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 de caja negra explicado en clase para obtener conjuntos de casos de prueba de unidad (método java). Realizaremos la selección de comportamientos a probar a partir 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.

GitHub

El trabajo de esta sesión también debes subirlo a *GitHub*. Todo el trabajo de esta práctica deberá estar en el directorio **P04-CajaNegra**, dentro de tu espacio de trabajo, es decir, dentro de la carpeta que contiene el directorio oculto .git. Cada ejercicio tendrá su subcarpeta propia.

Las soluciones debes guardarlas en ficheros de **imagen (jpg, png)**, o en formato de **texto (.txt, .md).**

Los nombres de los ficheros se indican en cada uno de los ejercicios.

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**: identifícalas con letras mayúsculas, p.ej. Entrada 1 (A) edad
- las **agrupaciones** de las entradas, si procede (p.ej. Entradas 2+3 (B): fecha+listaFacturas). Debes dejar claro qué entradas estás agrupando.
- las **particiones válidas** y **no válidas** (convenientemente etiquetadas) de cada una de las entradas/salidas y/o agrupaciones. Para cada partición no válida usa el prefijo **N**, p.ej. NA1, NA2 (si usamos A para identificar las particiones válidas de una entrada)
- las **combinaciones** de particiones ejercitadas por cada caso de prueba, y
- los **valores concretos** para las entradas y salidas en la tabla de casos de prueba.

Recuerda también que:

- tienes que indicar las **asunciones sobre los datos de entrada** de la tabla en el caso de que sea necesario
- los **valores** de la tabla siempre tiene ser **CONCRETOS** (porque, independientemente de quién vaya a automatizar la ejecución de los casos de prueba, los drivers asociados a todos ellos tienen que ejercitar exactamente los comportamientos representados en la tabla)
- **no debes asumir nunca un resultado esperado** que NO esté indicado en la especificación En ese caso usaremos particiones de salida con valor ?. P.ej. la partición NS1 de la tr. 17 de S04
- cuando apliques las **heurísticas**, deberás tener en cuenta que vamos a implementar en **Java**, y por lo tanto, tendrás que considerar el hecho de que cualquier tipo no primitivo en Java puede tomar el **valor null** (por ejemplo: un array, un tipo enumerado, o una cadena de caracteres).

➪ ➪ Ejercicio 1: especificación *importe_alquiler_coche()*

Crea la subcarpeta "**importe_alquiler**", en la que guardarás tu solución para 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 "**alquiler-<sufijo>.<extension>**", siendo **<sufijo>** la indicación del paso o pasos seguidos: por ejemplo alquiler-paso1.jpg, alquiler-pasos2-3.jpg,... o simplemente "**alquiler.<extension>**" si todo el ejercicio está resuelto en un fichero. **<extension>** denota el tipo de fichero: jpg, png, ...

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 validada 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 ó 2 días	100 euros/dia
más de 2 días	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.

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

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 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 una fecha válida).
- **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 (22/02/23) y la fecha de fin el viernes (24/02/23), 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 de la clase 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 *agregarGrabacion()*

Crea la subcarpeta "**agregarGrabacion**", en la que guardarás tu solución para 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 "**agregarGrabacion-<sufijo>.<extension>**

En una aplicación para gestionar la grabación de programas de TV queremos implementar un método para añadir una nueva grabación a una lista de grabaciones. El prototipo del método a probar es el siguiente :

```
void agregarGrabacion(ArrayList<Grabacion> grabaciones, Grabacion p)
                                                throws GrabacionException;
```

Cada grabación (clase **Grabacion**) está formada por los campos **canal**, **inicio** y **fin**. Todos de tipo entero, que representan el número de canal de la grabación, y el minuto inicial y final de dicha grabación, respectivamente.

Los canales están numerados desde 1 hasta 20. El sistema de grabación permite grabar en cualquiera de los canales disponibles, pero no puede simultanear dos grabaciones, es decir, las grabaciones no pueden "solaparse" en el tiempo.

En la lista de grabaciones nunca hay elementos que "solapan" entre ellos y en todo momento está ordenada por los valores de inicio de cada elemento de la lista. El sistema aceptará grabaciones con una duración mínima de un minuto.

Asumiremos que los valores de los atributos inicio y fin de la grabación a añadir siempre tendrán valores comprendidos entre 1 y 1440 (que son los minutos de un día). Y también asumiremos que la lista de grabaciones nunca va a tener el valor null.

La nueva grabación que se pasa por parámetro se añadirá a la lista sólo si no solapa con ninguna otra ya existente en dicha lista, en cuyo caso se añadirá en la posición que corresponda para mantener la lista ordenada.

Si la grabación que se quiere añadir solapa con alguna de las de la lista se lanzará la excepción GrabacionException con el mensaje "Solape de grabaciones". Si el canal indicado no existe se lanzará la excepción GrabacionException con el mensaje "No existe el canal", Si el intervalo de grabación a añadir tiene una duración cero, entonces se lanzará la excepción GrabacionException con el mensaje "Error en intervalo de grabación"

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

➪ ➪ ANEXO: Observaciones a tener en cuenta sobre la práctica P03

- El algoritmo de un drive (test JUnit) SIEMPRE es el mismo:
  1. ARRANGE: Preparamos loa datos de entrada y anotamos el resultado esperado

2. ACT: Invocamos a la SUT con los datos de entrada y obtenemos el resultado real
3. ASSERT: Verificamos que el resultado esperado coincide con el real.

Las tres secciones deben estar **claramente separadas** (a menos que JUnit nos lo impida)

- Evitaremos siempre duplicaciones innecesarias de código usando alguna de las anotaciones @BeforeEach, @AfterEach, @BeforeAll, @AfterAll. De momento sólo vamos a usar la anotación **@BeforeEach** (más adelante usaremos el resto de las anotaciones)
- La sentencia **assertDoesNotThrow()** tiene dos variantes:
  a) método void
  b) método que devuelve el resultado de la expresión lambda que se pasa por parámetro

  En los ejercicios de prácticas deberéis usar ambos. EN NINGÚN caso se permitirá que el método anotado con @Test propague la excepción, o la capture (ya debes saber por qué)
- Tenéis que usar el método **assertArrayEquals()**, para comparar dos objetos de tipo array
- Siempre que sea posible tenéis que **agrupar las aserciones** en una única sentencia, ya debes saber por qué.
- Cuando configuramos el valor de la variable "groups" y usamos varias etiquetas separadas por "coma", ejecutaremos los tests que contengan cualquiera de ellas (es decir, no es una operación AND)

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).
- 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. Es imprescindible tener claras qué entradas y salidas tendrá la unidad a probar (con este método y con cualquier otro, sea de caja blanca o caja negra), ya que de ello dependerá la estructura de la tabla de casos de prueba obtenida.
- Los valores posibles para 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. Además todas las particiones son disjuntas, y la unión de todas las particiones de una entrada tiene que contener todos los valores posibles para dicha entrada). 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 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.
- El objetivo es proporcionar el número mínimo de casos de prueba que garanticen que estamos probando todas y cada una de las particiones, al menos una vez, 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 definimos la partición de salida correspondiente con el valor "interrogante". El tester NO debe completar/cambiar la especificación.