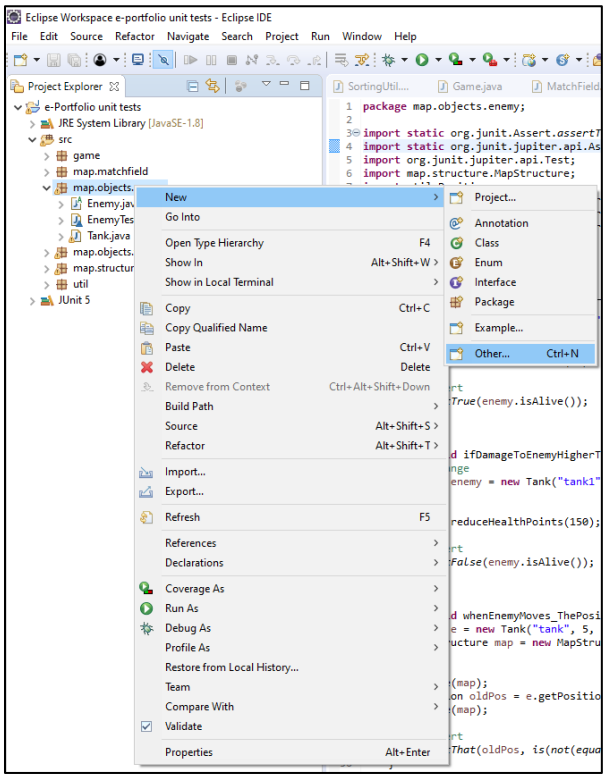
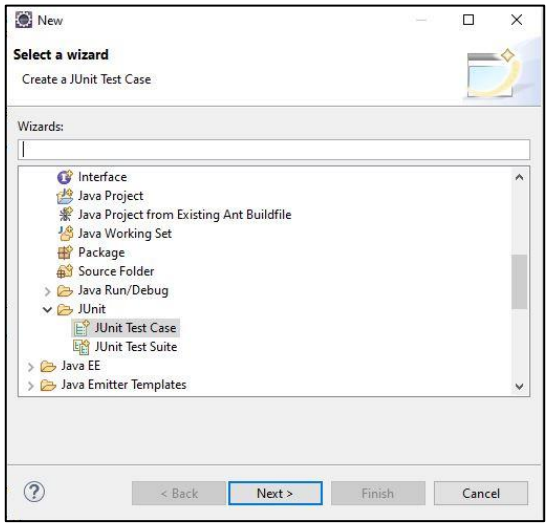
Tutorial Unit Testing in Eclipse

This tutorial is about unit testing with Junit in Eclipse. I want to show you how to create a test class and their test methods. To get some experience with JUnit and testing itself, I provided an eclipse project which is a part of the backend for the tower defense game my team is developing. I tried to write the code as good as possible for unit testing and its very good for beginners to write some own little tests.

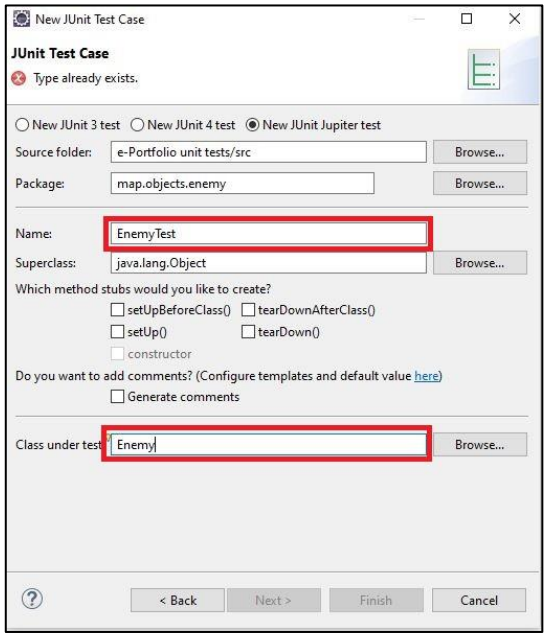
Setup a Junit Test Class:

To create a new Junit class, you right click on a package or folder where the class should be located. Then hover on New -> Other to create a new JUnit Test Case.





In the test case dialog, you write the name of the test class. Class under test is that class you want to test. With clicking on the button “finish” the class is created with a standard test method and looks like the following picture. Sometimes you have to add Junit 5 to the build path manually, to get the tests started.




```java
package map.objects.enemy;

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

class EnemyTest2 {

    @Test
    void test() {
        fail("Not yet implemented");
    }

}
```

To assign a test method you need the Annotation @Test. As I mentioned in the presentation, every unit test consists of the three parts arrangement, action and assertion. But the first thing we take a look at is the name. Our first example should test a tower object, that it is only shooting an enemy if the enemy is in range. As background DefTower and Tank are the implementations of Tower and Enemy. As this should be a tower defense game a Tower has a specific damage and range and a method to shoot an Enemy. An Enemy has a specific amount of health points and can move to a position.

