por Mari Chelo Rubio

Entornos de Desarrollo
Bloque 3
Tema 1: Pruebas de software. Tipos. Casos de prueba. Pruebas unitarias. Automatización.

3.1.1. Introducción.

Como estudiamos en el primer bloque, uno de los pasos en el desarrollo de una aplicación es la realización de las pruebas. Y no es una parte poco importante sino todo lo contrario. En este vídeo podréis observar cómo de importante es una buena y eficiente fase de pruebas.

Las pruebas de software consisten en la dinámica de la verificación del comportamiento de un programa en un conjunto finito de casos de prueba debidamente seleccionados. Son una serie de actividades que se realizan con el propósito de encontrar los posibles fallos de implementación, calidad o usabilidad de un programa u ordenador; probando el comportamiento del mismo.

3.1.2. Objetivos.

La prueba de software es un elemento crítico para la garantía del correcto funcionamiento del software. Entre sus objetivos están:

- Detectar defectos en el software.
- Verificar la integración adecuada de los componentes.
- Verificar que todos los requisitos se han implementado correctamente.
- Identificar y asegurar que los defectos encontrados se han corregido antes de entregar el software al cliente.
- Diseñar casos de prueba que sistemáticamente saquen a la luz diferentes clases de errores, haciéndolo con la menor cantidad de tiempo y esfuerzo.

3.1.3. Principios de las pruebas de software

- La prueba puede ser usada para mostrar la presencia de errores, pero nunca su ausencia.
- La principal dificultad del proceso de prueba es decidir cuándo parar.
- Evitar casos de pruebas no planificados, no reusables y triviales a menos que el programa sea verdaderamente sencillo.
- Una parte necesaria de un caso de prueba es la definición del resultado esperado.
- Los casos de pruebas tienen que ser escritos no solo para condiciones de entrada válidas y esperadas sino también para condiciones no válidas e inesperadas.
- El número de errores sin descubrir es directamente proporcional al número de errores descubiertos.

3.1.4. Etapas

- Seleccionar qué es lo que debe medir la prueba, es decir, cuál es su objetivo, para qué exactamente se hace la prueba.
- Decidir cómo se va a realizar la prueba, es decir, qué clase de prueba se va a utilizar para medir la calidad y qué clase de elementos de prueba se deben usar.

- Desarrollar los casos de prueba. Un caso de prueba es un conjunto de datos o situaciones de prueba que se utilizarán para ejecutar la unidad que se prueba o para revelar algo sobre el atributo de calidad que se está midiendo.
- Determinar cuáles deberían ser los resultados esperados de los casos de prueba y crear el documento que los contenga.
- Ejecutar los casos de prueba.

3.1.5. Evaluación de los resultados

Comparar los resultados de la prueba con los resultados esperados. Cualquier discrepancia entre ellos significa un error. Típicamente el error está en el sistema o unidad probada, pero también puede ser generado por algún aspecto del mismo proceso de prueba.

3.1.6. Clasificación de las pruebas

3.1.6.1. Pruebas unitarias

Pruebas para comprobar el correcto funcionamiento de una unidad de código. Por ejemplo una clase. Suelen ser ejecutadas por los desarrolladores y la técnica más habitual es la de caja blanca. Las características que tienen que intentar cumplir estas pruebas son:

- Automatizable
- Completas
- Repetibles
- Independientes: la ejecución de una prueba no debe afectar a la ejecución de otra
- Profesionales: Se deben realizar con la misma profesionalidad que el código.

3.1.6.2. Pruebas de sistemas

Deben demostrar que los sistemas cumplen con los requerimientos relativos a calidad y funcionalidad. Se clasifican en funcionales y no funcionales. Son realizadas por los desarrolladores o un equipo de pruebas en un ambiente controlado.

3.1.6.3. Pruebas de integración

Se realizan para encontrar errores en interfaces y en la interacción entre componentes. Son realizadas por los desarrolladores y algunas de las técnicas más conocidad son:

- Big bang: Integrarlo todo y probarlo todo de golpe.
- Top down: Se desarrollan y prueban los componentes siguiendo su jerarquía de arriba a abajo. Los componentes inferiores no desarrrollados se sustituyen por componentes auxiliares que simulan su comportamiento. Ventaja: Los interfaces entre componentes se prueban en una fase temprana.
- Bottom up: Los componentes que primero se desarrollan y se prueban son los de menor jerarquía. Ventaja: No se necesitan componentes auxiliares.
- Combinadas: Se combinan algunas partes se desarrollan y prueban siguiendo Top down y otras partes Bottom up.

3.1.6.4. Pruebas de aceptación

Estas pruebas las realizan los usuarios y jefes de proyecto para verificar que se cumplen los requerimientos de funcionalidad y calidad.

