Assignment 2 - Software Quality & Testing

1.2 Mock Objects

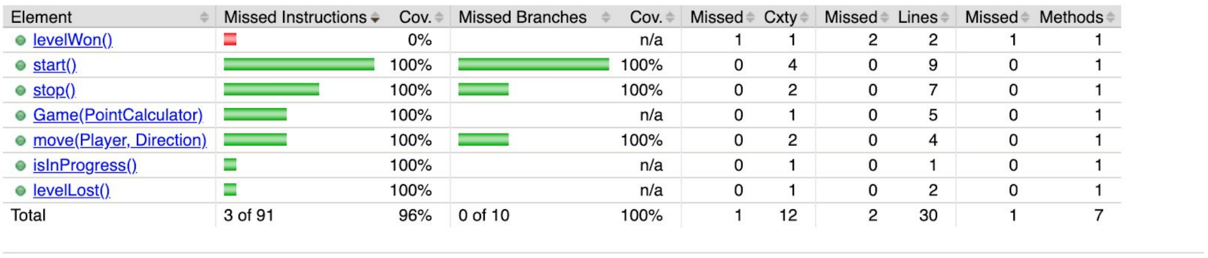
Exercise 1&2:

See *src/test/java/nl/tudelft/jpacman/level/MapParser*

1.3 Branch Coverage with Mocks

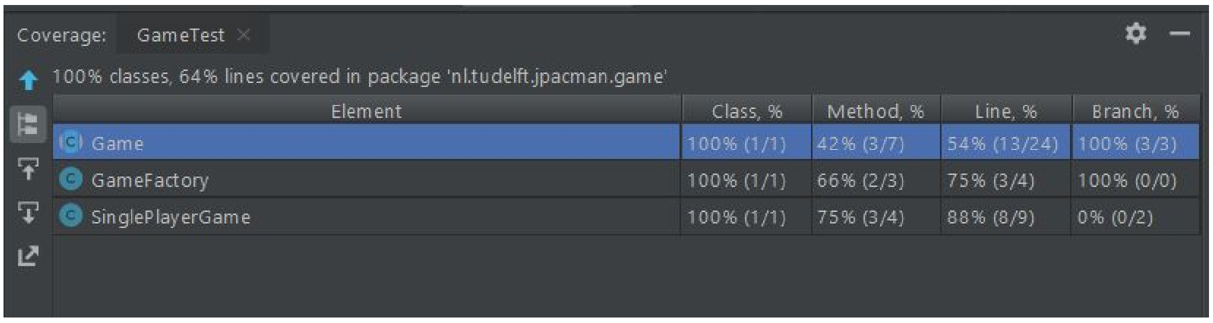
Exercise 3:

With all the tests (jacoco report):



Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods
levelWon()	🟥	0%		n/a	1	1	2	2	1	1
start()	🟩	100%	🟩	100%	0	4	0	9	0	1
stop()	🟩	100%	🟩	100%	0	2	0	7	0	1
Game(PointCalculator)	🟩	100%		n/a	0	1	0	5	0	1
move(Player, Direction)	🟩	100%	🟩	100%	0	2	0	4	0	1
isInProgress()	🟩	100%		n/a	0	1	0	1	0	1
levelLost()	🟩	100%		n/a	0	1	0	2	0	1
Total	3 of 91	96%	0 of 10	100%	1	12	2	30	1	7

With just the Game tests:



Coverage: GameTest

100% classes, 64% lines covered in package 'nl.tudelft.jpacman.game'

Element	Class, %	Method, %	Line, %	Branch, %
Game	100% (1/1)	42% (3/7)	54% (13/24)	100% (3/3)
GameFactory	100% (1/1)	66% (2/3)	75% (3/4)	100% (0/0)
SinglePlayerGame	100% (1/1)	75% (3/4)	88% (8/9)	0% (0/2)

1.4 Testing Collisions

Exercise 4:

Q: Analyze requirements (found in doc/scenarios.md) and derive a decision table for the JPacman collisions from it.

A:

	Player	**Pellet**	**Ghost**
Player	not possible	player earns points from the pellet, if it's the last one ends the game	player dies, end of the game
Pellet	player earns points from the pellet, if it's the last one ends the game	not possible	nothing happens
Ghost	player dies, end of the game	nothing happens	nothing happens

Exercise 5:

Q: 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". **Hint**: Use mocks.
The *PlayerCollision* class is far from ideal, as it does not scale well to more realistic collision maps. An alternative is the (reflection-based) *DefaultPlayerInteractionMap*, which makes use of the (more complicated) *CollisionInteractionMap*

A: See *src/test/java/nl/tudelft/jpacman/level/PlayerCollisionTest* and *src/test/java/nl/tudelft/jpacman/level/CollisionInteractionMapTest*

Exercise 6:

Q: Restructure your test suite from exercise 5 so that you can execute the same test suite on both *PlayerCollision* and *DefaultPlayerInteractionMap* objects. **Hint**: Use a parallel class hierarchy for your tests.

