

Pasos a seguir para automatizar las pruebas:

1. Identificar las dependencias externas de nuestro SUT
2. Asegurarnos de que nuestro código (SUT) es testeable, para ello tendré que poderse realizar un reemplazo controlado de cada dependencia externa por su doble sin modificar su código.
- 2.5 Realizar una posible refactorización de nuestro SUT para poder realizar dichos reemplazos controlados.
3. Proporcionar una implementación ficticia (DOBLE) que reemplazará al código real de cada dependencia externa durante las pruebas.
4. Implementación de DRIVERS
 - Basada en Estado: solo estamos interesados en comprobar el estado resultante de la invocación de nuestro SUT
 - Basada en comportamiento: nos interesa además verificar que las interacciones entre nuestro SUT y las dependencias externas se realizar correctamente.

Para que sea testeable, debe contener un SEAM

- Necesitaremos poder cambiar la dependencia real por su doble, si no tenemos un seam para cada dependencia externa que nos permita "inyectar" nuestro doble durante las pruebas.
- Como trabajamos en JAVA:
 - Nuestro doble debe implementar la misma interfaz que el colaborador o debe EXTENDER la misma clase que el colaborador.
- Formas de injectar el doble durante las pruebas:
 1. A través de un parámetro de nuestra SUT
 2. A través del constructor
 3. A través de un método Setter
 4. A través de un método de factoría local de la clase que contiene nuestra SUT, o clase factoría
- Si nuestra SUT NO es testeable → Refactorizar:
 1. Si añadimos un parámetro a nuestro SUT, estamos obligando a que cualquier código cliente de nuestra SUT tenga que conocer dicha dependencia antes de invocar a nuestra SUT.
 2. Si añadimos un parámetro al constructor de nuestra SUT (o un método Setter) estamos obligados a declarar la dependencia como un atributo de la clase que contiene nuestra SUT.
 3. No podemos añadir un método setter si el constructor realiza alguna acción significativa sobre nuestra dependencia.
 4. Si usamos un método de factoría local no se ven afectados, ni los clientes de nuestro SUT, ni la estructura de la clase que contiene nuestro SUT, aunque alteramos el comportamiento de la clase que contiene nuestro SUT al añadir un nuevo método. Una clase factoría implica añadir un código que puede ser innecesario en producción.

```
public Factura generarFactura(Cliente cli, Buscador buscar)
    throws FacturaException {
    Factura factura;
    int numElems = buscar.elemPendientes(cli);
    if (numElems > 0) {
        //código para generar la factura
        factura = ...;
    } else {
        throw new FacturaException("No hay ...");
    }
    return factura;
}
```

SUT

```
public class Buscador {    en src/main/java
    //código REAL de nuestro DO
    //en src/main/java
    public int elemPendientes(Cliente cli) {
        ...
    }
}
```

IMPLEMENTACIÓN REAL

```
public class BuscadorStub extends Buscador {
    int result;

    public void setResult(int salida) {
        this.result = salida;
    }
    //código de nuestro doble
    //en src/test/java
    @Override
    public int elemPendientes(Cliente cli) {
        return result;
    }
}
```

DOBLE (STUB)

Ejemplo

Factoría Local

```
public class GestorPedidos {  
    /src/main/java  
  
    public Factura generarFactura(Cliente cli)  
        throws FacturaException {  
        Factura factura = new Factura();  
        IService buscarDatos = new Buscador();  
        int numElems = buscarDatos.elemPendientes(cli);  
        if (numElems>0) {  
            //código para generar la factura  
            factura = ...;  
        } else {  
            throw new FacturaException("No hay nada  
                pendiente de facturar");  
        }  
        return factura;  
    }  
}
```

SUT NO TESTABLE!!!

```
public class Buscador implements IService {  
    @Override  
    public int elemPendientes(Cliente cli)  
    {...}  
    ...  
    /src/main/java  
  
    public interface IService {  
        /src/main/java  
        public int elemPendientes(Cliente cli);  
    }
```

