Universität Potsdam

Name: **Deniz Mert Tecimer, 811870 / Günther Wullaert, 780827** Labwork: **0**
Course: **Software Testing** Date: April 21, 2022

0 Part 0: Prerequisites

0.1 A First Test

0.1.1 Ex 1

The instructions were quite easy to follow. Adding the missing tests for the other 3 directions was also straightforward and were nearly a carbon copy:

```
@Test
    void testEast() {
        Direction north = Direction.valueOf("EAST");
        assertThat(north.getDeltaX()).isEqualTo(1);
        assertThat(north.getDeltaY()).isEqualTo(0);
    }
```

We also checked in the same test that we are not moving diagonally if we want to move east.

0.1.2 Ex 2

For this exercise we used the useful helper function hasSquare() to check if the unit has a squared before we access the square with getSquare(). The test to check if a unit initializes with no square is trivial. To check if a unit occupies the correct square after we call the occupy() function we accessed the getSquare() and assert that it is equal to the square we passed to the occupy function. We also checked the other direction, if the square has the unit as its occupant listed. For the reoccupy() function we were a bit unsure what was meant in the question with this test. Should it be tested that the same square is occupied twice or what happens if we occupy 2 different squares after each other. We decided upon the later but can easily add the other variant if needed. As the previous test already checks the correctness after one occupation, we only need to check what happens after the second occupation. Here we check just as in the occupy function that the links between the unit and the square are still correct. However, we now also check that the previous square has lost its reference to the unit.

For the assertion test we tried it on GitLab, however as there currently is no runner, all tests fail. However, without the -ea option the tests throw no error and run through. I would guess that assertions are not enabled on GitLab.

```
@Test
    void noStartSquare() {
        assertThat(unit.hasSquare()).isEqualTo(false);
    }

    /**
     * Tests that the unit indeed has the target square as its
       base after
     * occupation.
     */
    @Test
    void testOccupy() {
        BasicSquare bsquare = new BasicSquare();
        unit.occupy(bsquare);
        assertThat(unit.hasSquare()).isEqualTo(true);
        assertThat(unit.getSquare()).isEqualTo(bsquare);
        assertThat(bsquare.getOccupants().contains(unit)).isEqualTo(true);
    }

    /**
     * Test that the unit indeed has the target square as its base
       after
     * double occupation.
     */
    @Test
    void testReoccupy() {
        Square testSquare = new BasicSquare();
        Square otherSquare = new BasicSquare();
        unit.occupy(testSquare);
        unit.occupy(otherSquare);

        assertThat(unit.hasSquare()).isEqualTo(true);
        assertThat(unit.getSquare()).isEqualTo(otherSquare);
        assertThat(otherSquare.getOccupants().contains(unit)).isEqualTo(tru
    }
```

0.1.3 Ex 3

It is impossible to exhaustively test our entire software project, because the number of possibilities grows exponentially with parameters added. Take for example a simple sum function that adds up two 32-bit integers, here each integer we pass can take on $2^{32} - 1$ values. This is already a very large number, but to exhaustively test this function we would need to pass each combination of values for these 2 integers to the function. We should do a variety of different tests (automatic, manual, static), covering as much as possible.

0.1.4 Ex 4

If the same tests are run over and over again, then those tests will find less defects, until no defects can be found anymore with those tests. To avoid running into this software developers have to keep adding new tests or revising old test to test different parts of the software project. In a changing environment, causes of defects might also change, so that the developer, who uses same test suit over and over again, might end up being a doctor, who prescribes the same medicine all the time regardless patient's disease.

0.1.5 Ex 5

Because we want to have a large number of tests and ideally not waste any time executing tests manually. If tests can be executed automatically we can run these checks in version control software and for example only let a merge request be merged if all tests run-through in a CI/CD environment, which speeds up the development process. Otherwise we would wait for the test results to proceed to development.

Universität Potsdam

Name: **Deniz Mert Tecimer, 811870 / Günther Wullaert, 780827** Labwork: **1**
Course: **Software Testing** Date: April 21, 2022

1 Part I: Unit tests + Boundary Tests

1.1 Smoke Testing

1.1.1 Exercise 6

There are many classes that are not well tested. However, these are mostly classes meant for testing. JPacmanFuzzer for example has a 0%-line coverage, is however meant as a tool for easier testing the game with random moves until Pacman dies. The StartupSystemTest class, also with 0%-line coverage, is an Integration test class that tests if the system starts up.

1.1.2 Exercise 7

The "move" method in the game.Game is covered by the smoke test. After commenting the line, that calls the "move" method of level class, an assertion failed. It was expected the player to move and collect 10 points, however it did not move and failed to collect these points. The assertion for that points failed. Thus, we know that there must be a problem with the point collection code or with the move function, as that was the only function we called during the test. This is a significant reduction in the search space for finding the error. However, the move function has a lot of dependencies and calls other functions which can be hard to pin point. It definitely shows something is wrong with the code and it shouldn't be shipped to the user, however a full unit test suite on the side, which tests the internal functions in isolation would be better.

1.1.3 Exercise 8

There was an assertion exception again at the same point. Again, the expected points to be collected where not collected. However, this time it is much harder to pin point the error. The fault happens deep after many function calls. As stated above the Smoke test shows quickly if everything is working as expected and should be used to see if fundamentally things are wrong or not with the code. However, unit tests testing the sub functions here at play, for example the getDeltaX function would be more useful in pin pointing the error.

1.1.4 Exercise 9

The Game is the master class that sets up the correct game we want to play. It contains a reference to all playable units and to the current level that is played. The level class contains the main business logic of the game. It instantiates a game board and contains a reference to all units participating in this level. The board defines the playground and terrain of the current level. It also contains references to all squares that are on this playground. These squares all hold references to the units that occupy them. A unit is everything that moves on the board. This includes different players that play Pacman, but also includes the different types of ghosts.

1.2 Unit Testing

1.2.1 Exercise 10

1. Two good weather tests can be implemented based on the shyness value. If Clyde is more than shyness steps away from the player, it should get closer. If the distance between Clyde and the player is smaller than shyness, then it should run away.
   Two bad weather tests can be implemented for the things that are unlikely to happen. The first is that a player could not be found. The second is that there is no path to the player from Clyde. While unlikely to happen with the current code base, they should still be tested, as in the future we could implement a teleporting functionality for the player from one side of the map to the other for example, leaving the player shortly absent.

2. Before we did any coding, we changed the gradle file so that it works on our local machine, as you did in the master repository.
   The GhostMapParser needs a lot of dependencies to be setup before we can test the ghosts. It basically requires the whole game to be setup. We need a LevelFactory, which creates the level itself. We require a BoardFactory, which creates the board with all the squares. We also need a GhostFactory to create the ghosts itself. The Levelfactory requires a PointCalculator to hold the points. We just used the DefaultPointCalculator. Most of these classes needed a reference to sprites for the UI, here we just used the basic PacManSprites class. We also required a player to register it. We created one Player with the PlayerFactory. We put all of this in a setup method that runs before each test, so that we do not have to much code repetition. This could also be put in test tools, that setup a GhostMapParser, so that you can use the same setup for other ghosts as well.

```
@BeforeEach
public void initTests() {
    PacManSprites pacManSprites = new PacManSprites();
    GhostFactory ghostFactory = new
       GhostFactory(pacManSprites);
    PointCalculator pointCalculator = new
       DefaultPointCalculator();
    LevelFactory levelFactory = new
       LevelFactory(pacManSprites, ghostFactory,
       pointCalculator);
    BoardFactory boardFactory = new
       BoardFactory(pacManSprites);
```

```
        PlayerFactory playerFactory = new
            PlayerFactory(pacManSprites);
        player = playerFactory.createPacMan();
        ghostMapParser = new GhostMapParser(levelFactory,
            boardFactory, ghostFactory);
    }
```

