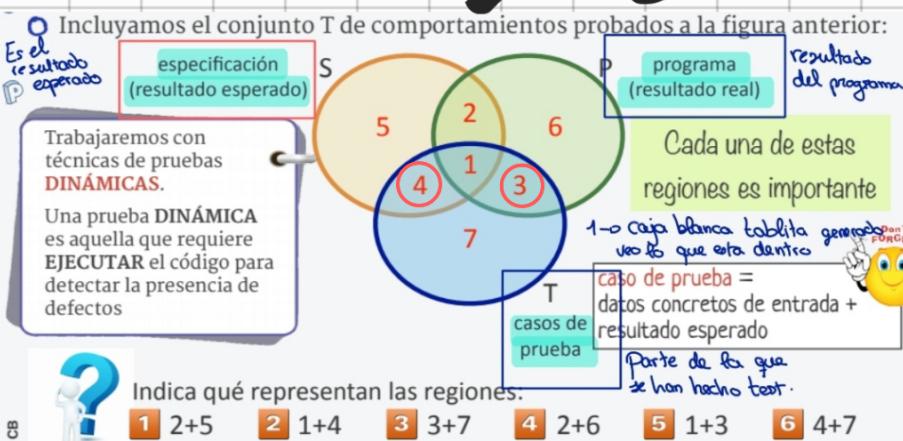


Caja Blanca



3 la caja blanca sería todo lo que se prueba sobre lo implementado sin tener en cuenta lo especificado.

4º Sería la especificación con caja negra donde no veríamos el código pero si la especificación y deberíamos hacer test con la misma

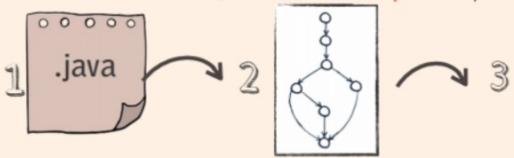
- Tabla creada a partir del procedimiento de camino básico en el tipo de pruebas de caja blanca.
- Tabla resultante del diseño de las pruebas:

Camino	Entrada 1	Entrada 2	...	Entrada n	Resultado Esperado
C1	d11	d12	...	d1n	r1
...					
C2	d21	d22	...	d2n	r2

• Camino básico

Los métodos de diseño de casos de prueba basados en el CÓDIGO:

- 1 Analizan el código y obtienen una representación en forma de GRAFO
- 2 Seleccionan un conjunto de caminos en el grafo según algún criterio
- 3 Obtienen un conjunto de casos de prueba que ejercitan dichos caminos



Salida esperada

Salida real

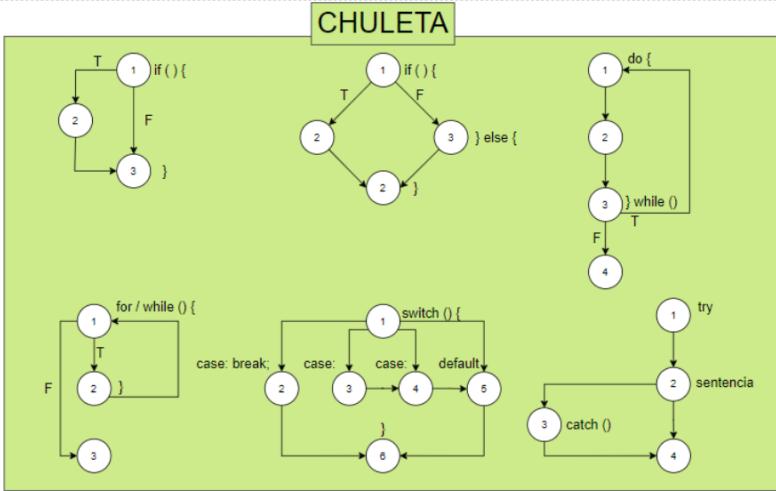
Variables que modificaron

Camino que probamos

OBSERVACIONES SOBRE LOS MÉTODOS ESTRUCTURALES

- Dependiendo del método concreto utilizado, obtendremos conjuntos DIFERENTES de casos de prueba. Pero el conjunto obtenido será EFECTIVO y EFICIENTE!!!
- Las técnicas o métodos estructurales son costosos de aplicar; por lo que suelen usarse sólo a nivel de UNIDADES de programa.
- Los métodos estructurales NO pueden DETECTAR TODOS los defectos en el programa (defecto = fault = bug). Aunque seleccionemos todas las posibles entradas, no podremos detectar todos los defectos si "faltan caminos" en el programa.
- De forma intuitiva, diremos que falta un camino en el programa si no existe el código para manejar una determinada condición de entrada.
- Por ejemplo: si la implementación no prevé que un divisor pueda tener un valor cero, entonces no se incluirá el código necesario para manejar esta situación