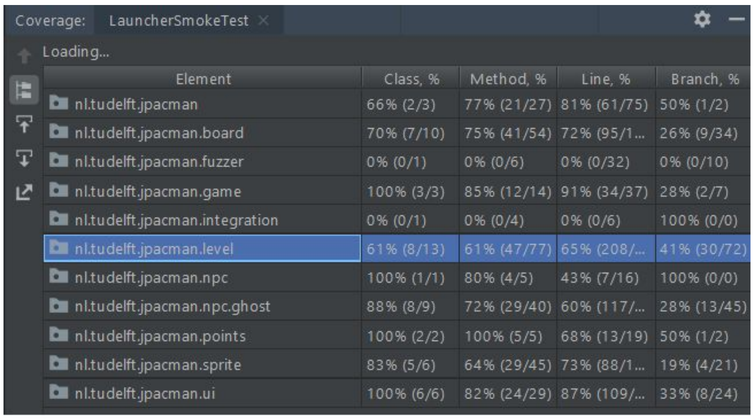
A: See *src/test/java/nl/tudelft/jpacman/level/DefaultPlayerInteractionMapTest*

Exercise 7:

Q: 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.

A:

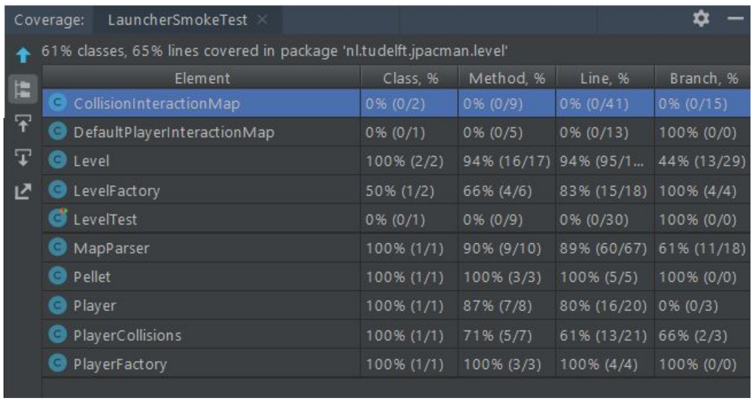
Before:



Coverage: LauncherSmokeTest

Loading...

Element	Class, %	Method, %	Line, %	Branch, %
nl.tudelft.jpacman	66% (2/3)	77% (21/27)	81% (61/75)	50% (1/2)
nl.tudelft.jpacman.board	70% (7/10)	75% (41/54)	72% (95/1...	26% (9/34)
nl.tudelft.jpacman.fuzzer	0% (0/1)	0% (0/6)	0% (0/32)	0% (0/10)
nl.tudelft.jpacman.game	100% (3/3)	85% (12/14)	91% (34/37)	28% (2/7)
nl.tudelft.jpacman.integration	0% (0/1)	0% (0/4)	0% (0/6)	100% (0/0)
nl.tudelft.jpacman.level	61% (8/13)	61% (47/77)	65% (208/...	41% (30/72)
nl.tudelft.jpacman.npc	100% (1/1)	80% (4/5)	43% (7/16)	100% (0/0)
nl.tudelft.jpacman.npc.ghost	88% (8/9)	72% (29/40)	60% (117/...	28% (13/45)
nl.tudelft.jpacman.points	100% (2/2)	100% (5/5)	68% (13/19)	50% (1/2)
nl.tudelft.jpacman.sprite	83% (5/6)	64% (29/45)	73% (88/1...	19% (4/21)
nl.tudelft.jpacman.ui	100% (6/6)	82% (24/29)	87% (109/...	33% (8/24)

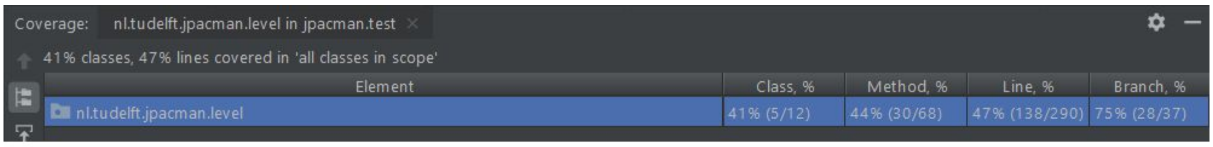


Coverage: LauncherSmokeTest

61% classes, 65% lines covered in package 'nl.tudelft.jpacman.level'

Element	Class, %	Method, %	Line, %	Branch, %
CollisionInteractionMap	0% (0/2)	0% (0/9)	0% (0/41)	0% (0/15)
DefaultPlayerInteractionMap	0% (0/1)	0% (0/5)	0% (0/13)	100% (0/0)
Level	100% (2/2)	94% (16/17)	94% (95/1...	44% (13/29)
LevelFactory	50% (1/2)	66% (4/6)	83% (15/18)	100% (4/4)
LevelTest	0% (0/1)	0% (0/9)	0% (0/30)	100% (0/0)
MapParser	100% (1/1)	90% (9/10)	89% (60/67)	61% (11/18)
Pellet	100% (1/1)	100% (3/3)	100% (5/5)	100% (0/0)
Player	100% (1/1)	87% (7/8)	80% (16/20)	0% (0/3)
PlayerCollisions	100% (1/1)	71% (5/7)	61% (13/21)	66% (2/3)
PlayerFactory	100% (1/1)	100% (3/3)	100% (4/4)	100% (0/0)

