SENG275 – Lab 6
Due Wednesday March 10, 11:55 pm

Welcome to Lab 6. Throughout this document, some sections will be in Red, and others will be in **bold**. The sections in Red will be marked. The questions in **bold** will not be graded. However, you should give these questions some thought, as these are the sorts of questions that could appear on a quiz or midterm. This lab is worth 4% of your final grade.

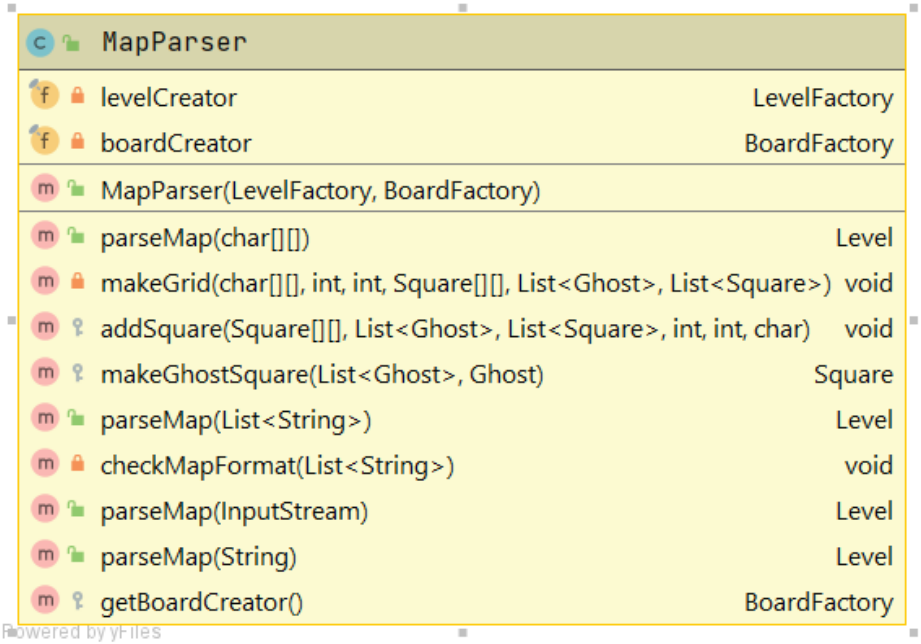
Quick summary of what you need to do:

1. Verify the mocks and stubs in Exercise 1, in test/…/level/MapParserTest.java.
2. Implement the bad-weather tests in Exercise 2, in the same file.
3. Provide branch coverage for Game.start() in Exercise 3, in the file test/…/game/GameTest.java.

You're done!

Overview

Consider the MapParser class:



We'd like to test this class (specifically the parseMap() methods), but when we read the constructor, we realize that even *creating* an object of type MapParser involves objects from two other classes. MapParser.parseMap(), in turn, involves several other classes (like Square and Pellet) in complicated ways.

This creates problems – if we see certain behaviour from MapParser, it's difficult to be certain that the behaviour results from the class under test, rather than one of its dependencies. The interdependence of these classes also make their boundaries unclear, and make it more difficult to ensure that our unit tests are actually testing the smallest possible units of the program.

Our solution is to mock these external classes. We replace them with minimal objects under our control, which we configure to give them only the minimum behaviour necessary to allow us to test our class of interest. Mockito is the mocking framework we'll use for this lab.

Basic Dummies

Basic dummies work well if the bare existence of an object of a particular class is enough, and we don't need any actual functionality. For example, to create instantiate a MapParser object, we need to pass the constructor objects of type LevelFactory and BoardFactory. To create a mock BoardFactory object:

```
BoardFactory mockedBoardFactory = mock(BoardFactory.class);
```

mockedBoardFactory is now a *dummy* of the BoardFactory type; it implements every method of the BoardFactory class, but all the dummy's methods return default values – zeros for numeric types, nulls for Objects, and false for booleans.

