P05- Dependencias externas 1: stubs

Dependencias externas

En esta sesión implementaremos **drivers** para automatizar pruebas unitarias dinámicas, teniendo en cuenta que las unidades a probar pueden tener **dependencias externas**, que necesitaremos controlar a través de sus dobles. El objetivo es realizar las pruebas "aislando" la ejecución del código de nuestra unidad. De esta forma nos aseguramos de que cualquier defecto que detectemos estará exclusivamente en el código de nuestra SUT, excluyendo así cualquier código del que dependa.

Las dependencias externas (métodos) tendremos que sustituirlas por sus dobles, concretamente por STUBS, por lo que la idea es que el doble controle las **entradas indirectas** de nuestro SUT. El doble reemplazará durante las pruebas a la dependencia externa real que se ejecutará en producción.

Recuerda que NO se puede alterar "temporalmente" el código a probar (SUT), pero sí se puede REFACTORIZAR, para que el código contenga un ***SEAM*** (uno por dependencia externa), de forma que sea posible inyectar el doble durante las pruebas, el cual reemplazará al colaborador correspondiente. Es importante que tengas claro que hay diferentes refactorizaciones posibles y que cada una de ellas tiene diferentes "consecuenciaas" para el resto del código en producción.

IMPORTANTE: no usaremos dobles para getter/setters, ni tampoco para métodos de la librería estándar de Java (a menos que queramos controlar lo que devuelve a la sut).

Los drivers que vamos a implementar realizan una **verificación basada en el estado**, es decir, el resultado del test depende únicamente del resultado de la ejecución de nuestra SUT. Observa que el algoritmo del driver es el mismo que el de sesiones anteriores, pero añadiendo más acciones en la fase de preparación de los datos, ya que tendremos que crear el doble e inyectarlo en nuestra SUT, antes de ejecutarla. Para implementar los drivers usaremos JUnit5. Para ejecutarlos usaremos Maven y Junit5.

GitHub

El trabajo de esta sesión también debes subirlo a *GitHub*. Todo el trabajo de esta práctica deberá estar en el directorio **P05-Dependencias1**, dentro de tu espacio de trabajo (carpeta que contiene el directorio oculto .git)

Ejercicios

En las sesiones anteriores, hemos trabajado con un proyecto IntelliJ que solamente contenía un proyecto Maven. En esta sesión vamos a usar **un proyecto** IntelliJ, que estará formado por varios ***módulos*** (cada módulo será un proyecto Maven). Recuerda que IntelliJ define un módulo como una unidad funcional, que podemos compilar, ejecutar, probar y depurar de forma independiente (ver pág. 5 de *P01-Pruebas_del_Software.pdf*).

Para **crear el proyecto IntelliJ**, simplemente tendremos que realizar lo siguiente:

- ***File→New Project***. Seleccionamos “Empty Project”
- ***Name*** : **P05-stubs**
- ***Location:*** *$HOME/ruta_de_tu_directorio_de_trabajo/P05-Dependencias1/*

  Es decir, tenemos que crear el proyecto dentro de nuestro directorio de trabajo, y del subdirectorio P05-Dependencias1.

IntelliJ habrá creado el proyecto ***P05-stubs*** con un único módulo, con el mismo nombre.

Vamos a eliminar este módulo porque no lo necesitamos. Para ello accedemos a ***File→Project Structure →Project Settings →Modules***.

Desde la ventana ***Project Structure***, seleccionamos el módulo P05-stubs y lo eliminamos pulsando sobre el icono con un signo "-". Nos preguntará si estamos seguros y le diremos que si.

OBSERVACIONES A TENER EN CUENTA PARA REALIZAR LOS EJERCICIOS.

Recuerda que debes **modificar el pom** para poder ejecutar tus tests JUnit a través del plugin *surefire*. Esto lo tendrás que hacer **para cada módulo nuevo** que añadamos al proyecto. Cuando modifiques el pom, para asegurarte de que Intellij "se ha dado cuenta" de dicho cambio, puedes usar la opción "***Maven→Reload Project***" desde el menú contextual del **módulo** que contiene el fichero pom.xml

- La configuración del ***plugin surefire*** será la misma que hemos usado en la práctica de drivers. De esta forma, veremos siempre los informes de maven en forma de árbol.