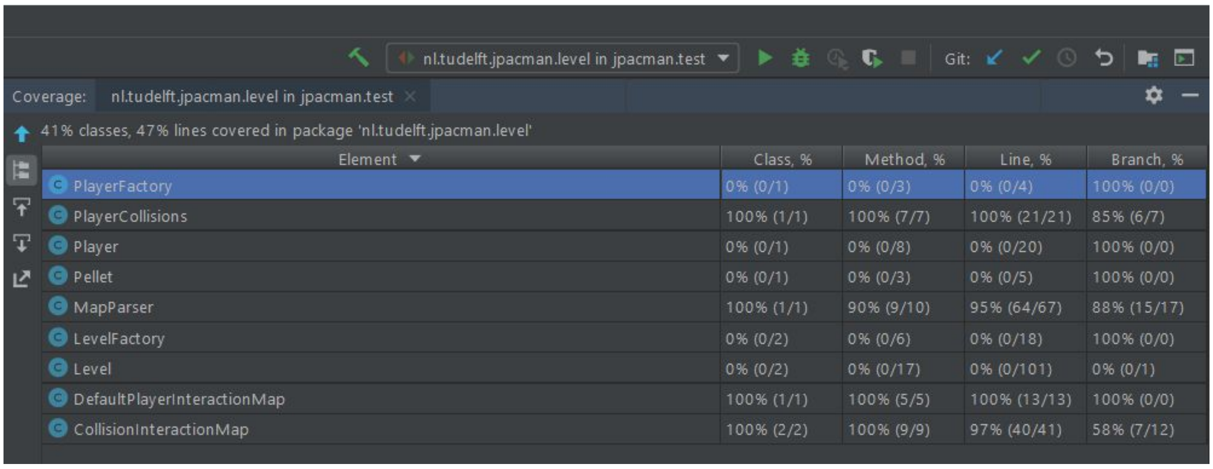
After:



Coverage: nl.tudelft.jpacman.level in jpacman.test

41% classes, 47% lines covered in 'all classes in scope'

Element	Class, %	Method, %	Line, %	Branch, %
nl.tudelft.jpacman.level	41% (5/12)	44% (30/68)	47% (138/290)	75% (28/37)



nl.tudelft.jpacman.level in jpacman.test

Git:

Coverage: nl.tudelft.jpacman.level in jpacman.test

41% classes, 47% lines covered in package 'nl.tudelft.jpacman.level'

Element	Class, %	Method, %	Line, %	Branch, %
PlayerFactory	0% (0/1)	0% (0/3)	0% (0/4)	100% (0/0)
PlayerCollisions	100% (1/1)	100% (7/7)	100% (21/21)	85% (6/7)
Player	0% (0/1)	0% (0/8)	0% (0/20)	100% (0/0)
Pellet	0% (0/1)	0% (0/3)	0% (0/5)	100% (0/0)
MapParser	100% (1/1)	90% (9/10)	95% (64/67)	88% (15/17)
LevelFactory	0% (0/2)	0% (0/6)	0% (0/18)	100% (0/0)
Level	0% (0/2)	0% (0/17)	0% (0/101)	0% (0/1)
DefaultPlayerInteractionMap	100% (1/1)	100% (5/5)	100% (13/13)	100% (0/0)
CollisionInteractionMap	100% (2/2)	100% (9/9)	97% (40/41)	58% (7/12)

In the previous exercises we tester *PlayerCollisions*, *DefaultPlayerInteractionMap* and *CollisionInteractionMap* and as we can see from the branch coverage of *PlayerCollisions* increased from 66% to 85%, *CollisionInteractionMap* and *DefaultPlayerInteraction* were not tested at all before and as we can see the first one reached a 58% branch coverage now and the latter has 100% coverage in all the cases.
The missing branch in the *PlayerCollisionsTest* is the case when our Unit is none of its subclasses (Player, Ghost or Pellet). This is because testing the superclass may cause problems in the future if we would like to add other subclasses to Unit.

1.5 Pragmatic Testing

Exercise 8:

Q: 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? *(max 100 words)*
A: A way to test these methods is through mocks: we need to supply a mocked number so the random method will always generate the move we want to test and we can test its behaviour. Furthermore, another way could be to supply a seed to the random algorithm, since these random numbers are generated according to the seed, by supplying the same seed everytime the number generated will always be the same.

Exercise 9:

Q: 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)*
A: *assertThat(player.isAlive()).isFalse();* can become a flaky test since it is based on the assumption that after a given time and moving a given amount of steps the player will be killed by the ghosts. In order to avoid flaky tests we should avoid dealing with things such as: assumptions, time, dynamic content, caching, concurrency and make sure that we have a clean state before every test.