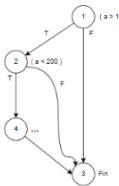
Representación de situaciones



Ejercicios Resueltos de formación de árboles

1

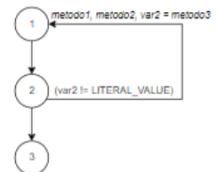
```
if ((a > 1) && (a < 200)) {
    ...
}
```



3

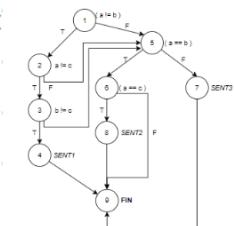
```
do {
    metodo1(var1, var2);
    metodo2(var3);

    var2 = metodo3();
} while (var2 != VALOR);
```



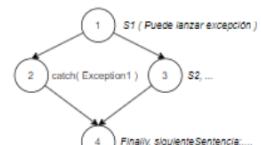
2

```
if ((a != b) && (a != c) && (b != c)) {
    ... SENT1
} else {
    if (a==b) {
        if (a==c) {
            ... SENT2
        }
    } else {
        ... SENT3
    }
}
```



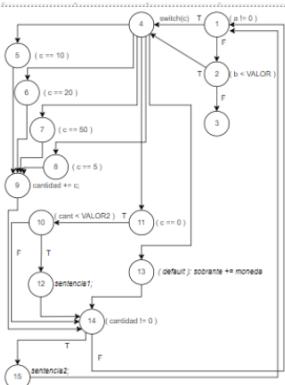
4

```
try {
    s1; //puede lanzar Exception1;
    s2; //no lanza ninguna excepción
    ...
} catch (Exception1 e) {
    ...
} finally {
    ...
}
siguienteSentencia;
```



5

```
while ((a!=0) || (b < VALOR)){
    switch (c) {
        case 5:
        case 10:
        case 20:
        case 50: cantidad += c; break;
        case 0: if (cantidad < VALOR2) {
                    sentencial;
                break;
            default: sobrante += moneda;
        }
        if (cantidad != 0) {
            sentencia2;
        }
    }
}
```



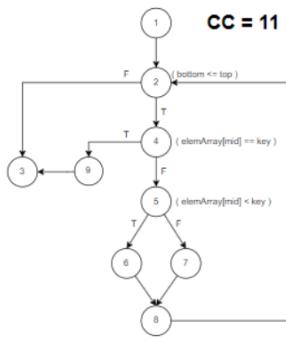
6

```
//Asumimos que la lista de elementos está ordenada de forma ascendente
class BinSearch
public static void search (int key, int [] elemArray, Result r) {
    int bottom = 0; int top = elemArray.length -1;
    int mid; r.found = false; r.index= -1;
    while (bottom <= top) {
        mid = (top+bottom)/2;
        if (elemArray [mid] == key) {
            r.index = mid;
            r.found = true;
            return;
        } else {
            if (elemArray [mid] < key)
                bottom = mid + 1;
            else top = mid -1;
        }
    } //while loop
} //search
} //class
```

Especificación del método search():
Dado un vector de enteros ordenados
ascendente, y dado un entero
(key) como entrada, el método
search() busca la posición de key en
el vector y devuelve el valor
found=true si lo encuentra, así como
su posición en el vector (dada por
índice). Si el valor de key no está en
el vector, entonces devuelve el valor
found=false

$$CC = 11 - 9 + (2) = 4$$

Se pone "?" porque no se ha especificado que es r.index cuando no encuentra el valor.



7

Calcula la CC para cada uno de estos códigos Java:

```
public int divide(int numberToDivide, int numberToDivideBy) throws BadNumberException{
    if(numberToDivide == 0){
        throw new BadNumberException("Cannot divide by 0");
    }
    return numberToDivide / numberToDivideBy;
}
```

Código 1

```
public void openFile(){
    try {
        int result = divide(n);
        System.out.println(result);
    } catch (BadNumberException e) {
        //do something clever with the exception
        System.out.println(e.getMessage());
    }
    System.out.println("Division attempt done");
}
```

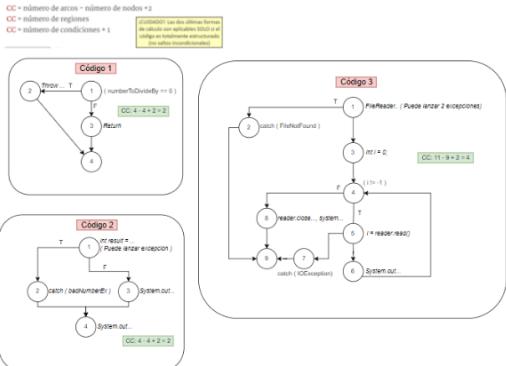
Código 2

CC = número de acciones - número de nodos = 2
 CC = número de regiones
 CC = número de condiciones + 1

Combinar con los últimos nodos de cada una de las regiones para aplicarle S002 al resultado final.