Debéis seguir las siguientes **normas para nombrar las nuevas clases** que necesitaréis añadir para implementar los drivers:

- A la clase que contiene la i**mplementación del doble** la debes llamar igual que la clase de la dependencia externa añadiéndole el sufijo **Stub**.
- Si el doble pertenece a la misma clase que la SUT, el doble lo implementarás en una clase con el mismo nombre que la clase que contiene la SUT añadiéndole el sufijo **Testable**.

Debéis seguir SIEMPRE las **normas** que ya hemos explicado **para implementar los drivers**:

- Hay que usar el patrón AAA.
- Los nombres de los tests deben ser *id_<sut>_should_<esperado>_when_<condiciones>*.

IMPORTANTE: Para implementar el driver tenemos que: detectar (PRIMERO) las dependencias externas, (SEGUNDO) comprobar si nuestro SUT es *testable* y refactorizar SOLO si es necesario, (TERCERO) implementar los dobles, y finalmente implementar el driver. Tienes que tener claro cada uno de los pasos para saber lo que estás haciendo en cada momento. Esto debes hacerlo para TODOS los ejercicios.

➪ ➪ Ejercicio 1: *drivers* para `calculaConsumo()`

Vamos a añadir un primer módulo a nuestro proyecto *P05-stubs*. Lo podemos hacer desde ***File→New→Module...***y nos aseguramos que en el menú de la izquierda esté seleccionado ***Java***:

- ***Name***: ***gestorLlamadas***
- ***Location***: *$HOME/ruta_de_tu_directorio_de_trabajo/P05-Dependencias1/**P05-stubs/***
- Seleccionamos la herramienta de construcción ***Maven***, y nos aseguramos de elegir el ***JDK 21***
- ***Parent:*** *<None>*
- Desde ***Advanced Settings***, indicamos las coordenadas de nuestro proyecto: ***GroupId***: *ppss.P05* ***ArtifactId:*** ***gestorLlamadas***

Finalmente pulsamos sobre ***Create.***

Si accedes de nuevo a la estructura de ese módulo con ***File→Project Structure→Project Settings→Modules→gestorLlamadas***, en la pestaña ***Sources*** veras cómo ha marcado IntelliJ los diferentes directorios de dicho proyecto maven.

Una vez que hemos creado el **módulo gestorLlamadas,** debes automatizar pruebas unitarias dinámicas sobre el método **GestorLlamadas.calculaConsumo()** (que pertenece al paquete **llamadas**) utilizando verificación basada en el estado.

Dicho método calcula el importe asociado una llamada teniendo en cuenta que se factura por minutos y que se aplicará una tarifa diurna de 20.7 si la llamada se realiza entre las 8:00 y 20:00, y una tarifa noctura 12.2 si la llamada se realiza en otra hora.

A continuación indicamos el código de nuestra SUT, y los casos de prueba que queremos automatizar.

```
//paquete llamadas

public class GestorLlamadas {
  private static final double TARIFA_NOCTURNA=12.2;
  private static final double TARIFA_DIURNA=20.7;
  public Calendario getCalendario() {
    Calendario c = new Calendario();
    return c;
  }

  public double calculaConsumo(int minutos) {
    Calendario c = getCalendario();
    int hora = c.getHoraActual();
    if(hora < 8 || hora > 20) {
      return minutos * TARIFA_NOCTURNA;
    } else {
      return minutos * TARIFA_DIURNA;
    }
  }
}
```

	minutos	hora	Resultado esperado
C1	10	12	207
C2	10	21	122

Debes tener claro en qué DIRECTORIOS debes situar cada uno de los fuentes.

