P04- Diseño de pruebas de caja negra

Diseño de pruebas de caja negra (functional testing)

En esta sesión aplicaremos el método de **diseño** de casos de prueba visto en clase para obtener conjuntos de casos de prueba de unidad (método java), partiendo de la especificación de dicha unidad (conjunto S). Recuerda que no sólo se trata reproducir los pasos de los métodos de forma mecánica, sino que, además, debes tener muy claro qué es lo que estás haciendo en cada momento, para así asimilar los conceptos explicados.

Debes tener claro el objetivo particular del método de diseño con el que vamos a practicar (particiones equivalentes), y que cualquier método de diseño nos proporciona una forma sistemática de obtener un conjunto de casos de prueba eficiente y efectivo. Obviamente, esa "sistematicidad" tiene que "verse" claramente en la resolución del ejercicio, por lo que tendrás que dejar MUY CLAROS todos y cada uno de los pasos que vas siguiendo.

Insistimos de nuevo en que el trabajo de prácticas tiene que servir para entender y asimilar los conceptos de la clase de teoría, y no al revés.

En esta sesión no utilizaremos ningún software específico, pero en la siguiente automatizaremos la ejecución de los casos de prueba (unitarias) que hemos diseñado aquí, por lo que necesitarás tus soluciones de esta práctica para poder trabajar en la próxima clase

Bitbucket

El trabajo de esta sesión también debes subirlo a *Bitbucket*. Todo el trabajo de esta práctica, que no será código, deberá estar en el directorio **P04-CajaNegra**, dentro de tu espacio de trabajo, es decir, dentro de tu carpeta: ppss-2020-Gx-apellido1-apellido2. Puedes subir los ficheros en formato png, jpg, pdf (al margen de que también subas ficheros con extensión doc, xml, u otros formatos particulares dependiendo de las herramientas de edición que uses).

Recuerda que si trabajas desde los ordenadores del laboratorio primero deberás configurar git y clonar tu repositorio de Bitbucket.

Ejercicios

A continuación proporcionamos la especificación de las unidades a probar (hemos definido una unidad como un método java). Se trata de **diseñar los casos de prueba** para cada una de las especificaciones utilizando el método de diseño de particiones equivalentes. Recuerda indicar claramente las entradas, salidas, las agrupaciones o no de las entradas, particiones válidas y no válidas de cada una de las entradas/salidas y/o agrupaciones, combinaciones de particiones, y los valores concretos para las entradas y salidas en la tabla de casos de prueba.

➪ ➪ Ejercicio 1: especificación *importe_alquiler_coche()*

En una aplicación de un negocio de alquiler de coches Dada la siguiente especificación, se quiere diseñar una tabla de casos de prueba para la unidad ***importe_alquiler_coche()*** utilizando el método de diseño de particiones equivalentes. Dicha unidad calcula el importe del alquiler de un determinado tipo de coche, durante un cierto número de días, a partir de una fecha concreta, y devuelve el importe del alquiler. Si no es posible realizar los cálculos devuelve una excepción de tipo ReservaException. El prototipo del método es el siguiente:

unidad

```
public float importe_alquiler_coche (TipoCoche tipo, Date fecha_inicio,
                                     int num_dias) throws ReservaException
```

TipoCoche es un tipo enumerado cuyos posibles valores son: (TURISMO, DEPORTIVO). Nos indican que si la fecha de inicio proporcionada no es posterior a la actual, entonces se lanzará la excepción ReservaException con el mensaje "Fecha no correcta". Si el tipo de coche no está disponible durante los días requeridos, o se intenta hacer una reserva de más de 30 días, entonces se lanzará la excepción ReservaException con el mensaje "Reserva no posible". El precio de la reserva por día depende del número de días reservados, según la siguiente tabla:

1 día	100 euros
2 días o más	50 euros/dia

Diseña los casos de prueba para la especificación anterior utilizando el método de particiones equivalentes.