Código 3

```
try {
    // constructor may throw FileNotFoundException
    FileReader reader = new FileReader("somefile");
    int i;
    while(i != -1){
        //read[i] may throw IOException
        i = reader.read();
        System.out.print((char)i);
    }
    reader.close();
    System.out.println("File End ---");
} catch (FileNotFoundException e) {
    //do something clever with the exception
} catch (IOException e) {
    //do something clever with the exception
}
```



Drivers

: los drivers se utilizan para automatizar las pruebas unitarias, en si los drivers son las pruebas que vamos a ejecutar, por cada comportamiento es decir por cada camino de una unidad deberemos implementar un driver.

Los drivers necesitarán una librería para su ejecución `import org.junit.jupiter.api.Test;`

la cual podemos resumir en "junit-jupiter-engine"

• SUT: Sistema Under test es el código que nos disponemos a probar, es decir las unidades estas se definen como métodos de código (métodos Java en JUnit) se guardan en : `Src/main/java`.

```
<groupId>ppss</groupId>
<artifactId>drivers</artifactId>
<version>1.0-SNAPSHOT</version>

<properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <maven.compiler.release>11</maven.compiler.release>
</properties>

<dependencies>
    <dependency>
        <groupId>org.junit.jupiter</groupId>
        <artifactId>junit-jupiter-engine</artifactId>
        <!-- No consigue resolver la versión 3.8.2 de JUNIT-->
        <version>5.8.2</version>
        <scope>test</scope>
    </dependency>
</dependencies>

<build>
    <plugins>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-compiler-plugin</artifactId>
            <version>3.9.0</version>
        </plugin>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-surefire-plugin</artifactId>
            <version>3.0.0-M5</version>
        </plugin>
    </plugins>
</build>

```

El pom

① Coordenadas de un proyecto combinación de :

- `groupId` → hace referencia al grupo del proyecto por ejemplo desarrollo, testing, producción...
- `ArtifactId` → es el nombre del proyecto
- Versión → número de versión

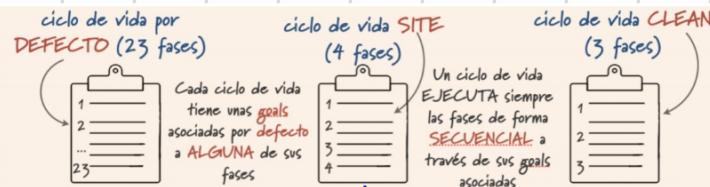
② Propiedades del proyecto que serían el formato el tipo de compilador

③ Dependencias: todas las librerías que se van a usar en nuestro proyecto.
la ruta en la que se guardarán será: \$HOME/.m2/repository

- org.pps:practica1:1.0-SNAPSHOT representa al fichero \$HOME/.m2/repository/org/pps/practica1/1.0-SNAPSHOT/practica1-1.0-SNAPSHOT.jar
- org.pps:practica1:2.0-SNAPSHOT representa al fichero \$HOME/.m2/repository/org/pps/practica1/2.0-SNAPSHOT/practica1-2.0-SNAPSHOT.jar
- org.pps:projeto3:war:1.0-SNAPSHOT representa al fichero \$HOME/.m2/repository/org/pps/projeto3/1.0-SNAPSHOT/projeto3-1.0-SNAPSHOT.war

Ejemplos de guardado de dependencias
en carpeta .m2

④ Build es un conjunto de plugins, los plugins son un conjunto de goals y los goals son comandos a realizar



Fase	plugin : goal	acciones realizadas por la goal
process-resources	maven-resources-plugin: resources	Copia *.* de /src/main/resources en target
compile	maven-compiler-plugin: compile	Compila *.java de /src/main/java
process-test-resources	maven-resources-plugin: testResources	Copia *.* de /src/test/resources en target
test-compile	maven-compiler-plugin: testCompile	Compila *.java de /src/test/java
test	maven-surefire-plugin: test	Ejecuta los tests unitarios
package	maven-jar-plugin:jar	Empaqueta *.class + recursos en un jar
install	maven-install-plugin: install	Copia el fichero jar en repositorio local
deploy	maven-deploy-plugin: deploy	Copia el fichero jar en repositorio remoto

Los son las goals del ciclo de vida.

