

by Mari Chelo Rubio

Development Environments

Block 3

Unit 5: Software testing. Test cases. Unit tests. Test automation

3.5.1. Introduction to software testing

As we saw in the 1st block of this module, one of the stages in the software development process is testing. And it's a really important stage. In this [video](#) you can see how important is to set up an appropriate and efficient test stage.

Software tests consist in the dynamic verification of the behavior of a program, with a properly selected set of test cases. Tests are performed in order to find possible bugs in the implementation, quality or usability of a given software.

3.5.1.1. Targets

The main targets of software testing are:

- Detect software failures or bugs, and make sure that every previously detected bug has been fixed.
- Verify the appropriate integration of the components.
- Verify that every requirement has been implemented.

3.5.1.2. Principles of software testing

- Test can help us find bugs, but not their absence.
- The most difficult part of the testing process is to decide when to stop.
- Try to avoid test cases that are not previously planned, not reusable and/or trivial, unless the program is really easy to test.
- Test cases must be written for both valid and not valid or unexpected input.
- The number of bugs to be found is directly proportional to the number of bugs already found. In other words, the more bugs we have detected with our test cases, the more bugs are waiting for us.

3.5.1.3. Stages

The stages of every testing process are:

1. Select what the test must detect, its main target.
2. Decide which kind of test is going to be performed and what kind of elements do we need to do it.
3. Implement the test cases. A *test case* is a set of data or input conditions that will be used in order to reveal something about the program, or the attribute(s) that we are checking.
4. Determine the expected results of the test cases and create a document with all of them.

5. Run the test cases.

3.5.1.4. Evaluating the results

Evaluating the results consists in comparing the test results with the expected results. Every difference between them means that there is a bug, and this bug is usually due to the unit or attribute that we are testing, although sometimes it can be due to the test process itself, if it hasn't been properly run.

3.5.2. Test types

There are different types of tests. Let's see them from lower to higher level.

3.5.2.1. Unit tests

Unit tests check the appropriate behavior of one code unit, such as a class. They are usually run by the development team, by applying the "white box" technique.

This type of test must be:

- Automatable
- Complete
- Repeatable
- Independent (a unit test must not affect the result of another one)

3.5.2.2. Integration tests

They try to find bugs in the interface connections and/or in the interaction between different components of an application. They are performed by the development team by applying some of these techniques:

- *Big bang*: it consists in integrating and testing everything at once (not recommended, unless the project is too simple)
- *Top down*: components are tested according to their hierarchy, from top to down. This way, bottom components that are not implemented or tested yet are replaced by auxiliary components that simulate their behavior. So, interfaces between components are checked in early stages of the project.
- *Bottom up*: bottom components are firstly implemented and tested, so we don't need any auxiliary component to replace them. As they are tested, then upper components can be integrated and tested as well.
- *Combined*: some parts are tested using a top down technique, and some others use a bottom up.

In this category we can also talk about **regression tests**. They consist in testing a given component whenever it has been modified, in order to find out any fault that was not previously checked. These faults can be found in either the modified code or in any other component integrated or related with the one that has been modified.

3.5.2.3. Acceptance or validation tests

These tests are performed by customers and project managers in order to check that every requirement registered in the analysis stage is being satisfied.

3.5.2.4. System tests

They must prove that the deployment of the application in its real environment is successful, and its behavior is as expected. In this test, the customer is also involved, along with the project manager or the development team.

3.5.3. Test cases

When we want to do any test over an application, we need to design the *test cases*. As we have said before, they are a set of conditions that can determine if software runs properly or not. The concrete definition according to the ISTQB (International Software Testing Qualifications Board) is: "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".

There are several formats for these test cases, but we must include the following data anyway:

- **Identifier:** it can be numeric or alphanumeric.
- **Name:** a descriptive (meaningful) name.
- **Preconditions:** what needs to be ready before starting the test, such as a given input file, the results of other test cases previously run...
- **Steps:** it defines the interaction with the user, such as entering a name, or pressing a button.
- **Test data:** data to be used in the test case, such as user name, password...
- **Expected result:** what the test must produce or output
- **Actual result:** result that we actually get when we run the test.

Example:

We want to check the behavior of a given form to log in an application. These are some of the test cases that we could specify:

ID	Name	Preconditions	Steps	Data	Expected result	Actual result
U1	ValidInput1	User pepe exists with password 1234	Type user and password	pepe 1234	OK	
U2	UserNotValid	User pepito does not exist	Type user and password	pepito 1234	error	
U3	PasswordNotValid	User pepe exists with password different than 4567	Type user and password	pepe 4567	error	
...						

Proposed exercises:

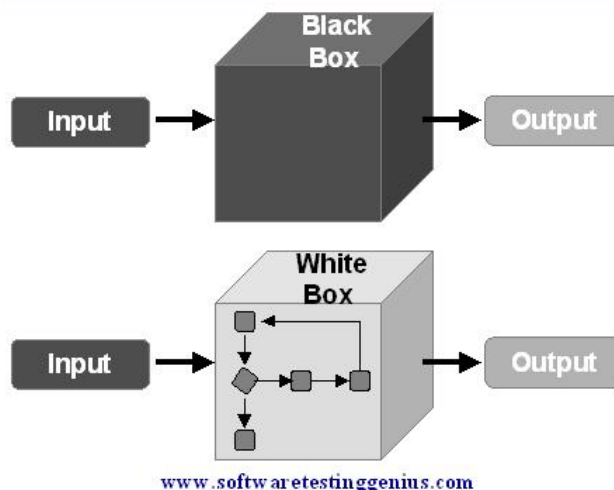
3.5.3.1. Design test cases to check the behavior of a function that returns a boolean indicating if the number specified as a parameter is even or not.