For the tests itself we noticed, that the maps were rotated, for easier test readability we created a tools.Tools class which will serve as tools for testing for us. In there we created a method for rotating the map by 90° and outputting the map on the console if needed. This uses the print function which is not allowed by our PMD checking. Therefore, in this case we suppressed the PMD warning for that one function as it is just used for debugging while working on the tests.
We follow the AAA method in our tests. This is the arrange part of one of our tests:

```
char[][] map = new char[][]{
        "############".toCharArray(),
        "#P         C#".toCharArray(),
        "############".toCharArray()
};
Level level = ghostMapParser.parseMap(Tools.rotateMap(map));

level.registerPlayer(player);
player.setDirection(Direction.EAST);

Clyde clyde = Navigation.findUnitInBoard(Clyde.class,
    level.getBoard());
```

As you can see we create the map in a readable format and then rotate the array with our tools. We register the player and set his direction and then find the unit we want to test (Clyde in this case).
Then we Act. We need this first Assertion because the variable clyde could be null. This should however never be the case if the test works correctly.

```
assertThat(clyde).isNotNull();
Optional<Direction> dir = clyde.nextAiMove();
```

The last part is the assertion. This is where we actually test if the expected behavior was acted out in the previous part:

```
assertThat(dir).isPresent();
assertThat(dir.get()).isEqualTo(Direction.WEST);
```

For the all the tests for Clyde we created a map with a single corridor. For the good weather tests, we put Clyde near the player and further away from the player. Either he will move closer or will move further away. For the bad weather tests, we once didn't register any player and for the other one we put a wall around them so they cannot reach

each other and the path finding fails.

1.2.2 Exercise 11

For the good weather cases we have that Inky follows Blinky if Blinky is between Inky and Player. The other case is that Inky seemingly runs away from the player, because it targets a faraway point, because blinky is far behind Inky and the player is in front of Inky. The bad weather cases are that we cannot find a nearest player or blinky. That there is no path between blinky and the tile 2 squares ahead of the player, regardless of terrain. Or that there is no path after that point when following the path again. The problem lies however with the case that ignores any terrain. To reach the inner part of that if is just impossible. The only case was this could happen if somehow one of the 2 units would be outside of the map, but then we wouldn't find that unit anymore on the map with the findNearestUnit method. Because this is catched in an earlier if statement that is not possible. As squares are traveled through step by step in the search methods it is also not possible to ever go out of bounds, as all the squares at the edge have a link back to themselves. In conclusion we left this test case in, but are not testing the correct if statement as we couldn't find a way to setup the level to trigger it.
We setup Inky the same way as Clyde. To have the correct map parsed we had to change the map parsing method to add blinky and inky:

```
switch (c) {
    case 'C':
        grid[x][y] = makeGhostSquare(ghosts,
           ghostFactory.createClyde());
        break;
    case 'B':
        grid[x][y] = makeGhostSquare(ghosts,
           ghostFactory.createBlinky());
        break;
    case 'I':
        grid[x][y] = makeGhostSquare(ghosts,
           ghostFactory.createInky());
        break;
    default:
        super.addSquare(grid, ghosts, startPositions, x, y, c);
}
```

For the good weather cases we just setup the map just as we explained the test cases. For one of the cases we put Blinky between the player and Inky and check if Inky moves towards the player(1). For another test we create a large map and see with Inky very close to the player, but Blinky at a further position, if Inky moves away (North) of the player(2):
(1)

```
char[][] map = new char[][]{
    "###############".toCharArray(),
    "#   P    B   I#".toCharArray(),
    "###############".toCharArray()
};
```

(2)

```
char[][] map = new char[][]{
    "################".toCharArray(),
    "#              #".toCharArray(),
    "#          P I #".toCharArray(),
    "#             B#".toCharArray(),
    "################".toCharArray()
};
```

The two bad weather cases if there is no player or Blinky are trivial.
The one where there is no path between Blinky and the Player is easy to create by just isolating them all:

```
char[][] map = new char[][]{
    "################".toCharArray(),
    "# P  #  B #  I#".toCharArray(),
    "################".toCharArray()
};
```

The problematic one is the one where the path between Blinky and the Player should be wrong:

```
char[][] map = new char[][]{
    "################".toCharArray(),
    "P       B    I#".toCharArray(),
    "################".toCharArray()
};
Level level = ghostMapParser.parseMap(Tools.rotateMap(map));

level.registerPlayer(player);
player.setDirection(Direction.WEST);

Inky inky = Navigation.findUnitInBoard(Inky.class,
   level.getBoard());
assertThat(inky).isNotNull();
Optional<Direction> dir = inky.nextAiMove();
assertThat(dir).isPresent();
assertThat(dir.get()).isEqualTo(Direction.WEST);
```

The Player is looking outside the map. The value of the squaresAheadOf function should be outside the map and not able to reach Blinky. Resulting in no movement from Inky. However, we get that Inky wants to move to the West. The reason for this is that the squaresAheadOf function returns the Square of the Player himself, because that Square has a reference to himself in the WEST direction. We tried changing that reference but, then the findNearestUnit function fails and we never reach that part of the function.

1.2.3 Bonus

We also did the bonus exercise of having unit tests for the squaresAheadOf function. As this is a function of the Unit class we created a UnitTest class under board. We looked at 3 cases: If the amount passed to the function is 0 we should return the current tile of the unit. If the amount is 1 then we should get the square in front of the unit. And if the amount is 2 but there is only one square ahead we should get null. If the amount would be 3 in the same case we get an exception. We don't know if that was intentional or not in the design, so we didn't test that case. In my opinion the code shouldn't throw an exception, but just return null, so you know there is no square there.

1.3 Boundary Testing

1.3.1 Exercise 12

Variable	Condition	Type	T1	T2	T3	T4	T5	T6	T7	T8
x	>= 0	On	0							
		Off		-1						
	< 4	On			4					
		Off				3				
	typical	In					0	1	2	3
y	>= 0	On					0			
		Off						-1		
	< 4	On							4	
		Off								3
	typical	In	0	1	2	3				
Result			T	F	F	T	T	F	F	T

1.4 Understanding your tests

1.4.1 Exercise 14

It is important to avoid redundant codes. In the arrange part of the unit test, we usually initialize needed objects and setup the environment that we need for our test to run. It is not logical to repeat this initialization in each test case. Thus, we use @BeforeEach notation to avoid it. This notation runs the code within before it runs every test case.

1.4.2 Exercise 15

Tests should run independently from each other. As if in an experiment, to see the effect of a variable, we have to set everything as same as possible and change only the variable that we want to observe. If we don't make a clean start, there might variables that are created or changed by other test cases. For example in our case, we might want to test the behaviour of Clyde when there is no player registered. If we don't make a clean start, there might still a player, that is registered by previous test cases.

1.4.3 Exercise 16

assertTrue(1==a) will return a boolean and check only if the assertion in it is true. However, if we use assertEquals(1,a), this will check if these values are equal and return us their values and the expected result. This gives us a better insight to detect what is wrong.

1.4.4 Exercise 17

Because we don't have a direct access to private methods, we can not isolate and test them directly. However, we test public methods that use those methods, which are also covered by the test. When we test the public method that contains another methods, we are usually more concerned about its outcome and behaviour. If it behaves correctly within the boundaries that we test, than private method is also doing its job.