- Recordemos que una fase no es un ejecutable sino un proceso lógico
- El nombre de un goal siempre va precedido por el nombre del plugin separado por ":"
- El proceso de construcción de maven genera un fichero empaquetado en el directorio target. Este target dependerá de la extensión del proyecto (.jar)

→ Cuando ejecutamos el ciclo de vida de la compilación y ejecución del proyecto nos creará la carpeta "target" donde se volverán los compilados (.jar), además de informes generados si hemos ejecutado pruebas se generan unos informes de las pruebas.

Ejemplo de ejecución:

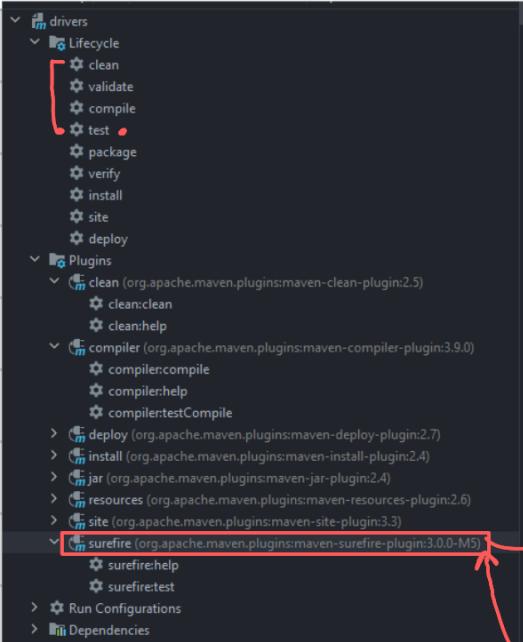
drivers [P02-Drivers] C:\Users\SApro\Desktop\UA\PPSS_UA\PP

- drivers [P02-Drivers]
- .idea
- src
 - main
 - java
 - ppss
 - Cine
 - dataArray
 - DataException
 - FicheroException
 - FicheroTexto
 - resources
 - test
 - java
 - ppss
 - CineTest
 - dataArrayTest
 - FicheroTextoTest

mvn clean test-compile test
pasos que deseamos ejecutar
* Si ejecutamos una fase intermedia
se ejecutarán todos los anteriores

target

- classes
 - ppss
 - Cine
 - DataArray
 - DataException
 - FicheroException
 - FicheroTexto
 - runConfigurationsIntelliJ
- generated-sources
- generated-test-sources
- maven-status
- surefire-reports
 - ppss.CineTest.txt
 - ppssdataArrayTest.txt
 - ppss.FicheroTextoTest.txt
 - TEST-ppss.CineTest.xml
 - TEST-ppssdataArrayTest.xml
 - TEST-ppssFicheroTextoTest.xml
- test-classes
 - ppss
 - CineTest
 - dataArrayTest
 - FicheroTextoTest



- * Ejemplo: Si ejecutamos "test" se ejecutarán先后 "clean → validate → compile" en ese orden
- * Si ejecutarmos el comando desde la terminal el ciclo de vida se rompe por lo que tendremos que forzarlo solo ejecutando el comando especificado
- * Si añadimos un plugin al pom este aparecerá en el apartado de plugins, si lo desplegamos aparecen los goals a ejecutar.

```

<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-compiler-plugin</artifactId>
      <version>3.9.0</version>
    </plugin>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-surefire-plugin</artifactId>
      <version>3.0.0-M5</version>
    </plugin>
  </plugins>
</build>

```

Funcionamiento de Junit.

```
-assert:
  assertEquals(resultadoEsperado, resultadoReal);
```

Comprobará que los dos parámetros especificados son iguales, en el caso de que los parámetros sean objetos si no tienen el método equals implementado o son el mismo objeto dará error.

```
-assertAll:
  assertAll("GrupoTestC3",
  ()-> assertEquals(arrayEsperado, colección.getColección()),
  ()-> assertEquals(numElemEsperado, colección.size())
);
//si no se usa assertAll, en caso de que el un assert falle no se ejecutan los restantes
```

Realizará los assertEquals uno por uno de tal manera que

aunque alguno de ellos falle se continuará con la ejecución, el fallo se verá reflejado en el informe.

```
@Each:
  @BeforeEach/@AfterEach
  void createOutputFile(){}
  @BeforeAll/@AfterAll
  static void initialState() {}
```

BeforeEach /AfterEach: Se ejecutarán antes y después de que finalice un driver.