Arrangement:
For our test we need a DefTower and one Tank. The tower is spawned at position (0,0) and the enemy is moved to the position (150, 150). The range of the tower is set with the constructor to 100, so there is no chance to shoot the enemy. Also the damage of the tower was set to 100 and the enemy has only 5 hp. This means one shot will kill the enemy instantly.

Action:
The action is very simple by calling the method fire of the DefTower. But the method needs a list of enemies which are spawned on the map. So we create a new ArrayList and put our tank inside. The tower will focus on the nearest enemy in the list but only if its in range. That means we want the tower not to shoot in this specific case.

Assertion:
If the healthpoints of an enemy are reduced to 0 it sets a boolean variable isAlive to false, so we can easily detect if its alive. For our test case we would expect, that the enemy is still alive. We will use a simple assertTrue for testing that.

```
@Test
void towerDontShootEnemyIfItsNotInRange() {
    //arrange
    Tower t = new DefTower("t1", 100, 100, 1, new Position(0, 0));
    Enemy e = new Tank("tank", 5, 950);
    e.moveToPosition(new Position(150, 150));

    //act
    t.fire(Arrays.asList(e));

    //assert
    assertTrue(e.isAlive());
}
```

Arrangement:
Our next test should prove that the tower shoots and kills an enemy if it is in range. Therefore, we still need our DefTower and a Tank. The Tower is set to a label “t1” with a range of 100, a damage of 100 a fire rate of 1 second and the position (0, 0). Our tank has the label “tank” and still 5 healthpoints and a speed of 950 which doesn’t matter in this test. The enemy is moved to position (50, 50) so it is in range of the tower.

Action:
Again our tower is called to shoot and gets as parameter a List which contains our new created enemy. The enemy is in range of the tower, so we expect the tower to shoot.

Assertion:
One hit of our tower will kill the enemy. So as the tower has shot we have to prove that the enemy is dead now. In this case we can use a simple assertFalse() for our boolean value isAlive from the enemy. This is the opposite of the first test we saw, so we only changed assertTrue() to assertFalse().

To show you some examples for other assertion types I used the 4 standard types which I introduced in the presentation.

- ➔ We can use asserTrue() if we change the boolean condition to isAlive==false or a negation of isAlive which would be asserTrue(!isAlive)
- ➔ assertEquals() compares the two values, which means we can compare two booleans. The assertion will be green if the both values are the same. So we assert e.isAlive() to be false.
- ➔ For asserThat() we need some hamcrest matchers, but as you can see the readability is very good. We assert that e.isAlive() is not equal to the boolean value true.
- ⇨ All assertions have the same result, but some are easier than others. While assertTrue() is very easy to write and often used, asserThat() is a lot more readable if the assertion gets more complex. Keep that in mind!

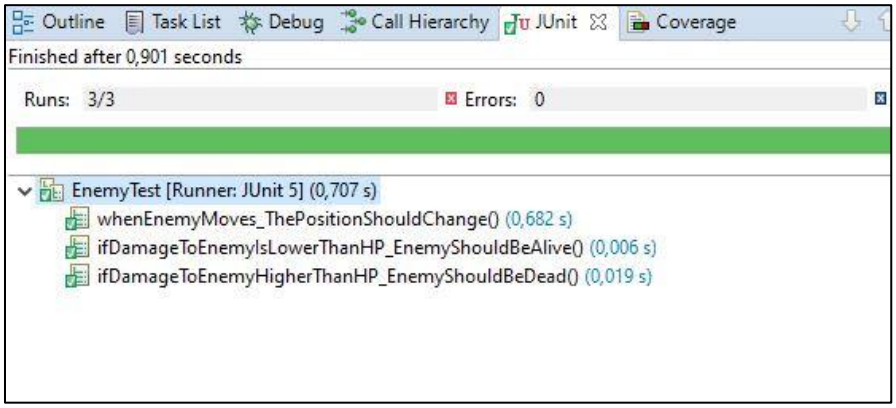
```
@Test
void towerKillsEnemyIfItsInRange() {
    //arrange
    Tower t = new DefTower("t1", 100, 100, 1, new Position(0, 0));
    Enemy e = new Tank("tank", 5, 950);
    e.moveToPosition(new Position(50, 50));

    //act
    t.fire(Arrays.asList(e));

    //assert
    assertFalse(e.isAlive());
    assertTrue(e.isAlive()==false);
    assertEquals(false, e.isAlive());
    assertThat(e.isAlive(), is(not(equalTo((true)))));
}
```