1.4.5 Exercise 18

With these new tests, we have tested for correct behavior of 2 out of 4 ghosts in the game, which is already the big part of our most important logic. These ghosts use a lot of different functions which all seem to be working correctly for the tests to pass. We now finally have a runner which always gives us a green check as we run gradle check before committing anything. If however an error slips through, the continuous integration server will notify us about it. We try to hold to a standard of how we write our commit messages. We commit as soon as we think the exercise is correct and push to the server. We maybe need to push more often as sometimes multiple class edits are in a single commit. We also had one case with merge conflicts in labwork 0 which left us with merge conflicts as we edited the end of the same file with different functions. Sometimes git also has some strange behavior and I had to commit the same commit twice before it recognized the changes. We keep every exercise in its own branch and only merge to master when the exercise is done. All in all, we are making good progress on testing this project.

Name: **Deniz Mert Tecimer, 811870 / Günther Wullaert, 780827** Labwork: **2**
Course: **Software Testing** Date: April 21, 2022

4 Part II - Structural testing and Mocking

4.2 Mock Objects

4.2.1 Exercise 20

The `level.MapParser` class is composed of 4 different `parseMap()` functions which are dependent on the `level.LevelFactory` and `board.BoardFactory`. They are in a hierarchy that they call the more underlying `parseMap()` function. To test these functions we will need quite a few Mocked classes. We need to Mock the `level.LevelFactory` and `board.BoardFactory`. But we also need some Mocks for the different Squares, Ghosts and Pellets that we should return to the `level.MapParser` so it can function. Because we need most of them for a lot of the tests, it does not make sense to recreate the same things before every test, therefore we create them in a setup method beforehand. We setup the `level.LevelFactory` and `board.BoardFactory` to return the respective Mocks for the ground, wall, pellet or ghost when the `level.MapParser` needs them.

```
private MapParser parser;

@Mock
private LevelFactory levelFactoryMock;
@Mock
private BoardFactory boardFactoryMock;

@Mock
private Square ground;
@Mock
private Square wall;
@Mock
private Pellet pellet;
@Mock
private Ghost ghost;

/**
 * Setup the Mock Objects for the Map Parser before each test.
 */
```

```
@BeforeEach
void setup() {
    initMocks(this);

    parser = new MapParser(levelFactoryMock, boardFactoryMock);

    // Setup returns for BoardFactory square creation
    when(boardFactoryMock.createGround()).thenReturn(ground);
    when(boardFactoryMock.createWall()).thenReturn(wall);
    when(levelFactoryMock.createPellet()).thenReturn(pellet);
    when(levelFactoryMock.createGhost()).thenReturn(ghost);

    // Setup better output for test output
    when(ground.toString()).thenReturn("Ground Square");
    when(wall.toString()).thenReturn("Wall Square");
    when(ghost.toString()).thenReturn("Ghost");
}
```

For better output in the tests we also changed the `toString()` methods for the different mock objects. This makes it so that instead of getting the output for example:

```
received: Mock Object #12345
expected: Mock Object #54321
```

We obtain the output:

```
recieved: Ground Square
expected: Wall Square.
```

Our first couple of tests focus on the `parseMap(char[][])` function. We first test if the the map we passed has the correct number of Dimensions. To do this we verify that the `level.BoardFactory.createBoard()` method has been called with a specific argument. We cannot use the return value of the `parseMap(char[][])` function as that only returns a `level.Level` which is created by the `level.LevelFactory` and requires output of the `board.BoardFactory.createBoard` as both of these functions have been mocked, we would have to recreate the level creation in our code which does not make sense. Therefore we use the argument that was passed to `level.BoardFactory.createBoard()` as our test result. Here we check the size of the grid and compare it to the what we expect with the `argThat()` method. This does work correctly, however does not output a nice error message on failure.

```
/**
 * Tests if the map parsers creates a grid with the correct width
   and height.
 */
@Test
void testIfMapHasCorrectDimensions() {
    parser.parseMap(new char[][] {{' '}, {' '}});

    verify(boardFactoryMock, times(1)).createBoard(any());
    verify(boardFactoryMock).createBoard(argThat(
        (Square[][] grid) -> grid.length == 2 && grid[0].length ==
           1 && grid[1].length == 1)
```

```
    );

    verify(boardFactoryMock, times(2)).createGround();
}
```

We then test all the different possible characters that could be as the map configuration. For this we check that the correct functions are called from the `level.LevelFactory` and `board.BoardFactory`. For example for the pellet:

```
/**
 * Tests if the map parsers calls the pellet creation correctly on
    '.'.
 */
@Test
void testIfMapCreatesPelletCorrect() {
    parser.parseMap(new char[][] {{'.'}});

    verify(boardFactoryMock, times(1)).createGround();
    verify(levelFactoryMock, times(1)).createPellet();
    verify(pellet, times(1)).occupy(ground);
}
```

This verifies that the ground was created where the pellet is and that the pellet was created. It also checks that the pellet's `occupy()` function was called.

For the Ghost for example we also check that the ghost list passed to the level creation is correct. This is also very similar to how we check if the player starting positions list is checked.

```
List<Ghost> ghosts = new ArrayList<Ghost>() {
    {
        add(ghost);
    }
};
[...]
verify(levelFactoryMock, times(1)).createLevel(
    any(),
    eq(ghosts),
    any()
);
```

For the `parseMap(List<String>)` function we passed it a list of strings and checked if the expected grid is passed to the `level.BoardFactory.createBoard()` function. We know because of the other tests, that if they pass and this one doesn't that there must be an error in this function.

```
/**
 * Tests if the map parser correctly converts the map from a list
    of strings to char array.
 */
@Test
void testIfMapIsCorrectlyConvertedFromStringListToCharArray() {
    parser.parseMap(new ArrayList<String>() {
```

```
        {
            add(" #");
            add("# ");
        }
    });
    final Square[][] expected = new Square[][]{
        {ground, wall},
        {wall, ground},
    };

    verify(boardFactoryMock).createBoard(eq(expected));
}
```

The `parseMap(InputStream)` was a lot more tricky. Here we had the problem that an `InputStream` is converted to a `Bufferedstream` and we thus didn't know exactly which functions get called that we could mock. In the end we decided upon using a `ByteArrayInputStream` which we can fully control and pass it a string. Again we use the hierarchy for the tests, this means if this test fails, but the others we described before pass, then we know this function has a problem.

```
/**
 * Tests if map is correctly read and created from input stream.
 * @throws IOException Could throw an error if the test is written
    incorrectly
 */
@Test
void testIfMapIsCorrectlyReadFromInputStream() throws IOException {
    InputStream anyInputStream = new ByteArrayInputStream("
       #\n##\n  ".getBytes());

    final Square[][] expected = new Square[][]{
        {ground, wall, ground},
        {wall, wall, ground},
    };

    parser.parseMap(anyInputStream);

    verify(boardFactoryMock).createBoard(eq(expected));
}
```

Lastly the good weather cases for `parseMap(String)` were required. Here we just have to pass a name of a file and the file will be opened and parsed. For this there was already a example file we could use in the test folder: "simplemap.txt". We used this for the test and checked for the correct result in the same way as the previous tests. Finally to get 100% coverage we also tested if the getter for the Board was correct.