BeforeAll / AfterAll: Se ejecutarán una vez antes de comenzar a realizar los drivers.

```

-assertThrows:
    Throwable exception = assertThrows(ExpectedException.class,
                                            () -> sut(e1,e2));
    assertEquals("a message", exception.getMessage());

```

Si alguna de nuestras SUTs lanza alguna excepción deberemos realizar el driver de la misma dentro del assertThrows de manera que la capture, de esta manera también podremos comprobar si el mensaje que lanza es el correcto.

```

-@Tag:
    @Test
    @Tag("taxes")
    @Tag("model")
    void testingTaxCalculation() {}

```

Este surijo sera el que debamos colocar para indicarle a maven que la función implementada es un test este lo podremos variar si lo deseamos

Mediante los tag podremos especificar cuales de los drivers especificados queremos ejecutar

```

<dependencies>
    <dependency>
        <groupId>org.junit.jupiter</groupId>
        <artifactId>junit-jupiter-engine</artifactId>
        <!-- No consigue resolver la version 3.8.2 de JUnit-->
        <version>5.8.2</version>
        <scope>test</scope>
    </dependency>
</dependencies>

```

```

-@ValueSource:
    @ParameterizedTest
    ④ @ValueSource(strings = {"racecar", "radar", "able was I ere I saw elba"})
    void palindromes(String candidate) {
        assertTrue(c.isPalindrome(candidate));
    }

```

• Solo para un parámetro

Estos son los test parametrizados
los utilizaremos siempre que queramos implementar unos test que posean la misma lógica pero queramos variar los parámetros con los que se ejecuta. ④ Parámetros que probaremos.

• Si nuestro método tiene más de un parámetro:

```

-@MethodSource:
    @ParameterizedTest
    @MethodSource("datos")
    void calculaTasaMatricula(int edad, boolean familiaNumerosa, boolean repetidor, float esperado) {
        Matricula a = new Matricula();
        assertEquals(esperado, a.calculaTasaMatricula(edad, familiaNumerosa, repetidor));
    }

    private static Stream<Arguments> datos(){
        return Stream.of(
            Arguments.of(19, false, true, 2000.00f),
            Arguments.of(68, false, true, 250.00f),
            Arguments.of(19, true, true, 250.00f),
            Arguments.of(19, false, false, 500.00f),
            Arguments.of(61, false, false, 400.00f)
        );
    }

```

Argumentos que vamos a probar

```

-assert con mensaje de error:
    assertEquals(expected, transp.canAccessTransporter(user),
                () -> generateFailureMessage("transporter", expected, user, alertStatus));

    private String generateFailureMessage(String system, boolean expected, Person user, Alert alertStatus) {
        String message = user.getFirstName() + " should";
        if (!expected) {
            message += " not";
        }
        message += " be able to access the " + system + " when alert status is " + alertStatus;
        return message;
    }

```

JUnit además nos permitirá personalizar el mensaje de error cuando nuestros test fallen

• Todos los métodos "assert" que generan una excepción del tipo "AssertionFailedError"
Si la asección no se cumple

Diseño de Pruebas: Caja Negra

Objetivo: el objetivo es obtener una tabla de casos de prueba a partir del conjunto de comportamientos especificados, debe de detectar dado un objetivo el máximo número de errores posibles en el código, con el mínimo número posible de filas.

Pasos a seguir:

1º Análisis de la especificación cada entrada y salida en conjuntos equivalentes.

2º Seleccionamos el comportamiento usando las particiones obtenidas.

3º Obtención de los conjuntos de prueba

- Cada entrada diferente con un resultado diferente es un comportamiento único.
- Podremos detectar comportamientos no implementados pero nunca los no especificados
- Los casos de prueba obtenidos son independientes de la implementación
- No debemos aplicar este diseño a problemas muy grandes
- El inconveniente de la caja negra no podremos detectar los comportamientos implementados pero no especificados
- Cada partición de entrada representa un subconjunto total de datos posibles de entrada que tienen el mismo comportamiento

¿Cómo identificar las particiones? Las particiones se identifican en base a condiciones de entrada/salida de la unidad a probar. Una condición de entrada/salida puede aplicarse a una única variable o subconjunto de ellas. Estas variables pueden o no ser parámetros de entrada de la unidad a probar.

Las clases de equivalencia (condiciones, particiones) de entrada pueden clasificarse como válidas o inválidas. Las particiones de entrada inválidas normalmente tienen asociadas clases inválidas.

Solo puede haber una partición inválida por caso de prueba.