3.5.3.1. Test case design

There are three approaches to design test cases:

- **Structural** approach or **white box**, in which we focus on the inner working of the units that we are testing.
- **Functional** approach or **black box**, in which we focus on the interface of the units that we are testing, this is, their inputs and outputs, but not in their inner behavior.
- **Random** approach, which consists in using statistical models to generate the possible input for the program. This way, we generate the test cases.

Comparison among Black-Box & White-Box Tests



www.softwaretestinggenius.com

3.5.3.2. White box tests

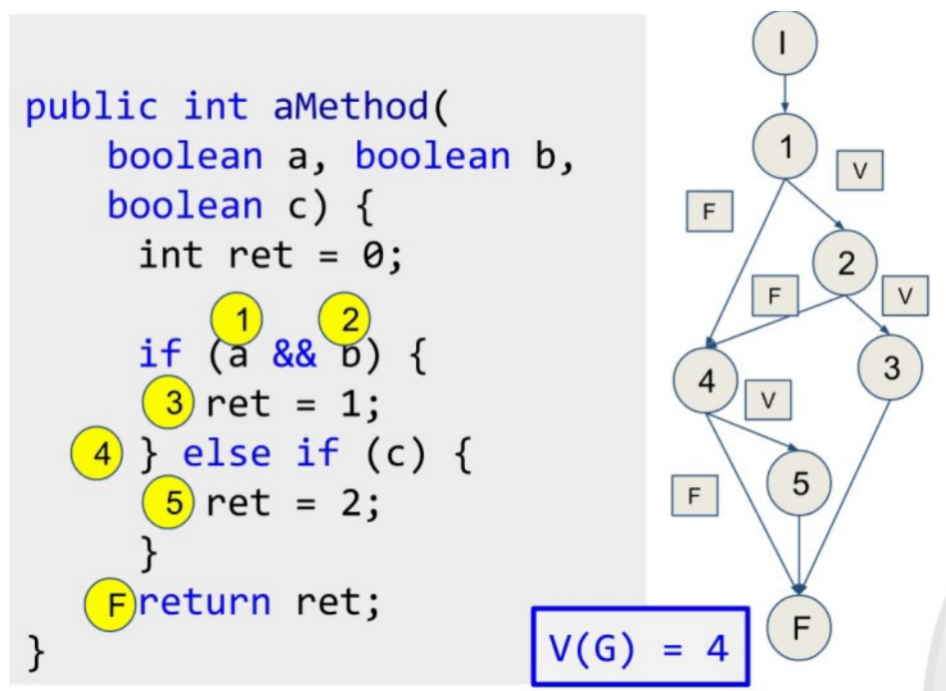
White box tests focus on the inner working of a program. This is the first tests that we must apply to a system, so that we can find basic shortcomings that are not related with user interface.

There are several types of white box tests, as we are going to see right now.

3.5.3.2.1. Basic path test

This method was focused on determining the complexity of a piece of code, so that we use this complexity to establish how many execution paths can be achieved.

It relies on a principle that sets that every procedural design can be represented as a flow graph. The cyclomatic complexity of this graph determines the number of independent paths. Each one of these paths corresponds to a new set of sentences or a new condition. Let's have a look a this short piece of code and the possible paths that can be run with it:



If we want to properly test this piece of code, we need to test the following paths:

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

Which correspond to the following tests:

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

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

Proposed exercises:

3.5.3.2. Design the white box test set for the following piece of code, using the *Basic path test* approach explained above. Determine the corresponding paths to be tested, and the test cases to test each path.

```
if (num1 > 10)
{
    if (num2 > 10)
        System.out.println("Both are greater");
    else
        System.out.println("First is greater");
} else {
    if (num2 > 10)
        System.out.println("Second is greater");
    else
        System.out.println("None is greater");
}
```

3.5.3.2.2. Condition tests

This method is similar to the previous one: it evaluates every possible path of the code, but it only focuses on the conditions of the code. Let's have a look at this example:

```
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;
}
```

Conditions:

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

From this set of conditions, we need to build the truth tables to check every possible combination:

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

As we can see, cases 3 and 4 lead to the same result regardless of the value of C3. And the same thing happens with cases 5 to 8 (condition C1 determines the final result regardless of the other two conditions). So the tests needed for this function are:

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

3.5.3.2.3. Loop tests

This test evaluates the possible paths for loops. For every loop with n iterations, we must check if:

- The loop is never iterated
- The loop is iterated only once
- The loop is iterated twice
- The loop is iterated m times, being $m < n$
- The loop performs n-1 and n-2 iterations.

If we have any nested loop, we must start exploring the inner loops and then go to the outer ones.

3.5.3.3. Black box tests

These tests focus on the input and output of the application or module to be tested. There are also some different techniques that we can apply to these tests.

3.5.3.3.1. Equivalent partition

It consists in dividing the possible inputs of the application in groups called *equivalence classes*. Some input will be valid inputs and some other will be not valid, so we must design test cases to check both valid and invalid equivalence classes.

