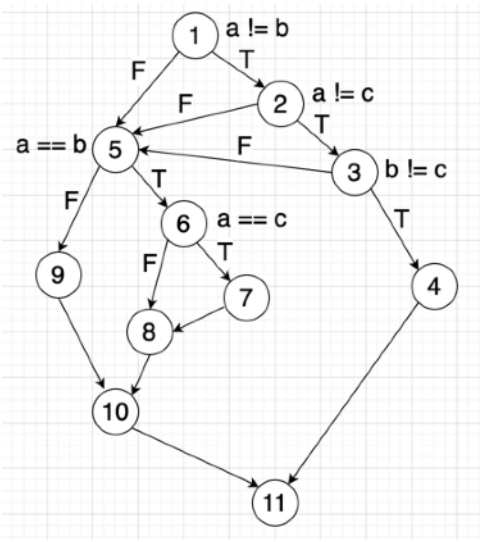
➪ ➪ ANEXO: soluciones de los ejercicios propuestos en teoría (S01)

Ejercicios propuestos en la tr. 11

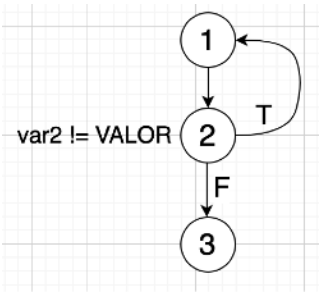
Grafo de flujo 1.



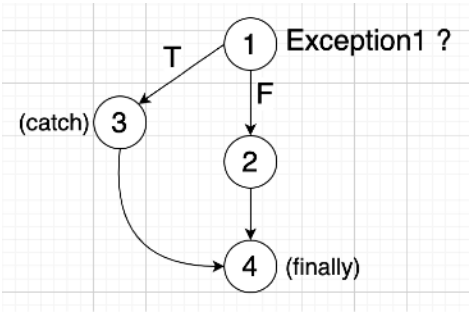
Grafo de flujo 2.



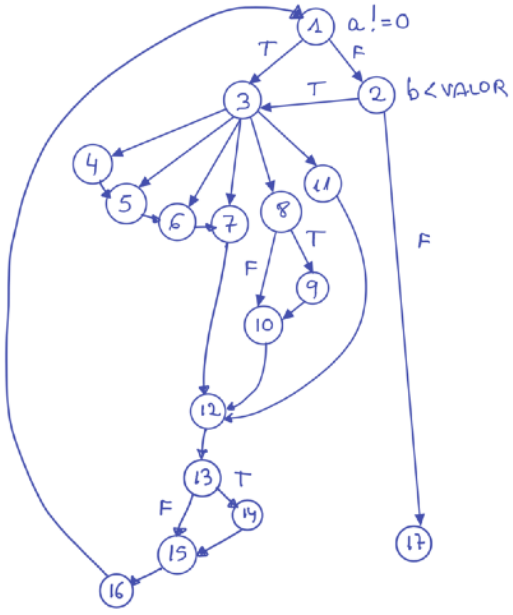
Grafo de flujo 3.



Grafo de flujo 4.

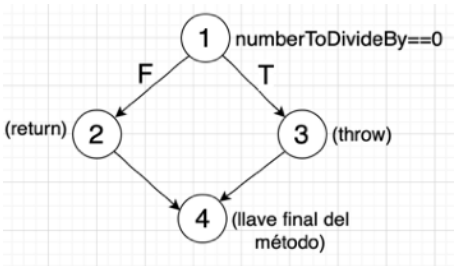


Grafo de flujo 5.