Identificación de las clases de equivalencia:

- Si la E/S especificada es un rango de valores válidos, definiremos una clase válida (dentro del rango) y dos inválidas (fuera de cada uno de los extremos)

- Si la E/S especifica un número N de valores definiremos una clase válida (número de entre 1 y N) y dos invalidas (ningún valor, más de N valores).
- Si la E/S especifica un conjunto de valores válidos definiremos una clase válida (valores que pertenecen al conjunto) y una invalida (valores que no pertenecen al conjunto)
- Si por alguna razón se piensa que cada uno de los valores de entrada se van a tratar de forma diferente en el programa, entonces definir una clase válida para cada valor de entrada.
- Si la E/S especifica una situación DEBESER definiremos una clase invalida y otra válida por ejemplo cuando se dice que un pin debe empezar por un dígito.
- Si por alguna razón se piensa que los valores de una partición van a ser tratados de forma distinta Subdividir la partición en particiones más pequeñas.

Identificar los casos de prueba de la siguiente forma:

- Hasta que todas las clases válidas estén probadas escribir un nuevo caso de prueba que cubra todas las clases válidas no probadas.
- Hasta que todas las clases invalidas estén probadas escribir un nuevo caso de prueba que cubra todas las clases válidas no probadas.
- Elegir un valor concreto para cada partición.
- El conjunto de pruebas debe contemplar todos mínimo una vez, habiendo solo una partición invalida por cada caso de prueba.

Ejemplos tema 3

Se eligen en una lista desplegable

ESPECIFICACIÓN: Método en el que, dados como entradas: un carácter X introducido por el usuario, un número N entre 5 y 10, y el valor "rojo" o "azul", devuelve (salida) una cadena de N caracteres X de color rojo o (N-l) caracteres de color azul, o bien el mensaje "ERROR: repite entrada" si el usuario proporciona un valor de N < 5 ó N >10.

Datos

- Entrada 1: puede ser cualquier carácter
 - ↳ Clase válida V₁ ✓
- Entrada 2: un valor comprendido entre 5 y 10
 - ↳ Clase válida V₂: un valor entre 5 y 10 ✓

- ↳ Clase invalida N1: valores menores que 5 ✓
- ↳ Clase invalida N2: valores mayores que 10 ✓

- Entrada 3: uno de los valores "rojo" y "azul"

- ↳ Clase valida V3 = Rojo ✓
- ↳ Clase valida V4 = azul. ✓

- Salida (cadena de N caracteres):

- ↳ Clase valida S1: Cadena de n caracteres rojos ✓
- ↳ Clase valida S2: Cadena de n-1 caracteres azules ✓
- ↳ Clase Invalida NS1: "Error repite entrada"

Clases

Datos de Entrada

Resultado Esperado

	E1	E2	E3	
• V1-V2-V3-S1	"h"	5	rojo	"hhhhh"
• V1-V2-V4-S2	"h"	5	azul	"hhhh"
• V1-N1-V3-NS1	"h"	4	rojo	"Error repite entrada"
• V1-N2-V3-NS1	"h"	11	rojo	"Error repite entrada"

• Dependencias

Los dependencias son todos aquellos llamados a funciones externas a la SUT

Proceso para aislar la unidad de sus dependencias externas:

1º Identificación de las dependencias externas

2º Refactorización de la unidad solo si es necesario para conseguir injectar los dobles de las dependencias externas.

3º Control de las dependencias externas implementando un doble (stub) para controlar las entradas indirectas al SUT.

4º Implementación del driver utilizando verificación basada en el estado.

Las pruebas de unidad dinámicas requieren ejecutar cada unidad de forma aislada.

- El código de la SUT tiene que ser exactamente el mismo código que se utilizará en producción, es decir que no estará permitido alterarlo circunstancialmente.
- Código Testable: aquel que permite que un código sea fácilmente probado de manera aislada. Siendo capaces de controlar sus dependencias externas, también denominadas colaboradores o Dtos
- Dependencia externa elemento con el que interactua nuestro código y no tenemos ningún control
- Para poder realizar este reemplazo necesitamos que nuestro código tenga uno o SEAMS
- Seams son partes de nuestro código (que pueden o no estar en el SUT) que nos permiten alterar el comportamiento del SUT sin modificar el código del mismo. Este nos permitirá injectar los dobles durante las pruebas.
- Dóbles Reemplaza el código de los colaboradores utilizados durante las pruebas para aislar el SUT

```

public class GestorLlamadas {
    static double TARIFA_NOCTURNA=10.5;
    static double TARIFA_DIURNA=20.8;

    public int getHoraActual() {
        Calendar c = Calendar.getInstance();
        return c.get(Calendar.HOUR);
    }

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

```

↓

Este es un SEAM porque nos permite cambiar el comportamiento del SUT sin cambiar el código del propio SUT

SUT: Función a testear ASILADAMENTE, nos da igual si el SUT llama a otras funciones (en este caso getHoraActual()) porque solo queremos probar el SUT aisladamente, por ello podemos cambiar el comportamiento de getHoraActual para cambiar el comportamiento del SUT

```

public class GestorLlamadasTestable extends GestorLlamadas {
    private int HoraActual=0;

    @Override
    public int getHoraActual() {
        return HoraActual;
    }

    public void setHoraActual(int hora) {
        HoraActual = hora;
    }
}

```

Al crear una clase Testable de la clase que alberga el SUT (heredando de la clase del SUT) podemos sobreescibir y cambiar el comportamiento de getHoraActual de esta manera, no tocamos el código, y podemos especificar nuestro valor propio que usaremos durante las pruebas, cambiando así el comportamiento del SUT, sin modificar el código del SUT.

Recordemos que el SUT es la función unitaria a probar, en concreto "calcularConsumo".

- 1º Identificamos la dependencia en el código en este caso getHoraActual()
- 2º Creamos la clase testable la cual heredará de nuestra clase original
- 3º Sobreescribimos el método que queremos que tenga un comportamiento en específico, también haremos un setter si es necesario para injectar el valor deseado.

Esto lo realizaremos para todas las métodos que tengan una dependencia

- Necesitamos poder cambiar la dependencia real por su doble, lo cual no será posible. Sino tenemos un seam para cada dependencia externa que nos permita injectar nuestro doble durante las pruebas.
- Los SEAMS que podemos encontrar en nuestro código son los siguientes:

1) Mediante un parámetro en el SUT

```
public class GestorLlamadas {
    static double TARIFA_NOCTURNA=10.5;
    static double TARIFA_DIURNA=20.8;

    public double calculaConsumo(int minutos, Calendar c) {
        int hora = c.get(Calendar.HOUR);
        if(hora < 8 ) {
            return minutos * TARIFA_NOCTURNA;
        } else {
            return minutos * TARIFA_DIURNA;
        }
    }
}
```

2) Si el constructor instancia dicho objeto pero no lo recibe por parámetro deberemos alterar su comportamiento haciendo una clase testeable

Si nuestro SUT usa de alguna función (DOC) de un atributo de la clase y ese objeto se define en el constructor, dicho constructor es el SEAM.

3) Si nuestro SUT usa de alguna función (DOC) de un atributo de la clase y ese objeto se define en un setter, dicho setter es el SEAM

<pre>public class GestorLlamadas { static double TARIFA_NOCTURNA=10.5; static double TARIFA_DIURNA=20.8; public int getHoraActual() { Calendar c = Calendar.getInstance(); return c.get(Calendar.HOUR); } public double calculaConsumo(int minutos) { int hora = getHoraActual(); if(hora < 8 hora > 20) { return minutos * TARIFA_NOCTURNA; } else { return minutos * TARIFA_DIURNA; } } }</pre>	<pre>public class GestorLlamadasTestable extends GestorLlamadas { private int HoraActual=0; @Override public int getHoraActual() { return HoraActual; } public void setHoraActual(int hora) { HoraActual = hora; } }</pre> <p>Al crear una clase Testable de la clase que alberga el SUT (herando de la clase del SUT) podemos sobreescribir y cambiar el comportamiento de getHoraActual de esta manera, no tocamos el código, y podemos especificar nuestro valor propio que usaremos durante las pruebas, cambiando así el comportamiento del SUT, sin modificar el código del SUT.</p> <p>Recordemos que el SUT es la función unitaria a probar, en concreto "calcularConsumo".</p>
--	---

4) Si nuestro SUT usa de alguna función (DOC) de un objeto devuelto por otra función de la clase que contiene el SUT. (Factoría local).

5) Si el constructor se encarga de definir el atributo u objeto que se usa en el Set y dicho objeto se recibe por parámetro en el constructor.

Si el objeto que se usa en el set se define a través de un setter de la propia clase

O si el atributo que se usa en el set tiene una visibilidad pública, protected o package

Si se da alguno de los anteriores casos no será necesario crear una clase testeable ya que no requerimos alterar ningún método para injectar nuestro STUB.

- Recordemos que si realizamos la clase testeable con sus respectivos override la realizaremos dentro de "Test/Java/paquete"

- Si nuestro SUT no es testeable entonces deberemos de refactorizar el código

- 1) Añadir un parámetro a nuestra SUT
- 2) Añadir un parámetro al constructor de nuestra SUT
- 3) Método de factoría local no se verán afectados ni clientes ni clase aunque añadamos un nuevo método.