3.1.6.5. Pruebas de regresión

Las Pruebas de Regresión consisten en volver a probar un componente, tras haber sido modificado, para descubrir cualquier defecto introducido, o no cubierto previamente, como consecuencia de los cambios. Los defectos pueden encontrarse tanto en el software que se ha cambiado como en algún otro componente. Se ejecutan cuando se cambia el software o su entorno.

3.1.7. Casos de prueba

Para la realización de las pruebas se deben diseñar los llamados ***casos de prueba*** que son una serie de condiciones que se establecen con el objetivo de determinar si el software funciona correctamente según lo esperado.

Definición según el ISTQB (International Software Testing Qualifications Board):

“test case Ref: After IEEE 610 A set of input values, execution preconditions, expected results and execution postconditions, developed for a particular objective or test condition, such as to exercise a particular program path or to verify compliance with a specific requirement”.

Un conjunto de valores de entrada, precondiciones de ejecución, resultados esperados y postcondiciones de ejecución, desarrollados para un objetivo particular de condición de prueba, tal como para ejercer una ruta de un programa en particular o para verificar el cumplimiento de un requisito específico

Para realizar estos casos de prueba existen diversos formatos pero los siguientes datos deben aparecer como mínimo:

- **Identificador**: Puede ser numérico o alfanumérico.
- **Nombre**: Nombre descriptivo.
- **Precondiciones**: Lo que se debe tener preparado para realizar la prueba. (Un archivo, la ejecución de otros casos de prueba...)
- **Pasos**: Define las acciones de usuario, tal como introducir nombre o darle al botón de enviar.
- **Dato de prueba**:Datos a usar en los pasos de la prueba. (nombre de usuario introducido o password...)
- **Resultado real**: Resultado obtenido.

Ejemplo: Se va a verificar el funcionamiento del formulario de entrada al sistema:

ID	Nombre	Precondiciones	Pasos	Dato	Resultado previsto	Resultado real
U1	EntradaValida1	Usuario **pepe** existe	introducir usuario y password	pepe 1234	OK	
U2	UsuarioNoValido	Usuario **pepito** no existe	introducir usuario y password	pepito 1234	error	
U3	PasswordNoValido	Usuario **pepe** existe contraseña 4567 no valida	introducir usuario y password	pepe 4567	error	

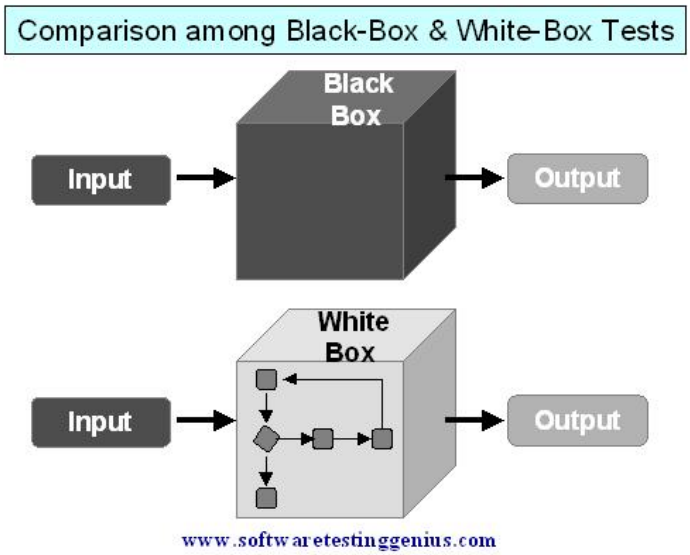
...

> **Ejercicios propuestos:**
>
> **3.1.7.1.** Diseña casos de prueba para verificar el funcionamiento de una función que devuelve un booleano indicando si el número recibido como parámetro es par.

3.1.7.1. Diseño de casos de prueba

Existen tres enfoques para el diseño de casos de prueba:

- Enfoque **estructural** o de **caja blanca**
- Enfoque **funcional** o de **caja negra**
- Enfoque **aleatorio** consiste en utilizar modelos estadísticos para las posibles entradas al programa y crear así los casos de prueba.



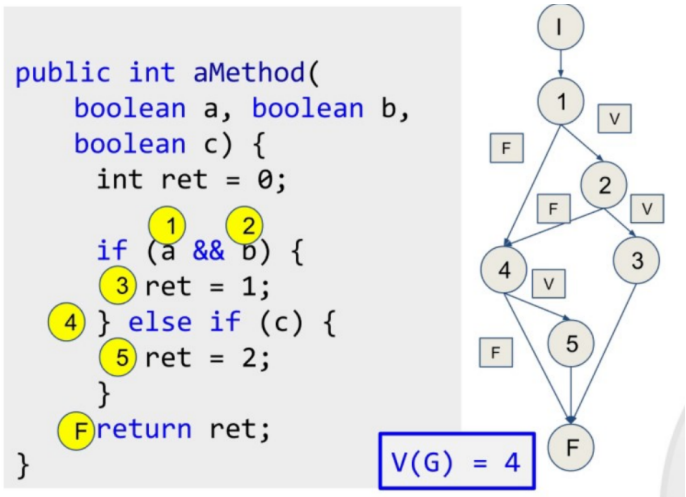
3.1.7.2. Pruebas de Caja Blanca

Las pruebas de caja blanca se centran en el funcionamiento interno del programa, observando y comprobando cómo se realiza una operación. Son las primeras pruebas que hay que realizar ya que revisan la estructura y funcionalidad interna del programa. Se pretende con ello encontrar defectos básicos de software no relacionados con la interfaz de usuario.