For instance, if we have a method to determine the total amount of a sale, given the concept (string starting with letter), product amount (integer other than 0) and product price (double greater or equal than zero), the possible equivalence classes are:

Input condition	Valid class	Invalid class
Concept not empty starting with letter	string=letter+*	empty string OR string starting with number OR string starting with special character
Amount integer other than 0	amount other than 0	amount 0 OR not integer
Price double greater or equal than 0	price>=0	price<0 OR not numeric

Once we define the equivalence classes, we can design the test cases:

ID	Name	Preconditions	Steps	Data	Expected result	Actual result
U1	Valid	<i>SalesList</i> object exists	Enter valid classes for concept, amount and price	concept="screw", amount=2, price=2	0, a new element is added	
U2	NotValidConcept1	<i>SalesList</i> object exists	Enter empty string as concept	concept="", amount=2, price=2	-1, no element added	
U3	NotValidConcept2	<i>SalesList</i> object exists	Enter string starting with number	concept="2screw", cantidad=2, precio=2	-1, no element added	
U4	NotValidConcept3	<i>SalesList</i> object exists	Enter string starting with special char	concept="@screw", amount=2, price=2	-1, no element added	
U5	ValidAmount	<i>SalesList</i> object exists	Enter negative amount	concept="screw", amount=-2, price=2	0, a new element is added	
U6	NotValidAmount	<i>SalesList</i> object exists	Enter amount of 0	concept="screw", amount=0, price=2	-1, no element added	
...						

Proposed exercises:

3.5.3.3. You have been asked to implement the tests for a class called *SalesList*, whose attribute is a `HashMap<String,Double>`. The string is the product description, and the number is the total amount of sales over this product. The class has the following methods:

- `addSale(String concept, int amount, double price)`: it adds a new element to the *HashMap* with the specified concept as product description. The incomes will be calculated by multiplying the amount and the price. It will return 0 if everything is OK, and -1 if there is any error. We will not be able to add sales with amount = 0 or price < 0, but we can add sales with negative amounts (but not negative prices).
- `getTotal()`: it will return the total sum of the incomes of the *HashMap*.
- `getAverage()`: it will return the income average.

Design the possible test cases for every method of the class. Regarding `addSale` method, you just have to complete the table shown in previous example. For `getTotal` and `getAverage` methods, you just need to set the preconditions to get the desired result, since they have no parameters.

3.5.3.3.2. Analysis of limit values

In order to design the test cases, we take into account input and output conditions:

- If the input condition is a range, we must design test cases for the limits of this range.
- If the input condition is a finite and consecutive set of values, we must define the test cases for the minimum and maximum value, along with the *minimum + 1* and *maximum - 1* values.
- We must apply these same rules for the output conditions.

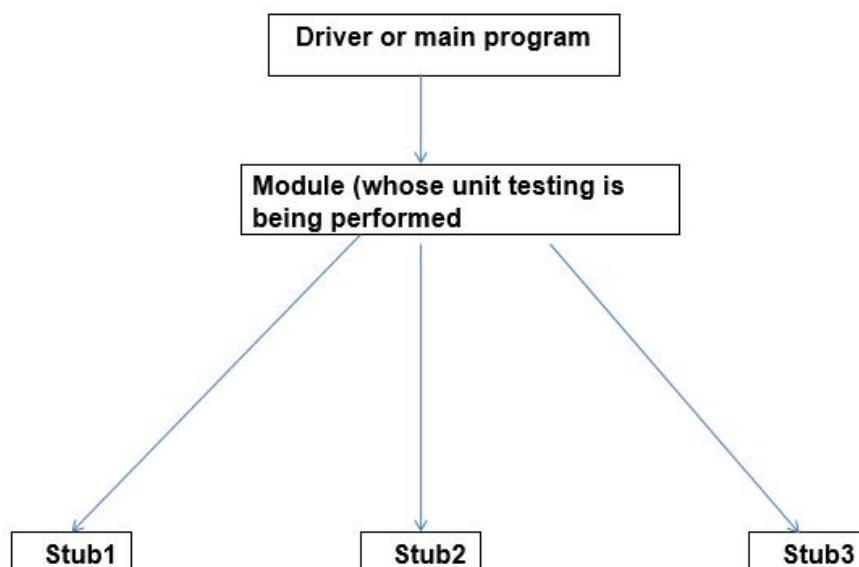
3.5.4. Test automation. JUnit

Test should be automatable, so that we can repeat them every time the code changes (regression tests) or we add new tests to the original set. In order to automate the tests, we need:

1. **Driver:** code that actually automates the test execution. It is in charge of:

1. Preparing test data.
2. Calling the units to be tested.
3. Storing the actual results and comparing them with the expected ones.
4. Generating a test report.

2. **Stub:** code that simulates a part of the program that is not being tested at a given moment. It is called by the units to be tested, so that it returns a given result for them in order to complete their task.



3.5.4.1. Introduction to JUnit. Setting up the project

Once the test cases have been designed, we start testing our application. In our case, we are going to use **JUnit**, a Java library for unit testing that also shows test reports that help us decide if we need to change the original code or not.