Ejemplo:

```
public class GestorPedidos {  
    public Factura generarFactura(Cliente cli) throws FacturaException {  
        Factura factura;  
        Buscador 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;  
    }  
}
```

Al instanciarse dentro de la clase no podemos injectar a nuestro doble por lo que no será un código testeable

Si utilizamos la opción de refactorización (Es la peor) quedaría tal que así:

```
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 nada pendiente de facturar");  
    }  
    return factura;  
}  
...
```

Versión
refactorizada (en
src/main/java)

→ De esta manera se añade un nuevo atributo a nuestra clase y ahora si que será testeable.

SUT REFACTORIZADA. AHORA SÍ ES TESTABLE!!!

Test Stub: es un objeto que actúa como un punto de control para entregar entradas indirectas al SUT, cuando se invoca a alguno de los métodos de dicho Stub

```
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 nada pendiente de facturar");
    }
    return factura;
}
```

Crearemos el Stub del buscador y realizaremos el override de las funciones que necesitamos además de los setters.

Clase Original

en src/main/java
public class Buscador {
 //código REAL de nuestro DOC
 //en src/main/java
 public int elemPendientes(Cliente cli) {
 D O C ...
 IMPLEMENTACIÓN REAL
 }

Stub generado

en src/test/java
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)

Implementación del Driver:

- Los drivers tienen tres estados → pass que ha funcionado, fail que ha realizado la prueba y ha fallado y el error que no se ejecuta.

```
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;
}
```

Los dos tests ejecutan el MISMO CÓDIGO!!! SUT

Podemos realizar 2 tipos de test, los unitarios, donde aislamos la unidad a probar y los de integración que incluyen varias unidades.

Test Unitario

```
public class GestorPedidosTest {
    @Test
    public void testGenerarFactura()
        throws Exception {
        Cliente cli = new Cliente(...);
        GestorPedidos sut = new GestorPedidos();
        BuscadorStub buscarStub = new BuscadorStub();
        buscarStub.setResult(10); Seteador
        Factura expectedResult = new Factura(...);
        Factura realResult =
            sut.generarFactura(cli, buscarStub);
        assertEquals(expectedResult, realResult);
    }
}
```

Aquí controlamos las entradas indirectas!

Test de Integración

```
public class GestorPedidosIT {
    @Test
    public void testGenerarFactura()
        throws Exception {
        Cliente cli = new Cliente(...);
        GestorPedidos sut = new GestorPedidos();
        Buscador buscar = new Buscador();
        Factura expectedResult = new Factura(...);
        Factura realResult =
            sut.generarFactura(cli, buscar);
        assertEquals(expectedResult, realResult);
    }
}
```

NO tenemos control sobre las entradas indirectas!

En los test unitarios deberemos generar los stub que necesitemos para aislar nuestro SUT

Ejemplo:

```
public class GestorPedidos { /src/main/java  
    public Factura generarFactura(Cliente cli)  
        throws FacturaException {  
        Factura factura = new Factura();  
        IService 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!!!

```
public interface IService { /src/main/java  
    public int elemPendientes(Cliente cli);  
}
```

```
public class Buscador implements IService {  
    @Override  
    public int elemPendientes(Cliente cli)  
        {...}  
    ...  
}
```

/src/main/java

Deberemos refactorizar nuestra unidad para poder injectar los dobles. Realizaremos la refactorización mediante factoría local. El método de factoría local será una nueva dependencia externa pero que pertenece a la misma clase que nuestra SUT.

SUT refactorizado

1)

```
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!!!

Esta será la factoría que hemos implementado la cual devolverá un objeto de tipo buscador.
En vez de crear un nuevo objeto dentro del código ahora llamaremos a la factoría

2)

```
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)

Implementamos el STUB desde el cual injectaremos el código, el constructor que seteará el resultado el cual injectaremos el código donde elemPendientes

3)

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

/src/test/java SUT TESTABLE

Generamos la clase testeable sobrescribiendo el método que necesitamos en este caso getBuscador que ahora no genera el buscador, solo lo devuelve, el setter también lo creamos para que devuelva nuestro STUB.

```

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

```

/src/test/java

DRIVER

Una vez hechas todas las pasos anteriores podemos implementar nuestro driver.

- 1) Generamos los objetos necesarios para nuestro Driver, tened en cuenta que usamos los nuevas clases generadas es decir el STUB y

- la testeable, que al ser clases heredadas o que implementan una interfaz son capaces de usar los métodos de sus padres
- 2) Utilizando el setter implementado injectamos el código que hemos generado en el STUB
 - 3) Generamos la salida esperada y la salida real desde nuestra clase testeable
 - 4) Realizamos el assert para comprobar el resultado.