Existen diversos tipos de pruebas dentro de las pruebas de caja blanca.

3.1.7.2.1. Prueba del camino básico

El método del camino básico se diseñó con el objetivo de obtener una medida de la complejidad lógica para usarla como guía de un conjunto de caminos de ejecución. Este método se basa en el principio que establece que cualquier diseño procedimental se puede representar por un grafo de flujo. La complejidad ciclomática de dicho grafo establece el número de caminos independientes, cada uno de esos caminos independientes se corresponde con un nuevo conjunto de sentencias o una nueva condición.



Para probar el método anterior, se puede observar que se deben probar los caminos:

- 1,2,3,F
- 1,4,5,F
- 1,2,4,5,F
- 1,2,4,F

Que se corresponderían a pasar al método las siguientes pruebas:

- a=true, b=true, c=true
- a=false, b=*,c=true
- a=true, b=false, c=true
- a=true, b=false, c=false

3.1.7.2.2. Prueba de condiciones

Parecido al método anterior ya que evalúa los caminos posibles pero únicamente producidos por los condicionales del código.

```
public boolean isLeapYear(int year)
{
    boolean result=false;
    if(year%4==0)
    {
        result=true;

        if(year%100==0)
        {
            result=false;

            if(year%400==0)
            {
                result=true;
            }
        }
    }
    return result;

}
```

Condiciones:

- if(year%4==0) : **C1**
- if(year%100==0): **C2**
- if(year%400==0): **C3**

A partir de estas condiciones se construyen tablas de verdad para verificar todas las opciones de estas condiciones.

N	C1	C2	C3	Resultado
1	true	true	true	true
2	true	true	false	false
3	true	false	true	true

N	C1	C2	C3	Resultado
4	true	false	false	true
5	false	true	true	false
6	false	true	false	false
7	false	false	true	false
8	false	false	false	false

Como se puede observar en la tabla de verdad anterior de los casos 3 y 4 sólo haría falta probar uno de ellos ya que la condición C2 a false ***cortocircuita*** la C3 de forma que el resultado será true independiente de el resultado de la C3. Y en los casos del 5 al 8 nos encontraríamos lo mismo con la condición C1 que ***cortocircuita*** el resto de condiciones ya que si la C1 es false el resultado será false independientemente de las otras condiciones. Por tanto los pruebas necesarias para esta función serían:

N	C1	C2	C3	Resultado
1	true	true	true	true
2	true	true	false	false
3	true	false	true	true
4	false	true	true	false

3.1.7.2.2. Prueba de bucles

Esta prueba evalúa las posibilidades de los bucles. Se debe evaluar el comportamiento de dicho bucle con n iteraciones si:

- El flujo del programa no entra nunca.
- Pasa una única vez
- Pasa 2 veces por el bucle
- Pasa m veces siendo m < n
- Hace n-1 y n+2 iteraciones.

Para los bucles anidados se deben comenzar las pruebas con los más internos e ir subiendo de nivel.

3.1.7.3. Pruebas de Caja Negra

Estas pruebas se centran en la entrada y salida de la aplicación a probar.

3.1.7.3.1. Partición equivalente

Consiste en dividir en grupos de datos (clases de equivalencia) las posibles entradas a la aplicación. Algunas serán entradas válidas y otras inválidas. Se deben diseñar casos de prueba para probar todas las clases válidas y las inválidas.

Por ejemplo si tenemos un método en una clase para sumar una venta que recibe como datos un concepto (cadena),cantidad (entero) y precio(double).