4.2.2 Exercise 21

One of the bad weather cases is what if the map parser is given an invalid character that has not been defined yet. For this we just passed it an invalid character: 'E' and checked if a `PacmanConfigurationException` was thrown:

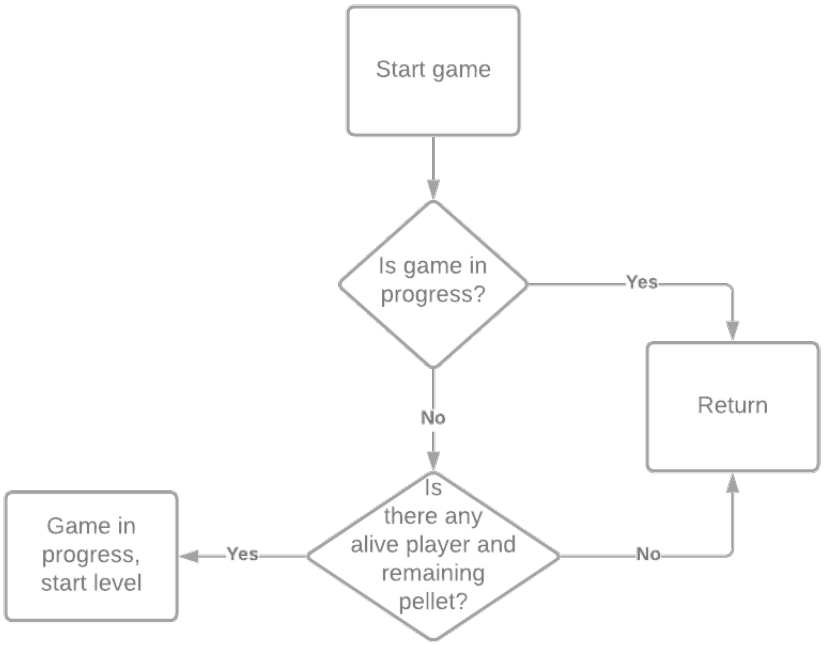
```
/**
 * Invalid character passed to map parser should throw
    PacmanConfigurationException.
 */
@Test()
void testIfWrongCharacterThrowsException() {
    assertThrows(PacmanConfigurationException.class,
        () -> parser.parseMap(new char[][]{{'E'}}));
}
```

Another case is that the map parser cannot be passed an null value. This should also throw an `PacmanConfigurationException`. We also checked if an empty list passed to the `parseMap(List<String>)` function also throws an `PacmanConfigurationException`. Another case we checked was if a list of empty strings was passed to the `parseMap(List<String>)` function also throws an `PacmanConfigurationException`. Another more likely case we tested was, if a map has different sizes for each row or column, that this also throws an `PacmanConfigurationException`. A common mistake with reading a file is that the path to the file does not exist. To test this we tried parsing a map with a path that never should exist. There are many more bad weather cases we didn't cover. For example what happens if one of the `level.LevelFactory` or `board.BoardFactory` is null. There would be a `NullException`, this is however very unlikely as you would have to create the `level.MapParser` without passing it the `level.LevelFactory` and `board.BoardFactory`, which technically isn't allowed. This means you would have to deliberately pass it uninitialized variables or null. There are also many different `IOException`'s that could occur, these are however quite hard to test, as we have no direct access to the `BufferedReader`. With our test suite the `level.MapParser` has a line coverage of 100%.

4.3 Branch Coverage with Mocks

4.3.1 Exercise 22

The method `Game.start()` method controls the logic, whether the game continues or stops. If the game is already in progress, method returns and if the game is not in progress method checks if the game can be started. To start the game, there has to be at least one player alive and there must be at least one pellet on the board. If these requirements are filled, `Level.start()` is called.



In the `game.GameUnitTest` we have mocked player, level and point calculator in order to create a single player game and to have controllability and observability on this game instance.

First, we have tested if game stops (not in progress anymore) when there is no player on the level, even though there is still a pellet. Then, we checked if game stops when there is no more pellet to consume on the level, however there is a player alive. Finally, we tested if level starts, when there is a pellet and a player alive. On the last test, we called the method `Game.start()` again, to have %100 branch coverage, so that we can test if method returns when the game is in progress.

In this exercise, we have have %100 branch coverage, but we don't have %100 condition coverage. Because on the second branch, we have an aggregated condition. The aim of the exercise is to maximize the branch coverage. As a result of that we did not test the second branch with different combinations.

4.4 Testing Collisions

4.4.1 Exercise 23

We have created Table 1 based on the scenarios in the document `doc/scenarios.md`.

Collider	Pacman	Pacman	Pacman	Ghost	Ghost
Collidee	Pellet	Wall	Ghost	Pellet	Pacman
Consequence	Pellet is consumed	Pacman does not move	Pacman dies	Pellet not visible anymore as long as ghost is on square	Pacman dies

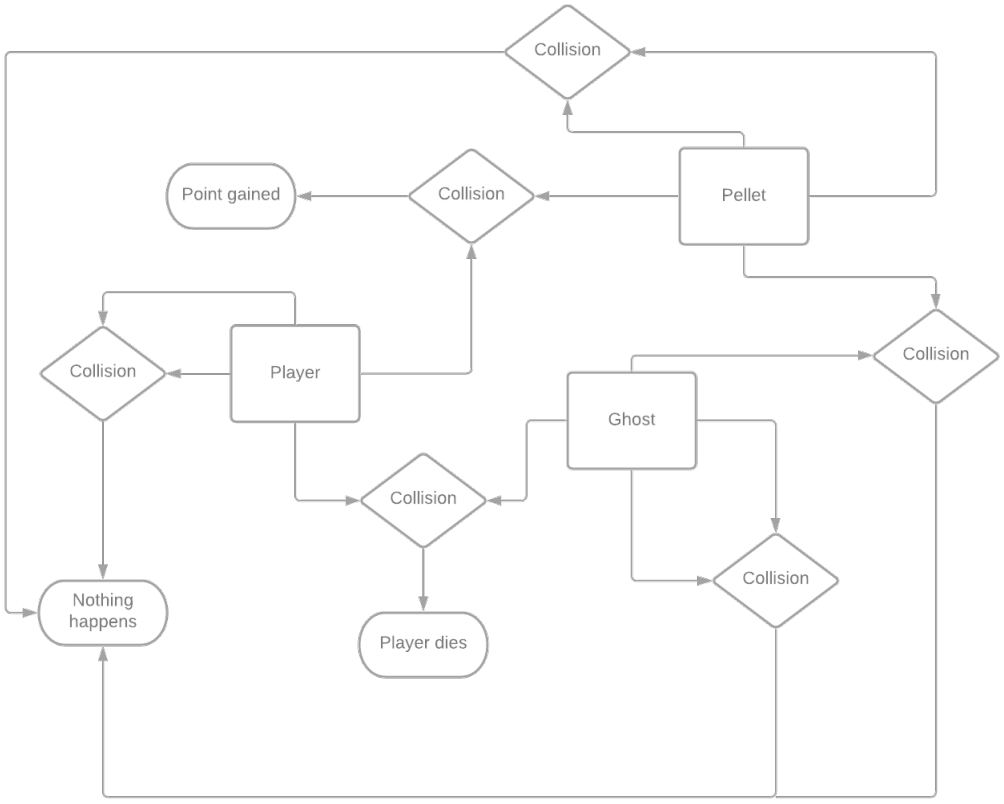
Table 1: Decision table for collisions of Pacman game

4.4.2 Exercise 24

In the test suit that we have implemented, we did not think about only good weather cases, that are described in the Table 1. We tested all possible collision combinations between player,

pellet and ghost which also contains collisions with themselves. We considered a wall as a collision object and defined a collision with a player in the Table 1, however as a result of a wall not being a unit, we can not test its collision and in the source code, there is no defined collision of a wall.

In the code, there are three different consequences defined as consuming of pellet, death of player or none. Therefore we have implemented those cases as separate methods.



Even though, the output does not change based on the type of the collider and collidee, we tested in both ways. For example, ghost colliding into a player and player colliding into a ghost result same. To check if necessary functions are triggered correctly, we tested both of these collisions.

On other cases, which are very unlikely to happen, such as pellet colliding into a pellet, we checked if none of these consequences take place.

You can see the collisions that we tested in the figure above.

4.4.3 Exercise 26

The line coverage in the original tests we were given, was 0% for the more complex `CollisionMap`: `CollisionInteractionMap` and `DefaultPlayerInteractionMap`. For the `PlayerCollisions` we had 78%. We increased the `PlayerCollisions` and `DefaultPlayerInteractionMap` both to 100% line coverage and increased the `CollisionInteractionMap` to 94% line coverage (48/51). The cases that were missing in the `PlayerCollisions` in the default tests provided, were the ghost colliding with the player and the pellet colliding with anything (for example the player). The cases we are still missing are the bad weather case that the `colliderKey` is set to null, the `collisionHandler` is set to null and the case were something is an assignable interface. These

cases are all very specific to the `CollisionInteractionMap` and would require specific tests for that class, that would not fit well the `PlayerCollisions` class we created.

4.4.4 Additional Information

We had to ignore the PMD warning: "PMD.JUnitTestsShouldIncludeAssert", in our abstract class, because we make use of functions that we can call like "gained points" to not have to repeat the same verify statements. However PMD does not see these and assumes we have tests, with no assert statements.

4.5 Pragmatic testing

4.5.1 Exercise 27

First of all the way the current code is written it is not very testable. Because `Random()` is instantiated in the function, there is no way to set the seed. It is more testable and efficient to have one Random variable that your program uses, and gets the next int from. If this is the case, then we can set the seed for the test and always get the same random numbers, as this is only a pseudo random number generator, which is basically only a big mathematical function.

4.5.2 Exercise 28

This `LauncherSmokeTest.smokeTest` depends on a lot of different things. We load a map, we open the launcher, etc. All things that are dependent on the operating system, which can fail at any point (file not readable for example). Secondly we use the `Thread.Sleep()` which is generally a bad idea. We do not know when our code runs, and we could sleep longer than the time we put there. We also do not know how fast the other threads are in the meantime. It also slows down the test execution. A Test can become Flaky, when we depend on external infrastructure which can behave differently with each test run, if we share the same resources (race condition) between threads. If for some reason a request times out, the test can fail. If another test is interacting with this test, due to dependencies, we also have a flaky test. We can avoid having a flaky test, by mocking external infrastructure and other threads. We also should always avoid having a test be dependent on another test.

4.5.3 Exercise 29

Code coverage defines the percentage of the lines that are hit, when test suite runs. However, the code that is covered might also remain untested. This is why sometimes code coverage can be misleading. For example we can call each function in our code once, to obtain 100% method coverage.

But it is a nice indicator to track the improvement when writing test suites and see how the test development is going. It also shows quickly at a glance if specific parts of a program have been reached by tests. However code coverage can become the goal of the project which leads to employees not writing good tests, but tests that cover the code well. When the metric becomes the goal it is detrimental for the project. Metrics like code coverage should be used with a goal in mind and not become goal.

4.5.4 Exercise 30

Sometimes your test can become quite complicated if you need to mock a lot of different classes that depend on each other and all need different functions to be mocked. This can also become a detriment when the tests become very specific to the implementation. When the implementation is then changed, then a lot of the test implementation needs to be changed, which costs a lot of effort. Also when there is too many things to arrange, such as classes that depend on each other, setup code might become complicated.

4.5.5 Exercise 31

If you do not mock dependencies to external resources. Sending network, file and system requests are all much slower then calculating on the CPU. Therefore we should mock such dependencies to not have our test take longer than necessary. When there are two classes under test, and an mocked object that works slower than those classes, this can result slowing down the entire test suite.

4.5.6 Exercise 32

You should use mocks for the classes in unit tests. However mocking simple classes can worsen the speed of the test. For integration tests, mocking internal classes does not make much sense, but mocking external dependencies does, as these take a long time to run. The reason we do not mock the internal classes, is that we want to test the interaction between two classes in an integration tests. For system tests, we wouldn't mock anything, as here we want to test the system as a whole.

4.5.7 Submit Part II

With our new classes we increased the coverage even more again. We split the exercises between us and each did our own exercise in a separate branch. In the end we merged the branches. The integration server helped us remove check-style errors that we didn't notice while working locally by immediately notifying us that we should change them. For example we noticed we made some breaking changes in our code. Previous tests were failing, because we had no player anymore. The reason was that we changed the `GhostMapParser`, so we parsed 'P' as Pinky, which removed the player start from the previous tests. This was a an excellent example of why tests are useful and CI Pipelines help your workflow. We didn't always commit changes immediately when they happened, however we will try to improve this for the next part. We usually have committed after each exercise. It would be better if we actually commit after every modification such as adding a new function or changing access modifiers of attributes of the test suit. Some of our commits were too large to easily see what has changed and what was wrong, therefore tracking the effects of those changes can be hard.

Universität Potsdam

Name: **Deniz Mert Tecimer, 811870 / Günther Wullaert, 780827** Labwork: **3**
Course: **Software Testing** Date: April 21, 2022

5 Part III: System tests, model and state-based testing

5.1 System Testing

5.1.1 Ex 33

We adhered for our integration test strongly to the user scenario as is apparent by the comments. To check that players cannot move again, we issue move commands and see if the player did not move. To check if ghosts are moving would require us to sleep and thus making the test flaky. This is because there is no way to access the ghosts directly without extensive mocking.

```
@Test
public void gameCanBeSuspended() {
    // Check the game is running, given the game has started
    assertThat(getGame().isInProgress()).isTrue();

    // Suspend the game, player clicks the "stop" button
    getGame().stop();

    // Assert that all moves from ghosts and the player are
       suspended.
    assertThat(getGame().isInProgress()).isFalse();

    // Check if players cannot move anymore
    for (Player p : getGame().getPlayers()) {
        Square sq = p.getSquare();
        getGame().move(p, Direction.NORTH);
        assertThat(p.getSquare()).isEqualTo(sq);
    }
}
```

5.1.2 Ex 34

We used a "before" method to not repeat ourselves before each test. This provided us the ability to change the map for each test if necessary. We could use the same map for the scenarios 2.1,

2.2 and 2.3 though by just surrounding the player with an empty square, a pellet and a wall. This made it very easy to just issue a move command and then checking everything worked correctly. We again tried to make the test very similar to the scenario as can be seen in the comments.

```
@Test
public void playerMovesOnEmptySquareTest() {
    before("/player_movements_map_test.txt");

    //Given the game has started
    assertThat(getGame().isInProgress()).isTrue();

    Player p = getGame().getPlayers().get(0);
    assertThat(p.getScore()).isEqualTo(0);
    Square squareNextToPlayer =
       p.getSquare().getSquareAt(Direction.NORTH);

    //and  my Pacman is next to an empty square;
    assertThat(squareNextToPlayer.getOccupants().size()).isEqualTo(0);

    //When  I press an arrow key towards that square;
    getGame().move(p, Direction.NORTH);

    //Then  my Pacman can move to that square,
    assertThat(p.getSquare()).isEqualTo(squareNextToPlayer);

    // and  my points remain the same.
    assertThat(p.getScore()).isEqualTo(0);
}
```

5.1.3 Ex 35

To test the scenarios 2.4 and 2.5 is a bit harder, because we are trying to test if the game correctly identifies that the player has lost or won the game, which is not saved in any variable or returned by a function. The solution is that we can use the same framework that the game itself is using to observe if the game was won or not with the Level.LevelObserver class. We extended that class and saved it in variables if we observed a win or not. We then have getters to access those values in a test.

```
public class TestObserver implements Level.LevelObserver {
    private boolean observedWin = false;
    private boolean observedLoss = false;

    @Override
    public void levelWon() {
        observedWin = true;
    }

    @Override
```

```
    public void levelLost() {
        observedLoss = true;
    }

    /**
     * If we observed a win.
     * @return true if we observed a win, false if not.
     */
    public boolean isObservedWin() {
        return observedWin;
    }

    /**
     * If we observed a loss.
     * @return true if we observed a loss, false if not.
     */
    public boolean isObservedLoss() {
        return observedLoss;
    }
}
```

in the test we then just have to instantiate that observer and register it to the current level:

```
TestObserver observer = new TestObserver();

getGame().getLevel().addObserver(observer);
```

And we can then assert if a game was lost later in the test:

```
assertThat(observer.isObservedLoss()).isTrue();
```

5.1.4 Ex 36

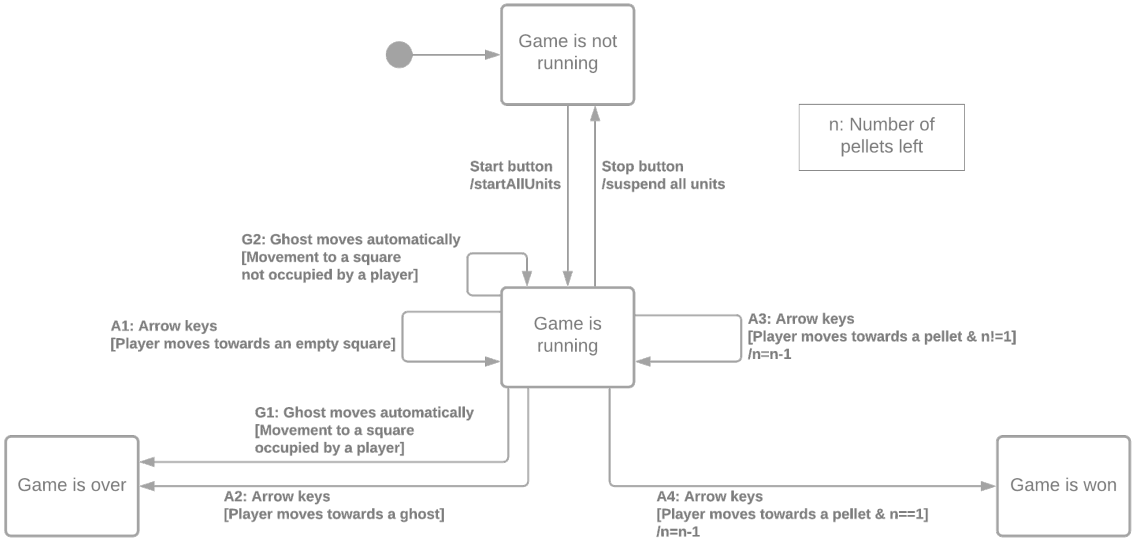
To actually test that the player lost, we can setup a ghost next to the player and move the player into it. To test that the player has won, we can only have one pellet on the board and let the player eat it. We again did the same, like for the last scenarios, so that the test looks very similar to the outlined user story scenario.

5.1.5 Ex 37

For User Story 3 it is even harder to test. The problem lies in the control-ability and observe-ability of the monsters. Because the NPCs are all stored in private variables with no public way of accessing them, it is very hard to know where they are and control there specific movements. To access them you would have to use a lot of mocking. A possibility we could also think of was to go through each tile of the board and check its occupants to get the reference to the ghost and its position. Controlling the ghost is however also very difficult as the ghost moves on it own, based on an timer which is not easily control able. This makes it very hard to test full scenarios which involve controlled behavior of the ghost.

5.2 State Machines

5.2.1 Ex 38

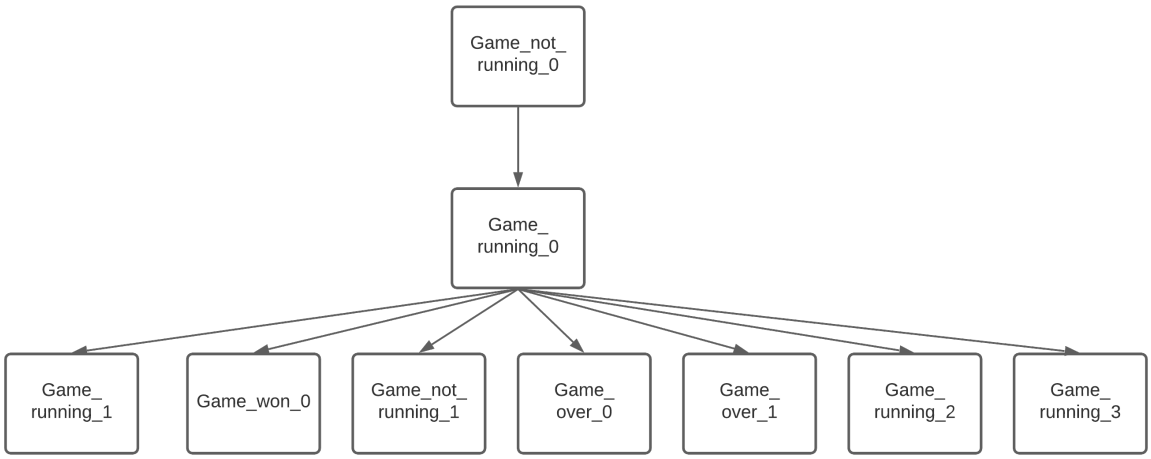


We first defined the initial state as GUI launched as described in scenarios.md file and added a state as game paused. However, later on, we found out that the GUI launched is just a different type of game paused. Thus, we merged these two states as game is not running, so that we simplify our chart, because they both have the same transitions and actions. In order to simplify distinguishing between triggers, we used short codes, such as A1 to refer the first trigger event of the player with arrow keys and G1 to refer the trigger event of the ghost.

As default, we start with the state game not running. When the user presses the start button, game starts all units and the state changes as game running. Game running could be defined as a super state, that includes the behaviors of the player and the ghosts separately. But because we wanted to focus on the states of the game in terms of being on progress, we did not used that approach and showed these transitions and actions with only game running state.

Depending on the conditions, if a ghost or a player moves towards an empty square, state does not change as in transitions A1, G2. Although, if player and ghost collide on each other as in transitions G1, A2, then the state changes as game is over. Until the last pellet is consumed, consuming a pellet does not lead a state change and game is still running as shown in A3. In contrast to that, when the last pellet consumed, the state changes as game is won, as shown in A4. All player transitions are triggered by its move via pressing an arrow key and might cause a state change depending on the conditions. But a ghost is controlled automatically, which is triggered by a tick event and also can cause a state change according to the conditions.

5.2.2 Ex 39



We derived a transition tree from the state chart, that we explained above. Thanks to that tree, we can now have a vision on non-recurrent paths that can be taken, starting from the initial state to other states. For example, a scenario can be interpreted from this tree: We start with the initial state as game not running and move to the state game running by pressing the start button, to the game winning state by eating the last pellet. This path is shown in the tree as the following path; Game_not_running_0, Game_running_0 and Game_won_0. From this tree, we can also derive seven good weather test cases with allowed transitions.

5.2.3 Ex 40

States	Events							
	start button	stop button	A1	A2	A3	A4	G1	G2
Game not running	Game running							
Game running		Game paused	Game running	Game over	Game running	Game won	Game over	Game running
Game won								
Game over								

Table 1: State (transition) table of JPacman game

We derived a transition table from the transition tree that is explained above to examine the interactions between events and states. Here, it is also easier to have an overview on the possible bad weather cases. The end states, as shown, has no transition to another state. To shorten the trigger events, we used the short codes that we introduced on the chart explanation.

5.2.4 Ex 41

We defined states and transitions as enums. This gives us the benefit, that we are not subject to typos and the autofill can let us develop code more quickly. As we learned in the lecture we used inspection methods and trigger methods. The inspection method is there to inspect the current game state and return us in which state we currently are. This we can then use in the tests to assert that we are in the correct state, without having to duplicate code for checking the game state everywhere.

```
public State inspectCurrentState() {
```

```
    if (game.isInProgress()) {
        return State.game_running;
    } else if (testObserver.isObservedLoss()) {
        return State.game_over;
    } else if (testObserver.isObservedWin()) {
        return State.game_won;
    } else {
        return State.game_not_running;
    }
}
```

The trigger function is then used in the tests to trigger an event. Basically we check what our current state is and then based on that trigger the correct action which should correspond to the correct transition in the graph. These can be simple actions like pressing the start button or more complex actions where we move the player in a specific direction.

```
public void trigger(Transition transition) {
        if (currentState == State.game_not_running && transition
           == Transition.start) {
            game.start();
        } else if (currentState == State.game_running) {
            gameRunningTrigger(transition);
        }  else if (currentState == State.game_won && transition
           == Transition.start) {
            game.start();
        } else if (currentState == State.game_over && transition
           == Transition.start) {
            game.start();
        }
    }
```

An example test case is shown below. First we set the map, that is created for the test scenario. Then, we check if the game is in its initial state at the beginning. Further more, we trigger a transition and check the state, the game is currently in, is the intended state and we trigger another transition and repeat the assertion phase. With the same method, we went through all seven paths derived from the tree.

```
@Test
public void test_transitions_start_stop() {
    before("/state_machine_test.txt");
    assertThat(currentState).isEqualTo(State.game_not_running);
    trigger(Transition.start);
    currentState = inspectCurrentState();
    assertThat(currentState).isEqualTo(State.game_running);
    trigger(Transition.stop);
    currentState = inspectCurrentState();
    assertThat(currentState).isEqualTo(State.game_not_running);
}
```

As said before, transition table can also be used to derive bad weather test cases to test unauthorized transitions. However, we already had tested those bad weather cases in unit test

section. And in order to have more controllability and observability, it would be more logical not to test them under system testing.

5.3 Multi-Level Games

5.3.1 Ex 42

For the user story of handling multiple levels we looked at how it works on the arcade machines for Pacman. If the player wins he should be able to go to the next level by clicking start. If he loses we should restart the game if he presses start.

Story 5: Handling Level Transitions

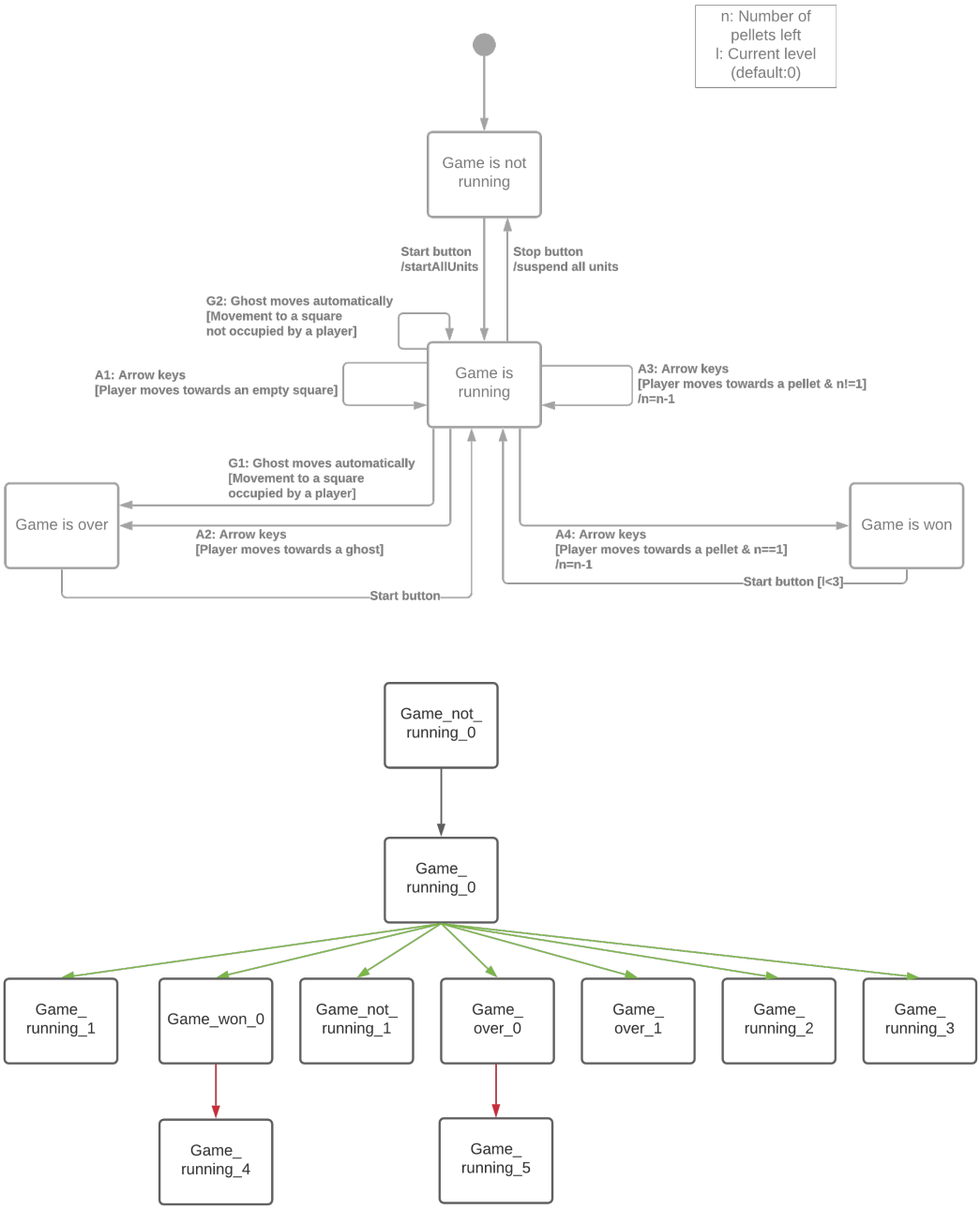
```
As a player,
 I want to be able to continue playing the game after a win or loss;
So  that I can continue playing and have fun.

Scenario S5.1: Restart the game.
Given the game is running;
When  the player loses the game;
Then  the game should be reset to its initial position with its initial map.

Scenario S5.2: Continue the game.
Given the game is running;
When  the player wins the game;
Then  the next level in a list of levels is loaded and the game is started again.
```

5.3.2 Ex 43

To accommodate, for this change, we had to add 2 new transitions. Previously, if you were in the Game is Over or Won state you could not go to any other state. With the new transition from the Game is Over state, you can, by triggering the start button event, go back into the game is running state. As long as the current level is not larger than or equal to 3. The new transition tree is very similar, but in two cases we are following the transition back to the game is running.



