P02- 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 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.

Así por ejemplo, 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 aquí, por lo que necesitarás tus soluciones de esta práctica para poder trabajar en las siguientes.

Bitbucket

El trabajo de esta sesión debes subirlo a *Bitbucket*. Todo el trabajo de esta práctica, tanto las tablas, como cualquier otro documento con vuestras notas de trabajo, deberán estar en el directorio **P02-CaminoBasico**, dentro de tu espacio de trabajo, es decir, dentro de tu carpeta: ppss-2020-Gx-apellido1-apellido2.

Recuerda que si trabajas desde los ordenadores del laboratorio primero deberás configurar git y clonar tu repositorio de Bitbucket. Usaremos un ÚNICO repositorio para guardar TODAS las prácticas realizadas.

Ejercicios

Puedes hacer los ejercicios en papel y subir luego una foto a Bitbucket (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. Cuando guardes tu trabajo, hazlo tanto en el formato por defecto (xml), por si quieres modificarlo en cualquier momento, como en formato png.

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

➪ ➪ Ejercicio 1

Queremos diseñar los casos de prueba para el método ***calculaTasaMatricula()*** (la especificación es la misma que la del ejercicio 3 de la práctica anterior). Dicha unidad calcula y devuelve el valor de las tasas de matriculación en función de la edad, de si es familia numerosa, y si es o no repetidor, de acuerdo con la siguiente tabla (asumiendo que se aplican siempre sobre un valor inicial de tasa=500 euros) :

	Edad < 25	Edad < 25	Edad 25..50	Edad 51..64	Edad ≥ 65
Edad	SI	SI	SI	SI	SI
Familia Numerosa	NO	SI	SI		
Repetidor	SI				
Valor tasa-total	tasa + 1500	tasa/2	tasa/2	tasa -100	tasa/2

La tabla anterior se "lee" por columnas. Por ejemplo, para la primera columna: "Si edad<25, y No familia numerosa y es repetidor" entonces el resultado es 500+1500. Las casillas en blanco indican que dicha condición es indiferente, por ejemplo "Si edad <25 y familia numerosa" independientemente de si es repetidor o no el resultado será 500/2.

Utiliza el método de casos de prueba del método que hemos visto en clase para diseñar los tests teniendo en cuenta que la implementación es la siguiente:

```
1. public float calculaTasaMatricula(int edad, boolean familiaNumerosa,
2.                                                boolean repetidor) {
3.    float tasa = 500.00f;
4.
5.    if ((edad < 25) && (!familiaNumerosa) && (repetidor)) {
6.      tasa = tasa + 1500.00f;
7.    } else {
8.       if ((familiaNumerosa) || (edad >= 65)) {
9.         tasa = tasa / 2;
10.      }
11.      if ((edad > 50) && (edad < 65)) {
12.        tasa = tasa - 100.00f;
13.      }
14.   }
15.   return tasa;
16.}
```

Debes subir tu solución a Bitbucket.

NOTA: Recuerda que los datos de entrada y salida esperada deben ser siempre valores concretos. Fíjate que en la tabla hay casillas en blanco, eso significa que dichos valores de entrada no afectan al resultado esperado. Aún así, tendremos que decidir un valor concreto cuando obtengamos la tabla de casos de prueba.

NOTA: Independientemente del método de DISEÑO de casos de prueba que usemos, todos ellos proporcionan un conjunto de casos de prueba eficiente y efectivo (evidenciar 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, la complejidad ciclomática (CC) nos indica 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 como 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

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

Dada una colección de enteros, que representan lecturas de la altura de un terreno, tomadas a intervalos equidistantes de 1 kilómetro (la medida se toma siempre al inicio del intervalo), se trata de detectar cuál es el tramo llano más largo de esas lecturas, y devolver un objeto de tipo Tramo con el origen (número de kilómetro donde comienza el llano) y la longitud (número de kilómetros) del llano encontrado. Consideraremos que hemos detectado un llano cuando haya dos o más kilómetros consecutivos con la misma altura. Además, tendremos en cuenta las siguientes consideraciones:

- La lista de lecturas nunca va a tener el valor null
- El primer kilómetro se considera como kilómetro cero
- Si hay varios llanos con la misma longitud, devolveremos el primero de ellos
- Si no hay ningún llano, se devolverá un Tramo con origen: 0, y con longitud: 0
- Los llanos pueden estar por debajo, por encima, o a nivel del mar (altura 0).

