Report Part 2

Exercise 1

Some good weather cases we thought of: - Verifying if methods from mocked classes in method makeGhostSquare are used. - Test for empty space ( ) in addSquare method. - Test for wall (#) in addSquare method. - Test for pellet (.) in addSquare method. - Test for ghost (G) in addSquare method. - Test for Pacman (P) in addSquare method.

Exercise 2

Some bad weather cases we thought of: - Test if exception is thrown when an invalid char is used in addSquare method. - Test if exception is thrown when an invalid position is used in addSquare method. - Test if exception is thrown when an text map is null in parseMap method. - Test if exception is thrown when an empty ArrayList is used in parseMap method. - Test if exception is thrown when an only an empty string inside the ArrayList is used in parseMap method. - Test if exception is thrown when text map lines, which are not of equal width, are used in parseMap method. - Test if exception is thrown when an empty string filename is used in parseMap method. - Test if exception is thrown when an inputStream with an empty resource is used in parseMap method.

Exercise 3

With these test together we have achieved 100% Class coverage , 57% Method coverage, 70% Line coverage

```
/**
 * Setting up the mocks for the tests.
 */
@BeforeEach
void setup() {
    pc = mock(PointCalculator.class);
    player = mock(Player.class);
    level = mock(Level.class);
    game = new SinglePlayerGame(player, level, pc);
    when(level.isAnyPlayerAlive()).thenReturn(true);
    when(level.remainingPellets()).thenReturn(1);
}

/**
 * Test to see if starting and stopping the game works.
 */
@Test
void testStartStop() {
    assertThat(game.isInProgress()).isFalse();
    game.start();
    assertThat(game.isInProgress()).isTrue();
    verify(level, times(1)).start();
    game.stop();
    assertThat(game.isInProgress()).isFalse();
}

/**
 * Test to see if the game not starts again after already being started.
 */
@Test
void testStartAndStartAgain() {
    assertThat(game.isInProgress()).isFalse();
    game.start();
    assertThat(game.isInProgress()).isTrue();
    verify(level, times(1)).start();
    game.start();
    assertThat(game.isInProgress()).isTrue();
    verify(level, times(1)).start();
}

/**
 * Test to see if the game isn't in progress if there are no players.
 */
@Test
void testStartNoPlayer() {
    when(level.isAnyPlayerAlive()).thenReturn(false);
    game.start();
    assertFalse(game.isInProgress());
}

/**
 * Test to see if the game isn't in progress if there are no pellets.
 */
```

```
@Test
void testStartNoPellet() {
    when(level.remainingPellets()).thenReturn(0);
    game.start();
    assertFalse(game.isInProgress());
}
```

Exercise 4

Collider	Player	Player	Player	Player	Player	Ghost	Ghost
Collidee	Empty square	Wall	Pellet	Last pellet	Ghost	Player	Pellet
Consequence	Move to square	Nothing	Move to square Pellet disappear Player gets points	Move to square Player wins End game	Move to square Player dies End game	Player dies End game	Pellet becomes invisible

Exercise 5

See level.PlayerCollisionTestOld

Exercise 6

See level.CollisionMapTest, level.PlayerCollisionTest and level.DefaultPlayerInteractionTest

Exercise 7

When comparing the original tests and the tests we wrote, we can see for the collisions of the game, that the original tests only covered the PlayerCollisions class and not the CollisionInteractionMap and DefaultPlayerInteractionMap. From the PlayerCollisions class not every method is covered completely. From the method "collide" the third if statement isn't covered. Also the method pelletColliding isn't covered.

The tests we wrote ourselves cover the DefaultPlayerInteractionMap and the CollisionInteractionMap almost completely (except for the method collide in the case the collisionHandler is null and getInheritance method for the if statement in the for-loop). Compared to the original tests, our tests cover all lines from the PlayerCollision class.

Exercise 8

One way to test the method is by observing it's behavior manually. If we execute the random method a thousand times and only the directions `East`, `West` and `South` are returned we can conclude that the random method is probably not well configured, This type of testing is very unreliable of course. A better approach to this problem would be using seeds for the random method. With this seed the random method will always return the same value. By using different seeds we can cover all decisions.

Exercise 9

The source of the flakiness is found in the `Thread.sleep()` line. This sleep is used to let the ghosts move around the board. However the movement of the ghost are not consistent and might therefore give different results with every test. As soon as a test relies on any form of randomness the test can be flagged as flaky. Writing big test may often result into flaky tests, as can be seen here. A simple solution would be by splitting the test and testing `eaten-by-ghost` separately. Using mocks to simulate necessary behavior is also a good solution against flaky tests.

Exercise 10

The biggest disadvantage is that 100% code coverage creates this illusion that everything has been tested and is therefore bug-free. But this is `NOT` the case. Even though every piece of code is covered, bugs might still be present. Code coverage should be used as a tool to indicate how much someone has not tested the software, rather than estimating how much testing has been done already.

Exercise 11

- The main disadvantage of mocking is that is that test can become very slow if there are many mocks involved.
- Since mocks are a reflection of the actual implementation it can slow down the testing process significantly.
- Mocking makes the setup of test very complicated (, much more complicated than test without mocks).

Exercise 12

Test tend to decrease in execution time when they grow bigger and/or when multiple mocks are used. To avoid slow test or to speed them up, a good place to start is to break up big tests into smaller ones. Three/four asserts at most per test. If your test requires multiple mocks

you should consider making your class less coupled.

Exercise 13

Mocking should be used whenever we need a dependency or something equivalent. However, mocking should be avoided if your class is too coupled

Exercise 14

When running the AmazingPointCalculator instead of the DefaultPointCalculator, it seems like the following four anomalies occur: * After eating around 28 pellets or more, the PointCalculator deducts 15 points after each pellet. * The score counter seems to suddenly change to an enormous negative number, possibly due to an overflow, after eating around 15 pellets or more and facing West. * After eating more than 34 pellets and facing North direction the game stops, and you get a Runtime exception * After eating more than 34 pellets Pacman suddenly dies, without any of the ghosts killing it.

Exercise 15

See attached .docx file

Exercise 16

The staticAnalysis tool indicates that the build has been successful without any warnings. The reason that AmazingPointCalculator doesn't trigger any PMD warnings is because the class executes an abnormal behavior that has code implemented similiar to other methods in the project. So PMD couldn't tell the difference from the way the code has been structured and therefore couldn't tell apart that this class will execute some abnormal behavior. Looking at the OWASP top 10 vulnerability list, we can say that this 'Using Components with Known Vulnerabilities'. This is because we replaced our PointCalculator component (DefaultPointCalculator) with that of a third party one into our application. Since we are unsure if the component works as it should do, because we got te component from a third party, their is a chance that this component has a vulnerability. To solve the vulnerability we should always check if we receive components from a trusted source and if the component is up to date. We should also check if the component works as expected.

Exercise 17

Dynamically loaded classes enables the developer to install software components at runtime. This means that classes are loaded on demand and at the last moment possible. The security problem that comes with this, is that third-party can use the component for malicious use, so the developer should check if the components comes from a trusted source. The developer should also create his own dynamically loaded classes and, in order to achieve isolation from dynamically loaded component, the developer should also separate it with by using different class loaders.

Assesment

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.