P04- Dependencias externas 1: stubs

Os recordamos que el plazo para entregar los ejercicios de esta práctica (P04) termina justo ANTES de comenzar la siguiente sesión de prácticas (P05) 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 P04 (de forma individual), con independencia de que se indiquen en clase las soluciones que os permitirán saber si las vuestras son correctas.

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 de nuestra unidad. De esta forma nos aseguramos de que cualquier defecto que detectemos estará exclusivamente en el código de nuestro 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, y que reemplazará al colaborador correspondiente. Es importante que tengas claro que hay diferentes refactorizaciones posibles y que cada una de ellas tiene diferentes "repercusiones" en el código en producción.

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 nuestro 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, programar su resultado e inyectarlo en nuestra SUT, antes de ejecutarla. Para implementar los drivers usaremos JUnit5.. Para ejecutarlos usaremos Maven y Junit5.

Bitbucket

El trabajo de esta sesión también debes subirlo a *Bitbucket*. Todo el trabajo de esta práctica deberá estar en el directorio **P04-Dependencias1**, dentro de tu espacio de trabajo, es decir, dentro de tu carpeta: ppss-2022-Gx-apellido1-apellido2..

Ejercicios

En las sesiones anteriores, hemos trabajado con un proyecto IntelliJ que solamente contenía un proyecto Maven. En esta sesión vamos a crear también un proyecto IntelliJ, pero inicialmente estará formado por un módulo **vacío**, e iremos añadiendo más módulos (proyectos Maven), en cada uno de los ejercicios.

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

- ***File→New Project***. Seleccionamos “Empty Project”
- ***Project name*** : "P04-stubs". ***Project Location:*** "$HOME/ppss-2022-Gx-.../P04-Dependencias1/P04-stubs". Es decir, que tenemos que crear el proyecto dentro de nuestro directorio de trabajo, y del directorio P04-Dependencias1.

Nuestro proyecto está formado por el módulo P04-stubs, puedes verlo dese ***File→Project Structure →Project Settings →Modules***

Realmente no vamos a necesitar este módulo, así que podemos eliminarlo, y así tendremos un proyecto totalmente vacío en el que añadiremos nuestros proyectos Maven (cada proyecto Maven será un módulo de nuestro proyecto IntelliJ).

Desde la ventana ***Project Structure***, seleccionamos el único módulo de nuestro proyecto y lo eliminamos pulsando sobre el icono con un signo -"-".

IntelliJ nos advierte de que el módulo se elimina del proyecto pero sus ficheros asociados permanecen en el disco duro. Realmente, el único fichero de este módulo en el disco duro es el fichero P04-stubs.iml (lo verás en la vista *Project*). Puedes seleccionarlo y borrarlo.

Ahora ya tenemos un proyecto IntelliJ totalmente vacío.

OBSERVACIONES A TENER EN CUENTA PARA REALIZAR LOS EJERCICIOS.

Recuerda que debes **modificar el pom** convenientemente 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

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 nombrar igual que la clase de la dependencia externa añadiéndole el sufijo "Stub".
- Si necesitas crear una **clase adicional para poder inyectar** el doble en la sut, dicha clase tendrá el mismo nombre que la clase de la sut, con el sufijo "Testable".

IMPORTANTE: Para implementar el driver tenemos que: detectar (PRIMERO) las dependencias externas, (SEGUNDO) comprobar si nuestro SUT es *testable*, (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 que usaremos en el Ejercicio1. En la ventana que nos muestra IntelliJ, desde ***File→Project Structure→Project Settings→module***, pulsamos sobre "***+→New Module***":

- Seleccionamos ***Maven***, y nos aseguramos de elegir el ***JDK 11,*** y pulsamos ***Next***
- El campo ***Parent***, asegúrate de que tiene el valor ***<none>***
- los campos ***Name*** y ***Location*** deben tener los valores: ***Name***: "***gestorLlamadas***". ***Location***: *"$HOME/ppss-2022-Gx-.../P04-Dependencias1/P04-stubs/**gestorLlamadas***". **No** edites los campos ***Name*** y ***Location***
- Las coordenadas de nuestro proyecto serán ***GroupId***: "*ppss.P04*"; ***ArtifactId:*** *"**gestorLlamadas***".

Finalmente pulsamos sobre *Finish* (automáticamente IntelliJ marcará los directorios de nuestro proyecto como directorios estándar de Maven, de forma que "sabrá" cuáles son los directorios de fuentes, de recursos, de pruebas,...).

Una vez que hemos creado el **módulo gestorLlamadas** en nuestro proyecto IntelliJ, lo usaremos para automatizar las pruebas unitarias dinámicas sobre el método **GestorLlamadas.calculaConsumo()** (que pertenece al paquete **ppss.ejercicio1**) utilizando verificación basada en el estado.

