Exercise 6

"Name 2 classes that are not well-tested, and explain why the smoke test does not cover it."

The classes `CollisionInteractionMap` and `InverseCollisionHandler` have 0% Class-coverage, because they are indirectly unused.
Their only usage is in the `DefaultPlayerInteractionMap`, a class that isn't used by any code. (consequently the Class-coverage of this class is 0% as well)

The SmokeTest would cover all 3 of these classes, if it instantiated the `DefaultPlayerInteractionMap` directly or indirectly.

Exercise 7

"Have a look at class game.Game, method move. Is it covered by our smoke test?"

Yes the method move is covered by the LauncherSmokeTest.

We are receiving the error that the player-score is 10, but we expected a score of 60.
The error message is quite helpful, as it tells us between which asserts the error is encountered.
Since we only call one method between the assert statements, we know precisely which method is causing the anomalous behaviour.

Exercise 8

"Change board.Direction.getDeltaX so that it returns dy instead of dx."

"Explain what you see. Was it easy to understand where the problem is?"

The test fails with this message:

```
org.opentest4j.AssertionFailedError:
Expecting:
 <0>
to be equal to:
 <10>
but was not.
Expected :10
Actual   :0

at nl.tudelft.jpacman.LauncherSmokeTest.smokeTest(LauncherSmokeTest.java:69)
```

In my opinion, the message provided by the smoke test isn't very clear. And the information that this test is failing is not particularly helpful.
This is because there are many possible error sources, as the test simply does "take one step east, are your points == 10?", it may be that the point calculator does not work, the level is set up incorrectly, something about the movement is broken, etc.

However, it is absolutely fine, that the smoke test provides such a general error, since it's purpose is to identify "does the game seem to work at a general level?" - and it correctly identified that it does *not*.

If we execute all test cases, it is easy to identify that something about navigating the board does not work correctly.
Unfortunately gradle does not execute our `DirectionTest` class, which makes the issue strikingly obvious, because the `DirectionTest` is not part of the `default-test`s.

Exercise 9

"Then, provide at most two paragraphs explaining how Game, Unit, Board, and Level classes are related to each other."

A List of Ghosts(which extends from Class `Unit`) and a `Board` is passed to the `Level` class on initialisation.
In the `Game` class the methods from the `Level` class are used to manage/control the running Game.

Exercise 10

see added test class `ClydeTest`
or alternatively the relevant Merge Request

Exercise 11

see added test class `InkyTest`
or alternatively the relevant Merge Request

Exercise 12

"Provide a domain matrix for the desired behavior of the boundary values in the withinBorders method."

Boundary conditions for "x >= 0 && x < getWidth() && y >= 0 && y < getHeight()"										
Variable	**Condition**	**type**	t1	t2	t3	t4	t5	t6	t7	t8
x	>= 0	on	0							
		off		-1						
	< getWidth()	on			width-1					
		off				width				
	typical	in					lerp(0, width-1, 0.2)	lerp(0, width-1, 0.4)	lerp(0, width-1, 0.6)	lerp(0, width-1, 0.8)
y	>= 0	on					0			
		off						-1		
	< getHeight()	on							height-1	
		off								height
	typical	in	lerp(0, height-1, 0.2)	lerp(0, height-1, 0.4)	lerp(0, height-1, 0.6)	lerp(0, height-1, 0.8)				
Result			TRUE	FALSE	TRUE	FALSE	TRUE	FALSE	TRUE	FALSE

Further Considerations:

`Width` and `Height` are variables as well from the set of positive natural numbers (including `0`).
We should test these boundaries ("`Width = 0`" and/or "`Height = 0`") as well, however, given the structure of two-dimensional Arrays, we cannot test for "`Width = 0 && Height > 0`".

Furthermore, assuming a `Width` of `0` reveals an inadequacy with the `Board` class, since `getHeight` will throw an exception.