Las posibles clases de equivalencia serían:

Condición de entrada	clase válida	clase inválida

Condición de entrada	clase válida	clase inválida
concepto de venta: cadena no vacía que comience por letra	cadena=letra+*	1)cadena vacía 2)cadena empiece por número 3)cadena empiece por carácter especial
cantidad:entero distinto de 0	cantidad>0 o cantidad<0	cantidad=0 o cantidad no es un número
precio: entero mayor que 0	precio>=0	precio <0 o no es un número

Una vez establecidas las clases de equivalencia se pasaría a diseñar los casos de prueba:

ID	Nombre	Precondiciones	Pasos	Dato	Resultado previsto	Resultado real
U1	Valido	existe objeto *SalesList*	pasar clases válidas de concepto, cantidad y precio	concepto="tornillo", cantidad=2,precio=2	O SalesList con un elemento más	
U2	conceptoNoValido1	existe objeto *SalesList*	pasar cadena vacía	concepto="", cantidad=2 y precio=2	-1 y no se añade nuevo elemento	
U3	conceptoNoValido2	existe objeto *SalesList*	pasar cadena empieza por número	concepto="2tornillo", cantidad=2, precio=2	-1 y no se añade nuevo elemento	
U4	conceptoNoValido3	existe objeto *SalesList*	pasar cadena empieza caracter especial	concepto="@tornillo", cantidad=2, precio=2	-1 y no se añade nuevo elemento	
U5	CantidadValida	existe objeto *SalesList*	pasar concepto="tornillo", cantidad<0,precio=2	concepto="tornillo", cantidad=-2,precio=2	0 y añade elemento	
U6	CantidadNoValida	existe objeto *SalesList*	pasar concepto="tornillo", cantidad=0,precio=2	concepto="tornillo", cantidad=0,precio=2	-1 y no se añade nuevo elemento	

...

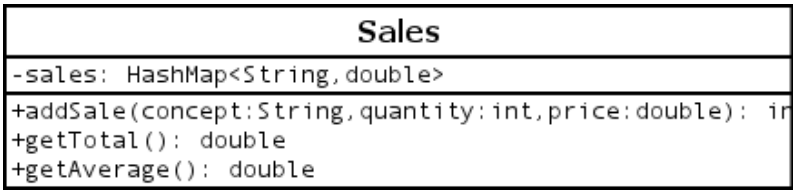
Ejercicios propuestos:

3.1.7.1. Se te encarga el desarrollo y las pruebas de una clase llamada *SalesList* cuyos datos son una HashMap<String,double>. La cadena es la descripción del producto y el número son los ingresos realizadas en ventas de ese producto. Debe tener los siguientes métodos:

- addSale(String ***concept***, int ***quantity***,double ***price***) : Incorpora un nuevo dato a la HashMap con el concepto de la cadena proporcionada como parámetro, y como ingreso el resultado de multiplicar el precio por la cantidad.

Devolverá un 0 si todo va bien y -1 si hay algún error. No se podrán incorporar ventas con cantidad 0 o precio 0 aunque sí con cantidad negativa pero no precio negativo.

- getTotal() :devolverá la suma total de todos los ingresos del HashMap.
- getAverage() : devolverá la media de todos los ingresos.



Diseña posibles casos de prueba para realizar las pruebas de los distintos métodos de esta clase. Del método *addSale* sólo debes completar la tabla anterior. De los métodos *getTotal* y *getAverage* sólo debes establecer las precondiciones para obtener el resultado deseado ya que no reciben parámetros.

3.1.7.3.2. Análisis de valores límite

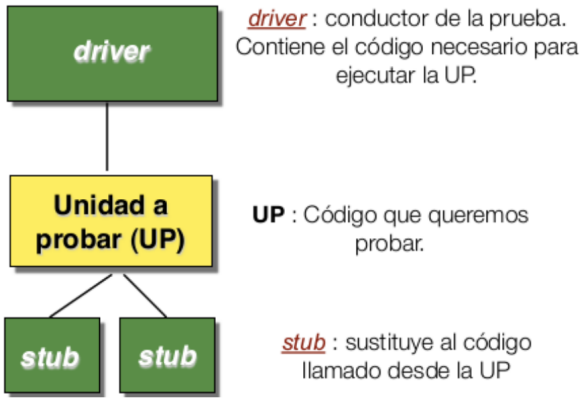
Para diseñar los casos de prueba se tienen en cuenta las condiciones de entrada y de salida:

- Si la condición de entrada es un rango, se diseñan casos de prueba para los extremos del rango.
- Si la condición de entrada especifica un número finito y consecutivo de valores, se escribirán casos para los números máximo, mínimo, uno más del máximo y uno menos del mínimo.
- Lo mismo para las reglas de salida.

3.1.8. Automatización

Las pruebas deberían poder automatizarse. Los test pueden repetirse cada vez que el código es modificado (pruebas de regresión)o se puede ampliar la extensión de las pruebas (se puede probar con un número mayor de datos de prueba). Para poder realizar las pruebas de forma automática necesitaremos:

1. **Driver**: código que automatiza la ejecución del conjunto de pruebas:
    1. Prepara los datos de pruebas.
    2. Realiza las llamadas a las unidades a probar (UP).
    3. Almacena los resultados y comprueba que los resultados son correctos.
    4. Genera un informe de pruebas
2. **Stub**: código que simula parte del programa llamado por la UP
    1. Simula el código al que llama la UP
    2. Devuelve el resultado a la UP.



3.1.8.1. JUnit

Una vez diseñados los casos de prueba, pasamos a probar la aplicación. La herramienta de automatización, en este caso JUnit, nos presentará un informe con los resultados de la prueba. En función de los resultados, deberemos o no, modificar el código.

Por ejemplo, tenemos una clase Persona que queremos probar. Esta clase se encuentra en un paquete llamado *persona.types*.

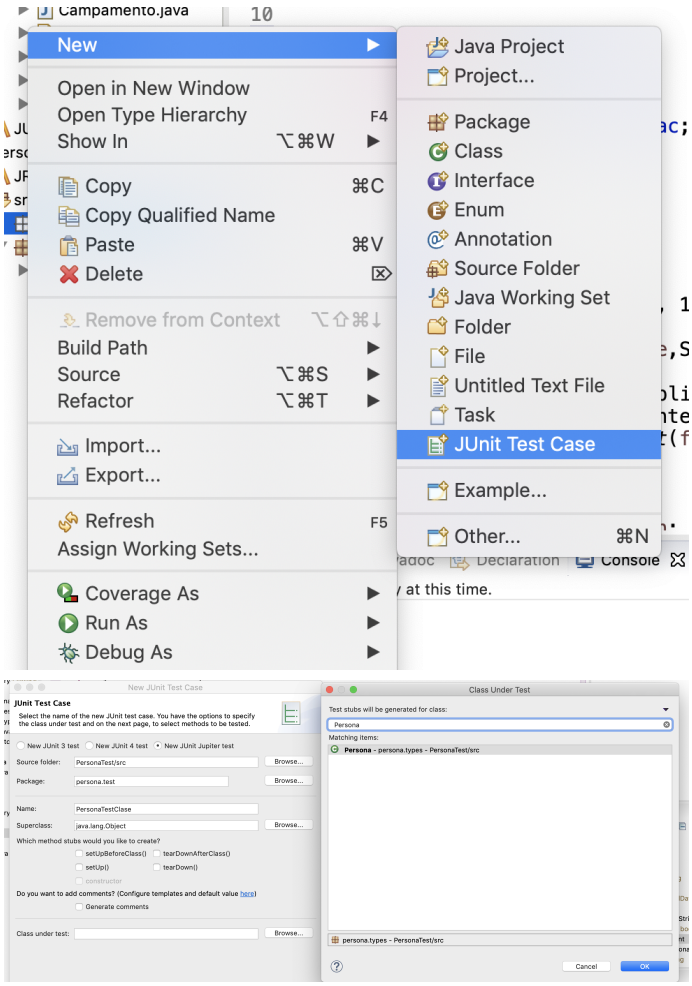
```
package persona.types;

import java.time.LocalDate;
import java.time.Period;

public class Persona implements Comparable<Persona>
{

    protected String nombre;
    protected String dni;
    protected String direccion;
    protected String telefono;
    protected  LocalDate fechaNac;

    public Persona()
    {
        nombre="";
        dni="";
        direccion="";
        telefono="";
        fechaNac=LocalDate.of(1, 1, 1);
    }

    public Persona(String nombre,String dni,String direccion,String telefono,String
fecha)
    {
        setFechaNac(fecha);
        this.nombre=nombre;
        this.dni=dni;
        this.direccion=direccion;
        this.telefono=telefono;
    }

    public String getNombre() {
        return nombre;
    }

    public void setNombre(String nombre) {
        this.nombre = nombre;
    }

    public String getDni() {
        return dni;
    }
```

```
    public void setDni(String dni) {
        this.dni = dni;
    }

    public String getDireccion() {
        return direccion;
    }

    public void setDireccion(String direccion) {
        this.direccion = direccion;
    }

    public String getTelefono() {
        return telefono;
    }

    public void setTelefono(String telefono) {
        this.telefono = telefono;
    }

    public String getFechaNac()
    {
        int day= fechaNac.getDayOfMonth();
        int month= fechaNac.getMonthValue();
        int year=fechaNac.getYear();
        return day+"/"+month+"/"+year;
    }

    public void setFechaNac(String fecha)
    {
        String[] fechas=fecha.split("/");
        if (fechas.length <3)
        {
            fechaNac=LocalDate.of(1, 1, 1);
        }
        else
        {
            fechaNac=LocalDate.of(Integer.parseInt(fechas[2]),
                Integer.parseInt(fechas[1]),Integer.parseInt(fechas[0]));
        }
    }

    @Override
    public boolean equals(Object p)
    {
        return (this.dni.equals(((Persona)p).dni));
    }

    public int calculaEdad()
    {
        LocalDate ahora =LocalDate.now();
        Period periodo=fechaNac.until(ahora);
        return periodo.getYears();
    }

    @Override
```

```
    public int compareTo(Persona p)
    {
        if(this.calculaEdad()>(p.calculaEdad()))
            return 1;
        else if (this.calculaEdad()<(p.calculaEdad()))
            return -1;
        else return 0;
    }

    public String ToString()
    {
        return nombre+" "+dni+" "+getFechaNac();
    }
}
```

Lo primero que deberíamos hacer sería crear un nuevo paquete en nuestro proyecto para las pruebas llamado ***persona.test***.

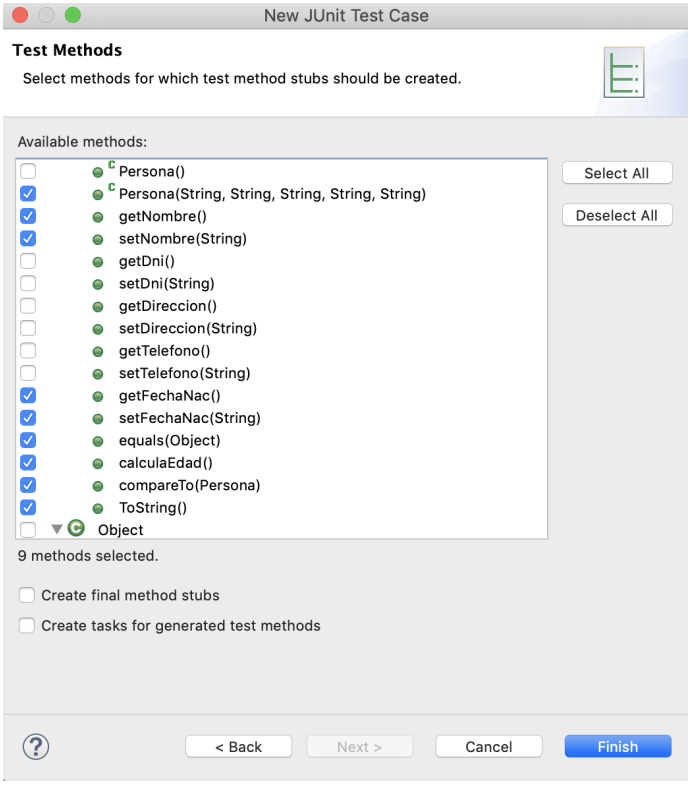
Nos situaríamos en eclipse sobre el paquete o el proyecto y con botón derecho en ***New*** le daríamos a JUnit test Case



En la siguiente ventana elegimos el nombre de la clase que vamos a generar con las pruebas, la versión de JUnit a utilizar (nosotros utilizaremos la última que es la 5 y que se corresponde con JUnit jupiter en la pantalla) y en el cuadro de texto del final debemos elegir la clase que vamos a probar dándole al botón de browse.

En este caso elegimos la clase Persona. Al darle a siguiente nos saldrán todos los métodos de nuestra clase y podremos elegir aquellos que queremos probar y nos generará automáticamente los métodos de prueba a rellenar para cada uno de

los métodos elegidos.



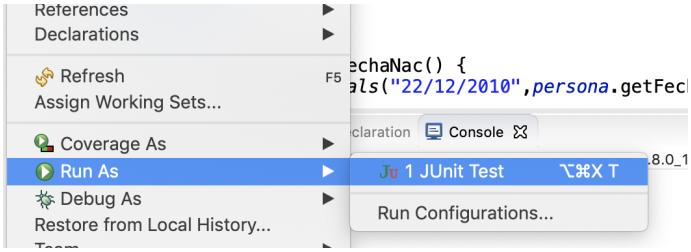
Una vez generado el fuente comenzaremos a rellenar esos métodos. Por ejemplo para probar el constructor:

```
@Test
void testPersonaStringStringStringStringString()
{
    Persona personaPrueba=new Persona("Prueba2","1111112K",
            "Calle Prueba,2","9222239","22/11/2009");
    assertEquals("Prueba2", personaPrueba.getNombre());
    assertEquals("1111112K",personaPrueba.getDni());
    assertEquals("Calle Prueba,2",personaPrueba.getDireccion());
    assertEquals("9222239",personaPrueba.getTelefono());
    assertEquals("22/11/2009",personaPrueba.getFechaNac());
}
```

La anotación *@Test* indica que el método de a continuación es una prueba. La cabecera es la que ha generado JUnit automáticamente.

En este método creamos una Persona con el constructor, que es lo que queremos probar y a continuación comprobamos si los datos se han creado correctamente. Utilizamos el método assertEquals(resultado esperado, resultado obtenido) que evalúa si el resultado que se esperaba ha sido el obtenido efectivamente. Este método está en *org.junit.jupiter.api.Assertions* junto con otros.

Para ejecutar esta prueba lo hacemos como cualquier otro proyecto en la opción de *Run as*



Saldrá entonces en el panel de JUnit el resultado de todas las pruebas si han ido bien (en verde) o si han fallado (en rojo) y el tiempo que han tardado lo que nos permite medir también la eficiencia del código que hemos construido.

Cuando necesitamos tener algunos datos previos a la prueba o establecer algunas condiciones podemos incorporarlas en un método con la anotación @BeforeEach o @BeforeAll. La primera afectará únicamente a la prueba a la que precede y la segunda a todas las pruebas de la clase. También existen las anotaciones @AfterEach y @AfterAll para cerrar o liberar recursos que hayamos podido utilizar.

Veamos otro ejemplo donde vamos a tener una instancia de Persona ya inicializada en al anotación @BeforeAll:

```
@BeforeAll
static void setUp()
{
    persona=new Persona("Prueba1","1111111K",
            "Calle Prueba,1","9568439","22/12/2010");
}
```

```
@Test
void testEqualsObject()
{
    Persona personaPrueba=new Persona("Prueba2","1111112K",
            "Calle Prueba,2","9222239","22/11/2009");
    personaPrueba.setDni(persona.getDni());
    assertTrue(persona.equals(personaPrueba));
    personaPrueba.setDni("222222");
    assertFalse(persona.equals(personaPrueba));
}
```

El método *assertTrue* evalúa un booleano, si es True la prueba se considerará válida y si no será un fallo.

En esta prueba hemos comprobado que la sobrecarga del método ***equals*** que hemos realizado en la clase Persona funciona correctamente, ya que igualábamos por DNI únicamente.

Como se puede observar hemos probado dos casos. En estas pruebas se debería incluir ***asserts*** para cubrir todos los casos de prueba que se hayan diseñado previamente. En este caso hemos comprobado el comportamiento de nuestro ***equals*** frente a 2 DNI iguales o diferentes.

> **Ejercicios propuestos:**
>
> **3.1.8.1.** Realiza las pruebas de los métodos que faltarían del ejemplo anterior según las que aparecen marcadas en la imagen de los métodos elegidos a probar.

3.1.8.2. Implementa una clase llamada *Access* con un método boolean *ValidUser(String user,String pass)* que devuelva true si el usuario es válido y false si no lo es. Para que sea un usuario válido deberá cumplir las siguientes condiciones:

- El usuario debe empezar por letra.
- El usuario debe tener una longitud mínima de 7 caracteres y máxima de 10.
- La contraseña debe tener una longitud mínima de 10 y contener alguna letra y algún número.

Realiza los casos de prueba utilizando el método de partición equivalente y posteriormente implementa dichas pruebas con JUnit.

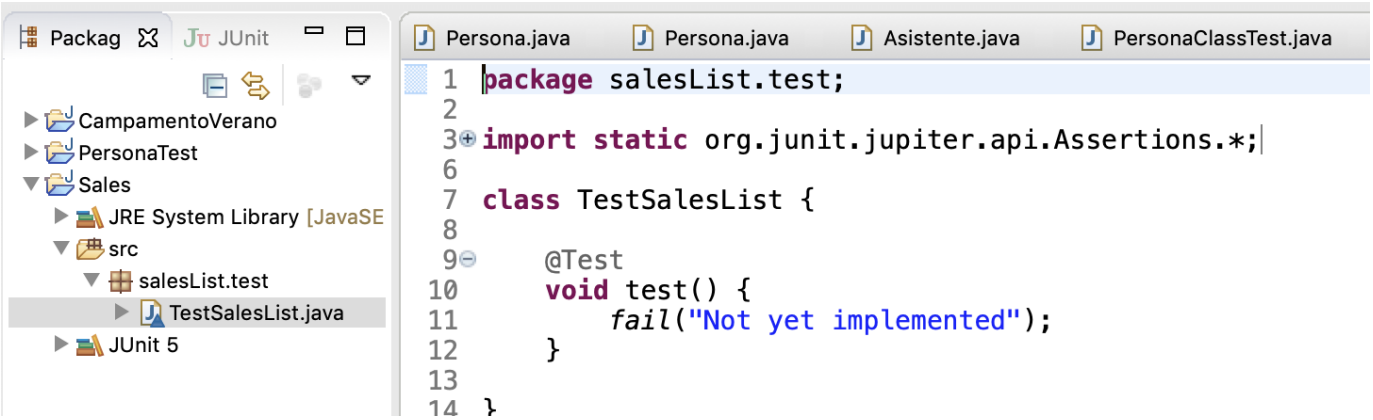
3.1.8.3. Añade a esta clase *Access* el método *boolean register(String user, String pass)* que realiza el registro. El registro consiste en añadir a un mapa el usuario y contraseña que se han pasado como parámetro. Tenemos un máximo de 10 usuarios de forma que si hemos superado ese máximo el método devolverá false y no lo añadirá al mapa y si hemos hecho el registro correctamente devolverá true. Realiza la prueba en JUnit de este método. Para poder comprobar lo de los 10 usuarios necesitaremos implementar un método con la anotación *@BeforeEach* que introduzca 10 usuarios y así cuando realize la prueba deberá devolver false.

3.1.9. Desarrollo dirigido por las pruebas (TDD)

Como ya vimos en el primer bloque del curso, existe la técnica de diseñar primero las pruebas e ir implementando nuestra aplicación según el resultado de dichas pruebas. Un ejemplo de este tipo de técnica lo podemos ver con el ejercicio **3.1.7.1.** De este ejercicios ya tenemos diseñadas la pruebas. Ahora vamos a implementarlas. Por ejemplo con la U1

ID	Nombre	Precondiciones	Pasos	Dato	Resultado previsto	Resultado real
U1	Valido	existe objeto *SalesList*	pasar clases válidas de concepto, cantidad y precio	concepto="tornillo", cantidad=2,precio=2	0 nuevo elemento añadido a la lista de ventas	

Primero creamos el proyecto y el package correspondiente a las pruebas y la prueba (esta vez no lo asociamos a ninguna clase porque todavía no existe ninguna clase):



Ahora comenzamos a implementar la prueba U1. Precondiciones que exista un objeto SalesList:

```
class TestSalesList {

   static SalesList salesPrueba;
```

```
    @BeforeAll
    static void setUp()
    {
        salesPrueba=new SalesList();
    }
```

Aquí nos sale un error de compilación ya que la clase SalesList no existe. Así que para corregir ese error creamos la clase SalesList.

```
package sales.types;

public class SalesList {

}
```

Ahora ya no sale el error de compilación en las pruebas, así que continuamos. Creamos la prueba para el método addSales que es el correspondiente a los casos de prueba que estamos implementando.

```
@Test
void addSalesTest() {
    assertEquals(0,salesPrueba.addSales("tornillo",2,2));
    }
```

Ahora sale error de compilación porque el método addSales no existe en la clase SalesList. Así que para corregir este error debemos crear el método. Para que pase la prueba debe recibir 3 parámetros y devolver un 0.

```
public int addSales(String concept,int quantity,double price)
{
    return 0;
}
```

```
@Test
void addSalesTest() {
    assertEquals(0,salesPrueba.addSales("tornillo",2,2));
    assertEquals(4,salesPrueba.getSale("tornillo"));
}
```

Ahora probamos con la segunda línea, el otro resultado esperado que es tener la línea introducida por parámetros como un elemento más de la lista de SalesList. Primer problema, el método ***getSale*** que sería el adecuado para consultar una línea de ventas, no está implementado, por lo que da error de compilación. Implementamos el método:

```
public double getSale(String concept)
{
```

```
        return (double)myList.get(concept);
}
```

Ahora el problema que nos encontramos es que no existe la lista de ventas todavía. Así que la creamos:

```
public class SalesList {
    Map myList;
        public SalesList()
        {
            myList=new HashMap<String,Double>();
        }
```

Ahora al ejecutar esta nueva prueba sale fallida con el error *NullPointerException*. Esto es debido obviamente porque el método addSale no añadió nada. Así que debemos añadir al método addSale la parte de añadir el dato correctamente.

```
public int addSales(String concept,int quantity,double price)
{
    myList.put(concept,(double)(quantity*price));
        return 0;
}
```

Con este último cambio la prueba sale bien y el primer caso de prueba estaría listo.

Pasamos al siguiente:

ID	Nombre	Precondiciones	Pasos	Dato	Resultado previsto	Resultado real
U2	conceptoNoValido1	existe objeto *SalesList*	pasar cadena vacía	concepto="", cantidad=2 y precio=2	-1 y no se añade nuevo elemento	

```
void addSalesTest()
{
    assertEquals(0,salesPrueba.addSales("tornillo",2,2));
    assertEquals(4,salesPrueba.getSale("tornillo"));
    assertEquals(-1,salesPrueba.addSales("", 2, 2));
}
```

La última evaluación falla ya que nuestro método addSale devuelve un 0 y debería devolver un -1 ya que el concepto introducido no es válido. Realizamos los cambios necesarios para que la prueba funcione.

```
public int addSales(String concepto,int cantidad,double precio)
{
    int resultado=0;

    if (concepto.isEmpty())
```

```
        {
            resultado=-1;
        }
        else
        {
            myList.put(concepto,(double)(cantidad*precio));
        }
        return resultado;
    }
```

Y así hasta completar todos los casos de pruebas.

> **Ejercicios propuestos:**
>
> **3.1.8.1.** Termina de implementar la clase SalesList utilizando TDD.

3.1.10. Más información

- https://www.slideshare.net/aracelij/pruebas-de-software/
- https://www.ecured.cu/Pruebasdesoftware
- https://www.youtube.com/watch?v=goaZTAzsLMk
- http://www.jtech.ua.es/j2ee/publico/lja-2012-13/sesion04-apuntes.html
- https://junit.org/junit5/docs/current/user-guide/