Como no tenemos ninguna manera de injectar el código testable debemos de refactorizar, usamos el método de **factoría local**

①

Refactorizamos la SUT

```
public class GestorPedidos {  
    public(IService getBuscador() {  
        IService buscar = new Buscador();  
        return buscar;  
    }  
  
    public Factura generarFactura(Cliente cli)  
        throws FacturaException {  
        Factura factura = new Factura();  
        IService buscarDatos = getBuscador();  
  
        int numElems = buscarDatos.elemPendientes(cli);  
        if (numElems>0) { //código para generar la factura  
            factura = ...;  
        } else {  
            throw new FacturaException("No hay ...");  
        }  
        return factura;  
    }  
}
```

/src/main/java

SUT REFACTORIZADA!!!

En el getter generamos el objeto.

→ Cambiamos el new por el getter

Es importante de disponer de un objeto que editaremos con el setter para modificar el comportamiento

② Implementamos el Stub con el override de la función

```
public class BuscadorSTUB implements IService {  
    int resultado;  
  
    public BuscadorSTUB(int salida) {  
        this.resultado = salida;  
    }  
  
    @Override  
    public int elemPendientes(Cliente cli) {  
        return resultado;  
    }  
}
```

/src/test/java DOBLE (STUB)

③ Generamos el SUT testable con el override de la factoría.

```
public class GestorPedidosTestable extends  
    GestorPedidos {  
    IServiceProvider busca;  
  
    @Override  
    public(IService getBuscador() {  
        return busca;  
    }  
  
    public void setBuscador(IService b) {  
        this.busca = b;  
    }  
}
```

SUT TESTABLE

/src/test/java

④ Implementamos el Driver

```
public class GestorPedidosTest {  
    @Test  
    public void testGenerarFactura() throws ... {  
        Cliente cli = new Cliente(...);  
        BuscadorSTUB stub = new BuscadorSTUB(10);  
        GestorPedidosTestable sut = new  
            GestorPedidosTestable();  
        sut.setBuscador(stub);  
        Factura expectedResult = new Factura(...);  
        Factura realResult = sut.generarFactura(cli);  
        assertEquals(expectedResult, realResult);  
    }  
}
```

/src/test/java

DRIVER

→ Usamos el Stub

→ Usamos nuestro testable

→ Injectamos nuestro testable

→ Hacemos el assert

Ejemplo Clase Factoría

```
public class GestorPedidos {           /src/main/java
    public Factura generarFactura(Cliente cli)      throws FacturaException {
        Factura factura = new Factura();
        Buscador buscarDatos = new Buscador();      DOC
        int numElems = buscarDatos.elemPendientes(cli);
        if (numElems>0) {
            //código para generar la factura
            factura = ...;
        } else {
            throw new FacturaException("No hay nada
                                         pendiente de facturar");
        }
        return factura;
    }
}
```

SUT NO TESTABLE!!!

- ① Detectamos las dependencias externas, sabemos que al hacer el new ahí no podemos inyectar nuestro doble por lo que tenemos que refactorizar.

Refactorización ②

```
public class GestorPedidos {           /src/main/java
    public Factura generarFactura(Cliente cli)      throws FacturaException {
        Factura factura = new Factura();
        Buscador buscarDatos = Factoria.create();      SUT TESTABLE!!!
        int numElems =
            buscarDatos.elemPendientes(cli);
        if (numElems>0) {
            //código para generar la factura
            factura = ...;
        } else {
            throw new FacturaException("No hay ...");
        }
        return factura;
    }
}
```

Ahora nuestra dependencia externa se crea desde una clase factoría y se llama al método create

Clase Factoría Implementada ③

```
public class Factoria {
    private static Buscador servicio= null;
    public static Buscador create() {
        if (servicio != null) {
            return servicio;
        } else {
            return new Buscador();
        }
    }
    static void setServicio (Buscador srv){
        servicio = srv;
    }
}
```