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

```java
//paquete ppss.ejercicio1

public class GestorLlamadas {
  static double TARIFA_NOCTURNA=10.5;
  static double TARIFA_DIURNA=20.8;
  public int getHoraActual() {
    Calendar c = Calendar.getInstance();
    int hora = c.get(Calendar.HOUR);
    return hora;
    }

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

	minutos	hora	Resultado esperado
C1	10	15	208
C2	10	22	105

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

➪ ➪ Ejercicio 2: drivers para *calculaConsumo()* Versión 2

Seguiremos trabajando en el módulo **gestorLlamadas** del ejercicio anterior. A partir de la tabla de casos de prueba del ejercicio 1, automatiza las pruebas unitarias sobre la siguiente implementación alternativa de ***GestorLlamadas.calculaConsumo()*** utilizando verificación basada en el estado**.** En este caso, la unidad a probar pertenece al paquete **ppss.ejercicio2** (que deberás crear), del **módulo gestorLlamadas.**

Para este ejercicio necesitamos también la clase Calendario

```java
//paquete ppss.ejercicio2

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

```java
//paquete ppss.ejercicio2
public class GestorLlamadas {
  static double TARIFA_NOCTURNA=10.5;
  static double TARIFA_DIURNA=20.8;

  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;
    }
  }
}
```

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

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

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

- Seleccionamos ***Maven***, y nos aseguramos de elegir el ***JDK 11,*** y pulsamos ***Next***
- El campo ***Parent***, asegúrate de que tiene el valor ***<none>***
- los campos ***Name*** y ***Location*** deben tener los valores: ***Name***: "***alquiler***". ***Location***: *"$HOME/ppss-2022-Gx-.../P04-Dependencias1/P04-stubs/**alquiler***". **No** edites los campos ***Name*** y ***Location***
- Las coordenadas de nuestro proyecto serán ***GroupId***: "*ppss.P04*"; ***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 ***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. 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 un String mediante:

```
LocalDate fecha = LocalDate.of(2022, Month.MARCH, 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*. 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.
Tipo coche 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 nuestro SUT.

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	2022-05-18	1o	false para todos los días	Ticket (75)
C2	CARAVANA	2022-06-19	7	true solo para los días 20 y 24	Ticket (62,5)
C3	TURISMO	2022-04-17	8	false para todos los días, y lanza excepción en 18, 21, y 22	("Error en dia: 2022-04-18; Error en dia: 2022-04-21; Error en dia: 2022-04-22; ")

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

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

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

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

- Seleccionamos ***Maven***, y nos aseguramos de elegir el ***JDK 11,*** y pulsamos ***Next***
- El campo ***Parent***, asegúrate de que tiene el valor ***<none>***
- los campos ***Name*** y ***Location*** deben tener los valores: ***Name***: "***reserva***". ***Location***: *"$HOME/ppss-2022-Gx-.../P04-Dependencias1/P04-stubs/**reserva***". **No** edites los campos ***Name*** y ***Location***
- Las coordenadas de nuestro proyecto serán ***GroupId***: "*ppss.P04*"; ***ArtifactId:*** *"reserva*".

Dado el código de la unidad a probar, que proporcionamos más adelante, se trata de implementar y ejecutar los drivers (usando verificación basada en el estado) automatizando así las pruebas unitarias de la siguiente tabla de casos de prueba.

	login	password	ident. socio	Acceso BD	isbns	Resultado esperado
C1	"xxxx"	"xxxx"	"Luis"	(1)	{"11111"}	ReservaException1
C2	"ppss"	"ppss"	"Luis"	(2)	{"11111", "22222"}	No se lanza excep.
C3	"ppss"	"ppss"	"Luis"	(3)	{"33333"}	ReservaException2
C4	"ppss"	"ppss"	"Pepe"	(4)	["11111"}	ReservaException3
C5	"ppss"	"ppss"	"Luis"	(5)	{"11111"}	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”.

ReservaException1: Excepción de tipo ReservaException con el mensaje: "ERROR de permisos; "

ReservaException2: Excepción de tipo ReservaException con el mensaje: "ISBN invalido:33333; "

ReservaException3: Excepción de tipo ReservaException con el mensaje: "SOCIO invalido; "

ReservaException4: Excepción de tipo ReservaException con el mensaje: “CONEXION invalida;"

(1): No se invoca al método reserva()

(2): El método reserva() NO lanza ninguna excepción

(3): El método reserva() lanza la excepción IsbnInvalidoException

(4): El método reserva() lanza la excepción SocioInvalidoException

(5): El método reserva() lanza la excepción ConexiónInvalidaException

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

El código de la unidad a probar es el siguiente:

```
//paquete ppss.P04
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 Exception {

    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.P04
public enum Usuario {
  BIBLIOTECARIO, ALUMNO, PROFESOR
}
```

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

```
//paquete ppss.P04.excepciones
public class JDBCException extends Exception { }
```

```
//paquete ppss.P04.excepciones
public class ReservaException extends Exception {
  public ReservaException(String message) { super(message);}
}
```

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

```
//paquete ppss.P04
public interface IOperacionBO {
  public void operacionReserva(String socio, String isbn)
              throws IsbnInvalidoException, JDBCException, SocioInvalidoException;
}
```

➪ ➪ ANEXO 1: Observaciones sobre el desarrollo de los ejercicios de P03

◉ 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:
Entradas2 : (**F**) Agrupamos Fecha 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

◉ 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

◉ 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 - 2022

entradas				salida
tipo_coche	**fecha_inicio**	**disponible**	**ndias**	**importe ó excepción ReservaException**
turismo	25-03-2022	true	1	100
deportivo	30-03-2022	true	10	50*10 = 500
turismo	23-01-2022	true	1	"Fecha no correcta"
null	26-03-2022	true	1	???
turismo	27-04-2022	true	-8	???
deportivo	16-05-2022	true	35	"Reserva no posible"
deportivo	14-05-2022	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 el nombre de la asignatura es "ppss" en todos los casos

entradas				salida
fecha_Inicio	**fecha_fin**	**hora inicio**	**dia**	**Lista de eventos o ParseException**
21-02-2022	14-03-2022	null	2	{("ppss",22-02-2022, null, -1) ("ppss",01-03/2022, null, -1) ("ppss",08-03-2022, null,-1)}
21-02-2022	14-03-2022	"10:00"	3	{("ppss", 23-02-2022, "10:00", 120) ("ppss", 02-03-2022, "10:00", 120) ("ppss", 09-03-2022, "10:00", 120)}
21-02-2023	30-03-2020	"10:00"	1	{ }
21-02-2022	23-02-2022	null	5	{ }
21-02-2022	27-04-2022	"10:00"	-6	ParseException
21-02-2022	14-03-2022	"10:00"	35	ParseException
21-02-2022	14-03-2022	"ab"	4	ParseException
21-02-2022	14-03-2022	"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: **matriculaAlumno**

Suponemos que el estado inicial de la base de datos es el siguiente:

Alumnos	Asignaturas	Matricula	
Nif (*)	código (**)	Nif	código
00000000A	Del 1 al 10	00000000A	1

Suponemos que los nif 0000000A y 11111111B son válidos

(*) El resto de datos (nombre, direccion, email, telefonos, fechaNacimiento) de todos los alumnos serán: ("nombre", "direccion", "nombre@email.com", ["123456789"], "01-01-2000")
(**) Los nombres de las asignaturas son "A1", "A2", ... "A10", todas son de 6 créditos.
(***) "msg*X*" denota el mensaje "Error al matricular la asignatura *x*" de la excepción BOException

Asumimos que el nombre de la asignatura es "ppss" en todos los casos

Datos Entrada			Resultado Esperado
Alumno(*) nif	Asignaturas códigos	acceso BD (para cada asignatura)	MatriculaTO (alumno, asignaturas,errores)/ BOException
11111111B	[1, 2, 3]	acceso sin fallos	MatriculaTO(11111111B, [1, 2, 3],[ ])
00000000A	[2]	acceso sin fallos	MatriculaTO(00000000A ,[2] ,[ ])
Null	[1]	acceso sin fallos	BOException("El nif no puede ser nulo")
999999999	[1]	acceso sin fallos	BOException("Nif no válido")
00000000A	[2, 3,4]	Fallo en asig. 3	MatriculaTO(00000000A, [2, 4] ,[msg3]) (***)
00000000A	[2]	Fallo al obtener datos alumno	BOException("Error al obtener los datos del alumno")
11111111B	[1]	Fallo al dar de alta alumno	BOException("Error en el alta del alumno")
00000000A	NULL	acceso sin fallos	BOException("Faltan las asignaturas de matriculación")
00000000A	[ ]	acceso sin fallos	BOException("Faltan las asignaturas de matriculación")
00000000A	[1]	acceso sin fallos	BOException("El alumno con nif 00000000A ya está matriculado en la asignatura con código 1")
11111111B	[1,2,3,4,5,6]	acceso sin fallos	BOException("El número máximo de asignaturas es 5")

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.
- Un DOBLE siempre heredará 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
- Para poder inyectar el doble durante las pruebas, es posible que tengamos que REFACTORIZAR nuestro SUT, para proporcionar un "seam enabling point"

IMPLEMENTACIÓN DE LOS TESTS

- El driver debe, durante la preparación de los datos, 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 nuestro SUT (pasándole las entradas DIRECTAS del SUT, mientras que las entradas INDIRECTAS las obtendrá de los STUBS). El driver comparará el resultado real obtenido y finalmente generará un informe.
- Dado que el informe de pruebas dependerá exclusivamente del ESTADO resultante de la ejecución de nuestro SUT, nuestro driver estará realizando una VERIFICACIÓN BASADA EN EL ESTADO..