For instance, let's create a new Java project called `PersonTest`. Then, create a new package called `person.types` and place this `Person` class in it:

```
package person.types;

public class Person
{
    protected String name;
    protected String idCard;

    public Person()
    {
        name="";
        idCard="";
    }

    public Person(String name,String idCard)
    {
        this.name=name;
        this.idCard=idCard;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

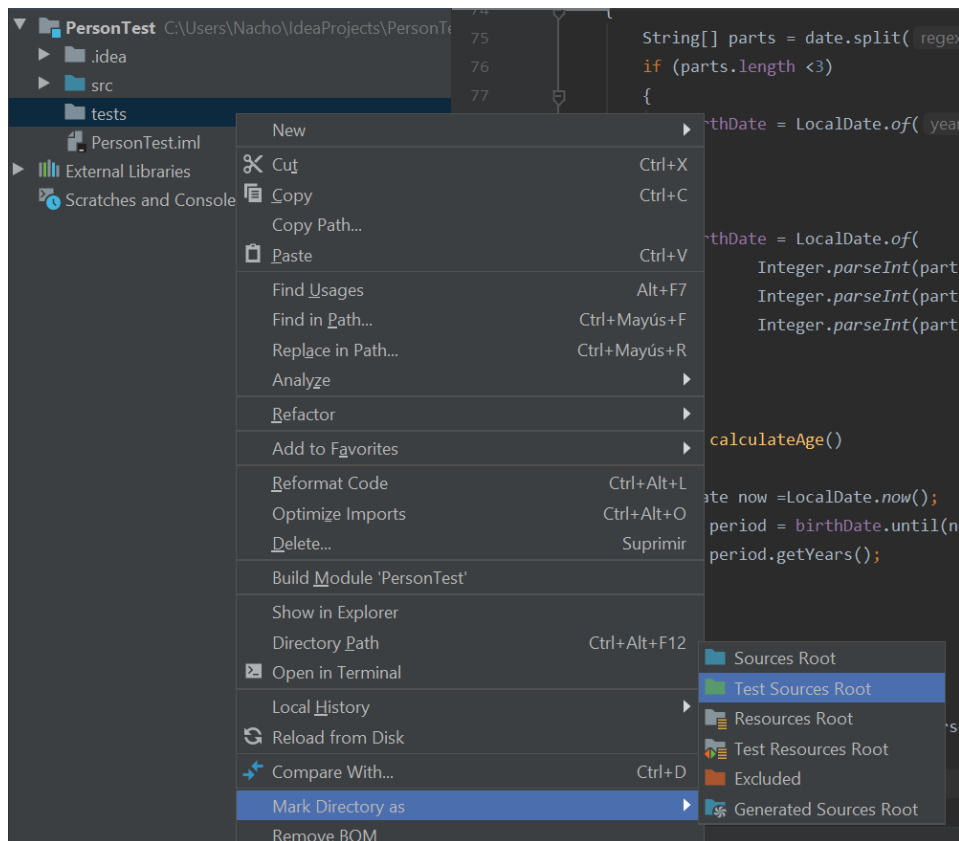
    public String getIdCard() {
        return idCard;
    }

    public void setIdCard(String idCard) {
        this.idCard = idCard;
    }

    @Override
    public boolean equals(Object p)
    {
        return (this.idCard.equals(((Person)p).idCard));
    }

    @Override
    public String toString()
    {
        return name + " " + idCard;
    }
}
```

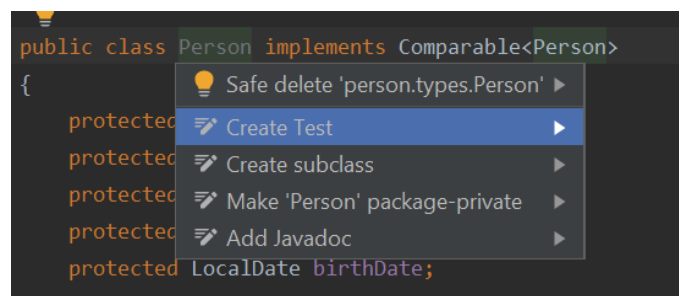
Now, we are going to learn how to create a unit test associated to this class, and define some test cases. First of all, it is recommended to **create an additional source folder to place all our tests in**, so that we don't mix them with the original source code. We can create a new directory called `tests` and right click on it, then we choose *Mark Directory as > Test Sources Root*.



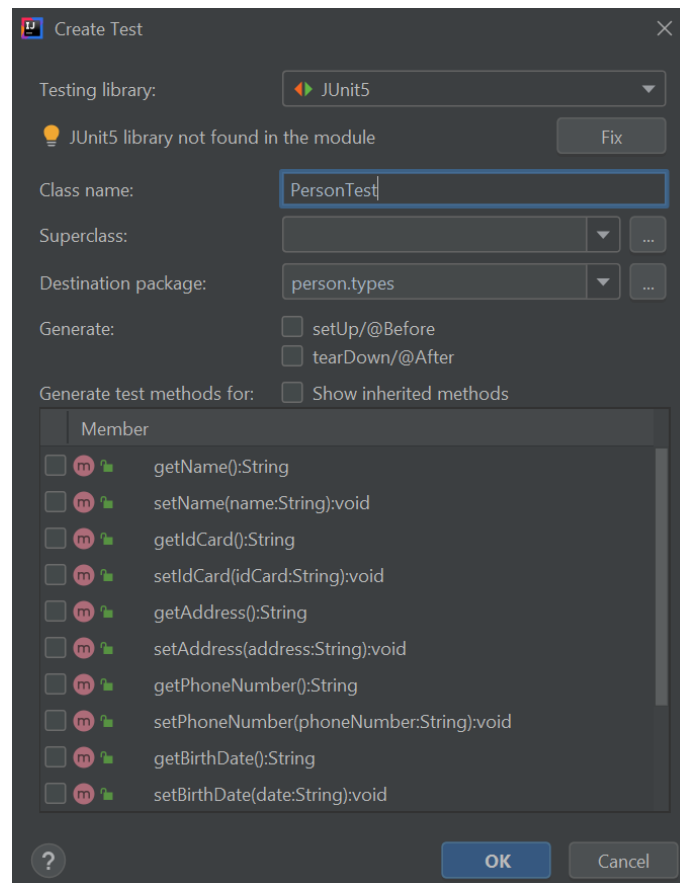
From now on, every test that we create will be automatically placed inside this source folder, with the same package name than the class that is being tested.

3.5.4.2. Creating test classes

In order to create a new unit test over `Person` class, place the cursor in the class name to be tested (`Person`, in our case) and press `Alt` + `Enter`. In the context menu, choose *Create test*.