Recuerda que el código en producción estará en src/main/java, y el código de pruebas estará en src/test/java

Para este ejercicio necesitamos también la clase Calendario

```
//paquete llamadas

public class Calendario {
  public int getHoraActual() {
        throw new UnsupportedOperationException ("Not yet implemented");
  }
}
```

Para ejecutar los tests de este ejercicio debes usar la ventana Maven de IntelliJ.

➪ ➪ Ejercicio 2: drivers para *validaAsignaturas()*

Para este ejercicio añadiremos **un nuevo módulo *asignaturas***:

Desde ***File→Project Structure→Project Settings→module***, pulsamos sobre "***+→New Module***":

- ***Name***: ***asignaturas***
- ***Location***: *$HOME/ruta_de_tu_directorio_de_trabajo/P05-Dependencias1/P05-stubs/*
- Seleccionamos el lenguaje ***Java***, la herramienta de construcción ***Maven***, y nos aseguramos de elegir el ***JDK 21***
- ***Parent:*** *<None>*
- Desde ***Advanced Settings***, indicamos las coordenadas de nuestro proyecto: ***GroupId***: *ppss.P05*; ***ArtifactId:*** ***asignaturas***

La unidad a probar en este ejercicio es el método ***ppss.MatriculaAlumno.validaAsignaturas()***, que recibe el dni de un alumno y una lista de códigos de asignatura de las que se quiere matricular, y devuelve un justificante de la matricula, que incluye el dni, la lista de asignaturas de las que se puede matricular y la lista de asignaturas de las que no se puede matricular (en cuyo caso se incluirá para cada asignatura el mensaje "no existe" o "asignatura ya cursada", en el caso de que se trate de un código de asignatura incorrecto, o ya haya sido cursada, respectivamente.

La comprobación de si un alumno se puede matricular en una asignatura se lleva a cabo en el método Operacion.compruebaMatricula() que devuelve las excepciones AsignaturaIncorrectaException o AsignaturaCursadaException (ambas de tipo *checked*), cuando no se puede matricular de una determinada asignatura. Siempre se comprueban todas las asignaturas de la lista, independientemente de que algunas no sean válidas

Disponemos de la siguiente implementación del método MatriculaAlumno.validaAsignaturas()

```java
public class MatriculaAlumno {
   protected Operacion getOperacion() {
      return new Operacion();
   }

   public JustificanteMatricula validaAsignaturas(String dni, String[] asignaturas) {
     JustificanteMatricula justificante = new JustificanteMatricula();
     ArrayList<String> validas = new ArrayList<>();
     ArrayList<String> listaErrores = new ArrayList<>();

     Operacion op = getOperacion();
     for (String asignatura: asignaturas) {
        try {
           op.compruebaMatricula(dni, asignatura);
           validas.add(asignatura);
        } catch (AsignaturaIncorrectaException ex) {
            listaErrores.add("Asignatura " + asignatura + " no existe");
        } catch (AsignaturaCursadaException ex) {
            listaErrores.add("Asignatura " + asignatura + " ya cursada");
        }
     }

     justificante.setDni(dni);
     justificante.setAsignaturas(validas);
     justificante.setErrores(listaErrores);

     return justificante;
    }
}
```

```java
public class JustificanteMatricula {
  private String dni;
  private ArrayList<String> asignaturas;
  private ArrayList<String> errores;
  //getters y setters
}
```

Debes implementar el código necesario en src/main/java para poder compilar todo el código en producción. Tienes que tener en cuenta que las excepciones son de tipo *checked* (heredan de *Exception*).

Queremos automatizar las pruebas unitarias sobre ***validaAsignaturas()*** usando verificación basada en el estado, a partir de los siguientes casos de prueba.