/src/main/java 19

- ④ Implemento el stub de nuestra clase externa con el método set para indicar lo que queremos devolver y el override de la dependencia

```
public class BuscadorStub extends Buscador {
    int result;

    public void setResult(int salida) {
        this.result = salida;
    }
    //código de nuestro doble
    //en src/test/java
    @Override
    public int elemPendientes(Cliente cli) {
        return result;
    }
}
```

/src/test/java STUB

⑤ Implementamos el Driver

```
public class GestorPedidosTest {
    @Test
    public void testGenerarFactura() throws Exception {
        Cliente cli = new Cliente(...);
        GestorPedidos sut = new GestorPedidos();
        BuscadorStub buscadorStub = new BuscadorStub();
        buscadorStub.setResult() = 10;
        Factoria.setServicio(buscadorStub);
        Factura expResult = new Factura(...);
        Factura result = sut.generarFactura(cli);
        assertEquals(expResult, result);
    }
}
```

Generamos el Cli con el que se llamará al Sut y al propio Sut

Usando nuestro Stub creamos el buscador y le seteamos su variable interna. Luego usando la factoría y el set injectamos el set en la factoría, para que luego se inyecte en el Sut.

Creamos el resultado esperado, generamos el resultado y comparamos.

Exercici 1: drivers para calculaConsumo()

Vamos a añadir un primer módulo. En la ventana que nos muestra IntelliJ, desde **File→Project Structure→Project Settings→module**, pulsamos sobre "+→New Module":

- Los campos **Name** y **Location** deben tener los valores: **Name: "gestorLlamadas"**. **Location: "\$HOME/ppss-2023-Gx.../P04-Dependencias1/P04-stubs/"**
- Seleccionamos el lenguaje **Java**, la herramienta de construcción **Maven**, y nos aseguramos de elegir el **JDK 11**
- Desmarcamos la casilla **Add sample code**
- Desde **Advanced Settings**, indicamos las coordenadas de nuestro proyecto: **GroupId: "ppss.P04"; Artifid: "gestorLlamadas".**

Finalmente pulsamos sobre **Create** (automáticamente IntelliJ creará nuestro proyecto maven, y marcará los directorios del 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 nuestra SUT, y los casos de prueba que queremos automatizar:

//paquete ppss.ejercicio1

```
private class GestorLlamadas {
    private static final double TARIFA_NOCTURNA=10.5;
    private static final 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

Nuestro SUT es **calculaConsumo** del que notamos que posee una dependencia externa **getHoraActual** al ser una dependencia de la propia clase del SUT lo que vamos a hacer será un **override** del método en una clase que herede de **GestorLlamadas**

```
public class GestorLlamadasTestable extends GestorLlamadas {
    int Hora;
    public void setHora(int HoraSet) {
        this.Hora = HoraSet;
    }
}
```

@Override

```
public int getHoraActual () {
    return Hora;
}
```

```
}
```

```
public class GestorLlamadasTest {
    GestorLlamadasTestable gestor;
```

Para ejecutar los tests de este ejercicio crea la **Run Configuration** con nombre **gestorLlamadas-1**, en la carpeta **intelliJ-configurations** situada en la raíz de tu proyecto maven (**gestorLlamadas**).

@BeforeEach

```
void setup () {
    gestor = new GestorLlamadasTestable();
```

```
}
```

Podríamos haber hecho solo un test parametrizado con **@ParameterizedTest** y **@MethodSource**

@Test

```
void GestorC1() {
    int minutos = 10;
    int hora = 15;
    double esperado = 208;
    gestor.setHora(hora);
    double obtenido = gestor.calculaConsumo(minutos);
    assertEquals(esperado, obtenido);
}
```