Exercise 10

Q: 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)*
A: In my opinion, 100% code coverage is something to aim for when we are dealing with simple and small systems but when they get more complex, it becomes very hard to achieve it. Having 100% code coverage gives us insurance that the code is properly tested and functional. However, the more a program is tested the less effective testing becomes which means the bigger a program gets, the number of tests grow a lot faster making it very resource and time consuming. That time could be used to improve the system's functionality. Moreover, higher code coverage doesn't imply higher quality code.

Exercise 11

Q: 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)*
A: Using mocks can give a fake sense of confidence since the behaviour of some classes is mocked and the real behaviour might be different than the expected one. For this reason, integration problems might pass.

Exercise 12

Q: 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)
A: One scenario that can lead to long executing time is just like one of the tests in the smoking tests that waits a certain amount of time, since it has to wait that amount of time before proceeding it always increases the execution time by a lot. Another reason could be that the test needs to parse too much data. In order to minimize these issues, stopping the

tests (using *Thread.sleep()*) for a given amount of time should be avoided and also the use of small data sets in the tests in order to parse through the minimum amount of data.

Exercise 13

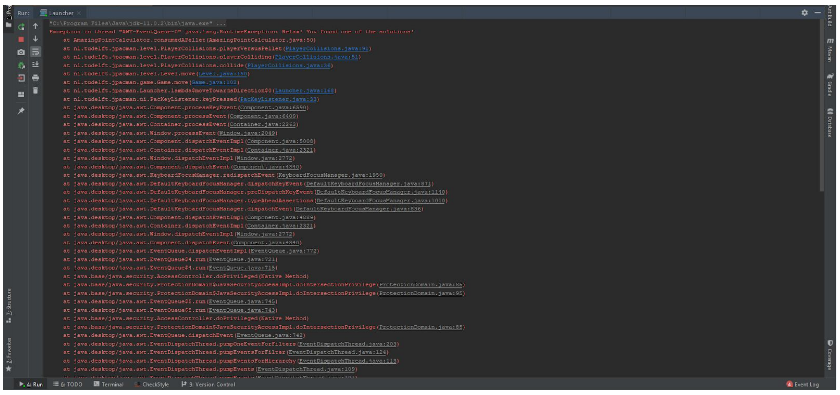
Q: There are occasions in which we should use the class' concrete implementation and not mock it. In what cases should one mock a class? In what cases should one not mock a class?
A: A class should be mocked when it's a dependency of the class being tested and it is something that is not in the control of the user, for example, responses from an external API or data access objects that access databases. The classes that should not be mocked are the main class being tested and any other class that does not depend on external resources.

1.6 Security Testing

Exercise 14:

Q: Have you noticed any weird behavior in the game? Inspect by playing the game a number of times and report what you observe. **Hint**: There are 4 anomalous behaviors associated to the score and the direction.
A: After playing the game a few times with the `AmazingPointCalculator` enabled, we noticed 4 different unexpected behaviours that we didn't encountered with the `DefaultPointCalculator`.

1. If the player eats at least 15 pellets and she's direction is *WEST*, the points number result in an overflow as 2147483647 points are added and the score increases over the maximum that the program can handle. If she keeps going *WEST* the points will alternate between an overflow and an increase of the previous score by 18 points.
2. If the player eats at least 28 pellets everytime the score is greater than 0, 15 points will be deducted from the total score.
3. If the player eats more than 34 pellets and she goes *NORTH* the game throws an exception (see **Figure 1**) and the pellets are not eaten anymore.
4. If the player eats more than 34 pellets and she goes in any other direction (except *NORTH*) she dies without colliding with any ghost.