Proporcionamos la siguiente implementación asociada a la especificación anterior:

```
1. public Tramo buscarTramoLlanoMasLargo(ArrayList<Integer> lecturas) {
2.   int lectura_anterior =-1;
3.   int longitud_tramo =0, longitudMax_tramo=0;
4.   int origen_tramo=-1, origen_tramoMax=-1;
5.   Tramo resultado = new Tramo(); //el origen y la longitud es CERO
6.
7.   for(Integer dato:lecturas) {
8.      if (lectura_anterior== dato) {//detectamos un llano
9.         longitud_tramo ++;
10.         if (origen_tramo == -1 ) {//marcamos su origen
11.             origen_tramo = lecturas.indexOf(dato);
12.         }
13.      } else {  //no es un llano o se termina el tramo llano
14.          longitud_tramo=0;
15.          origen_tramo=-1;
16.      }
17.      //actualizamos la longitud máxima del llano detectado
18.      if (longitud_tramo > longitudMax_tramo) {
19.         longitudMax_tramo = longitud_tramo;
20.         origen_tramoMax = origen_tramo;
21.      }
22.      lectura_anterior=dato;
23.   }
24.   switch (longitudMax_tramo) {
25.     case -1:
26.     case  0: break;
27.     default: resultado.setOrigen(origen_tramoMax);
28.              resultado.setDuracion(longitudMax_tramo);
29.   }
30.
31.   return resultado;
32.}
```

Se pide:

A) Representa el CFG asociado al siguiente código, que implementa esta especificación, calcula su CC, y obtén el conjunto de caminos independientes.

B) Selecciona datos de entrada para recorrer todos los caminos obtenidos, verás que aparece un camino que es **IMPOSIBLE** de recorrer con ningún dato de entrada (**pista**: el problema está en las líneas 24..29). ¿Debemos simplemente ignorar dicho camino?

C) Modifica el código para eliminar las sentencias que nunca se va a ejecutar. Representa el nuevo grafo y obtén el conjunto de casos de prueba (aplicando el método del camino básico). ¿Podemos asegurar que el código es correcto? Piensa en algún caso de prueba adicional que pudiera poner de manifiesto un defecto en nuestro código.

Sube tu solución a Bitbucket

➪ ➪ Ejercicio 3: método *realizaReserva()*

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

Se quiere llevar a cabo la reserva de una serie de libros 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, 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 realizaReserva), el cual puede lanzar varias excepciones. Si la reserva puede realizarse con éxito, el método realizaReserva termina normalmente. En el caso de que no se pueda hacer efectiva la reserva de algún libro, el método realizaReserva() 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 ";".

Las excepciones que puede provocar el método que hace efectiva la reserva de un libro son:

- `IsbnInvalidoExceptio, 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).
- `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.
- `JDBCException`, con el mensaje `"CONEXION invalida"`, si no se puede acceder a la BD.

Por ejemplo: suponiendo que el login y password del bibliotecario son "biblio", "1234", que el identificador de socio existe en la base de datos, que la lista de isbns es (12345, 23456, 34567), y que el segundo y tercer isbns son incorrectos, se obtendría como resultado una excepción de tipo ReservaException con el mensaje: "ISBN invalido: 23456; ISBN invalido: 34567;"

En la Figura 1 se muestra una implementación del método re***alizaReserva()***.

A partir del código y la especificación proporcionadas, diseña una tabla de casos de prueba para el método *realizaReserva().*

> **NOTA**: En este ejemplo 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 pueden ocurrir dos cosas: (a) O bien se completa la especificación por quien corresponda y añade el nuevo comportamiento a S, o bien (b) Se elimina el código con el comportamiento no especificado.

Recuerda subir tu solución a Bitbucket.

IMPORTANTE: Necesitarás las tablas con los casos de prueba de los ejercicios de esta sesión para poder realizar la próxima práctica, en la que automatizaremos la ejecución de nuestras pruebas con JUnit5.

```
1. public void realizaReserva(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.}
```

Figura 1. Implementación del método *realizaReserva()*

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 pero no implementados.

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 conjunto de caminos independientes sea $\leq$ que CC).
- Es posible que haya caminos imposibles o que detectemos comportamientos implementados pero no especificados.
- Los datos de entrada y los datos de salida esperados siempre deben ser concretos.