```
}
```

@Test

```
void GestorC2() {
    int minutos = 10; int hora = 22; double esperado = 105
    gestor.setHora(hora)
    assertEquals(esperado, gestor.calculaConsumo(minutos));
}
```

```
,
```

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

```
//paquete ppss.ejercicio2
public class Calendario {
    public int getHoraActual() {
        throw new UnsupportedOperationException ("Not yet implemented");
    }
}
```

```
//paquete ppss.ejercicio2
public class GestorLlamadas {
    private static final double TARIFA_NOCTURNA=10.5;
    private static final 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;
        }
    }
}
```

Identificamos las dependencias

- Get Calendario la cual devolverá un objeto Calendario
- getHoraActual un método de la clase Calendario

Para ejecutar los tests de este ejercicio crea la Run Configuration con nombre **gestorLlamadas-2**, en la carpeta **intellij-configurations** situada en la raíz de tu proyecto maven (**gestorLlamadas**).

① Creamos el Stub de la clase Calendario

```
public class CalendarioStub extends Calendario {
    int hora;

    public void setHora (int hora) {
        this.hora = hora;
    }

    @Override
    public int getHoraActual () {
        return hora;
    }
}
```

② Creamos nuestro testeable para GestorLlamadas

```
public class GestorLlamadasTestable extends GestorLlamadas {
    Calendario calendar;
    public void setCalendario (Calendario c) {
        calendar = c;
    }

    @Override
    public Calendario getCalendario () {
        return calendar;
    }
}
```

③ Generamos nuestros Test

```
public class GestorLlamadasTest {
    CalendarioStub c;
    GestorLlamadasTestable gestor;

    @BeforeEach
    void Setup () {
        c = new CalendarioStub ();
        gestor = new GestorLlamadasTestable ();
        c.setHora (15);
        gestor.setCalendario (c);
        assertEquals (208, gestor.CalculaConsumo (10));
    }

    @Test
    void GestorC1 () {
        c.setHora (15);
        gestor.setCalendario (c);
        assertEquals (208, gestor.CalculaConsumo (10));
    }
}
```

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

④ Test

```
void GestorC2 () {
    c.setHora (22);
    gestor.setCalendario (c);
    assertEquals (105, gestor.CalculaConsumo (10));
}
```

Ejercicio 3: drivers para `calculaPrecio()`

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

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

- Los campos **Name** y **Location** deben tener los valores: **Name: "alquiler"**. **Location: "\$HOME/ppss-2023-Gx.../P04-Dependencias1/P04-stubs"**
- Seleccionamos el lenguaje **Java**, la herramienta de construcción **Maven**, y nos aseguramos de elegir el **JDK 11**.
- 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.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 **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  
}
```

- * Al ser de tipo **protected** y pertenecer nuestro test al mismo package podemos injectar el Stub de calendario ahí
- * Esta no es una dependencia externa ya que es lo que devuelve nuestro método
- * Otra dependencia externa que debemos refactorizar con un getter, un setter y un Stub.

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" días 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**. 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.

Esto nos dice que debemos usar una factoría local

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

Id	Datos Entrada				Resultado Esperado
	Tipo	fechalinicio	días	es_festivo()	Ticket (importe) o MensajeException
C1	TURISMO	2023-05-18	10	false para todos los días	Ticket (75)
C2	CARAVANA	2023-06-19	7	true solo para los días 20 y 24	Ticket (62,5)
C3	TURISMO	2023-04-17	8	false para todos los días, y lanza excepción los días 18, 21, y 22	("Error en dia: 2023-04-18; Error en dia: 2023-04-21; Error en dia: 2023-04-22;")

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

① Refactorizar nuestra SUT

```

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

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

        String observaciones = "";
        IService servicio = getService();
        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
}

```

② Debemos añadir en nuestra SUT el método getter para que el comportamiento sea el mismo

```

public IService getService() {
    return New IService();
}