IMPORTANTE: Si necesistas refactorizar, debes tener en cuenta que no puedes añadir ninguna clase adicional en producción ni ningún atributo.

	dni	asignaturas	JustificanteMatricula (dni, asignaturas, errores)
C1	"00000000T"	{"MD", "ZZ", "FBD","P1"}	("00000000T", {"MD", "FBD"}, {"Asignatura ZZ no existe", "Asignatura P1 ya cursada"})
C2	"00000000T"	{"PPSS", "ADA", "P3"}	("00000000T", {"PPSS", "ADA", "P3"}, {})

Suponemos que el alumno con dni "0000000T" ya ha cursado las asignaturas "P1", "FC" y "FFI" ,y que los únicos códigos de asignaturas que no existen son "YYY" y "ZZ".

Para ejecutar los tests de este ejercicio debes usar la ventana Maven de IntelliJ.

➪ ➪ Ejercicio 3: drivers para *calculaPrecio()*

Para este ejercicio añadiremos **un nuevo módulo *alquiler***:

Desde ***File→Project Structure→Project Settings→module***, pulsamos sobre "***+→New Module***":

- ***Name***: ***alquiler***
- ***Location***: *$HOME/ruta_de_tu_directorio_de_trabajo/P05-Dependencias1/P05-stubs/*
- Seleccionamos el lenguaje ***Java***, la herramienta de construcción ***Maven***, y nos aseguramos de elegir el ***JDK 21***
- ***Parent:*** *<None>*
- Desde ***Advanced Settings***, indicamos las coordenadas de nuestro proyecto: ***GroupId***: *ppss.P05*; ***ArtifactId: alquiler***.

La unidad a probar en este ejercicio es el método ***calculaPrecio()***, el cual calcula el precio de alquiler de un determinado tipo de coche durante un determinado número de días, a partir de una fecha que se pasa también como parámetro. El precio para cada día depende de si es festivo o no, en cuyo caso se suma o se resta un 25% sobre un precio base obtenido mediante una consulta a un servicio externo. La comprobación de si es festivo o no puede lanzar una excepción, en cuyo caso, ese día no se contabilizará en el precio. Si no se ha producido ningún error en las comprobaciones, se devolverá un ticket con el precio total, en caso contrario se lanzará una excepción con un mensaje indicando los días en los que se han producido errores

Proporcionamos el siguiente código del método ***ppss.Alquilacoches.calculaPrecio()***..

```
public class AlquilaCoches {
  protected Calendario calendario = new Calendario();

  public Ticket calculaPrecio(TipoCoche tipo, LocalDate inicio, int ndias)
                                                        throws MensajeException {
   Ticket ticket = new Ticket();
   float precioDia,precioTotal =0.0f;
   float porcentaje = 0.25f;

   String observaciones = "";
   IService servicio = new Servicio();
   precioDia = servicio.consultaPrecio(tipo);
   for (int i=0; i<ndias;i++) {
      LocalDate otroDia = inicio.plusDays((long)i);
      try {
         if (calendario.es_festivo(otroDia)) {
           precioTotal += (1+ porcentaje)*precioDia;
         } else {
           precioTotal += (1- porcentaje)*precioDia;
         }
      } catch (CalendarioException ex) {
          observaciones += "Error en dia: "+otroDia+"; ";
      }
   }

   if (observaciones.length()>0) {
      throw new MensajeException(observaciones);
   }

   ticket.setPrecio_final(precioTotal);
   return ticket;
  }
}
```

```
public class Ticket {
  private float precio_final;
  //getters y setters
}
```

Debes tener en cuenta que el tipo ***LocalDate*** representa una fecha y pertenece a la librería estándar de Java (*java.time.LocalDate*). La sentencia `inicio.plusDays(i)` devuelve la fecha resultante de añadir "i" dias a la fecha "inicio".

Podemos obtener una representación de tipo String a partir de un LocalDate, de la siguiente forma:

```
String fechaS = fecha.toString(); //el valor de fechaS es : "aaaa-mm-dd"
```

Podemos crear un objeto de tipo LocalDate a partir de tres enteros (año, mes y día):

```
LocalDate fecha = LocalDate.of(2022, 12, 2);
```

Las clases ***Calendario*** y ***Servicio*** están siendo implementadas por otros miembros del equipo.

Tendréis que crear las clases *Calendario*, *Servicio*, así como la interfaz *IService* y las excepciones *CalendarioException* y *MensajeException* (todas de tipo *checked*). Son clases que se usarán en producción (por lo tanto deben estar en src/main/java), pero que no es necesario implementar. Si no las definimos, lógicamente el código no compilará.

La implementación de los métodos de *Calendario* y *Servicio* debe ser:

```
throw new UnsupportedOperationException ("Not yet implemented");
```

TipoCoche es un tipo enumerado (fichero TipoCoche.java):

```
public enum TipoCoche {TURISMO,DEPORTIVO,CARAVANA}; //ej. de uso: TipoCoche.TURISMO
```

Queremos automatizar las pruebas unitarias sobre ***calculaPrecio()*** usando verificación basada en el estado, a partir de los siguientes casos de prueba:
IMPORTANTE: si necesitas refactorizar no puedes añadir ningún atributo en la clase que contiene nuestra SUT, ni tampoco alterar en modo alguno la forma de invocar a nuestra sut desde otras unidades, así como tampoco puedes añadir ninguna clase adicional en producción.

Asumimos que el precio base para cualquier día es de 10 euros, tanto para caravanas como para turismos.

Id	Datos Entrada				Resultado Esperado
	Tipo	fechaInicio	dias	es_festivo()	Ticket (importe) o MensajeException
C1	TURISMO	2024-05-18	10	false para todos los días	Ticket (75)
C2	CARAVANA	2024-06-19	7	true solo para los días 20 y 24	Ticket (62,5)
C3	TURISMO	2024-04-17	8	false para todos los días, y lanza excepción los días 18, 21, y 22	(“Error en dia: 2024-04-18; Error en dia: 2024-04-21; Error en dia: 2024-04-22; ”)

Nota: el formato de la fecha es "aaaa-mm-dd" (año-mes-dia)

Para ejecutar los tests de este ejercicio debes usar la ventana Maven de IntelliJ.

➪ ➪ Ejercicio 4: drivers para *reserva()*

Para este ejercicio añadiremos **un nuevo módulo *reserva***:

Desde ***File→Project Structure→Project Settings→module***, pulsamos sobre "***+→New Module***":

- ***Name***: "***reserva***"
- ***Location***: *"$HOME/ruta_de_tu_directorio_de_trabajo/P05-Dependencias1/P05-stubs/*"
- Seleccionamos el lenguaje ***Java***, la herramienta de construcción ***Maven***, y nos aseguramos de elegir el ***JDK 21***
- El valor del campo parent asegúrate de que es <None>
- Desmarcamos la casilla *Add sample code*
- Desde ***Advanced Settings***, indicamos las coordenadas de nuestro proyecto: ***GroupId***: "*ppss.P05*"; ***ArtifactId:*** *"**reserva***"

Dado el código de la unidad a probar (método ***realizaReserva()***), se trata de implementar y ejecutar

```
//paquete ppss
public class Reserva {

  public boolean compruebaPermisos(String login, String password, Usuario tipoUsu) {
     throw new UnsupportedOperationException("Not yet implemented");
  }

  public void realizaReserva(String login, String password,
                String socio, String [] isbns) throws ReservaException {

    ArrayList<String> errores = new ArrayList<>();
    if(!compruebaPermisos(login, password, Usuario.BIBLIOTECARIO)) {
        errores.add("ERROR de permisos");
    } else {
        IOperacionBO io = new Operacion();
      try {
         for(String isbn: isbns) {
            try {
              io.operacionReserva(socio, isbn);
            } catch (IsbnInvalidoException iie) {
              errores.add("ISBN invalido" + ":" +  isbn);
            }
         }
      } catch (SocioInvalidoException sie) {
            errores.add("SOCIO invalido");
      } catch (JDBCException je) {
            errores.add("CONEXION invalida");
      }
    }
    if (errores.size() > 0) {
       String mensajeError = "";
       for(String error: errores) {
         mensajeError += error + "; ";
       }
       throw new ReservaException(mensajeError);
    }
  }
}
```

```
//paquete ppss
public enum Usuario {
  BIBLIOTECARIO, ALUMNO, PROFESOR
}
```

los drivers (usando verificación basada en el estado) automatizando así las pruebas unitarias de la tabla de casos de prueba proporcionada.

Las excepciones debes implementarlas en el paquete "**ppss.excepciones**" (por ejemplo):

```
public class JDBCException extends Exception { }
```

```
public class ReservaException extends Exception {
  public ReservaException(String message) { super(message);}}
```

Definición de la interfaz (paquete: **ppss**):

```
public interface IOperacionBO {
  public void operacionReserva(String socio, String isbn)
              throws IsbnInvalidoException, JDBCException, SocioInvalidoException;}
```

La implementación del método: Operacion.operacionReserva() debe ser:

```
throw new UnsupportedOperationException ("Not yet implemented");
```

La tabla de casos de prueba es la siguiente:

	login	password	ident. socio	isbns	{reserva()}	Resultado esperado
C1	"xxxx"	"xxxx"	"Luis"	{"11111"}	--	ReservaException1
C2	"ppss"	"ppss"	"Luis"	{"11111", "22222"}	{NoExcep, NoExcep.}	No se lanza excep.
C3	"ppss"	"ppss"	"Luis"	{"11111","33333", "44444",}	{NoExcep,IsbnEx, IsbnEx}	ReservaException2
C4	"ppss"	"ppss"	"Pepe"	["11111"}	{SocioEx}	ReservaException3
C5	"ppss"	"ppss"	"Luis"	{"11111", "22222"}	{NoExcep,JDBCEx}	ReservaException4

Suponemos que el login/password del bibliotecario es "ppss"/"ppss"; que "Luis" es un socio y "Pepe" no lo es; y que los isbn registrados en la base de datos son "11111", "22222".

-- significa que no se invoca al método reserva()
NoExcep. ⇢ El método reserva no lanza ninguna excepción
IsbnEx ⇢ Excepción IsbnInvalidoException
SocioEx ⇢ Excepción SocioInvalidoException
JDBCEx ⇢ Excepción JDBCException
ReservaException1: Excepción de tipo ReservaException con el mensaje: "ERROR de permisos; "
ReservaException2: Excepción de tipo ReservaException con el mensaje: "ISBN invalido:33333; ISBN invalido:44444;"
ReservaException3: Excepción de tipo ReservaException con el mensaje: "SOCIO invalido; "
ReservaException4: Excepción de tipo ReservaException con el mensaje: "CONEXION invalida;"

Para este ejercicio, si tienes que refactorizar, usa una clase factoría.

Para ejecutar los tests de este ejercicio debes usar la ventana Maven de IntelliJ.

➪ ➪ ANEXO 1: Observaciones sobre el desarrollo de los ejercicios de P04

◉ PRIMERO tienes que determinar las entradas y salidas de la unidad a probar. Cada entrada tiene que estar perfectamente identificada.

Por ejemplo:
Entrada 1 (**C**): (tipo_coche) parámetro de entrada de tipo enumerado.

◉ SEGUNDO tienes que decidir si vas a agrupar alguna entrada y cuáles vas a agrupar.

Por ejemplo:
Entradas1-2-3 : (**F**) Agrupamos Fecha de inicio (inicio)+Fecha de fin (fin)+dia semana (d)

◉ TERCERO: tienes que identificar claramente las condiciones que determinan cada partición válida e inválida sobre las entradas y salidas.

Por ejemplo:
Entrada2 (**F**): (fecha_inicio): fecha de inicio
Clases válidas : **F1**: fecha_inicio > fecha_actual
Clases NO válidas : **NF1**: fecha_inicio <= fecha_actual

IMPORTANTE: Igual que hay que dejar claras QUÉ ENTRADAS/SALIDAS tenemos, hay que dejar claro cuáles son las particiones VÁLIDAS y las INVÁLIDAS, para CADA ENTRADA/SALIDA

◉ No puede haber particiones inválidas con una condición OR.

Por ejemplo:
Entrada 2 : (**C**) canal de televisión (canal)
Clases NO válidas : **NC1:** canal > 20
NC2: canal < 1

NO se puede poner: NCX:(canal > 20) ∨ (canal < 1)

◉ A partir de las particiones, tiene que ser posible rellenar la tabla sin tener que volver a leer la especificación, por lo que es FUNDAMENTAL que dejes claras las condiciones que deben cumplir los valores de cada partición, tanto de entrada como de salida.

Por ejemplo:
Salida (**S**): lista de eventos o excepción de tipo ParseException
Clases válidas :
S1: Lista con eventos de todo el día (duración = -1) comprendidos entre la fechas de entrada de inicio de curso y de fin, todas las semanas, el día de la semana que se indique como entrada

◉ Cada entrada/salida es una variable con su dominio de valores posibles. Y cada partición es un subconjunto del dominio de valores de esa variable obtenido al aplicar las condiciones sobre ella. Las particiones deben ser disjuntas, y cada valor posible debe pertenecer a alguna partición. La unión de todas las particiones debe ser igual a todos los valores del dominio de la variable.

◉ Las heurísticas 4 y 6 también se aplican a las salidas, de forma que deben explicitarse los tipos de comportamientos diferentes.

Por ejemplo:
Salida (**S**): importe o excepción de tipo BOException
Clases válidas :
S1: importe = 100 euros * ndias (siendo ndias el valor correspondiente de entrada)
S2: importe = 50 euros * ndias
Cada partición representa un comportamiento diferente (dependiendo de los valores de entrada, hay dos tipos de tarifa que se calcularán de forma diferente en el código)

◉ CUARTO: a partir de las particiones se generan las combinaciones aplicando el algoritmo visto en clase, y se proporcionan valores CONCRETOS en la tabla de casos de prueba.

Tienes que indicar todas las asunciones que hagas sobre las entradas, y/o sobre cualquier dato de la tabla.

➪ ➪ ANEXO 2: Tablas de casos de prueba que deberías haber obtenido en P03

SUT: **importe_alquiler_coche**

Asumimos que la fecha actual es: 23 - marzo - 2024

entradas				salida
tipo_coche	**fecha_inicio**	**disponible**	**ndias**	**importe o excepción ReservaException**
turismo	25-03-2024	true	1	100
deportivo	30-03-2024	true	10	50*10 = 500
turismo	23-01-2024	true	1	"Fecha no correcta"
null	26-03-2024	true	1	???
turismo	27-04-2024	true	-8	???
deportivo	16-05-2024	true	35	"Reserva no posible"
deportivo	14-05-2024	false	18	"Reserva no posible"

Los casos de prueba 3, 6 y 7 generan como salida una excepción de tipo **ReservaException** con el mensaje indicado en las filas correspondientes

SUT: **generaEventos**

Asumimos que "ppss" es el nombre de la única asignatura válida

entradas					salida
nombre	**fecha_Inic**	**fecha_fin**	**hora inicio**	**dia**	**Lista de eventos o ParseException**
"ppss"	19-02-2025	12-03-2025	null	4	{("ppss",20-02-2025, null, -1) ("ppss",27-02/2025, null, -1) ("ppss",6-03-2025, null,-1)}
"ppss"	19-02-2025	12-03-2025	"10:00"	5	{("ppss",21-02-2025,"10:00",120) ("ppss",28-00-2025."10:00",120) ("ppss",7-03-2025,"10:00",120)}
"abcd"	19-02-2025	12-03-2025	"10:00"	5	??
"ppss"	19-02-2025	30-01-2020	"10:00"	1	{ }
"ppss"	19-02-2025	20-02-2025	null	5	{ }
"ppss"	19-02-2025	27-04-2025	"10:00"	-6	ParseException
"ppss"	19-02-2025	14-03-2025	"10:00"	35	ParseException
"ppss"	19-02-2025	14-03-2025	"ab"	4	ParseException
"ppss"	19-02-2025	14-03-2025	"34:00"	4	ParseException

Nota: los valores dd-mm-aaaa representan el día-mes-año del objeto de tipo LocalDate
Nota: Cada evento de la lista de salida es una tupla con los valores (asignatura, fecha_inicio, hora_inicio, duración)

SUT: ***agregarGrabacion***

Suponemos que:
el valor de inicio y fin siempre está comprendido entre 1..1444
la lista de grabaciones de entrada nunca puede tener el valor null

Datos Entrada				Resultado Esperado
lista grabaciones	Grabación (canal, inicio, fin)			lista grabaciones / excepción
	canal	inicio	fin	
[ ]	2	30	60	[(2,30,60)]
[(5, 10,20), (6,40,60)]	1	20	40	[(5, 10,20), (1.20,40),(6.40,60)]
[(5, 10,20), (6,40,60)]	25	25	35	"No existe el canal"
[(5, 10,20), (6,40,60)]	-3	70	100	"No existe el canal"
[(5, 10,20), (6,40,60)]	4	25	50	"Solape de grabaciones"
[(5, 10,20), (6,40,60)]	4	50	50	"Error en intervalo de grabacion"
[(5, 10,20), (6,40,60)]	4	55	50	???
[ ]	4	30	30	"Error en intervalo de grabacion"
[ ]	4	30	10	???

Nota1: Cada grabación de las listas de entrada y salida es una tupla de con valores (canal, inicio, fin), en ese orden

Nota2: Los casos de prueba 3..6 y el caso de prueba 8 generan como salida una excepción de tipo **GrabacionException** con el mensaje indicado en las filas correspondientes

Resumen

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

DEPENDENCIAS EXTERNAS

- Cuando realizamos pruebas unitarias la cuestión fundamental es AISLAR la unidad a probar para garantizar que si encontramos evidencias de DEFECTOS, éstos se van a encontrar "dentro" de dicha unidad..
- Para aislar la unidad a probar, tenemos que controlar sus entradas indirectas, proporcionadas por sus colaboradores o dependencias externas (DOCs). Dicho control se realiza a través de los dobles de dichos colaboradores
- Durante la pruebas realizaremos un reemplazo controlado de las dependencias externas por sus dobles de forma que el código a probar (SUT) será IDÉNTICO al código de producción.
- Nuestro DOBLE siempre pertenecerá a una clase heredada de la clase que contiene nuestro DOC, o implementará su misma interfaz. y será inyectado en la unidad a probar durante las pruebas. Hay varios tipos de dobles, concretamente usaremos STUBS
- La implementación de un STUB tiene como objetivo controlar las entradas indirectas de nuestra SUT.
- El código del doble debe ser lo más simple posible, y también lo más genérico posible. No usaremos dobles para getters/setters ni para dependencias externas que sean de la librería estándar de java (a menos que queramos controlar el resultado que nos proporciona dicha dependencia)
- Para poder inyectar el doble durante las pruebas, es posible que tengamos que REFACTORIZAR nuestra SUT, para proporcionar un "seam enabling point"

IMPLEMENTACIÓN DE LOS TESTS

- El driver debe, durante la preparación de los datos (Arrange), crear los dobles (uno por cada dependencia externa), y debe inyectar dichos dobles en la unidad a probar a través de uno de los "enabling seam points" de nuestro SUT.
- A continuación el driver ejecutará nuestra SUT. (Act) Las entradas directas de la SUT las proporciona el driver, mientras que las entradas indirectas llegan a nuestra unidad a través de los dobles (STUBS).
- Después de ejecutar la SUT, el driver comparará el resultado real obtenido con el esperado y finalmente generará un informe (Assert).
- Dado que el informe de pruebas dependerá exclusivamente del ESTADO resultante de la ejecución de nuestra SUT, nuestro driver estará realizando una VERIFICACIÓN BASADA EN EL ESTADO..