➪ ➪ Ejercicio 2: especificación *generaTicket()*

En una aplicación de un comercio, tenemos un método que genera tickets de venta en función de los artículos comprados por un determinado cliente. Concretamente, el prototipo del método será el siguiente:

unidad

```
public Ticket generaTicket(Cliente cliente,
                           List<String> codArticulos)
                                 throws BOException;
```

Para dicho método, dados un cliente y la lista de artículos que desea comprar (identificados por su código), el método *generaTicketc()* genera un ticket de compra que incluye, para **cada artículo**, las **unidades** solicitadas, y el **precio total** para dicho **artículo**. También incluye el precio **total** de la **compra** (resultante de sumar todos los totales de todos los artículos comprados). La lista de artículos puede contener **códigos repetido**s. Si por ejemplo queremos comprar dos unidades de un artículo, el código de ese artículo puede aparecer dos veces en la lista. Cada **cliente** se caracteriza por su **nif**, y su **estado**. El nif habrá sido validado previamente en otra unidad. En el caso de que no tengamos registrado el nif del cliente, o se trate de un cliente con valor null, se lanzará la excepción **BOException** con el mensaje ***"El cliente no puede realizar la compra"***. El **estado** del cliente puede ser "**normal**", si no tiene cuentas pendientes de abonar, o "moroso" (si tiene pagos pendientes). En el caso de que el cliente sea "**moroso**", se comprobará si su deuda es superior a 1000 euros, en cuyo caso se generará la excepción BOException con el mensaje ***"El cliente no puede realizar la compra"*** (en otro caso se continuará con el proceso de compra). Si alguno de los artículos no existe en la base de datos se generará la excepción BOException con el mensaje ***"El artículo no está en la BD"***. Si en algún momento se produce un error de acceso a la base de datos se generará la excepción BOException con el mensaje ***"Error al recuperar datos del articulo"***.

Diseña los casos de prueba para la especificación anterior utilizando el método de particiones equivalentes.

A continuación se proporcionan las estructuras de datos utilizadas en el método a probar:

```
public class Cliente {
    String nif;
    EstadoCliente estado;
    float deuda; ...
}
enum EstadoCliente {normal,moroso};
```

```
public class Articulo {
    String cod;
    float precioUnitario;
    ...
}
```

```
public class Ticket {
  Cliente cliente;
  List<LineaVenta> lineas;
  float precioTotal; ...
}
```

```
public class LineaVenta {
  Articulo articulo;
  int unidades;
  float precioLinea;
}
```

➪ ➪ Ejercicio 3: especificación *matriculaAlumno()*

Supongamos que queremos probar un método que realiza el proceso de matriculación de un alumno. Se trata del método `MatriculaBO.matriculaAlumno()`:

unidad

```
public MatriculaTO matriculaAlumno(AlumnoTO alumno,
                List<AsignaturaTO> asignaturas) throws BOException
```

Dicho método tiene como entradas los datos de un alumno, contenidos en un objeto AlumnoTO más la lista de asignaturas de las que se quiere matricular. El método devuelve un objeto MatriculaTO que contiene: la información sobre el alumno, la lista de asignaturas de las que se ha matriculado con éxito, y una lista con el informe de error para cada una de la asignaturas de las que no se haya podido matricular. Los tipos de datos que vamos a utilizar son los siguientes:

```
public class AlumnoTO implements Comparable<AlumnoTO> {
    String nif; // NIF del alumno
    String nombre; // Nombre del alumno
    String direccion;  // Direccion postal del alumno
    String email; // Direccion de correo electronico del alumno
    List<String> telefonos; // Lista de telefonos del alumno
    Date fechaNacimiento;  // Fecha de nacimiento del alumno
    ...
}
```

```
public class AsignaturaTO {
    int codigo;
    String nombre;
    float creditos;
    ...
}
```

```
public class MatriculaTO {
    AlumnoTO alumno;
    List<AsignaturaTO> asignaturas;
    List<String> errores;
    ...
}
```

El método `matriculaAlumno()`, dada la información sobre los datos del **alumno** y las **asignaturas** de las que se quiere matricular, hace efectiva la matriculación, actualizando la base de datos utilizando la información que recibe como entrada. Asume que el método recibirá un objeto de tipo `AlumnoTO` no nulo. Los datos del alumno (excepto el nif) han sido validados previamente. En el caso particular del nif, éste podrá ser nulo, un nif válido, o un nif no válido.

Si el nif del alumno es nulo, se devuelve un error (de tipo **BOException**). con el mensaje ***"El nif no puede ser nulo"***. **Nota**: a menos que se diga lo contrario cuando se devuelve un error, éste será de tipo BOException.

Si el nif no es válido, devolverá el mensaje de error: ***"Nif no válido"***. Si el alumno no está dado de alta en la base de datos, se procederá a dar de alta a dicho alumno (con independencia de que luego se produzca un error o no en las asignaturas a matricular). Al comprobar si el alumno está o no dado de alta, puede ser que se produzca un error de acceso en la base de datos, generándose el mensaje de error ***"Error al obtener los datos del alumno"***, o bien que se produzca algún error durante el proceso de dar de alta, generándose un error con el mensaje: ***"Error en el alta del alumno"***. Si la lista de asignaturas de matriculación es vacía o nula, se genera el mensaje de error: ***"Faltan las asignaturas de matriculación"***. De la misma forma, si para alguna de las asignaturas de la lista ya se ha realizado la matrícula, se devolverá el mensaje de error: ***"El alumno con nif nif_alumno ya está matriculado en la asignatura con código código_asignatura"***. **Nota**: asumimos que todas las asignaturas que pasamos como entrada ya existen en la BD.

El número máximo de asignaturas a matricular será de 5. Si se rebasa este número se devolverá el error ***"El número máximo de asignaturas es cinco"***. El proceso de matriculación sólo se llevará a cabo si no ha habido ningún error en todas las comprobaciones anteriores. Durante el proceso de matricula, puede ocurrir que se produzca algún error de acceso a la base de datos al proceder al dar de alta la matrícula para alguna de las asignaturas. En este caso, para cada una de las asignaturas que no haya podido hacerse efectiva la matrícula se añadirá un mensaje de error en la lista de errores del objeto MatriculaTO. Cada uno de los errores consiste en el mensaje de texto: ***"Error al matricular la asignatura cod_asignatura"*** (siendo cod_asignatura el código de la asignatura correspondiente). El campo asignaturas del objeto MatriculaTO contendrá una lista con las asignaturas de las que se ha matriculado finalmente el alumno.

Diseña los casos de prueba para la especificación anterior utilizando el método de particiones equivalentes. En el caso del objeto de tipo ***AlumnoTO***, puedes considerar como dato de entrada únicamente el atributo ***nif***. En el caso de los objetos de tipo ***AsignaturaTO*** puedes considerar únicamente el dato de entrada el atributo ***codigo***, que representa el código de la asignatura.

Resumen

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

MÉTODOS DE DISEÑO DE CAJA NEGRA

- Permiten seleccionar de forma SISTEMÁTICA un subconjunto de comportamientos a probar a partir de la especificación. Se aplican en cualquier nivel de pruebas (unitarias, integración, sistema, aceptación), y son técnicas dinámicas.
- El conjunto de casos de prueba obtenidos será eficiente y efectivo (exactamente igual que si aplicamos métodos de diseño de caja blanca, de hecho, la estructura de la tabla obtenida será idéntica).
- Pueden aplicarse sin necesidad de implementar el código (podemos obtener la tabla de casos de prueba mucho antes de implementar, aunque no podremos detectar defectos sin ejecutar el código de la unidad a probar).
- No pueden detectar comportamientos implementados pero no especificados.

MÉTODO DE PARTICIONES EQUIVALENTES

- Cada entrada y salida de la unidad a probar se particiona en clases de equivalencia (todos los valores de una partición de entrada tendrán su imagen en la misma partición de salida). Cada partición (tanto de entrada como de salida) se etiqueta como válida o inválida. Todas las particiones deben ser disjuntas.
- El objetivo es proporcionar el número mínimo de casos de prueba que garanticen que estamos probando todas y cada una de las particiones, y que todas las particiones inválidas se prueban de una en una (sólo puede haber una partición inválida de entrada en cada caso de prueba).
- Cada caso de prueba será una selección de un comportamiento especificado. Puede ocurrir que la especificación sea incompleta, de forma que al hacer las particiones, no podamos determinar el resultado esperado. En ese caso debemos poner un "interrogante". El tester NO debe completar/cambiar la especificación.