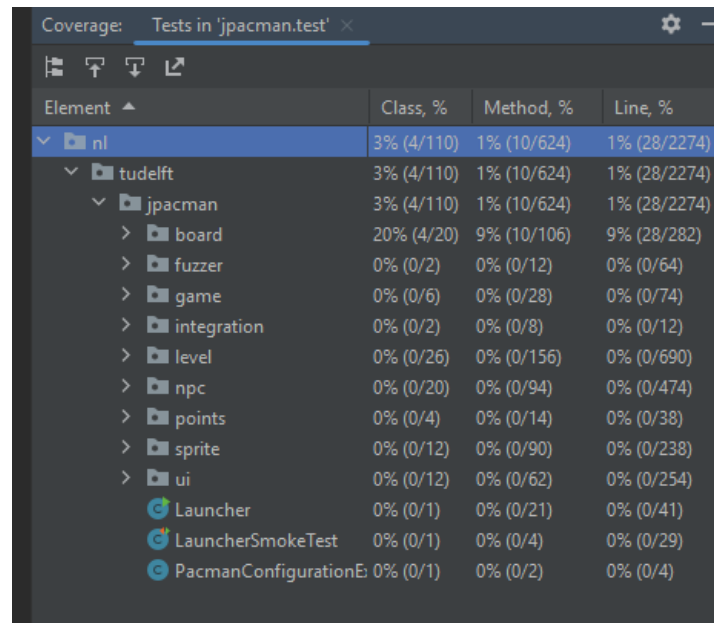


Isabella Capriotti
Dr. Businge
CS 472
28 January 2023

Report: JPacman Dynamic Analysis

Task 2.1

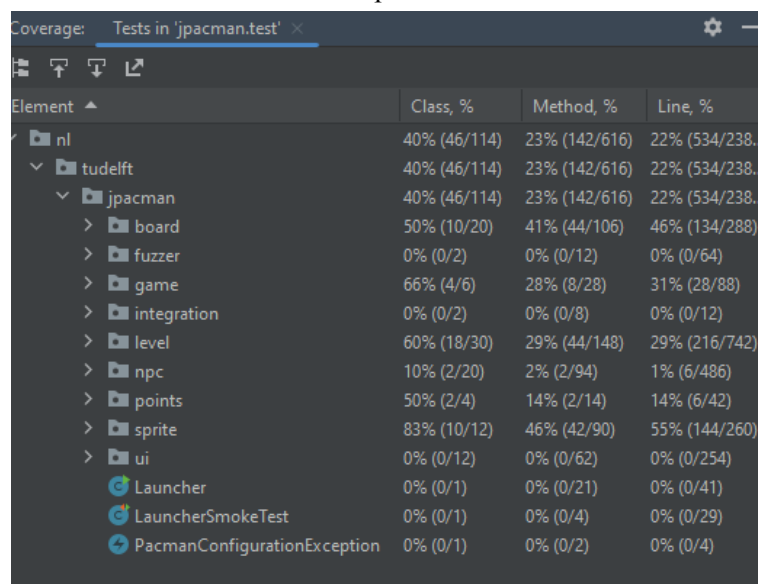
Before adding additional unit test cases, this was the IntelliJ coverage report for the JPacman project:



The screenshot shows the IntelliJ coverage report for the JPacman project. The table lists the coverage for various elements, including classes and methods. The 'nl' package is highlighted, showing 3% class coverage (4/110) and 1% method coverage (10/624). The 'tudelft' package is also highlighted, showing 3% class coverage (4/110) and 1% method coverage (10/624). The 'jpacman' package is highlighted, showing 3% class coverage (4/110) and 1% method coverage (10/624). The 'board' package is highlighted, showing 20% class coverage (4/20) and 9% method coverage (10/106). The 'fuzzer' package is highlighted, showing 0% class coverage (0/2) and 0% method coverage (0/12). The 'game' package is highlighted, showing 0% class coverage (0/6) and 0% method coverage (0/28). The 'integration' package is highlighted, showing 0% class coverage (0/2) and 0% method coverage (0/8). The 'level' package is highlighted, showing 0% class coverage (0/26) and 0% method coverage (0/156). The 'npc' package is highlighted, showing 0% class coverage (0/20) and 0% method coverage (0/94). The 'points' package is highlighted, showing 0% class coverage (0/4) and 0% method coverage (0/14). The 'sprite' package is highlighted, showing 0% class coverage (0/12) and 0% method coverage (0/90). The 'ui' package is highlighted, showing 0% class coverage (0/12) and 0% method coverage (0/62). The 'Launcher' class is highlighted, showing 0% class coverage (0/1) and 0% method coverage (0/21). The 'LauncherSmokeTest' class is highlighted, showing 0% class coverage (0/1) and 0% method coverage (0/4). The 'PacmanConfigurationException' class is highlighted, showing 0% class coverage (0/1) and 0% method coverage (0/2).