Verifying Interactions

Our goal is to confirm that our class under test is fulfilling its contract – it is calling exactly the methods that it should. For example, when mapParser creates a map containing one empty space,

```
mapParser.parseMap(List.of(" "));
```

the BoardFactory should call createGround(). We can *verify* that this has happened by asserting:

```
verify(mockedBoardFactory).createGround();
```

If our mockedBoardFactory ever called its createGround() method, this will pass. Otherwise the test will fail.

Sometimes we want to verify that a method was called, but that method takes arguments. If we want to check for a specific method, we can provide it in the method call itself, and if we don't care what the arguments are, we can write our verify statement with one of the any() argument substitution functions.

```
verify(mockedLevelFactory).createLevel(any(), anyList(), anyList());
```

Other times we want to be more specific about our verification. For example, we may wish to verify that createGround() was called exactly twice:

```
verify(mockedBoardFactory, times(2)).createGround();
```

If we want to verify that NO methods were called on a mock object, we can use:

```
verifyZeroInteractions(mockedBoardFactory);
```

If we want to verify that NO other methods were called on a mocked object OTHER than the ones we have already verified, we can use:

```
verifyNoMoreInteractions(mockedBoardFactory);
```

An aside: Logging

How do we know what methods we should verify?

It would help us if the documentation clearly stated what external class methods it invokes. For example, MapParser.parseMap() could state that it calls MapParser.makeGrid(), which in turn calls MapParser.addSquare(), which finally calls BoardFactory.createGround(). Then we would know that BoardFactory.createGround() is an external interaction, and we would know we need to mock BoardFactory and verify createGround().

Unfortunately, the documentation doesn't state this. In lieu of documentation, we sometimes have to resort to code tracing. We would follow the chain of internal method calls as above, and stop when we reach the first non-MapParser method call.

A (perhaps) easier way to do this is to use Mockito's logging capabilities. We can turn BoardFactory from a dummy into (sort of) a spy, and then get it to report back to us the methods that were called on it.

```
mockedBoardFactory = mock(BoardFactory.class,
withSettings().verboseLogging());
```

When we then run parseMap(List.of(" ")), we get:

```
############ Logging method invocation #1 on mock/spy ########

boardFactory.createGround();

   invoked: -> at
nl.tudelft.jpacman.level.MapParser.addSquare(MapParser.java:113)

   has returned: "null"


############ Logging method invocation #2 on mock/spy ########

boardFactory.createBoard([[null]]);

   invoked: -> at
nl.tudelft.jpacman.level.MapParser.parseMap(MapParser.java:75)

   has returned: "null"
```

So one approach is to mock every external class you might need with logging enabled, and see the full list of interactions from your class under test.

Mocking

By default, test doubles created using the mock() method are dummies - their methods return values, but those values are `0`, `null` and `false`. We may need to turn this dummy into a Mock, by specifying other return values for its methods.

For example, if we call parseMap("."), we're asking the LevelFactory to create a level with one Square in it, and that square should contain a Pellet. MapParser.addSquare specifies:

```
levelCreator.createPellet().occupy(pelletSquare);
```

Our dummy LevelFactory gets the createPellet() call, and returns null. The result is

```
null.occupy(pelletSquare);
```

Which gives us a null pointer exception – we've tried to call occupy() on null.

The solution is to Mock LevelCreator:

```
Pellet mockedPellet = mock(Pellet.class);
when(mockedLevelFactory.createPellet()).thenReturn(mockedPellet);
```

We've identified that when MapParser.parseMap() gets a map that has a pellet on it, we have a new external interaction with the Pellet class, so we mock that. We then specify that our mocked LevelFactory should return a dummy Pellet, so that occupy() can be called on a valid object reference, rather than null.

Worked example:

Our goal: Suppose that if MapParser.parseMap(" ") is called (one empty space) then createGround() and createBoard() are called on BoardFactory, and createLevel() is called on LevelFactory. We wish to verify this.