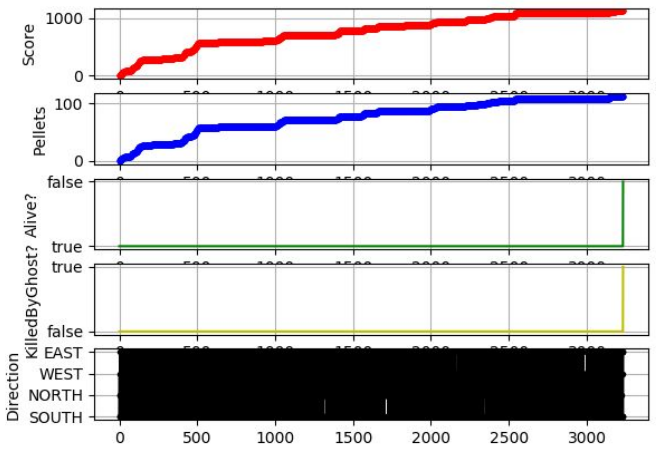
```
"C:\Program Files\Java\jdk-11.0.2\bin\java.exe" ...
Exception in thread "AWT-EventQueue-0" java.lang.RuntimeException: Relax! You found one of the solutions!
	at AmazingPointCalculator.consumedAPellet(AmazingPointCalculator.java:50)
	at nl.tudelft.jpacman.level.PlayerCollisions.playerVersusPellet(PlayerCollisions.java:91)
	at nl.tudelft.jpacman.level.PlayerCollisions.playerColliding(PlayerCollisions.java:51)
	at nl.tudelft.jpacman.level.PlayerCollisions.collide(PlayerCollisions.java:36)
	at nl.tudelft.jpacman.level.Level.move(Level.java:190)
	at nl.tudelft.jpacman.game.Game.move(Game.java:102)
	at nl.tudelft.jpacman.Launcher.lambda$moveTowardsDirection$0(Launcher.java:168)
	at nl.tudelft.jpacman.ui.PacKeyListener.keyPressed(PacKeyListener.java:33)
	at java.desktop/java.awt.Component.processKeyEvent(Component.java:6590)
	at java.desktop/java.awt.Component.processEvent(Component.java:6409)
	at java.desktop/java.awt.Container.processEvent(Container.java:2263)
	at java.desktop/java.awt.Window.processEvent(Window.java:2049)
	at java.desktop/java.awt.Component.dispatchEventImpl(Component.java:5008)
	at java.desktop/java.awt.Container.dispatchEventImpl(Container.java:2321)
	at java.desktop/java.awt.Window.dispatchEventImpl(Window.java:2772)
	at java.desktop/java.awt.Component.dispatchEvent(Component.java:4840)
	at java.desktop/java.awt.KeyboardFocusManager.redispatchEvent(KeyboardFocusManager.java:1950)
	at java.desktop/java.awt.DefaultKeyboardFocusManager.dispatchKeyEvent(DefaultKeyboardFocusManager.java:871)
	at java.desktop/java.awt.DefaultKeyboardFocusManager.preDispatchKeyEvent(DefaultKeyboardFocusManager.java:1140)
	at java.desktop/java.awt.DefaultKeyboardFocusManager.typeAheadAssertions(DefaultKeyboardFocusManager.java:1010)
	at java.desktop/java.awt.DefaultKeyboardFocusManager.dispatchEvent(DefaultKeyboardFocusManager.java:836)
	at java.desktop/java.awt.Component.dispatchEventImpl(Component.java:4889)
	at java.desktop/java.awt.Container.dispatchEventImpl(Container.java:2321)
	at java.desktop/java.awt.Window.dispatchEventImpl(Window.java:2772)
	at java.desktop/java.awt.Component.dispatchEvent(Component.java:4840)
	at java.desktop/java.awt.EventQueue.dispatchEventImpl(EventQueue.java:772)
	at java.desktop/java.awt.EventQueue$4.run(EventQueue.java:721)
	at java.desktop/java.awt.EventQueue$4.run(EventQueue.java:715)
	at java.base/java.security.AccessController.doPrivileged(Native Method)
	at java.base/java.security.ProtectionDomain$JavaSecurityAccessImpl.doIntersectionPrivilege(ProtectionDomain.java:85)
	at java.base/java.security.ProtectionDomain$JavaSecurityAccessImpl.doIntersectionPrivilege(ProtectionDomain.java:95)
	at java.desktop/java.awt.EventQueue$5.run(EventQueue.java:745)
	at java.desktop/java.awt.EventQueue$5.run(EventQueue.java:743)
	at java.base/java.security.AccessController.doPrivileged(Native Method)
	at java.base/java.security.ProtectionDomain$JavaSecurityAccessImpl.doIntersectionPrivilege(ProtectionDomain.java:85)
	at java.desktop/java.awt.EventQueue.dispatchEvent(EventQueue.java:742)
	at java.desktop/java.awt.EventDispatchThread.pumpOneEventForFilters(EventDispatchThread.java:203)
	at java.desktop/java.awt.EventDispatchThread.pumpEventsForFilter(EventDispatchThread.java:124)
	at java.desktop/java.awt.EventDispatchThread.pumpEventsForHierarchy(EventDispatchThread.java:113)
	at java.desktop/java.awt.EventDispatchThread.pumpEvents(EventDispatchThread.java:109)
```

Figure 1

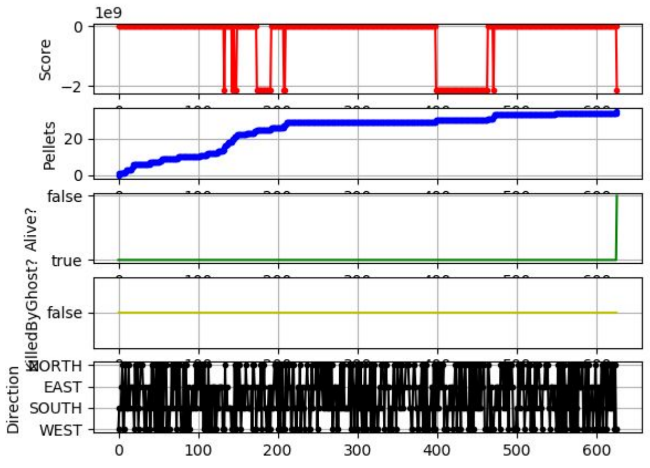
Exercise 15:

Q: Run the fuzzer multiple times and plot the generated logs. Perform this for both *Default* and *Amazing Point Calculators*, and compare their logs. Report what differences you see in their behaviors. The solution should contain snapshots of both normal and abnormal behavior, followed by an explanation of what behavior you observed and which condition triggered it. Find all 4 anomalous behaviors in this way.