Element	Class, %	Method, %	Line, %
nl	3% (4/110)	1% (10/624)	1% (28/2274)
tudelft	3% (4/110)	1% (10/624)	1% (28/2274)
jpacman	3% (4/110)	1% (10/624)	1% (28/2274)
board	20% (4/20)	9% (10/106)	9% (28/282)
fuzzer	0% (0/2)	0% (0/12)	0% (0/64)
game	0% (0/6)	0% (0/28)	0% (0/74)
integration	0% (0/2)	0% (0/8)	0% (0/12)
level	0% (0/26)	0% (0/156)	0% (0/690)
npc	0% (0/20)	0% (0/94)	0% (0/474)
points	0% (0/4)	0% (0/14)	0% (0/38)
sprite	0% (0/12)	0% (0/90)	0% (0/238)
ui	0% (0/12)	0% (0/62)	0% (0/254)
Launcher	0% (0/1)	0% (0/21)	0% (0/41)
LauncherSmokeTest	0% (0/1)	0% (0/4)	0% (0/29)
PacmanConfigurationException	0% (0/1)	0% (0/2)	0% (0/4)

On top of the required test for `isAlive()` in the `Player` class, the three additional methods I chose to test were `isAnyPlayerAlive()` in the `Level` class, `consumedAPellet()` in the `DefaultPointCalculator` class, and `getPlayers()` in the `SinglePlayerGame` class. After adding the new unit tests, this was the new coverage report:




The screenshot shows the IntelliJ coverage report for the JPacman project after adding additional unit test cases. The table lists the coverage for various elements, including classes and methods. The 'nl' package is highlighted, showing 40% class coverage (46/114) and 23% method coverage (142/616). The 'tudelft' package is highlighted, showing 40% class coverage (46/114) and 23% method coverage (142/616). The 'jpacman' package is highlighted, showing 40% class coverage (46/114) and 23% method coverage (142/616). The 'board' package is highlighted, showing 50% class coverage (10/20) and 41% method coverage (44/106). The 'fuzzer' package is highlighted, showing 0% class coverage (0/2) and 0% method coverage (0/12). The 'game' package is highlighted, showing 66% class coverage (4/6) and 28% method coverage (8/28). The 'integration' package is highlighted, showing 0% class coverage (0/2) and 0% method coverage (0/8). The 'level' package is highlighted, showing 60% class coverage (18/30) and 29% method coverage (44/148). The 'npc' package is highlighted, showing 10% class coverage (2/20) and 2% method coverage (2/94). The 'points' package is highlighted, showing 50% class coverage (2/4) and 14% method coverage (2/14). The 'sprite' package is highlighted, showing 83% class coverage (10/12) and 46% method coverage (42/90). The 'ui' package is highlighted, showing 0% class coverage (0/12) and 0% method coverage (0/62). The 'Launcher' class is highlighted, showing 0% class coverage (0/1) and 0% method coverage (0/21). The 'LauncherSmokeTest' class is highlighted, showing 0% class coverage (0/1) and 0% method coverage (0/4). The 'PacmanConfigurationException' class is highlighted, showing 0% class coverage (0/1) and 0% method coverage (0/2).