```

③ Creamos el Stub de nuestro Calendario

```

public class CalendarioStub extends Calendario {
    List<integer> diafest;
    List<integer> exception;
    // Constructor parametrizado para injectar días
    // Festivos y de excepción
    Public CalendarioStub (List<integer> f, List<integer> ex) {
        diafest = f;
        exception = ex;
    }

    @Override
    Public Boolean es_festivo (LocalDate dia) {
        if (diafest.contains (dia.getDayOfMonth())) {
            return True;
        } else if (exception.contains (dia.getDayOfMonth())){
            throw new CalendarioException ("Error en dia" + dia);
        } else {
            return false;
        }
    }
}

```

④ Creamos el Stub de Servicio

```

public Class ServicioStub implements IService {
    int Precio;
    public void setPrecio(int Precio) {
        Precio = Precio;
    }

    @Override
    public float consultaPrecio (TipoCochе tipo) {
        return precio;
    }
}

```

⑤ Creamos el testable de nuestra clase

```

public class AlquilaCochesTestable extends AlquilaCoches {
    IService serv;

    @Override
    public IService getService() {
        return serv;
    }

    public void setServicio(IService serv) {
        serv = serv;
    }

    public setCalendario (Calendario c) {
        Calendario = c;
    }
}

```

6 Comenzamos a implementar nuestros Test

```
public class AlquilaCochesTest {
```

```
    CalendarioStub cal;
```

```
    ServicioStub seru;
```

```
    AlquilaCochesTestable alquila;
```

```
@BeforeEach
```

```
void setup() {
```

```
    seru = new ServicioStub();
```

```
    alquila = new AlquilaCochesTestable();
```

```
}
```

```
@Test
```

```
void alquilaC1() {
```

```
    cal = new CalendarioStub(new ArrayList<>(), new ArrayList<>());
```

```
    alquila.calendario = cal;
```

```
    LocalDate fechaInicio = new LocalDate(2023, 05, 18);
```

```
    Ticket ticketObtenido = assertDoesNotThrow(() -> alquila.calculaPrecio(TipoCochе.Turismo, fecha, 10));
```

```
    assertEquals(75, ticketObtenido.getPrecioFinal());
```

```
}
```

```
@Test
```

```
void alquilaC2() {
```

```
    cal = new CalendarioStub(Arrays.asList(20, 24), new ArrayList<>());
```

```
    alquila.calendario = cal;
```

```
    LocalDate fechaInicio = new LocalDate(2023, 06, 19);
```

```
    Ticket ticketObtenido = assertDoesNotThrow(() -> alquila.calculaPrecio(TipoCochе.Caravana, fecha, 7));
```

```
    assertEquals(62,5, ticketObtenido.getPrecioFinal());
```

```
}
```

```
@Test
```

```
void alquilaC3() {
```

```
    cal = new CalendarioStub(new ArrayList<>(), Arrays.asList(18, 21, 22));
```

```
    alquila.calendario = cal;
```

```
    LocalDate fechaInicio = new LocalDate(2023, 04, 17);
```

```
    MensajeException exp = assertThrows(MensajeException.class, () -> alquila.calculaPrecio(TipoCochе.Turismo, fecha, 8));
```

```
    String errorEsperado = "Error dia 2023-04-18; Error dia 2023-04-21;
```

```
    Error dia 2023-04-22;";
```

```
    assertEquals(errorEsperado, exp.getMessage());
```

```
}
```

Id	Datos Entrada				Resultado Esperado
	Tipo	fechainicio	dias	es_festivo()	
C1	TURISMO	2023-05-18	10	false para todos los días	Ticket (75)
C2	CARAVANA	2023-06-19	7	true solo para los días 20 y 24	Ticket (62,5)
C3	TURISMO	2023-04-17	8	false para todos los días, y lanza excepción los días 18, 21, y 22	("Error en dia: 2023-04-18; Error en dia: 2023-04-21; Error en dia: 2023-04-22;")

Ejercicio 4: drivers para reserva()