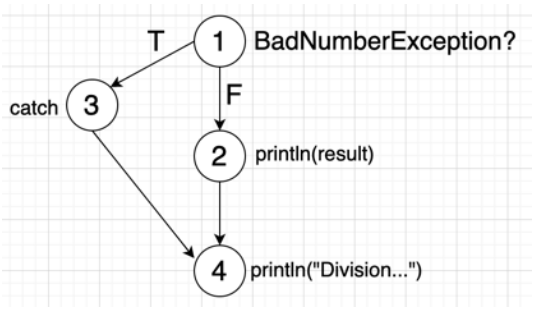


Valores de CC para los códigos de la Tr. 23

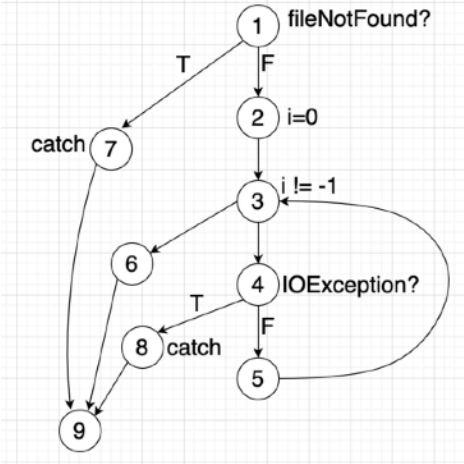
Código 1. CC = 2



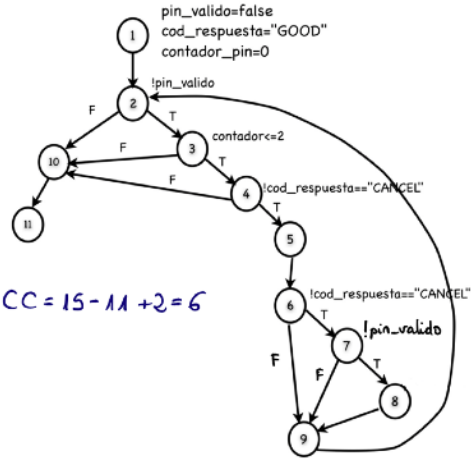
Código 2. CC = 2



Código 3. CC = 4



Grafo + CC ejercicio tr. 24:



Resumen

¿Qué conceptos y cuestiones me deben quedar CLAROS después de hacer la práctica?

MÉTODOS DE DISEÑO ESTRUCTURALES

- Permiten seleccionar un subconjunto de comportamientos a probar a partir del código implementado. Sólo se aplican a nivel de unidad, y son técnicas dinámicas.
- No pueden detectar comportamientos especificados que no han sido implementados.
- Cualquier método de este tipo nos permite obtener un conjunto de casos de prueba eficiente y efectivo para un objetivo concreto (que vendrá determinado por el criterio de selección del método en cuestión).
- El conjunto de casos de prueba seleccionados se representan en forma de tabla

MÉTODO DE DISEÑO DEL CAMINO BÁSICO

- Usa una representación de grafo de flujo de control (CFG).
- El objetivo es proporcionar el número mínimo de casos de prueba que garanticen que estamos probando todas las líneas del programa, y todas las condiciones en sus vertientes verdadera y falsa, al menos una vez.
- A partir del grafo, el valor de CC indica una cota superior del número de casos de prueba a obtener.
- Una vez que obtenemos el conjunto de caminos independientes, hay que proporcionar los casos de prueba que ejerciten dichos caminos (admitiremos que el número de caminos independientes sea ≤ que CC en todos los casos).
- Es posible que haya caminos imposibles o que detectemos comportamientos implementados pero no especificados, pero nunca podremos darnos cuenta de que nos faltan comportamientos por implementar.
- Si hay algún nodo/arista que no se puede recorrer con ninguna entrada posible tendremos que refactorizar el código a probar, ya que estaremos incumpliendo el objetivo del método.
- Todos los caminos independientes obtenidos deben poder recorrerse con algún dato de entrada.
- Los datos de entrada y los datos de salida esperados siempre deben ser concretos.
- Las entradas de una unidad no tienen por qué ser los parámetros de dicha unidad.
- Si no tenemos claro qué y cuantas entradas y salidas tiene dicha unidad a probar NO podremos diseñar los casos de prueba.
- El resultado esperado siempre debemos obtenerlo de la especificación de la unidad a probar.
- Si el resultado esperado no está especificado lo indicaremos con un interrogante "?", y no automatizaremos dicho caso de prueba hasta no haber determinado un valor concreto para ese resultado esperado.