A:



`DefaultPointCalculator` behaviour for *log_2.txt*

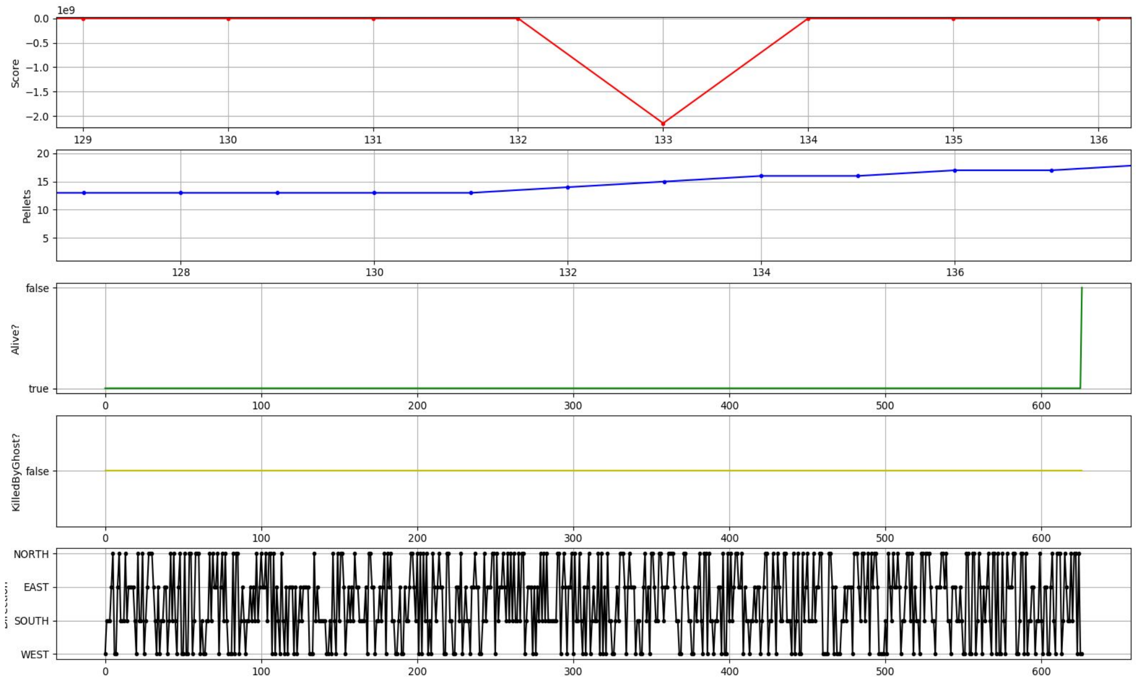


`AmazingPointCalculator` behaviour for *log_2.txt*

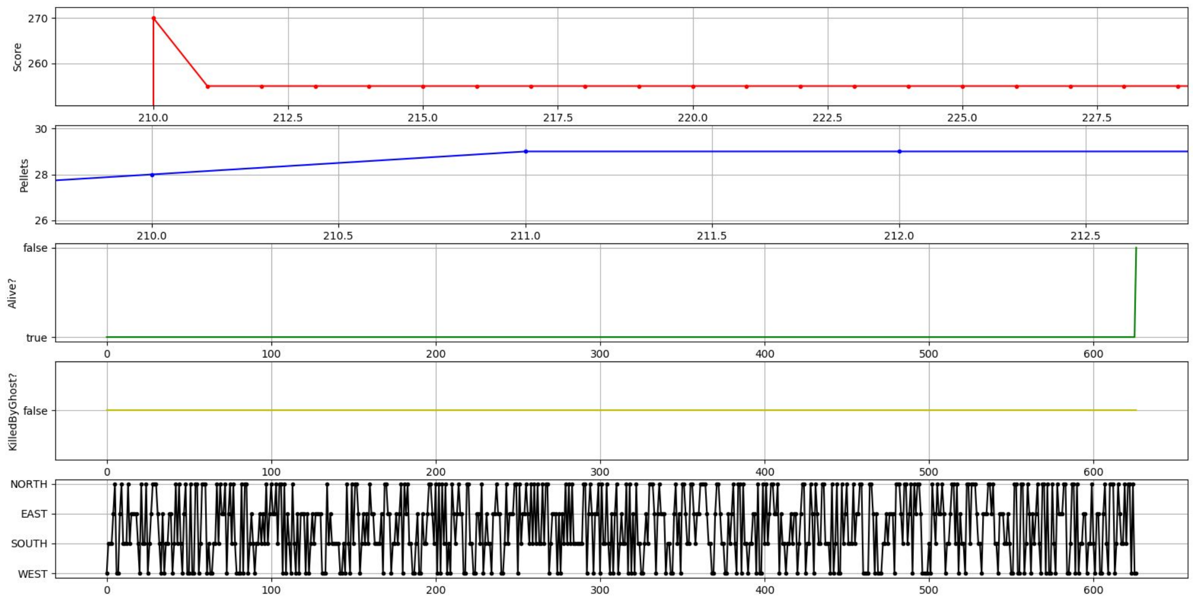
As we can see from the two graphs, in the first case, the score increases constantly while in the second case the score reaches a certain number, from there it starts bouncing between overflow and positive till it constantly keeps resulting in an overflow. Moreover, we can see how the drastic change in the score is affected by the number of pellets that the player ate.