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

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

- Los campos **Name** y **Location** deben tener los valores: **Name: "reserva"**. **Location: "\$HOME/ppss-2023-Gx.../P04-Dependencias1/P04-stubs/"**
- Seleccionamos el lenguaje **Java**, la herramienta de construcción **Maven**, y nos aseguramos de elegir el **JDK 11**
- Desmarcamos la casilla **Add sample code**
- Desde **Advanced Settings**, indicamos las coordenadas de nuestro proyecto: **GroupId: "ppss.P04"**; **ArtifactId: "reserva"**

- * Dependencia externa en misma clase debemos hacer un override
* Dependen externa debemos refactorizar según enunciado hay que hacer clase factoría

Dado el código de la unidad a probar (método **realizaReserva()**), se trata de implementar y ejecutar los drivers (usando verificación basada en el estado) automatizando así las pruebas unitarias de la tabla de casos de prueba proporcionada.

```
//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 [] isbn) throws ReservaException {
        ArrayList<String> errores = new ArrayList<>();
        if(!compruebaPermisos(login, password, Usuario.BIBLIOTECARIO)) {
            errores.add("ERROR de permisos");
        } else {
            IOperacionB0 io = new Operacion();
            try {
                for(String isbn: isbn) {
                    try {
                        io.operacionReserva(socio, isbn);
                    } catch (IsbnInvalidoException iie) {
                        errores.add("ISBN invalido" + ":" + isbn);
                    }
                }
            } catch (SocioInvalidoException sie) {
                errores.add("SOCIO invalido");
            } catch (JDBCEception 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
}
```

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

```
public class JDBCEception extends Exception {}

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

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

```
public interface IOperacionB0 {
    public void operacionReserva(String socio, String isbn)
        throws IsbnInvalidoException, JDBCEception, 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"	{"111111"}	--	ReservaException1
C2	"ppss"	"ppss"	"Luis"	{"111111", "222222"}	{NoExcep, NoExcep.}	No se lanza excep.
C3	"ppss"	"ppss"	"Luis"	{"111111", "333333", "444444", "555555"}	{NoExcep, IsbnEx, IsbnEx}	ReservaException2
C4	"ppss"	"ppss"	"Pepe"	{"111111"}	{SocioEx}	ReservaException3
C5	"ppss"	"ppss"	"Luis"	{"111111", "222222"}	{NoExcep, JDBCEEx}	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 "111111", "222222".

-- significa que no se invoca al método reserva()

NoExcep. → El método reserva no lanza ninguna excepción

IsbnEx → Excepción IsbnInvalidoException

SocioEx → Excepción SocioInvalidoException

JDBCEEx → Excepción JDBCEception

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.

① Creamos nuestra clase factoría

```
public class OperacionFactory {  
    private IOperacionBO operacion = null;  
    public static IOperacionBO create() {  
        if (operacion == null) {  
            return new Operacion();  
        } else {  
            return operacion;  
        }  
    }  
  
    public void setOperacion(IOperacion op) {  
        operacion = op;  
    }  
}
```

Este If se pone para no modificar el comportamiento del Set si no se ha injectado ninguna operación.

② Refactorizamos nuestra Set con la factoría

```
//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[] isbn) throws ReservaException {  
  
        ArrayList<String> errores = new ArrayList<>();  
        if (!compruebaPermisos(login, password, Usuario.BIBLIOTECARIO)) {  
            errores.add("ERROR de permisos");  
        } else {  
            IOperacionBO io = OperacionFactory.create();  
            try {  
                for (String isbn : isbn) {  
                    try {  
                        io.operacionReserva(socio, isbn);  
                    } catch (IsbnInvalidoException iie) {  
                        errores.add("ISBN invalido" + ":" + isbn);  
                    }  
                }  
            } catch (SocioInvalidoException sie) {  
                errores.add("SOCIO invalido");  
            } catch (JDBCEception 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  
}
```

