P03- Diseño de pruebas de caja negra

Os recordamos que el plazo para entregar los ejercicios de esta práctica (P03) termina justo ANTES de comenzar la siguiente sesión de prácticas (P04) en el laboratorio (los grupos de los lunes, por lo tanto, el próximo lunes, los de los martes, el próximo martes, ...). Pasado este plazo los profesores no revisarán vuestro trabajo sobre P03 (de forma individual), con independencia de que se indiquen en clase las soluciones que os permitirán saber si las vuestras son correctas.

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

En esta sesión aplicaremos el método de **diseño** de casos de prueba de caja negra explicado 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 saber qué es lo que estás haciendo en cada momento, para así asimilar los conceptos explicados.

Es importante tener presente el objetivo particular del método de diseño usado (**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 y seguir las normas explicadas en clase sobre cómo indicar las entradas, salidas, particiones, ...

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.

Bitbucket

El trabajo de esta sesión también debes subirlo a *Bitbucket*. Todo el trabajo de esta práctica deberá estar en el directorio **P03-CajaNegra**, dentro de tu espacio de trabajo, es decir, dentro de tu carpeta: ppss-2022-Gx-apellido1-apellido2. Puedes subir los ficheros en formato png, jpg, pdf, texto, o con extensión .md.

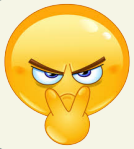
Cada uno de los ejercicios debería estar contenido en **un único fichero**. Si hay más de un fichero por ejercicio, debería estar claro en el nombre del fichero lo que contiene cada uno.

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** a partir de las especificaciones utilizando el método de diseño de **particiones equivalentes**. Recuerda indicar CLARAMENTE:

- cada entrada y salida
- las agrupaciones o no de las entradas,
- las particiones válidas y no válidas (convenientemente etiquetadas) de cada una de las entradas/ salidas y/o agrupaciones,
- las combinaciones de particiones asociadas a cada caso de prueba, y
- los valores concretos para las entradas y salidas en la tabla de casos de prueba.

Recuerda que tienes que indicar las asunciones sobre los datos de entrada de la tabla en el caso de que sea necesario, y que la tabla siempre tiene que tener valores concretos.



Tienes que concretar TODAS las entradas y salidas esperadas de la tabla porque, independientemente de quién vaya a automatizar la ejecución de los casos de prueba, los drivers asociados a todos ellos tienen que ejercitar los MISMOS comportamientos que has seleccionado en dicha tabla..

➪ ➪ Ejercicio 1: especificación *importe_alquiler_coche()*

Crea la subcarpeta "**importe_alquiler**", en la que guardarás tu solución para este ejercicio.

En una aplicación de un negocio de alquiler de coches necesitamos una unidad denominada ***importe_alquiler_coche()***. 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 de dicho alquiler. Si no es posible realizar los cálculos devuelve una excepción de tipo *ReservaException*. El prototipo del método es el siguiente:

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

TipoCoche es un tipo enumerado cuyos posibles valores son: (TURISMO, DEPORTIVO). Asumimos que la fecha de inicio ha sido validad en otra unidad. 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 teniendo en cuenta la especificación anterior utilizando el método de particiones equivalentes.

➪ ➪ Ejercicio 2: especificación *generaEventos()*

Crea la subcarpeta "**generaEventos**", en la que guardarás tu solución para este ejercicio.

En una aplicación de matriculación de una universidad, queremos implementar una unidad denominada ***generaEventos(),*** que devuelve una lista de eventos de calendario para todas las sesiones de clase de una asignatura, (asumiremos 1 única clase por semana), o bien una excepción de tipo *ParseException*, de acuerdo con las siguientes reglas sobre sobre las asignaturas:

- **R1**. Asumimos que ni el nombre de la asignatura ni ninguno de los objetos que representan la fecha (tipo LocalDate) serán null (y además la fecha será correcta).
- **R2.** Si la hora de inicio tiene un formato o valores incorrectos, o el día de la semana no es uno de los valores válidos, se devolverá una instancia de *ParseException*
- **R3.** La fecha de inicio especificada puede ser posterior a la de fin. En tal caso se devolverá una lista de eventos vacía. También se devolverá una lista de eventos vacía si el día de la semana no está incluido en el rango de fechas del curso académico (por ejemplo, si la fecha de inicio de curso fuese miércoles (23/02/22) y la fecha de fin el viernes (25/02/22), y el día de la semana en la que se imparte la asignatura fuese los martes, la salida será una lista vacía)
- **R4**. Si la hora de inicio es null, se considerará un evento de todo el día y la duración será -1 (la hora de inicio del evento también será null). En caso contrario, la duración contendrá un valor de 120 minutos.

El prototipo del método a probar será el siguiente:

```
List<EventoCalendario> generaEventos(HorarioAsignatura horario) throws ParseException;
```

Los tipos *HorarioAsignatura* y *EventoCalendario* se definen como:

```
public class HorarioAsignatura {
   String asignatura;
   LocalDate fechaInicioCurso;
   LocalDate fechaFinCurso;
   String horaInicioClase; // Formato “hh:mm”
   int diaSemana;  // Valores válidos:
                   //    1(lunes),2,3,4,5,6(sábado)
}
```

```
public class EventoCalendario {
   String nombreAsig;
   LocalDate fechaDeSesion;
   String horaInicio;
   int duracion;
}
```

Diseña los casos de prueba teniendo en cuenta la especificación anterior utilizando el método de particiones equivalentes.

➪ ➪ Ejercicio 3: especificación *matriculaAlumno()*

Crea la subcarpeta "**matriculaAlumno**", en la que guardarás tu solución para este ejercicio.

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. 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 con la información que recibe por parámetros. Asumimos que el parámetro alumno nunca va a ser nulo. Los datos del alumno (excepto el nif) han sido validados previamente.

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

Si el nif no es válido, devolverá una excepción con 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 la excepción con el mensaje ***"Error al obtener los datos del alumno"***, o puede que se produzca algún error durante el proceso de dar de alta, generándose una excepción con el mensaje: ***"Error en el alta del alumno"***. Si la lista de asignaturas de matriculación es vacía o nula, se producirá una excepción con el mensaje: ***"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 la excepción con el mensaje: ***"El alumno con nif nif_alumno ya está matriculado en la asignatura con código código_asignatura"***, siendo ***nif_alumno*** y ***codigo_asignatura*** el nif del alumno y el código de la asignatura, respectivamente.

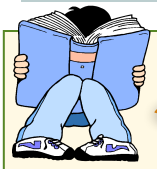
Nota1: asumimos que todas las asignaturas que pasamos como entrada ya existen en la BD.

Nota2: todos los accesos a la BD se realizan a través de una unidad externa.

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

- Necesitamos identificar previamente todas las entradas y salidas de la unidad a probar para poder aplicar correctamente el método. Este paso es igual de imprescindible si aplicamos un método de diseño de pruebas de caja blanca., ya que de ello dependerá la estructura de la tabla de casos de prueba obtenida.
- 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). Las particiones pueden realizarse sobre cada entrada por separado, o sobre agrupaciones de las entradas, (si la validez de una partición de entrada, depende de otra/s de las entradas, entonces hay qua agruparlas y particionar todas ellas a la vez). Cada partición (tanto de entrada como de salida, agrupadas o no) 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 en cada caso de prueba (sólo puede haber una partición de entrada inválida en cada caso de prueba). Para ello es importante seguir un orden a la hora de combinar las particiones: primero las válidas, y después las inválidas, de una en una.
- Cada caso de prueba será 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" como resultado esperado. El tester NO debe completar/cambiar la especificación.