In both cases the player is always alive when performing the different movers, but as we can notice by the third graph, with the `DefaultPointCalculator` the player dies only when she's killed by a ghost, that's not the case with the `AmazingPointCalculator` where the player dies without colliding with any ghost.
Also the Direction graph is quite different as with the `DefaultPointCalculator` the player performs more moves compared to the second case.

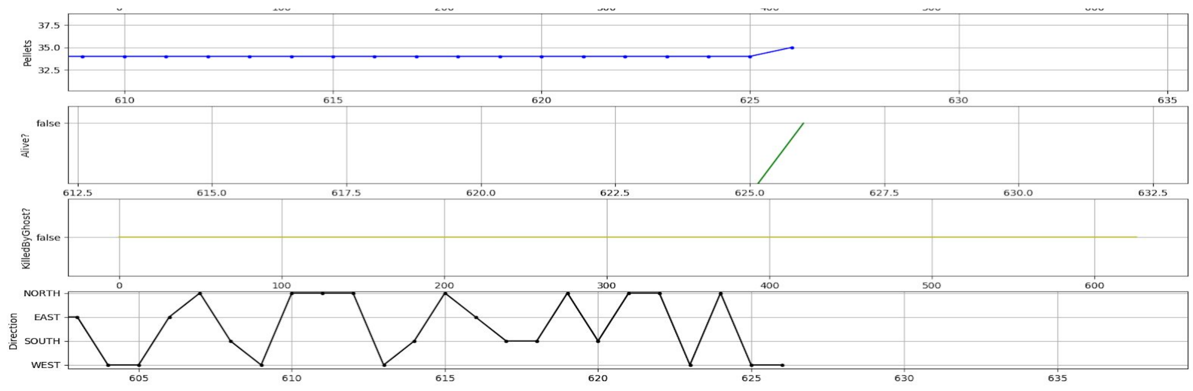
1. The player eats the 15th pellet, the score results in an overflow (*AmazingPointCalculator*).