③ Creamos el Stub de operación

```
public class IOperacionStub implements IOperacionBO {  
    List<String> ISBNsExcept;  
    Boolean socioExcept = false;  
    Boolean conexionExcept = false;  
  
    void setIsBNsExcept(List<String> ISBNs) {  
        ISBNsExcept = ISBNs;  
    }  
  
    void setSocioExcept(Boolean e) {  
        socioExcept = e;  
    }  
  
    void setConexionExcept(Boolean e) {  
        conexionExcept = e;  
    }  
}
```

④ override

```
public void OperacionReserva(String Socio, String ISBN)  
throws ISBNInvalidoException, JDBCEception,  
SocioInvalidoException {  
    if (socioExcept) {  
        throw new SocioInvalidoException;  
    } else if (ISBNsExcept.contains(ISBN)) {  
        throw new ISBNInvalidoException;  
    } else if (conexionExcept) {  
        throw new JDBCEception;  
    }  
}
```

4) Creamos nuestra clase Testable

```

public class ReservaTestable extends Reserva {
    private String validLogin;
    private String validPassword;
    private Usuario validUser;

    public void setvalidLogin(String v){validLogin = v}
    public void setvalidPassowrd(String v){validPassword = v}
    public void setvalidUser(Usuario v){validUser = v}

    @Override
    public Boolean compruebaPermisos (String us, String pass, Usuario u) {
        if (us == validLogin and pass == validPassword and u == validUser){
            return true
        } else {
            return false;
        }
    }
}

```

5) Hacemos la clase de test

```

public class ReservaTest {
    ReservaTestable reserva;
    IOperacionStub operacion;
    String validUser = "ppss";
    String validPass = "ppss";
    Usuario validUser = Usuario.Bibliotecario;

    @BeforeEach
    void setup() {
        reserva = new ReservaTestable();
        operacion = new operacion();
        OperacioFactory.setOperacion(operacion);
        reserva.setvalidLogin(validUser);
        reserva.setvalidPass(validPass);
        reserva.validUser(validUser);
    }
}

```

@Test

```

void ReservaC1 () {
    String ISBN [] = {"1111", "2222"};
    ReservaException ex = AssertThrows (ReservaException.class, () -> RealizaReserva ("xxxx", "xxxx", "Luis", ISBN));
    assertEquals ("ERROR de Permisos", ex.getMessage());
}

```

@Test

```

void ReservaC2 () {
    String ISBN [] = {"1111", "2222"};
    assertDoesNotThrow ( () -> RealizaReserva ("ppss", "ppss", "Luis", ISBN));
}

```

	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 excepc.
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;"

② Test

```
void ReservaC3() {
    String ISBN[] = {"11111", "33333", "44444"};
    Operacion.setISBNsExcept(Array.asList(ISBN[1], ISBN[2]));
    String exEsperada = "ISBN invalido: 33333; ISBN invalido: 44444";
    ReservaException op = AssertThrows(ReservaException.class, () -> RealizaReserva("ppss", "ppss", "luis", ISBN));
    assertEquals(exEsperada, op.getMessage());
}
```

② Test

```
void ReservaC4() {
    String ISBN[] = {"11111"};
    Operacion.setSocioExcept(true);
    String exEsperada = "Socio invalido";
    ReservaException op = AssertThrows(ReservaException.class, () -> RealizaReserva("ppss", "ppss", "luis", ISBN));
    assertEquals(exEsperada, op.getMessage());
}
```

② Test

```
void ReservaC5() {
    String ISBN[] = {"11111", "22222"};
    Operacion.setISBNsExcept(Array.asList(ISBN[2]));
    String exEsperada = "CONEXION INVALIDA";
    Operacion.setConexionExcept(true);
    ReservaException op = AssertThrows(ReservaException.class, () -> RealizaReserva("ppss", "ppss", "luis", ISBN));
    assertEquals(exEsperada, op.getMessage());
}
```