We're testing MapParser. So we need to create a MapParser object. To do this, we need LevelFactory and BoardFactory dummies. Let's ARRANGE our test by creating them:

```
mockedLevelFactory = mock(LevelFactory.class);
mockedBoardFactory = mock(BoardFactory.class);

mapParser = new MapParser(mockedLevelFactory,
mockedBoardFactory);
```

Now, we ACT – we call parseMap:

```
mapParser.parseMap(List.of("#"));
```

Finally, we ASSERT that the contract is met:

```
verify(mockedBoardFactory, times(1)).createWall();

verify(mockedLevelFactory, times(1)).createLevel(any(),
anyList(), anyList());

verify(mockedBoardFactory, times(1)).createBoard(any());
```

Exercise 1

(2%) - Please test the MapParser class in a file named test/…/level/MapParserTest.java. To do so, you will need to create test doubles for LevelFactory and BoardFactory. You may also have to create doubles for other classes; use a combination of code reading and logging to determine what mocks are required. Write tests in MapParserTest to verify the following:

1. If the map consists of one space (" "), createGround() and createBoard() are called on the dummy BoardFactory, and createLevel() is called on the dummy LevelFactory. The code to do this is in the section above.
2. If the map consists of one wall ("#"), createWall() and createBoard() are called on the dummy BoardFactory, and createLevel() is called on the dummy LevelFactory. IN ADDITION, EACH OF THESE METHODS IS CALLED EXACTLY ONCE.
3. If the map consists of one player and one empty space ("P "), createBoard() is called on the dummy BoardFactory EXACTLY TWO TIMES.
4. If the map consists of one pellet ("."), createPellet() is called on the mocked LevelFactory, and occupy() is called on a mocked Pellet (you will need to mock the pellet class, and perhaps more, to make this test work).
5. If the map consists of one ghost ("G"), createGhost() is called on the mocked LevelFactory. You will need to mock the Ghost and Square classes to make this work. In addition, test that occupy() is called on the mocked Ghost, and that NO METHODS are called on the Square Mock.
6. If the map consists of one space (" "), createGround() and createBoard() are called on the dummy BoardFactory, and createLevel() is called on the dummy LevelFactory (just like in #1 above). In addition, test that NO OTHER METHODS are called on either of those two dummy classes.

Bad-weather testing

So far we have been giving parseMap valid maps only – we have not tried breaking the map parser. Thus our testing is incomplete. If parseMap() is given invalid input, it should throw a `PacmanConfigurationException`.

We can test for exceptions using `assertThrows:`

```
assertThrows(someExceptionType.class, someMethodReference);
```

The second argument to assertThrows is a 'method reference' – a pointer to a particular method in a particular class. If the method we want to reference doesn't take parameters, we can simply refer to it by name:

```
SomeClass::someMethod
```

This is a reference to the method SomeClass.someMethod().

However, if the method takes parameters, we have to wrap the method call in an anonymous function, by using a lambda expression. If someClassObj is an object of SomeClass, and we want to pass a and b to someClassObj.someMethod(), we need to write:

```
() -> {someClassObj.someMethod(a, b)}
```

Exercise 2

(1%) - In the same MapParserTest.java file, add three bad-weather tests for the MapParser.parseMap() methods. Hint: check the addSquare() and checkMapFormat() methods for ideas.

Branch coverage

We wish to write tests to achieve Branch coverage of some method. However, IntelliJ's branch coverage metrics (using Run With Coverage and the Tracing run configuration option) report statistics for the entire class. We'll have to resort to looking at the coverage annotations in the gutter to the left of the source code to confirm branch coverage instead.

Exercise 3

(1%) - Write tests to achieve 100% BRANCH coverage of the Game.start() method. Write your solution in test/…/game/GameTest.java. Mock any appropriate classes. Hint: the following methods may be useful:

- Game.start()
- Game.isInProgress()
- Level.remainingPellets()
- Level.isAnyPlayerAlive()

If you wish, you may use the following code as a basis for your solution:

```
package nl.tudelft.jpacman.game;

import nl.tudelft.jpacman.level.Level;
import nl.tudelft.jpacman.level.Player;
import nl.tudelft.jpacman.points.PointCalculator;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;

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

class GameTest {

    private Player mockPlayer;
    private Level mockLevel;
    private PointCalculator mockPointCalculator;
    private SinglePlayerGame mockGame;

// your code will go here...

}
```

You will know you have been successful when after running your tests with coverage, every non-conditional statement in the method is annotated with green, like this: