# P4- Diseño de pruebas de caja negra

## Diseño de pruebas de caja negra *(functional testing)*

El objetivo de esta práctica es aplicar los métodos de diseño de casos de prueba vistos en clase para obtener conjuntos de casos de prueba de unidad (método java), y pruebas de sistema, partiendo de la especificación del correspondiente elemento a probar. Utilizaremos el método de particiones equivalentes para diseñar pruebas unitarias, y el método de transición de estados para diseñar pruebas de sistema. 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.

En esta sesión no utilizaremos ningún software, pero en la siguiente sesión 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 la próxima clase.

## Bitbucket

El trabajo de esta sesión también debes subirlo a Bitbucket. Todo el trabajo de esta práctica, tanto el código fuente, como cualquier otro documento con vuestras notas de trabajo, deberán estar en la carpeta **P4** de vuestro repositorio**.**

## Ejercicios

A continuación proporcionamos la especificación de los elementos a probar. Se trata de diseñar los casos de prueba para cada una de las especificaciones utilizando uno de los dos métodos que hemos visto en clase.

1. **ESPECIFICACIÓN 1.** 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 es el siguiente:

```
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), 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 repetidos. Si por ejemplo queremos comprar dos unidades de un artículo, el código de ese artículo deberá 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”. Si alguno de los artículos no existen 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;
}
```

2. **ESPECIFICACIÓN 2.** Supongamos que queremos probar un método que realiza el proceso de matriculación de un alumno. Se trata del método `MatriculaBO.matriculaAlumno()`:

```
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 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 ya en la base de datos, se procederá a dar de alta a dicho alumno (independientemente 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 `codigo`, que representa el código de la asignatura.

**Nota sobre la base de datos:** Tenemos las tablas **alumnos**, **asignaturas**, y **matricula**. Las tablas **alumnos** y **asignaturas** contienen la información de las entidades correspondientes (ver objetos `AlumnoTO`, `AsignaturaTO` y `MatriculaTO`). Los campos de la tabla **matricula** son el nif del alumno (clave ajena de la tabla **alumnos**) y el código de la asignatura (clave ajena de la tabla **asignaturas**). La clave primaria de la tabla **matricula** es la combinación de ambos campos.

3. **ESPECIFICACIÓN 3.** Se trata de probar las acciones realizadas en un terminal de un supermercado. Para poder utilizar el sistema, el encargado del terminal deberá autenticarse en el sistema utilizando un código de acceso (se permite cualquier número de intentos). Una vez autenticado, podrá comenzar a pasar por el escáner los productos (de uno en uno) que el cliente desee comprar. Cada vez que se escanee un producto, se mostrara el subtotal del importe de la compra en la pantalla. Si el escáner no detecta bien el código de un producto, será necesario volver a situar dicho producto por el escáner. Finalmente se procederá al pago de la compra. Se admitirá cualquier tarjeta de pago de cualquier banco. Si se produce un error al procesar la tarjeta, se deberá volver a intentar el procesamiento del pago. Una vez procesado el pago, se procederá a entregar el ticket de compra al cliente, mostrando los códigos de los productos adquiridos, su precio, el número de unidades y el total de la compra. El cliente puede cancelar la compra en el momento del pago. Cuando termina su turno, el encargado del terminal procederá a "salir" del sistema, teniendo en cuenta que deberá completar el proceso de compra para el cliente que en ese momento esté en curso. Aplica el método de transición de estados para diseñar la tabla de casos de prueba correspondiente.