Report Part 1

Exercise 6

The class LevelFactory is not well tested. Line 111 calls an red enemy sprite when no color is called. This is not tested because the color of the sprite is always defined. CollisionInteractionMap.java is not covered at all. This implies that this class is never called by executing this test.

Exercise 7

Yes the move method is covered by the smoke test

The test expected the player to move East but this did not happen. The failure shows that the expected value is expected to be 10 but was 0. From this information it is clear that there is no movement.

Exercise 8

The test gives exactly the same failure. But now you can not be sure were the error originates from. The error is too high up the hierarchy to be found immediately. Therefore the error will be hard to find and resolve

Exercise 9

The Game class can be seen as container as such. It creates the point system, fetches the players and level. This class is also in charge for ending the game.

The Level class creates all units, which includes the player, enemies and pellet's. It also creates the grid and board that the game is played on.

Exercise 10

Good weather: Implementing two test cases will make sure Clyde works as expected. - Test Clyde is close to Pacman and if he is moving away from him. - Test if Clyde is moving near to Pacman if their distance is greater than 8 squares.

Bad weather: - Test Clyde's behaviour if the player isn't on the board. - Test Clyde's behaviour if there is a null square from the board

See source code

Exercise 11

Good weather: - Test Inky's behaviour if Inky is behind Blinky. - Test Inky's behaviour if Inky is in front of Blinky.

Bad weather: - Test Inky's behaviour if Blinky isn't on the board. - Test Inky's behaviour if player isn't on the board. - Test Inky's behaviour if Blinky and Inky are encapsulated.

Exercise 12

■ = false
■ = true

Boundary test for method Board.withinBorders x >= 0 && x < getWidth() && y >= 0 && y < getHeight()									
Variable	condition	type	Test cases(x, y)						
			T1	T2	T3	T4	T5	T6	T7	T8
x	=> 0	On	0							
		Off		<0						
	< getWidth()	On			getWidth()					
		Off				<getWidth()				
	typical	In					0 =< x < getWidth()			
y	>= 0	On					0			
		Off						< 0		
	< getHeight()	On							getHeight()	
		Off								< getHeight()
	typical	In	0 =< y < getHeight()							

Example: Board of 3x4

Boundary test for method Board.withinBorders x >= 0 && x < getWidth() && y >= 0 && y < getHeight()									
Variable	condition	type	Test cases(x, y)						
			T1	T2	T3	T4	T5	T6	T7	T8
x	=> 0	On	0							
		Off		-1						
	< getWidth()	On			3					
		Off				2				
	typical	In					1	1	2	0
y	>= 0	On					0			
		Off						-1		
	< getHeight()	On							4	
		Off								3
	typical	In	0	1	2	3				

Exercise 13

```
    @ParameterizedTest
    @CsvSource({
        "0, 0, true", "-1, 1, false", "3, 2, false", "2, 3, true",
        "1, 0, true", "1, -1, false", "2, 4, false", "0, 3, true"
    })
    public void TestWithinBorders(int x, int y, boolean expected) {
        Square[][] grid = {
            { mock(Square.class), mock(Square.class), mock(Square.class),
mock(Square.class) },
            { mock(Square.class), mock(Square.class), mock(Square.class),
mock(Square.class) },
            { mock(Square.class), mock(Square.class), mock(Square.class),
mock(Square.class) }
        };
        board = new Board(grid);

        assertEquals(board.withinBorders(x,y), expected);

    }
```

Exercise 14

If a specific part of the code is being tested several times over again with only different inputs we can use `@ParamaterizedTest` from JUnit5. This way we only have to write the test method just once and add parameters to this test method. The inputs we want to test are saved as CSV under the `@CsvSource`-tag.

Exercise 15

The advantage of creating the same board for every test class is that the test will always give consistent results. If we would have used the same board over and over again we might have gotten different results for the same test cases. This is because other test might adjust this board and cause other test the fail. This holds for all tests which rely on an extra instance or object for the test case.

Exercise 16

In general, both assertions are the same. The only difference is the error message which will be printed if the test fails. If the `assertTrue` fails, we will not know what the correct int should have been. Whereas if we would have used `assertEquals` the default error message also prints what the actual value should have been. This feedback is crucial in order to be able to find and resolve your errors

Exercise 17

Private methods do not need te be tested in a isolated way. They should be tested from a public class or interface which invokes the private method.

Exercise 18

After spending some analyzing and understating Inky's behavior we have added a test case for Inky and adjusted the addSquare() method so we can add Inky to the gameboard.

In general, we committed once per exercise. Sometimes multiple commits were needed because a small discussion changed our final answer. CI sometimes failed because we didn't always execute Gradle check. So this also resulted into multiple commits per exercise