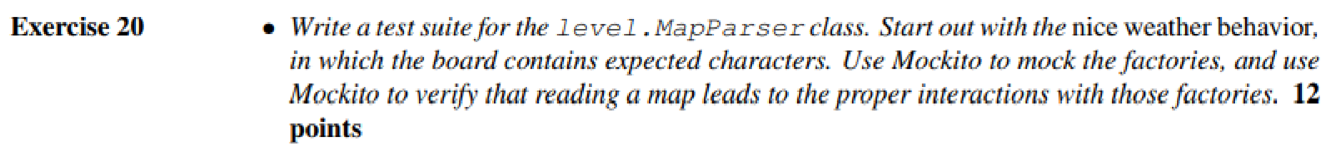
Students

- Eric Jarosch - 227271
- Lukas Fritzsche - 227771

Exercise 20



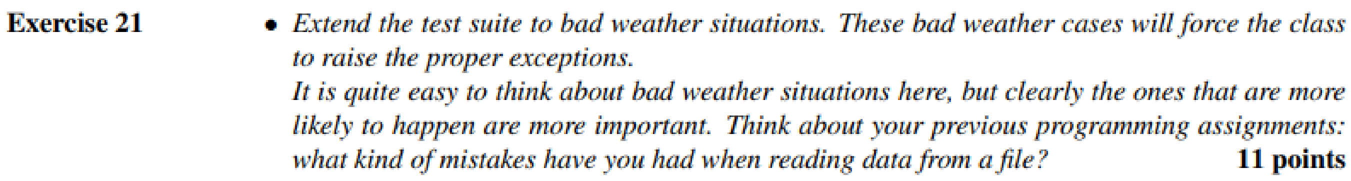
Exercise 20

- *Write a test suite for the* `level.MapParser` *class. Start out with the* nice weather behavior, *in which the board contains expected characters. Use Mockito to mock the factories, and use Mockito to verify that reading a map leads to the proper interactions with those factories.* **12 points**

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

"Nice Weather"-Test	Method under Test	Summary
test_parseMap_normalMap	parseMap(char[][] map)	Provides a typical Map and verifies that the Tiles `empty`, `wall`, `pellet` and `ghost` are created correctly.
test_parseMap_validPlayerPosition	parseMap(char[][] map)	Provides a typical Map and verifies that the Player is correctly placed in the world.
test_parseMap_validStringList	parseMap(List text)	Verify that the overload is working as expected.
test_parseMap_validInputStream	parseMap(InputStream source)	Verify that the overload is working as expected.
test_parseMap_validMapName	parseMap(String mapName)	Verify that the overload is working as expected.

Exercise 21



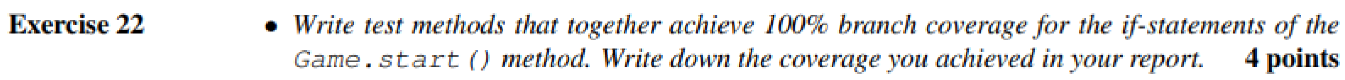
Exercise 21

- *Extend the test suite to bad weather situations. These bad weather cases will force the class to raise the proper exceptions.*
  It is quite easy to think about bad weather situations here, but clearly the ones that are more likely to happen are more important. Think about your previous programming assignments: what kind of mistakes have you had when reading data from a file? **11 points**

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

"Bad Weather"-Test	Method under Test	Summary
test_parseMap_emptyMap	parseMap(char[][] map)	Provides a completely empty map and verifies that it is handles correctly.
test_parseMap_invalidCharacter	parseMap(char[][] map)	Verify that an exception is thrown when maps contain unknown characters.
test_parseMap_invalidFormat	parseMap(List text)	Verify that invalidly formatted maps raise an exception.
test_parseMap_invalidMapName	parseMap(String mapName)	Verify that trying to parse a map that does not exist raises an exception.
test_parseMap_throwsIOException	parseMap(String mapName)	Verify behaviour of parseMap, when the InputStream overload throws an exception.

Exercise 22

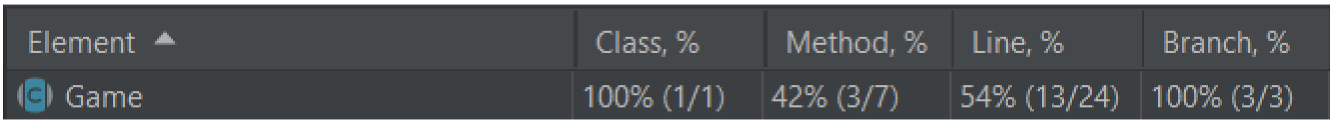


Exercise 22

- *Write test methods that together achieve 100% branch coverage for the if-statements of the* `Game.start()` *method. Write down the coverage you achieved in your report.* **4 points**

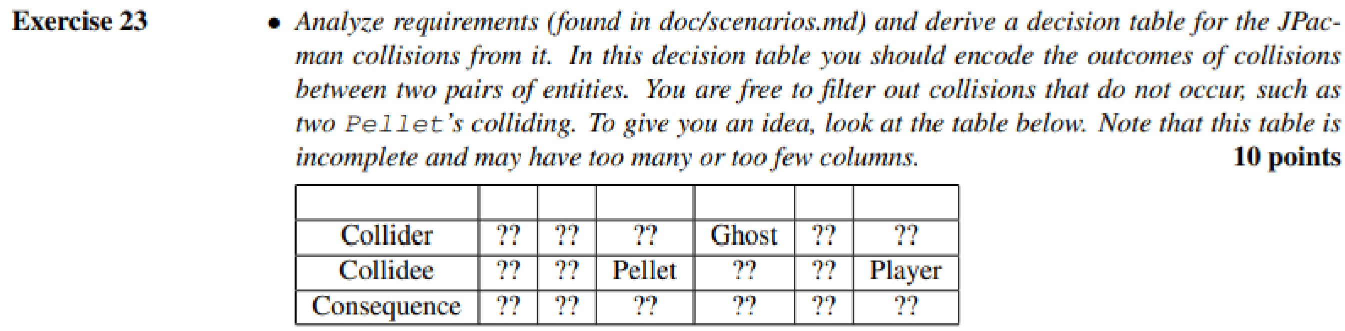
see added test class `GameTest`

or alternatively the relevant Merge Request



Element	Class, %	Method, %	Line, %	Branch, %
Game	100% (1/1)	42% (3/7)	54% (13/24)	100% (3/3)

Exercise 23



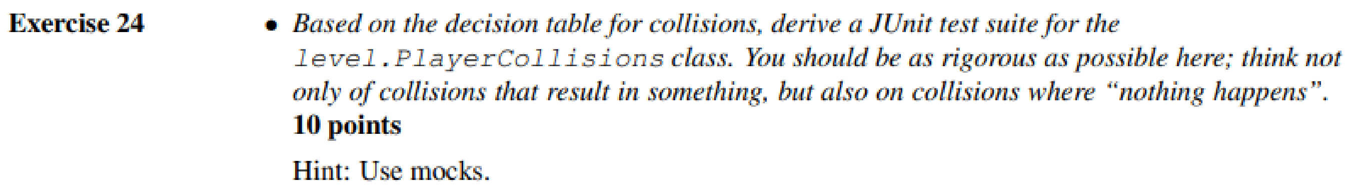
Exercise 23

- *Analyze requirements (found in doc/scenarios.md) and derive a decision table for the JPac-man collisions from it. In this decision table you should encode the outcomes of collisions between two pairs of entities. You are free to filter out collisions that do not occur, such as two* `Pellet`*'s colliding. To give you an idea, look at the table below. Note that this table is incomplete and may have too many or too few columns.* **10 points**

Collider	??	??	??	Ghost	??	??
Collidee	??	??	Pellet	??	??	Player
Consequence	??	??	??	??	??	??

Collider	**Collidee**	**Consequence**
Player	empty Square	points stay the same
Player	Wall	move is not conducted
Player	Player	players occupy the same square
Player	Pellet	earn Points, Pellet disappears
Player	Ghost	Pacman dies, Game over
Ghost	emptySquare	ghost occupies empty square
Ghost	Wall	move is not conducted
Ghost	Player	Game over
Ghost	Pellet	Pellet isn´t visible anymore
Ghost	Ghost	ghosts occupy the same square

Exercise 24



Exercise 24

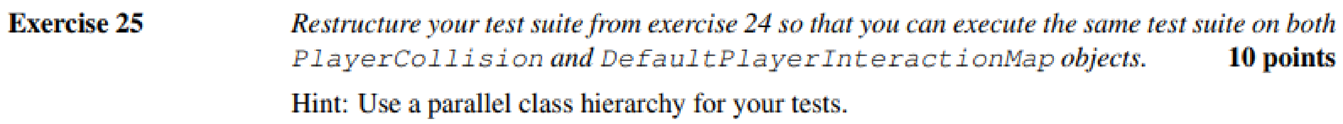
- *Based on the decision table for collisions, derive a JUnit test suite for the `level.PlayerCollisions` class. You should be as rigorous as possible here; think not only of collisions that result in something, but also on collisions where "nothing happens".* **10 points**

  Hint: Use mocks.

see added test class `PlayerCollisionsTest`

or alternatively the relevant Merge Request

Exercise 25



Exercise 25

Restructure your test suite from exercise 24 so that you can execute the same test suite on both `PlayerCollision` and `DefaultPlayerInteractionMap` objects. **10 points**

Hint: Use a parallel class hierarchy for your tests.

see modified test class `PlayerCollisionsTest`

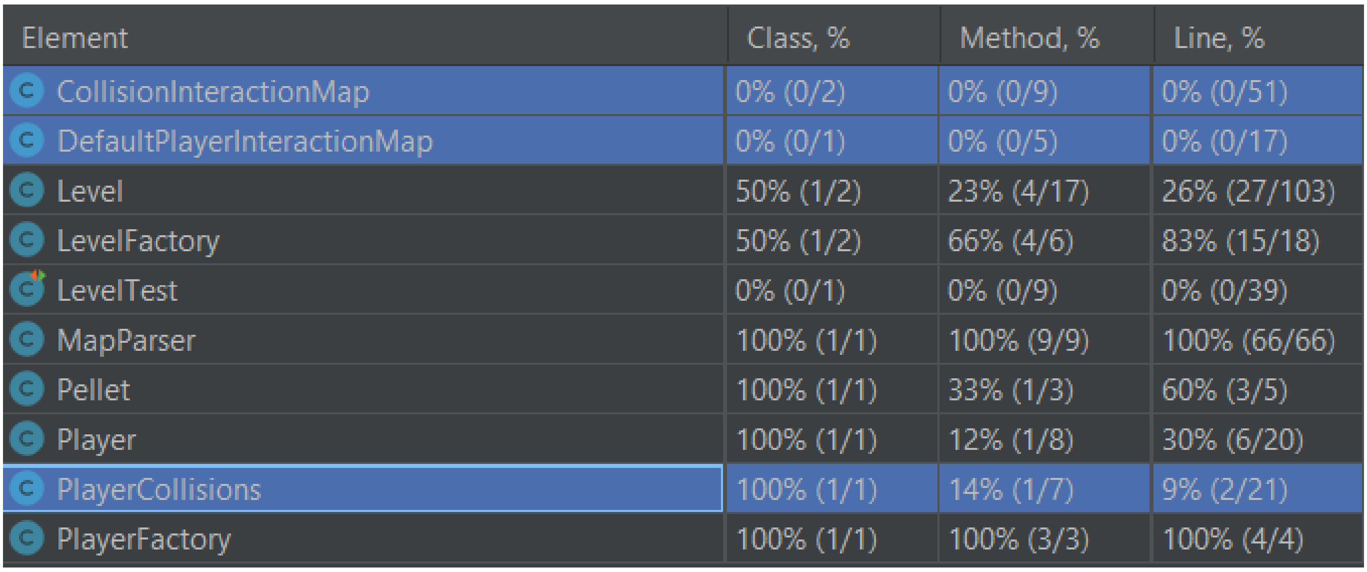
or alternatively the relevant Merge Request

Exercise 26

Exercise 26

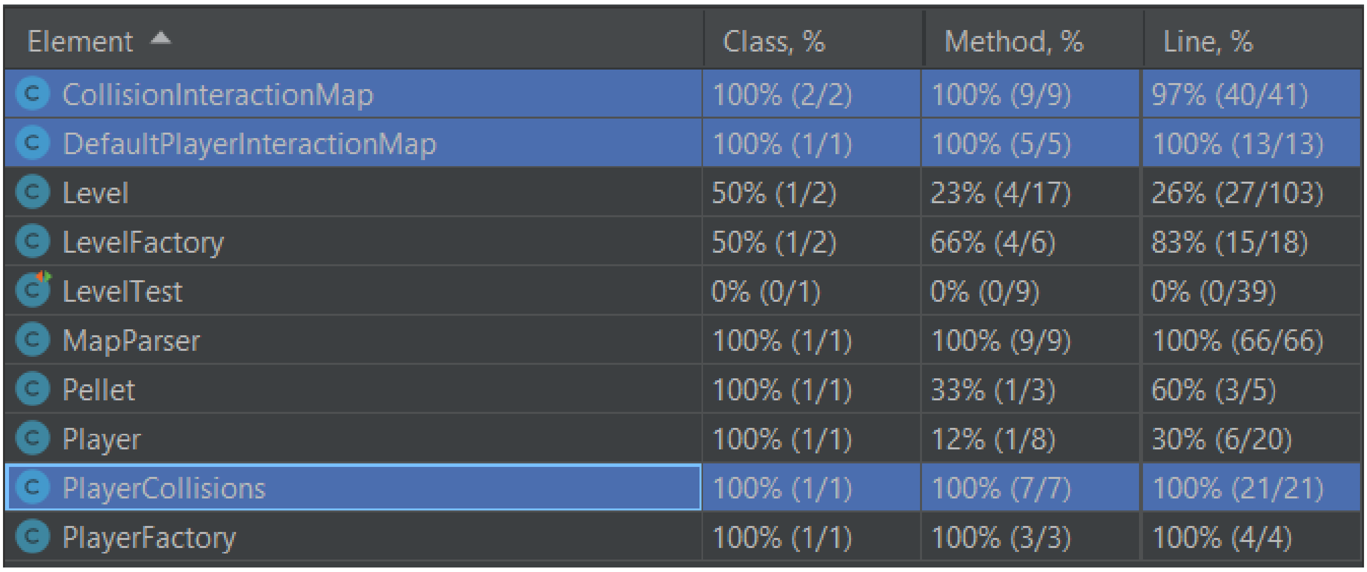
- *Analyze the increase in coverage compared to the original tests we gave you at the beginning, and discuss what collision functionality you have covered additionally, and which (if any) collision functionality is still unchecked.* **6 points**

Coverage before:



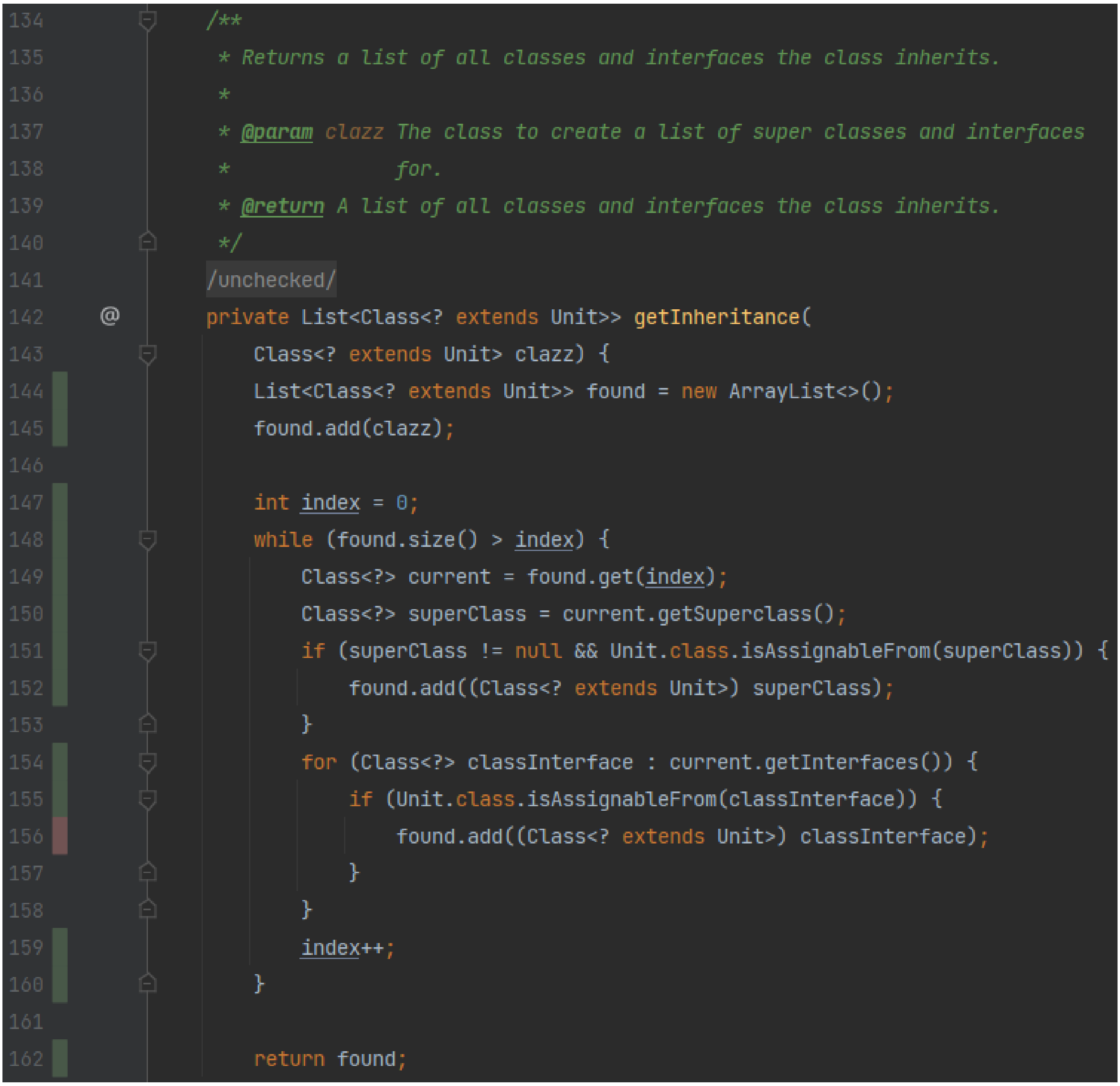
Element	Class, %	Method, %	Line, %
CollisionInteractionMap	0% (0/2)	0% (0/9)	0% (0/51)
DefaultPlayerInteractionMap	0% (0/1)	0% (0/5)	0% (0/17)
Level	50% (1/2)	23% (4/17)	26% (27/103)
LevelFactory	50% (1/2)	66% (4/6)	83% (15/18)
LevelTest	0% (0/1)	0% (0/9)	0% (0/39)
MapParser	100% (1/1)	100% (9/9)	100% (66/66)
Pellet	100% (1/1)	33% (1/3)	60% (3/5)
Player	100% (1/1)	12% (1/8)	30% (6/20)
PlayerCollisions	100% (1/1)	14% (1/7)	9% (2/21)
PlayerFactory	100% (1/1)	100% (3/3)	100% (4/4)

Coverage after:



Element ▲	Class, %	Method, %	Line, %
CollisionInteractionMap	100% (2/2)	100% (9/9)	97% (40/41)
DefaultPlayerInteractionMap	100% (1/1)	100% (5/5)	100% (13/13)
Level	50% (1/2)	23% (4/17)	26% (27/103)
LevelFactory	50% (1/2)	66% (4/6)	83% (15/18)
LevelTest	0% (0/1)	0% (0/9)	0% (0/39)
MapParser	100% (1/1)	100% (9/9)	100% (66/66)
Pellet	100% (1/1)	33% (1/3)	60% (3/5)
Player	100% (1/1)	12% (1/8)	30% (6/20)
PlayerCollisions	100% (1/1)	100% (7/7)	100% (21/21)
PlayerFactory	100% (1/1)	100% (3/3)	100% (4/4)

The remaining uncovered line in `CollisionInteractionMap`



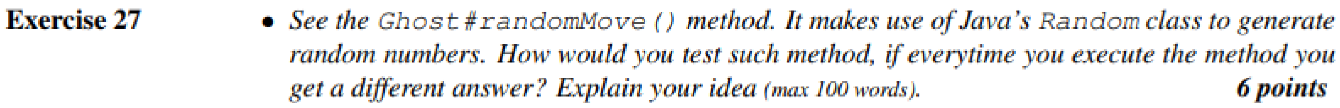
```java
/**
 * Returns a list of all classes and interfaces the class inherits.
 *
 * @param clazz The class to create a list of super classes and interfaces
 *              for.
 * @return A list of all classes and interfaces the class inherits.
 */
/unchecked/
private List<Class<? extends Unit>> getInheritance(
    Class<? extends Unit> clazz) {
    List<Class<? extends Unit>> found = new ArrayList<>();
    found.add(clazz);

    int index = 0;
    while (found.size() > index) {
        Class<?> current = found.get(index);
        Class<?> superClass = current.getSuperclass();
        if (superClass != null && Unit.class.isAssignableFrom(superClass)) {
            found.add((Class<? extends Unit>) superClass);
        }
        for (Class<?> classInterface : current.getInterfaces()) {
            if (Unit.class.isAssignableFrom(classInterface)) {
                found.add((Class<? extends Unit>) classInterface);
            }
        }
        index++;
    }

    return found;
```

The added `PlayerCollisionsTest` supports testing Classes that implement the `CollisionMap` interface *and* make use of a `PointCalculator`.
It exercises all scenarios described in Exercise_23, except for those that do not involve `Unit`s ("empty Square" and "Wall").
Currently collisions between Units that are assumed to be stationary are not checked (e.g. Pellet on Pellet).

Exercise 27

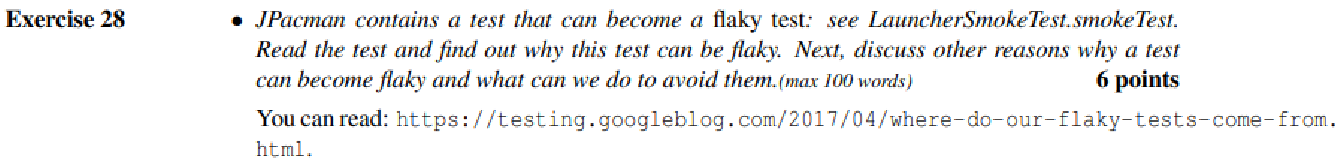


Exercise 27

- *See the `Ghost#randomMove()` method. It makes use of Java's `Random` class to generate random numbers. How would you test such method, if everytime you execute the method you get a different answer? Explain your idea (max 100 words).* ***6 points***

In this case I would test that one of the possible results of this method is present as the return value of this function.
Currently, the `randomMove()` method creates a new Random Number Generator every time it is called - that smells! Instead it should obtain a new random number from an existing generator.
If that was the case, it would also mean that we can mock the generator and eliminate the randomness in our tests.

Exercise 28



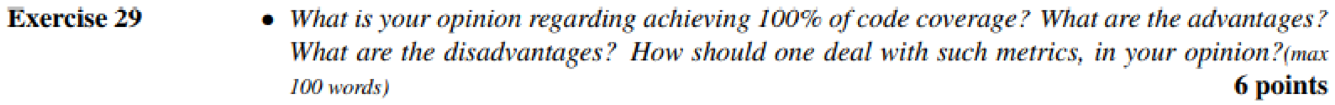
Exercise 28

- *JPacman contains a test that can become a flaky test: see LauncherSmokeTest.smokeTest. Read the test and find out why this test can be flaky. Next, discuss other reasons why a test can become flaky and what can we do to avoid them.(max 100 words)* **6 points**

  You can read: `https://testing.googleblog.com/2017/04/where-do-our-flaky-tests-come-from.html`.

The smoke test only works in the current level. In another level it may fail. In addition, it can also fail if something is changed in the "attack" algorithms of the ghosts. The likelihood of flaky tests increases with the size of our tests. The larger it is, the flakier it can be. We can reduce this by considering beforehand what we want to test in order to avoid unnecessary tests and thus to keep the tests smaller.

Exercise 29

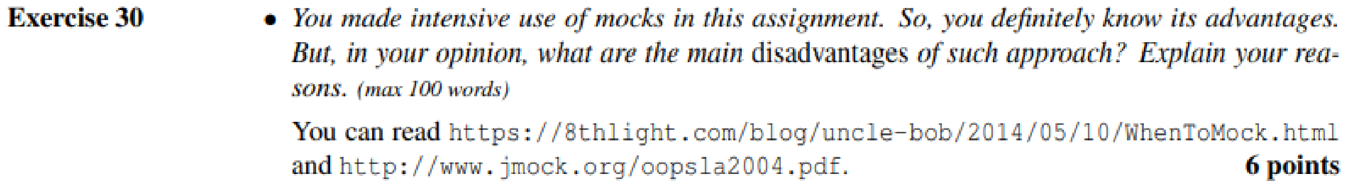


Exercise 29

- *What is your opinion regarding achieving 100% of code coverage? What are the advantages? What are the disadvantages? How should one deal with such metrics, in your opinion?(max 100 words)* **6 points**

100% code coverage is a nice thing, but in most cases it is not necessary or not realistic.
The advantage is that you can be sure that the code does what you expect.
The disadvantage is that you usually need a lot of resources to achieve 100% code coverage.
Such metrics are good for assessing how error-free the code is. For core elements of the code, you should achieve a very high percentage of the metrics, but for less important elements, in my opinion, 80% is enough.

Exercise 30



Exercise 30

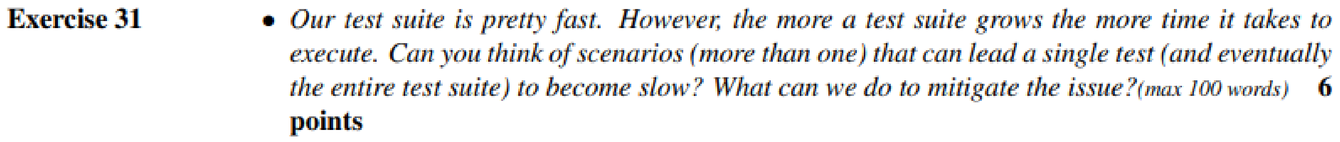
- *You made intensive use of mocks in this assignment. So, you definitely know its advantages. But, in your opinion, what are the main* disadvantages *of such approach? Explain your reasons.* *(max 100 words)*

  You can read `https://8thlight.com/blog/uncle-bob/2014/05/10/WhenToMock.html` and `http://www.jmock.org/oopsla2004.pdf`. **6 points**

I see three possible disadvantages with mocking:

1. Performance - Mocking makes use of reflection, which *can* be slower than the classes that are being mocked.
2. Refactoring - From what I've seen, mocking can get closely coupled to the class that is being mocked. That means that seemingly simple refactoring could lead to a long string of Tests that need to be updated.
3. Bugs obscured by mocking - typically, getters are only indirectly tested (because "it's a getter, what could go wrong here"), but if every test is mocking this getter, we never ensure that the getter actually works at all.

Exercise 31



Exercise 31

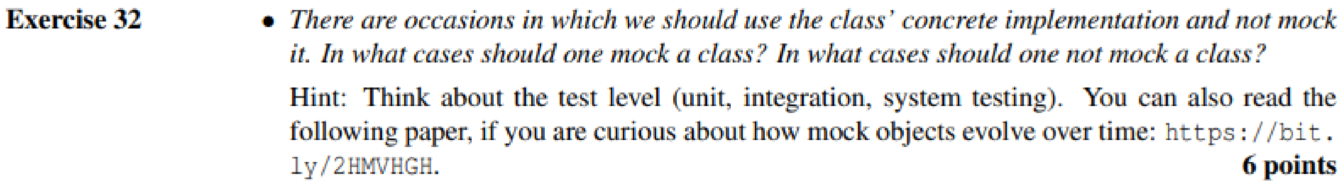
- *Our test suite is pretty fast. However, the more a test suite grows the more time it takes to execute. Can you think of scenarios (more than one) that can lead a single test (and eventually the entire test suite) to become slow? What can we do to mitigate the issue?* *(max 100 words)* **6 points**

Unit Tests may become slow when the unit that is being tested involves too many dependencies.
We can solve this issue by using Mocks and/or limiting our tests to only exercise the behaviour of the Unit under test.

Complex tests can also slow down the Test Suite.
Here we should identify what exactly is being tested and reducing the test accordingly.

Lastly, The Test Suite may become slow if there are too many tests / the setup of tests is too complex.
Since the setup will be executed once for every test, the effort spent on executing setup scales proportionally with the amount of tests.
There is not much we can do to fix this, but one possibility is to identify the exact requirements of each test and minimizing the setup-code accordingly.

Exercise 32



Exercise 32

- *There are occasions in which we should use the class' concrete implementation and not mock it. In what cases should one mock a class? In what cases should one not mock a class?*

  Hint: Think about the test level (unit, integration, system testing). You can also read the following paper, if you are curious about how mock objects evolve over time: `https://bit.ly/2HMVHGH`.

  6 points

In my opinion, if the desired behaviour can easily be achieved without mocking (just using the concrete implementation), then that class should not be mocked.
Conversely, the class probably should be mocked, if the desired behaviour is difficult to cause normally (e.g. testing anomalous behaviour).

In regards to test levels, I would argue that Mocking has no place in System Testing, as the goal is to verify the System as a whole.
In Integration Testing, Mocking can be reasonable, but should be limited to the Modules that are not exercised by the Test.
For example: if testing the communication between Module-A and Module-B requires the presence of Module-C which is *not* intended to be under test, then Mocking Module-C is reasonable.

Submission

"Explain or eliminate checkstyle or SpotBugs violations that remain"

The same Checkstyle violation from report1 is still present ("Avoid inline conditionals") and has now tripled. Because of this (and a lack of feedback to Part1), we have disabled the `AvoidInlineConditionals`-module.

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

There are three new test classes:

- `PlayerCollisionTest` - verifies behaviour of CollisionMap implementations.
- `GameTest` - verifies that the `Game::start()` method behaves as expected.
- `MapParserTest` - extensively exercises the `MapParser` class.

"Also reflect on your continuous integration server results"

This time our Github-Actions actually caught a failing test on the master branch.
This has lead to a re-evaluation of the correctness of our `PlayerCollisionsTest`, which otherwise may have gone unnoticed.

"And [reflect on] your commit behaviour"

For the most part, our commit behaviour has not changed much.
But I do believe that the team has become familiar with the workflow.
Commits seem to occur frequently, as of writing there are 121 commits after completing 32 exercises.

"And [reflect on] the new knowledge acquired."

This labwork gave us good insights into working with mocks and handling with coverage.
We have also become more familiar with the intricacies of JPacman, specifically map-parsing and unit-collisions.