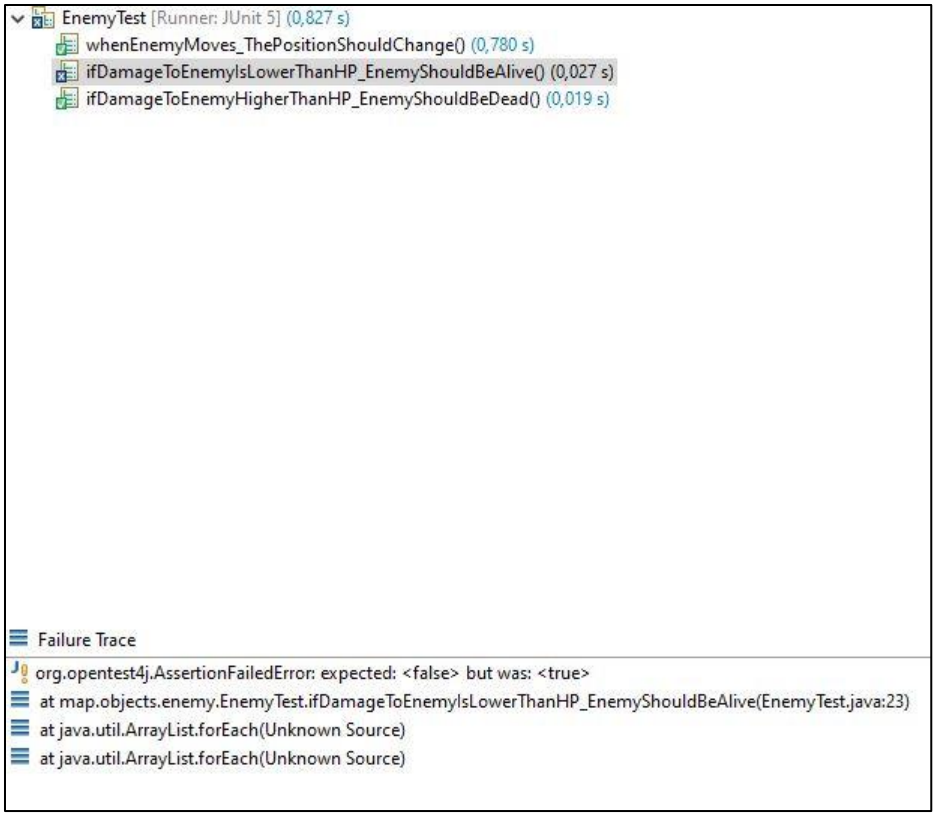
If the import of the hamcrest CoreMatchers doesn’t work automatically, try to add the imports manually, like the picture on the bottom.

```
import static org.hamcrest.CoreMatchers.is;
import static org.hamcrest.CoreMatchers.not;
import static org.hamcrest.CoreMatchers.equalTo;
```

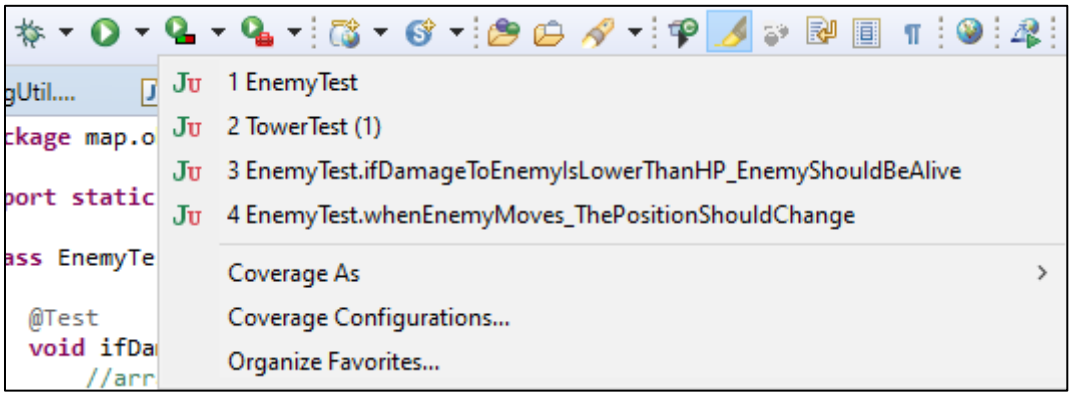
Get tests running:

To run a unit test or all test of a test class you can use the normal “run” button in eclipse. There … When the tests are finished, you get an overview of the result. In the first example all tests are green. That means no Exception was thrown while testing. In the second example we can see what happens if an assertion was wrong. The assertFalse() method got an true as parameter so it throwed an exception which leads to the error message, we can see at the bottom. “Expected false but was true”





A nice feature of eclipse is the test coverage button. To the right of the normal “run” button there is a second one, where you can run your tests. The result will be the same, but you get an overview about the test coverage in all classes. As you can see in the following pictures our EnemyTest does also test lots of other classes and their methods. The tests used more than 90 percent of the lines in the class MapStructure. This overview can help to find classes where the coverage has to be improved. And we can see how effective our unit tests are. Especially by using Mockito it happens that too much methods are mocked which could be tested as well, if we would improve our code by extracting some methods and so on.