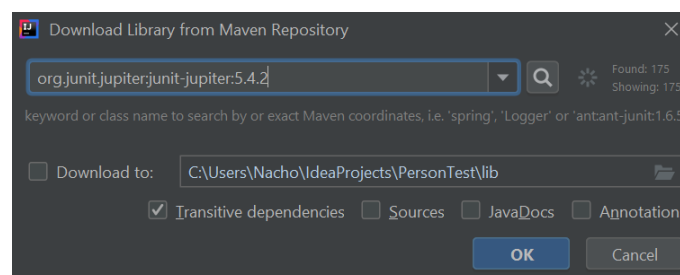


Then, a new dialog will appear to specify the contents of this unit test:



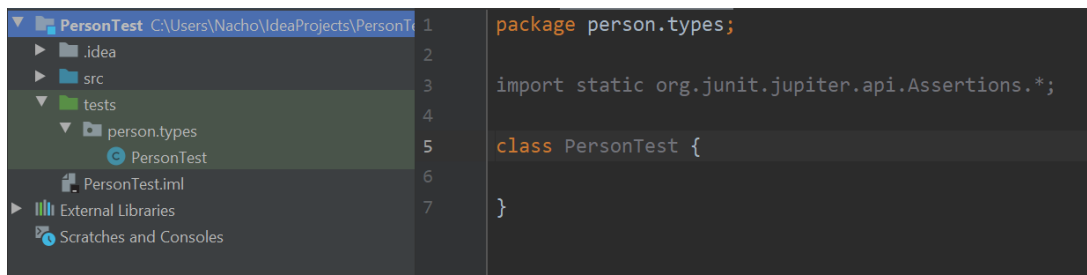
In the dialog that will be shown, we can:

- Specify JUnit version. We will use the last one, which is version 5, and it corresponds to *JUnit Jupiter*. If a message appears indicating that *JUnit 5 library not found in the module*, we can press the *Fix* button and automatically download and add the library to the project:



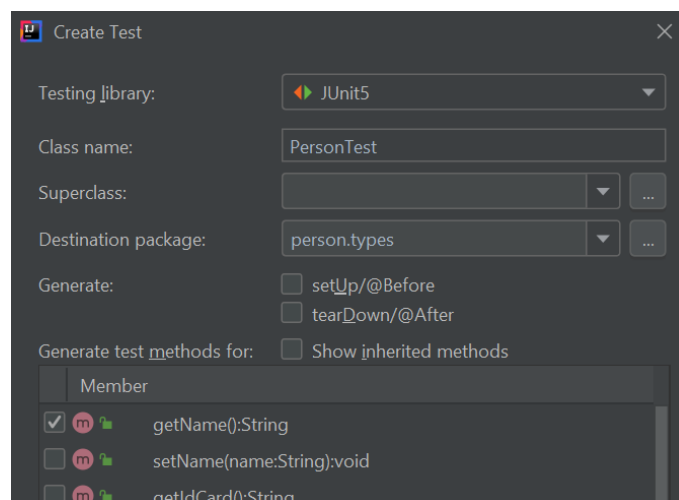
- We can also change the test class name and/or package name for this test class (although it is not recommended, nor usual)
- Finally, we can also choose which methods from the original class are going to be tested. We don't need to check any method now if we don't know which one(s) we need to test. We can add as many as we want later.

So, for now, we leave the default options of this dialog and click on the *OK* button. A new class called `PersonTest` has been created in the `person.types` package inside the `tests` source folder.



3.5.4.3. Adding test methods to the test class

Now, let's try to define a test method. For instance, let's create a test method for the `getName` method of `Person` class. To do this, we go again to `Person` class name, press `Alt` + `Enter` keys, and then choose the method(s) that we want to add to the test:



IntelliJ will ask us to confirm that we want to update existing test class, then we can proceed. Any old content of this class will be preserved, and then new method(s) will be added.

```

package person.types;

import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.*;

class PersonTest
{
    @Test
    void getName()
    {
    }
}

```

In order to test this method, we can, for instance, create a new `Person` object with a given name, and then check if this name is the one that we expect. For this checkings, we can use `assertXXXXX` methods that are available through static import of `org.junit.jupiter.api.Assertions` package:

```
package person.types;

import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.*;

class PersonTest
{
    @Test
    void getName()
    {
        Person p = new Person("James", "11223344A");
        assertEquals("James", p.getName());
    }
}
```

Let's explain this code more in depth:

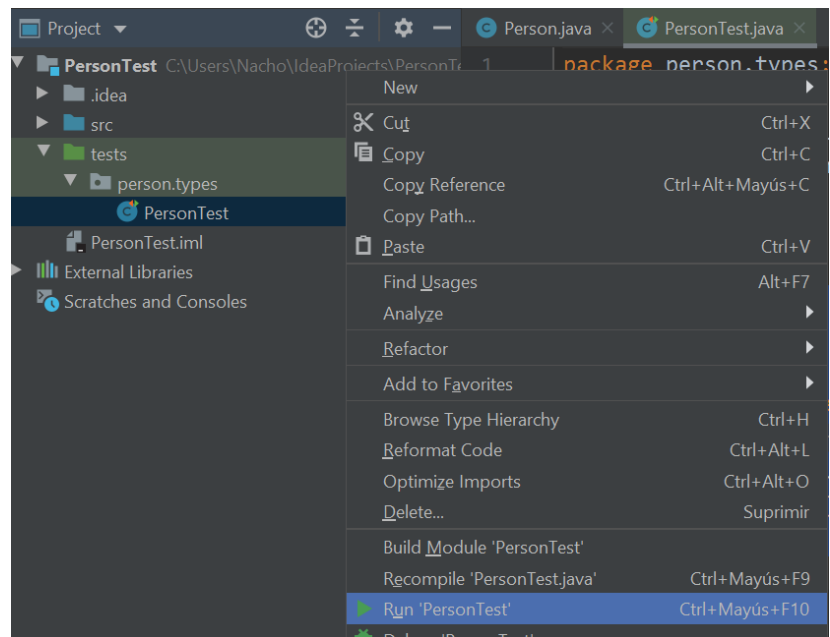
- `@Test` annotation indicates that the method above is a test. The method header has been automatically generated by JUnit.
- In this test we create a `Person` object with the (second) constructor
- Next, we check if `name` attribute is the one that we expect. We use `assertEquals` method with two arguments (expected result and actual result), to evaluate with the corresponding *getter* if the attribute has the expected value after creating the object. This method is in `org.junit.jupiter.api.Assertions` class in JUnit 5. There are also some other useful methods, such as `assertTrue`, `assertFalse` (we will see an example of these two methods later), `assertNull`, or `fail`, which can be used to automatically emit a failure according to some condition(s).