5.3.3 Ex 44

We highlighted the branches that didn't change to green and the branches that are new or are changed in red. With this color coding you can easily see that if you have a test, which has a completely green path, the test does not need to be changed. If however you have red paths, than the test needs to be changed. This means the test that we need to change is the test were we go with A4 into the Game_won state, now this needs to continue with a start event back into the Game_running state.
The other test that needs to change is the test were we go with A2 into the Game_over state, now this needs to continue with a start event back into the Game_running state.

5.3.4 Ex 45, 46

MultiLevelGame and MultiLevelLauncher completely mirror what game and the launcher classes are doing. There were a few exceptions though, we added in the GameFactory class an extra function which returns a MultiLevelGame, which is instansiated:

```
public MultiLevelGame createSinglePlayerGameWithMultipleLevels(
    Level level, PointCalculator pointCalculator) {
    return new MultiLevelGame(playerFactory.createPacMan(), level,
       pointCalculator);
}
```

Which we then called from a function we have overwritten in the MultiLevelLauncher:

```
@Override
public MultiLevelGame makeGame() {
    GameFactory gf = getGameFactory();
    Level level = makeLevel();
    multiGame = gf.createSinglePlayerGameWithMultipleLevels(
        level, new PointCalculatorLoader().load());
    return multiGame;
}
```

5.3.5 Ex 47

For the tests to be run for both the Launcher and the MultiLevelLauncher, we did the exact same as in the previous labwork. We made it so that the MultiLevelLauncherTest extends from the StateMachine (LauncherTest). With a simple getter and setter, we can then set which launcher we actually want to use and we can add MultiLevelLauncher specific tests to the MultiLevelLauncherTest test class.

5.3.6 Ex 48

The tests look very similar to the tests we did in 5.2 Just with the new transitions tested.

```
@Test
public void test_transitions_start_a2_start() {
    before("/state_machine_test.txt");
    assertThat(getCurrentState()).isEqualTo(State.game_not_running);
    trigger(Transition.start);
    setCurrentState(inspectCurrentState());
    assertThat(getCurrentState()).isEqualTo(State.game_running);
    trigger(Transition.a2);
    setCurrentState(inspectCurrentState());
    assertThat(getCurrentState()).isEqualTo(State.game_over);
    trigger(Transition.start);
    setCurrentState(inspectCurrentState());
    assertThat(getCurrentState()).isEqualTo(State.game_running);
}
```

We had to however add new triggers, which basically only call the game.start() method.

```
else if (currentState == State.game_won && transition ==
   Transition.start) {
    getGame().start();
} else if (currentState == State.game_over && transition ==
   Transition.start) {
    getGame().start();
}
```

5.3.7 Ex 49

To make the tests work, the launcher needs to keep track of which level we are currently in and how many and which levels exist for the game.

```
private final String[] levels = {"/board.txt", "/board2.txt",
   "/board3.txt"};
private int currentLevel = 0;
```

If we then need to have access to the current level map we can with these two variables return the correct one.

```
@Override
protected String getLevelMap() {
    return levels[currentLevel];
}
```

We found out that it is hard to reset the GUI without making changes to the launcher or the PacManGUI without relaunching the window. We decided to let the user after he has lost or won the game, restart the launcher with the correct game loaded and started when he presses the start button, like it is shown on the Transition Graph. There is however one small little limitation of this approach: The input is not accepted in the new window until you focus on the window. Focusing on the window only works by clicking a button, so you either have to click start again or click stop shortly if you do this in person.
Because the new transitions happen if we have won or lost the game, we can override the functions that are called by game in case of these events. Here we save in MultiLevelGame that we won the game or lost it, so we can reference it later in the start method, which triggers the change.

```
@Override
public void levelWon() {
    currentLevelWon = true;
    super.levelWon();
}

@Override
public void levelLost() {
    currentLevelLost = true;
    super.levelLost();
}
```

In the start method we can then check if we have lost or won and the game is not in progress. However we cannot change the level from the game itself, so we notify the launcher that we

want to be restart, or go to the next level. This is done with an Runnable, where the launcher can register itself to the Runnable on creation of the game.

```
@Override
public void start() {
    if (!isInProgress() && currentLevelLost) {
        currentLevelLost = false;
        this.restartGameRunnable.run();
    }
    if (!isInProgress() && currentLevelWon) {
        currentLevelWon = false;
        this.loadNextLevelRunnable.run();
    }

    super.start();
}
[...]
public void registerRestartRunnable(Runnable restart) {
        restartGameRunnable = restart;
}
```

In the launcher we then pass the following functions on game creation with the registerRestartRunnable(Runnable) function. These handle then the actual level switching, by setting the currentLevel variable to the correct value and disposing the launcher and restarting it with the correct dimensions. Finally starting the game immediately. This also handles the case were when we are at the last map, we cannot do more levels.

```
private void restartGame() {
    currentLevel = 0;
    dispose();
    launch();
    getGame().start();
}

private void loadNextLevel() {
    if (currentLevel + 1 >= levels.length) {
        return;
    }
    currentLevel += 1;
    dispose();
    launch();
    getGame().start();
}
```

5.3.8 Ex 50

MultiLevelLauncher: 92% line coverage
MultiLevelGame: 100% line coverage

GameFactory: 83% line coverage

The MultiLevelLauncher had one line which was not covered by our tests. This was the if statement that checked if we reached the last map and if so, does not load the next level. This is not covered by the tests suggested by our state tree. To cover this transition, we added another tests, which tests this branch. In hindsight this should have been an extra transition in our graph. We think, that then would have been in the transition tree. The MultiLevelLauncher does not have 100% line coverage, because we put a main method in the class. This main method can be used to start the whole project just like in the launcher. This is however not tested, as you do not call a main method from a test. This main method is also what you would call too simple to test. As the main method, it only instantiates a MultiLevelLauncher and launches that launcher.
MultiLevelGame is fully covered and everything we added to the GameFactory, is still covered. However there are some unused lines that are not covered in the GameFactory, which make the GameFactory not fully covered. This is however from before we started our project.

5.4 Submit

5.4.1 Ex 51

With this project we increased our coverage of the codebase substantially. The tests we implemented were all interesting and brought with each their own sets of challenges. Due to this project we could see how we can apply the test practices we learned in the lecture to a real project. We learned also new ways to write tests. We didn't know about using inheritance in tests, for example to run the same tests for different classes. By having to use git for handing in our code, we refreshed our skills and got in the habit of committing more often.

One of the most annoying things is the heavy privatization of the variable fields without many getters to access them. The biggest one that comes to mind is that there is no way to access the ghosts. The solution would be to have a getter in the level that lets you access the ghosts, there is no reason why an immutable list of the ghosts shouldn't be allowed to be accessed outside of the level, if the players are allowed to be accessed. This would make writing tests much easier.
The ghost scheduler is also not very test friendly. We found this to be irritating where we had to put sleep timers in our tests so that we can wait for the ghost to at a random point in time. The solution would be that you can extend or even overwrite which scheduler is used, so that you can decide when the ghost AI is triggered.
The check style configuration was also counterproductive for integration or system tests. The limit of only 5 asserts is way to low for these more complex tests. If the state tree becomes a little larger it is just impossible to correctly test with only 5 asserts. Here the solution is very simple you would just have to edit the configuration to be a larger number, or completely remove the rule.

The code however is written very modular and extending the codebase, with for example the MultiLevelLauncher was very easy and didn't require many changes.
The collision system with the CollisionInteractionMap is a very interesting way of handling collisions and is again a good showcase how easy extending this Pacman framework is.
Writing the tests for the ghosts was also very interesting and where like little puzzles you had

to solve. In the tests that resulted out of it are very cool ways to test the specifics of each AI we tested.