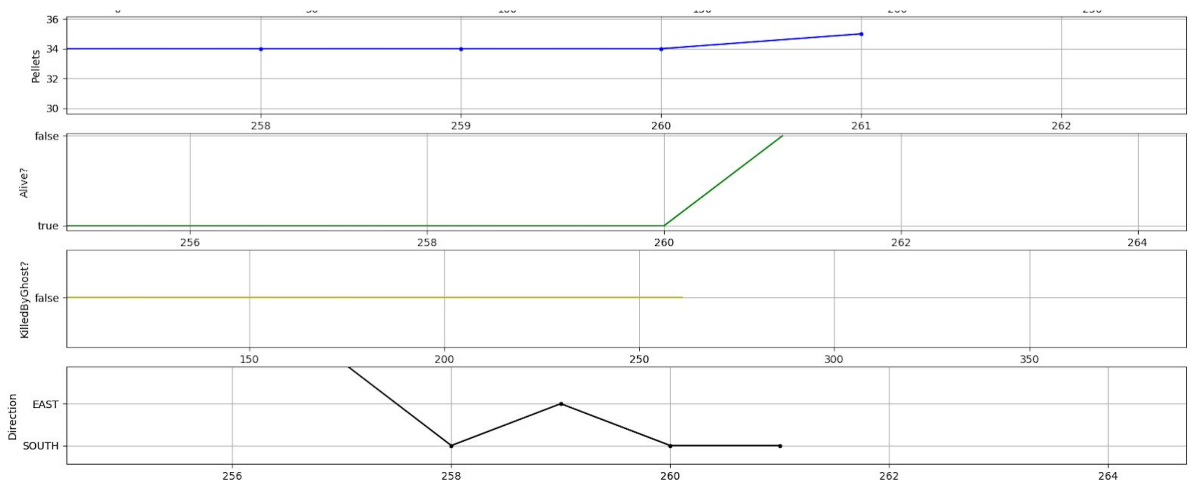


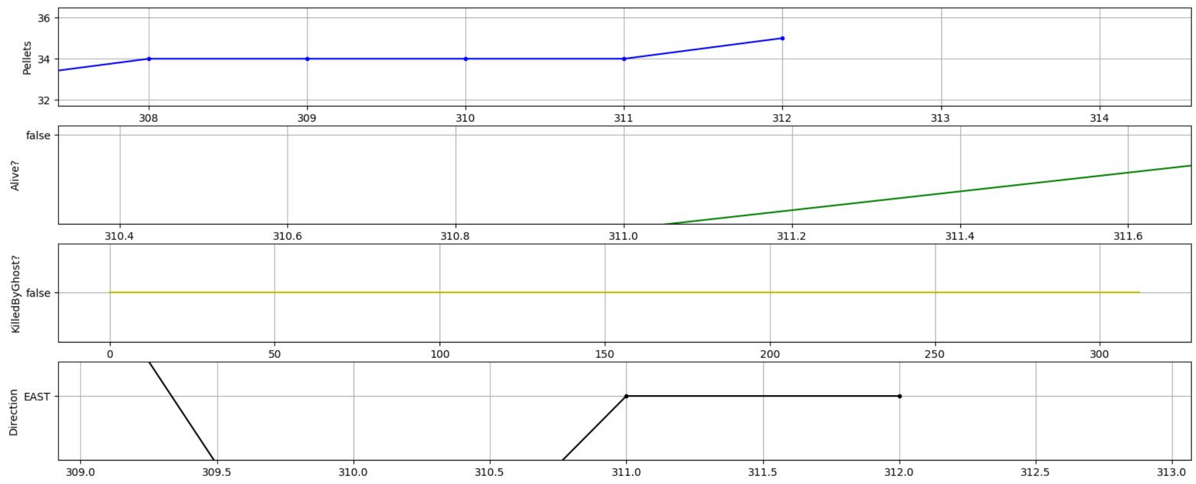
2. The player eats more than 28 pellets and the score is greater than 0 the points decrease from 270 to 255 (*AmazingPointCalculator*).



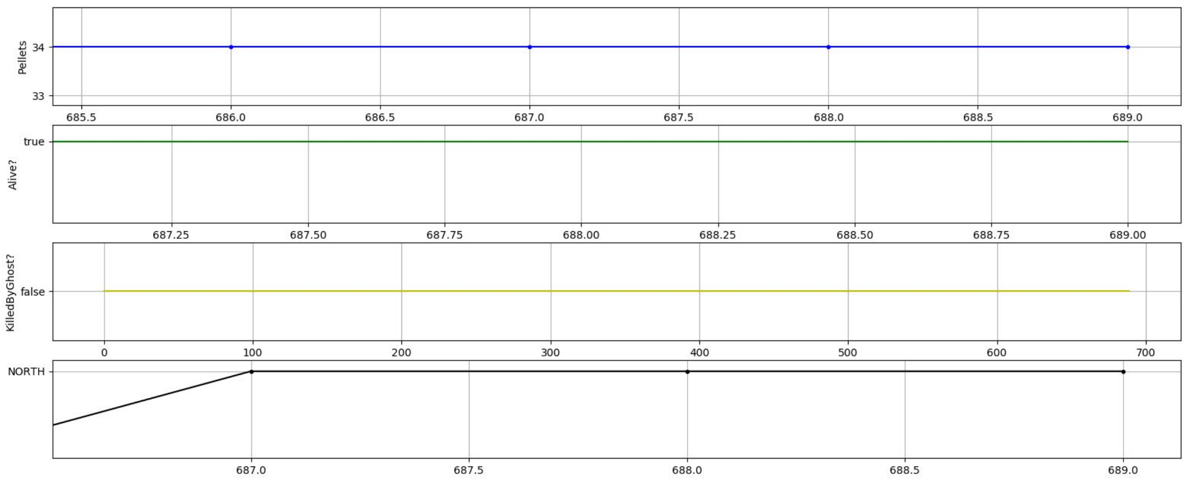
3. The player eats more than 34 pellets, when her direction is **not** *NORTH* she dies (*AmazingPointCalculator*).







4. The player eats 34 pellets and her direction is *NORTH, the number of pellets doesn't increase anymore* (*AmazingPointCalculator*).



Exercise 16:

Q: Do you see any security warnings associated to this piece of code reported by SpotBugs? Determine why (or why doesn't) SpotBugs give a security warning. Go through the **OWASP Top 10** vulnerability list and determine which one applies to that piece of code. Briefly reflect on it.
A: SpotBugs doesn't report any warning referred to that piece of code, that is because we are performing static analysis. In order to spot security vulnerabilities, we need to perform dynamic analysis instead. From the OWASP Top 10 list of vulnerabilities in 2017, the one that adhere to our case is the first one: **Injection**.
That is because *PointCalculator* is loaded dynamically in the *PointCalculatorLoader* class. In this way an attacker can send untrusted data to the interpreter and she can exploit information she is not authorized to get.

Exercise 17:

Q: How can the security problem(s) associated to dynamic class loading be fixed? Briefly explain at least three different possibilities. *(max 100 words)*

A:

1. **SecurityManager**: a Security Manager is an object that defines a security policy for an application. Thanks to this policy we can specify which actions to allow and which ones to deny.
2. **ClassLoader**: a custom ClassLoader can be used to load the untrusted code as used in our assignment.
3. **Separate JVM**: Use a separate JVM in order to run the code in order to isolate it from the rest of the code.
4. **Own Thread**: Use a different thread to run the code which can help to prevent infinite loops and such.
5. Don't use dynamic classes at all.