Our resulting domain matrix may look like this:

Boundary conditions for "x >= 0 && x < getWidth() && y >= 0 && y < getHeight()"													
Variable	**Condition**	**type**	t1	t2	t3	t4	t5	t6	t7	t8	t9	t10	t11	t12
x	>= 0	on	0											
		off		-1										
	< getWidth()	on			width-1									
		off				width								
	typical	in					lerp(0, width-1, 0.2)	lerp(0, width-1, 0.4)	lerp(0, width-1, 0.6)	lerp(0, width-1, 0.8)	lerp(0, width-1, 0.3)	lerp(0, width-1, 0.5)	lerp(0, width-1, 0.7)	lerp(0, width-1, 0.9)
y	>= 0	on					0							
		off						-1						
	< getHeight()	on							height-1					
		off								height				
	typical	in	lerp(0, height-1, 0.2)	lerp(0, height-1, 0.4)	lerp(0, height-1, 0.6)	lerp(0, height-1, 0.8)					lerp(0, height-1, 0.3)	lerp(0, height-1, 0.5)	lerp(0, height-1, 0.7)	lerp(0, height-1, 0.9)
width	> 0	on									1			
		off										0		
	typical	in	2	3	4	5	6	7	8	9			10	11
height	> 0	on											1	
		off												0
	typical	in	11	10	9	8	7	6	5	4	3	0		
Result			TRUE	FALSE	TRUE	FALSE	TRUE	FALSE	TRUE	FALSE	TRUE	FALSE	TRUE	FALSE

height has to be 0 here, because width is 0

Exercise 13

see modified test class `BoardTest`
or alternatively the relevant Merge Request

Exercise 14

"What can we do to avoid code repetition during the Arrange part of the unit test?"

It is important to specify the unit test as precisely as possible in the arrange phase.
This saves you from calling classes / methods etc. that are not absolutely necessary for the unit-test.
The less complex the test, the lower the probability of code repetitions.

Exercise 15

"Note that our tests always make use of 'clean instances' of the class under test. What are the advantages of such approach?"

Testing "clean instances" means that the individual tests can easily prepare the instance for their individual tests.
Most importantly it prevents tests from affecting each other. If they did, executing one test before another may result in a test failure, whereas the reverse order does not. Additionally, changing a test may cause different tests to fail.

Exercise 16

"Which one is a better assertion, supposing some `int a`? 1) `assertEquals(1, a);` or 2) `assertTrue(1 == a)`? Discuss the differences between both assertions."

It is generally always better to use the more "specific" assertion: `assertEquals` in this case.
assertEquals means "are two things equal?", whereas assertTrue just means "is this boolean true?".
When using `assertTrue`, the information that we're comparing two values for equality is not communicated to JUnit.

The error message of assertEquals is also more clear than assertTrue: `Expected: 1 | Actual: 5` is more helpful than `Expected: True | Actual: False`

Using assertEquals also ensures that changing the type of `a` to a non-primitive type does not affect the assertion, as assertEquals uses the overridable `Object::equals` method, whereas `==` does not.

Exercise 17

"Do we need to have specific tests for [private methods] ? Why? Why not?"

We do not need to test private methods.
Private methods are not supposed to be directly accessible from the outside, that means two things:

1. Assuming they are not unused (and no reflection shenanigans are involed), there must be a public method that uses the private method.
   We should test the private method indirectly through the public method(s) that access it.
2. Private methods are not supposed to be accessed from the outside, so testing them like that seems inappropriate to me.

Exercise 18

"Explain or eliminate checkstyle or SpotBugs violations that remain"

Currently, there is one remaining checkstyle violation:

```
[ant:checkstyle] [WARN] D:\Schule\Studium\Semester
5\SoftwareTesting\SoftwareTesting_LabWork\jpacman\src\main\java\nl\tudelft\jpacman
\board\Board.java:61:13: Avoid inline conditionals. [AvoidInlineConditionals]
```

I have actively chosen to ignore this warning, because I personally strongly disagree with it.
I believe that ternary expressions for single (or sometimes two) conditions are at least just as readable as `if` statements, while offering a reduced footprint / less clutter.

"Include a brief assessment of the additional adequacy achieved in JPACMAN, thanks to your new classes."

There are two new test classes: `ClydeTest` and `InkyTest` which aim to identify incorrect behaviour of the movement logic for the Ghosts `Clyde` and `Inky`.

"Also reflect on your continuous integration server results"

Seeing our Github-Actions finally working correctly is quite satisfying, however - given that they have only recently been implemented - we have yet to get much use out of them.
Regardless, I am sure it won't be long until they spot an issue that we didn't before pushing.

"And [reflect on] your commit behaviour"

I believe that our commit behaviour is - for the most part - presentable.
When committing individually, we try to create merge requests per exercise for the other person to review and merge.
Occassionally this is easier said than done, because one exercise may depend on the changes applied in another exercise.

When committing together, commits may get more messy, as both parties have already seen the changes, but we still try to create merge requests as a means of documenting completed exercises.