We can add as many assertions as we want in a test method. For instance, we can also check if name is not null before checking if it is "James":

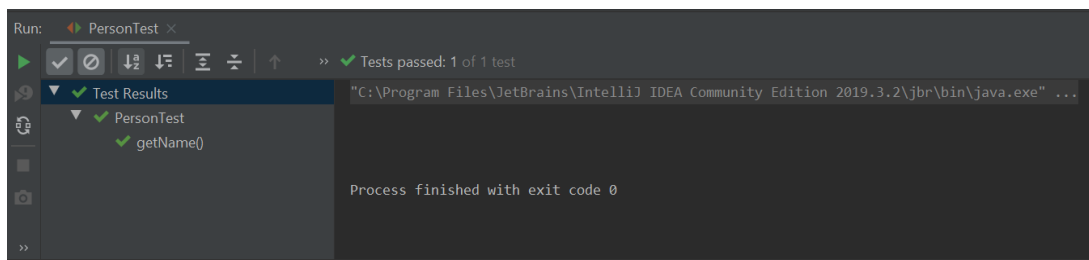
```
@Test
void getName()
{
    Person p = new Person("James", "11223344A");
    assertNotNull(p.getName());
    assertEquals("James", p.getName());
}
```

3.5.4.4. Running tests

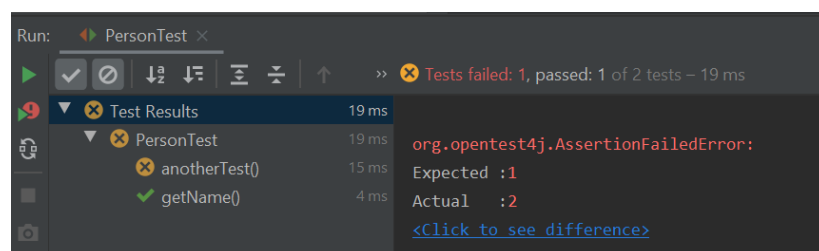
If we want to run this test class, we can right click on it in the left panel and choose *Run PersonTest*.



Then, we will see a JUnit panel with the results of every test contained in the test class:



If every test has been successful, then we will get a green icon, otherwise we will see an error icon next to each test method that has failed.



3.5.4.5. Initializations and closings.

If we need to have some previously initialized data before starting the tests, or some conditions previously established, we can use the annotation `@BeforeEach`. We can also use the annotation `@AfterEach` to close or free some resources after the tests have been performed.

There is a method called `setUp` that is usually employed to initialize data for every test case before they are launched. So we can use the annotation `@BeforeEach` with this method to initialize some data. This method can be added from the test dialog in IntelliJ, when we add new test methods to the test class. For instance, we can initialize a shared `Person` object for all the test methods, and use it in every test method that we want. For instance, here we instantiate a `Person` object in the `setUp` method, and use it in the

`getName` test method and also in the `equals` test method (we consider that two `Person` objects are the same if they have the same ID card):

```
class PersonTest
{
    Person person;

    @BeforeEach
    void setUp()
    {
        person = new Person("James", "11223344A");
    }

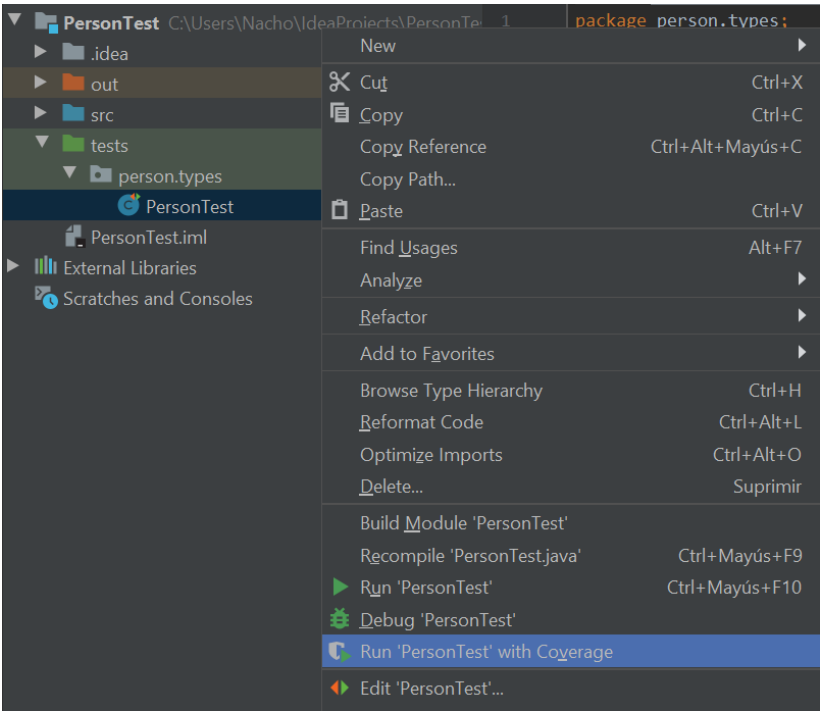
    @Test
    void getName()
    {
        assertNotNull(person.getName());
        assertEquals("James", person.getName());
    }

    @Test
    void testEquals()
    {
        Person testPerson = new Person("Test2", "1111112K");
        testPerson.setIdCard(person.getIdCard());
        assertTrue(person.equals(testPerson));
        testPerson.setIdCard("222222");
        assertFalse(person.equals(testPerson));
    }
}
```

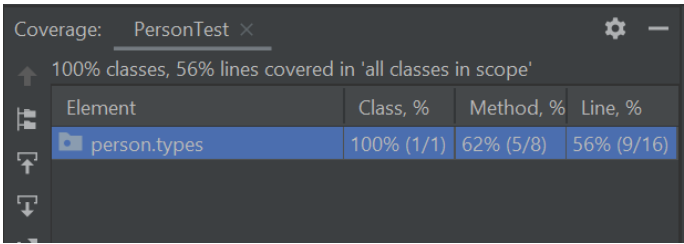
There are also some other common methods that can be used, such as `tearDown` along with `@AfterEach` annotation to free resources for every test after their execution. Again, this method can be added from the test class dialog in IntelliJ.

3.5.4.6. Checking code coverage

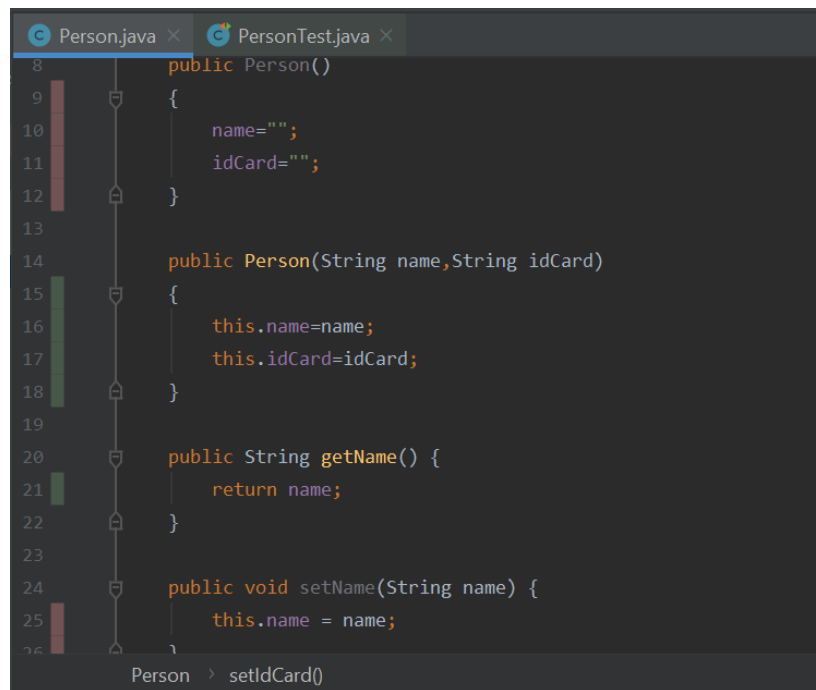
We have checked two methods so far, and we have added some different kinds of *asserts* in each one to determine if they work properly or not with different inputs. But IntelliJ can also show how much of the original code has been tested. This feature is called **code coverage**, and we can get to it by right clicking on the test source file and choosing *Run XXXXX with Coverage*, being XXXXX the class name.



Then, a new panel with some stats will appear:



According to these stats, our test covers 62% of `Person` class. If we edit this class, we can see green, vertical bars in the left margin for lines of code that are being executed, and red, vertical bars for the lines of code that no test is exploring yet.



```
8 public Person()
9 {
10     name="";
11     idCard="";
12 }
13
14 public Person(String name,String idCard)
15 {
16     this.name=name;
17     this.idCard=idCard;
18 }
19
20 public String getName() {
21     return name;
22 }
23
24 public void setName(String name) {
25     this.name = name;
26 }
```

Proposed exercises:

3.5.4.1. Implement the tests for every method of previous exercise 3.1.3.3., according to the test case design defined in that exercise. Use a project called `SalesList` for this purpose, with separate source folders for the original code and the classes.

3.5.4.2. Create a new project called `Access`. Implement a class called `Access` with a method called `validUser(String user,String pass)` that returns `true` if user is valid and `false` if it is not. In order for a user to be valid, it must meet the following conditions:

- `user` parameter must start with a letter.
- `user` parameter must have a length between 7 and 10 characters.
- Password must have a minimum length of 10 characters, and it must have at least one letter and one number.

Design the test cases using the technique of the equivalent partition, and then implement these test cases with JUnit.

3.5.4.3. Add a new method to the `Access` class: `boolean register(String user, String pass)` that will register the user. The registration will consist in adding the user to a map. There will be a maximum of 10 allowed users in the map, so that if we exceed this limit, the method will return `false`. If the registration is correct, it will return `true`. Also, if a user with the existing name already exists in the map, it will return `false`. You are also asked to implement the tests in JUnit.

3.5.5. Test Driven Development (TDD)

As we have already seen in the units of the first block of this module, there is a technique that consists in starting by the tests and then implement the application according to the results of these tests. We can apply

this technique in the project started in exercise **3.1.3.3**.. We already have a set of test cases, such as this one:

ID	Name	Preconditions	Steps	Data	Expected result	Actual result
U1	Valid	<i>SalesList</i> object exists	Enter valid classes for concept, amount and price	concept="screw", amount=2,price=2	0, a new element is added	

First of all, let's create a project called *SalesListTDD*, with no classes in it for now. We just define a new source folder called *tests*, as we did in previous exercises, and we define a package called `sales` in both *src* and *tests* folders. This time, we are going to start by creating the test class first. To do this, we right click on `tests/sales` package and choose to create a new simple class on it. We call it `SalesListTest`, and we leave it with the following appearance:

```
package sales;

public class SalesListTest
{

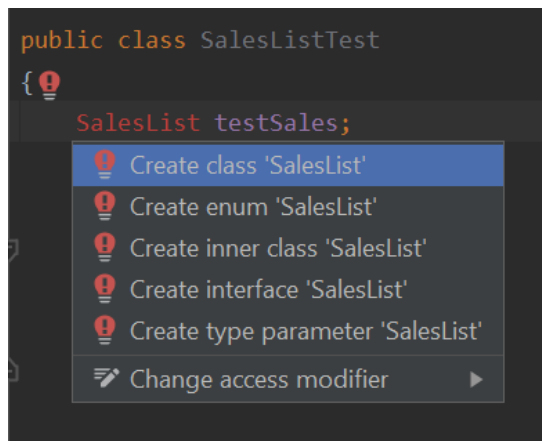
}
```

Now we start implementing the test for U1. As a precondition, we need to define a *SalesList* object, so we add this code to the test class, with the corresponding *import* for the `@BeforeEach` annotation. We may also need to add JUnit library to the project following the wizard:

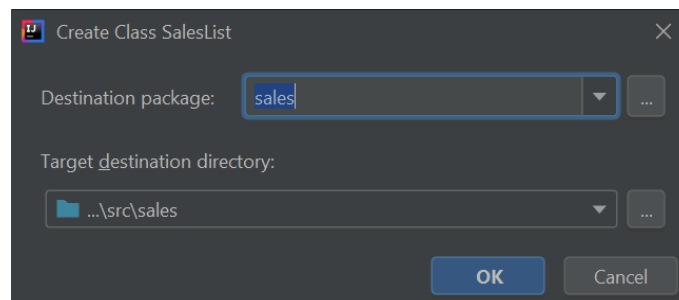
```
public class SalesListTest
{
    SalesList testSales;

    @BeforeEach
    void setUp()
    {
        testSales=new SalesList();
    }
}
```

However, we will get a compilation error, since `SalesList` class is not yet defined. To fix this error, we create the class in our source folder, inside any package (`sales` package, for instance). We can `Alt` + `Enter` over the compilation error and choose **Create class SalesList`*:



Then we choose the location of this new class (package *src/sales*):



This new class will be empty for now:

```
package sales;

public class SalesList
{
}
```

Next, we add the corresponding *import* to our `SalesListTest` class to fix the error, and then we go on. Now, we create a test for the `addSales` method in our `SalesListTest`, along with the *import* for the `@Test` annotation and the `assertEquals` method. This is how our `SalesListTest` class should look like after this step:

```
package sales;

import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.*;

public class SalesListTest
{
    SalesList testSales;

    @BeforeEach
    void setUp()
    {
        testSales=new SalesList();
    }

    @Test
    void addSales() {
        assertEquals(0,testSales.addSales("screw",2,2));
    }
}
```

Now we get an error because `addSales` method does not exist in `SalesList` class. So we define it (`Alt` + `Enter` over the compilation error to automatically create the method in `SalesList` class), and make it return a 0 so that we can pass the test.

```
package sales;

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

Now, we add this second line to our `addSales` test method, to check if we can retrieve the total amount for a given product:

```
@Test
void addSales() {
    assertEquals(0, testSales.addSales("screw", 2, 2));
    assertEquals(4, testSales.getSale("screw"), 0.00001);
}
```

The last parameter of this second line is a *delta*, the maximum difference allowed between the first two parameters (which are double) in order to consider the test valid. However, `getSale` method does not exist, so we implement it in `SalesList` class:

```
public double getSale(String concept) {
    return (double)myList.get(concept);
}
```

Now the problem is that there is no internal sales list in `SalesList` class. We define and initialize it:

```
public class SalesList {

    Map myList;

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

If we run the test now, it will fail due to a *NullPointerException* error, because `addSale` method did not add anything to the map. So we add the code to this method, so that it adds the new element properly:

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

Now the test succeeds and we are done with our first test case. Let's go with next one:

ID	Name	Preconditions	Steps	Data	Expected result	Actual result
U2	NotValidConcept1	<i>SalesList</i> object exists	Enter empty string as concept	concept="", amount=2, price=2	-1, no element added	

We add this new line at the end of our `addSales` test method:

```
void addSales()
{
    assertEquals(0, testSales.addSales("screw", 2, 2));
    assertEquals(4, testSales.getSale("screw"), 0.00001);
    assertEquals(-1, testSales.addSales("", 2, 2));
}
```

If we run the test now, it will fail because our `addSales` method always return 0, and it should return -1 if there is any error with the input parameters. In this case, the specified concept is not valid (it is an empty string). So we make these changes to the method to make it work again:

```
public int addSales(String concept, int quantity, double price) {
    int result=0;

    if (concept.isEmpty())
        result=-1;
    else
        myList.put(concept, (double)(quantity*price));
    return result;
}
```

This way, we can go on until we complete every test case.

Proposed exercises:

3.5.5.1. Add some more test cases to our SalesList project using TDD.

3.5.6. More information

- <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/>