Element	Class, %	Method, %	Line, %
nl	40% (46/114)	23% (142/616)	22% (534/238...)
tudelft	40% (46/114)	23% (142/616)	22% (534/238...)
jpacman	40% (46/114)	23% (142/616)	22% (534/238...)
board	50% (10/20)	41% (44/106)	46% (134/288)
fuzzer	0% (0/2)	0% (0/12)	0% (0/64)
game	66% (4/6)	28% (8/28)	31% (28/88)
integration	0% (0/2)	0% (0/8)	0% (0/12)
level	60% (18/30)	29% (44/148)	29% (216/742)
npc	10% (2/20)	2% (2/94)	1% (6/486)
points	50% (2/4)	14% (2/14)	14% (6/42)
sprite	83% (10/12)	46% (42/90)	55% (144/260)
ui	0% (0/12)	0% (0/62)	0% (0/254)
Launcher	0% (0/1)	0% (0/21)	0% (0/41)
LauncherSmokeTest	0% (0/1)	0% (0/4)	0% (0/29)
PacmanConfigurationException	0% (0/1)	0% (0/2)	0% (0/4)




















The overall coverage of all classes in the project increased from 3% to 40%. The level, points, and game packages, which previously all had 0% class coverage, are now at 60%, 50%, and 66% class coverage respectively. The method and line coverage for these packages similarly increased.

Task 3

Below is a screenshot of the JaCoCo coverage report referenced in the following questions.

 jpacman

jpacman

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
default		0%		0%	12	12	21	21	5	5	1	1
nl.tudelft.jpacman.points		60%		75%	1	11	5	21	0	9	0	2
nl.tudelft.jpacman.level		67%		58%	73	155	103	344	21	69	4	12
nl.tudelft.jpacman		69%		25%	12	30	18	52	6	24	1	2
nl.tudelft.jpacman.npc.ghost		71%		55%	56	105	43	181	5	34	0	8
nl.tudelft.jpacman.ui		77%		47%	54	86	21	144	7	31	0	6
nl.tudelft.jpacman.board		86%		58%	44	93	2	110	0	40	0	7
nl.tudelft.jpacman.game		87%		60%	10	24	4	45	2	14	0	3
nl.tudelft.jpacman.sprite		88%		62%	29	70	10	113	5	38	0	5
nl.tudelft.jpacman.npc		100%	n/a	n/a	0	4	0	8	0	4	0	1
Total	1,204 of 4,694	74%	290 of 637	54%	291	590	227	1,039	51	268	6	47

Question 1: Are the coverage results from JaCoCo similar to the ones you got from IntelliJ in the last task? Why so or why not?

The coverage results from JaCoCo are similar, but not identical, to the values obtained with IntelliJ in the previous task. This is likely because JaCoCo uses a different formula that factors in branch coverage while calculating the overall coverage. For example, the IntelliJ coverage report puts the level package at 60% coverage, while the JaCoCo coverage report puts the level package at 67% coverage. This could be because adding unit tests for the methods `isAlive()` and `isAnyPlayerAlive()` from the level package positively impacted the coverage on a high number of code branches because of how frequently these methods are used in the source code.

Question 2: Did you find helpful the source code visualization from JaCoCo on uncovered branches?

Yes, I found this visualization helpful. This is because analyzing the branches that execution can take gives a more realistic view of test coverage than looking at the presence of a test for a method alone. Even if a method is covered by a test, if that test doesn't verify a key execution path or a vulnerable edge case, it is not safe to say that it is fully covered. Therefore, this visualization is very useful for identifying potential scenarios that still need to be tested.

Question 3: Which visualization did you prefer and why? IntelliJ's coverage window or JaCoCo's report?

Overall, I prefer JaCoCo's visualization because it gives a more detailed breakdown of the coverage by including the branch coverage and visualization of the source code. Additionally, whereas IntelliJ's coverage window puts the number of methods/classes/lines covered in terms of how many out of the total are covered, JaCoCo puts it in terms of how many are missed. This leads to a more actionable report, since developers can target the items that are missed instead of only thinking about what is already

covered. Finally, in JaCoCo, you can drill down into specific methods within classes, while IntelliJ's granularity stops at the class level. Again, this leads to a deeper understanding on the developer side of which specific workflows in the program need more thorough testing, as opposed to just having a high level overview like the one in IntelliJ. One advantage of the IntelliJ coverage window is that the UI is more intuitive; however, once I learned how to read the JaCoCo report, I found that it contains more valuable information.