EnemyTest (29.10.2019 19:30:09)

Element	Coverage	Covered Instructio...	Missed Instructions
e-Portfolio unit tests	38,9 %	661	1.038
src	38,9 %	661	1.038
map.matchfield	0,0 %	0	330
map.objects.tower	0,0 %	0	316
map.structure	74,7 %	384	130
Field.java	32,3 %	53	111
MapStructure.java	94,7 %	286	16
FieldDescription.java	93,8 %	45	3
util	31,5 %	45	98
SortingUtil.java	0,0 %	0	98
Position.java	100,0 %	45	0
map.objects.enemy	73,7 %	232	83
Enemy.java	68,3 %	164	76
EnemyTest2.java	0,0 %	0	7
EnemyTest.java	100,0 %	61	0
Tank.java	100,0 %	7	0
game	0,0 %	0	81
Game.java	0,0 %	0	81

The test coverage is also shown by some special highlighting of your code. A green highlighting shows the code which was executed. If statements are often yellow because they were executed but not every possible case has occurred. A red marker means the code was not executed. A good tested class should contain as much as green highlighting as possible except the getter and setter methods. But as unit testing also can cost a lot of time, often it is not possible to write so much tests. Then you have to focus on the important functionalities.

```
public boolean move(MapStructure map) {
    if(actualField==null) {
        actualField=map.getFieldForEnemy(progress);
        progress++;
        moveToPosition(actualField.getSpawnPoint());
        return true;
    }
    if(actualField.getSpawnPoint().equals(getPosition())) {
        actualField = map.getFieldForEnemy(progress);
        progress++;
        if(actualField!=null) {
            System.out.println(label + " is moving to a new field [" + actualField.getFieldPositionX() + actualField.getFieldPositionY()
        }
    }

    if(actualField!=null) {
        Position pos = actualField.getSpawnPoint();
        if(pos.getX()-x<0) {
            moveTo(x-1, y);
        }else if(pos.getX()-x>0){
            moveTo(x+1, y);
        }else if(pos.getY()-y<0) {
            moveTo(x, y-1);
        }else {
            moveTo(x, y+1);
        }
        return true;
    }
    System.out.println(label + " reached the target");
    reachedTarget=true;
    return false;
}
```

Solution for Exercises:

The first test should control, the enemy to be dead, if the healthpoints are reduced to zero or less. Our arrangement is simple. We only need one tank, in this case we set the healthpoints to 100. Important for the action is, that we reduce the healthpoints about 100 or more points. For the test we assert enemy.isAlive() to be false.

```
@Test
public void ifDamageToEnemyHigherThanHP_EnemyShouldBeDead() {
    //arrange
    Enemy enemy = new Tank("tank1", 100, 900);

    //act
    enemy.reduceHealthPoints(150);

    //assert
    assertFalse(enemy.isAlive());
}
```

For the second test you need to know the code of the gameengine a little bit better. But to write tests it is important to get into the code you want to test. Now we want to test, if an enemy moves on a given map by its own. For that we will take a look at the position before and after the move operation.

Arrangement:
First we need to create an enemy. The parameters of our tank are not important for this test. A standard map is automatically created by calling the constructor of the class MapStructure. Our Enemy needs that object to perform the move action.

Action:
If we create a new enemy, it has no position at the beginning. To spawn it on the map, we need to execute the move method one time. Then the enemy moves to the defined spawn point of the map we created in the arrangement. Now we will keep the spawn position in a local variable for the assertion. The second call of the move method should move the enemy one unit in a defined direction. But to keep the test simple we only try to test, if the enemy changed the position or not.

Assertion:
In the assertion we have to make sure that the new position doesn't equals the position of the point the enemy was spawned. We could use asserEquals() as well as asserThat(). It is just about comparing the position we saved after the first move and the position after the second move.

```
@Test
public void whenEnemyMoves_ThePositionShouldChange() {
    Enemy e = new Tank("tank", 5, 950);
    MapStructure map = new MapStructure();

    //act
    e.move(map);
    Position oldPos = e.getPosition();
    e.move(map);

    //assert
    assertThat(oldPos, is(not(equalTo(e.getPosition()))));
}
```