

Alleyn Murphy
Dr. Businge
CS 472
January 30, 2023

JPacman Testing Coverage Analysis

Fork Repository: <https://github.com/NyellalleyN/jpacman>
Group Repository: <https://github.com/justin-negron/SoftwareProductDesign>

The initial coverage report for the JPacamn project for several packages was 0%. After implementing the unit test `isAlive()` for the method class player in Task 2 we are left with the following coverage report:

Element ^	Class, %	Method, %	Line, %
▼ nl	16% (18/110)	9% (60/624)	8% (190/2306)
▼ tudelft	16% (18/110)	9% (60/624)	8% (190/2306)
▼ jpacman	16% (18/110)	9% (60/624)	8% (190/2306)
> 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	15% (4/26)	6% (10/156)	3% (26/700)
> npc	0% (0/20)	0% (0/94)	0% (0/474)
> points	0% (0/4)	0% (0/14)	0% (0/38)
> sprite	83% (10/12)	44% (40/90)	52% (136/260)
> 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 most notable change was the level coverage jumped to 15%, however, this total coverage is not enough thus additional unit tests were implemented to analyze the total coverage.

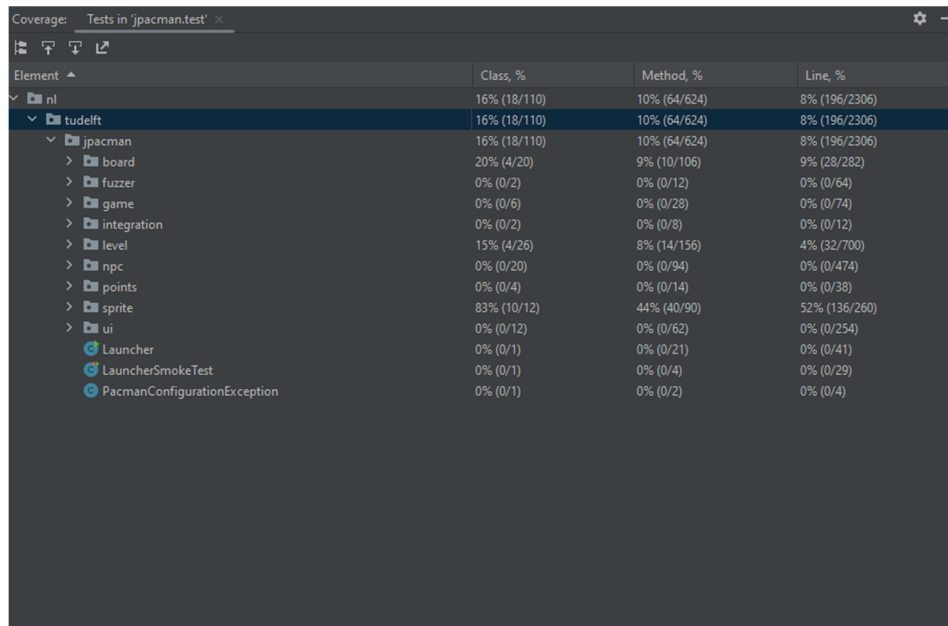
Task 2.1 Coverage Updates

To analyze the total coverage the following unit tests were implemented:

1. `Player.getSore()`

```
@Test
void testGetScore()
{
    assertThat(ThePlayer.getScore()).isEqualTo( expected: 0 );
    ThePlayer.addPoints(100);
    assertThat(ThePlayer.getScore()).isEqualTo( expected: 100 );
}
```

The unit test for `Player.getScore()` was first tested by instantiating a `Player` as well as the necessary objects such as the `PlayerFactory` and so on. This is needed as without a `Player` we are unable to test the `getScore()` method. A newly created `Player` object should contain a score of 0 thus this the method `getScore()` was tested if it was first equal to 0. Afterwards the `Player` score was adjusted by 100 points using `Player.addScore(100)`. The `getScore()` method was used again to see if it matched 100 points. The results of the test coverage from these implementations can be found below:



The screenshot shows the Coverage tool in IntelliJ IDEA. The table displays coverage data for various elements in the 'jpacman.test' suite. The 'tudelft' package is highlighted, showing an increase in method and line coverage.

Element	Class, %	Method, %	Line, %
nl	16% (18/110)	10% (64/624)	8% (196/2306)
tudelft	16% (18/110)	10% (64/624)	8% (196/2306)
jpacman	16% (18/110)	10% (64/624)	8% (196/2306)
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	15% (4/26)	8% (14/156)	4% (32/700)
npc	0% (0/20)	0% (0/94)	0% (0/474)
points	0% (0/4)	0% (0/14)	0% (0/38)
sprite	83% (10/12)	44% (40/90)	52% (136/260)
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)

Looking at the new test coverage we can see that the package level has increased its method% to 8% as well as the line percentage to 4%.

2. Unit.setDirection()

```
@Test
void testSetDirection()
{
    Direction east = Direction.valueOf( name: "EAST");
    assertThat(ThePlayer.getDirection()).isEqualTo(east);
    Direction north = Direction.valueOf( name: "NORTH");
    ThePlayer.setDirection(north);
    assertThat(ThePlayer.getDirection()).isEqualTo(north);
}
```

The next unit method that was unit tested was `unit.setDirection()` (package **board**). Like the previous unit test, a `Player` was instantiated as it extends the `Unit` class. The default direction for a unit that is newly instantiated is to face east. The command `unit.getDirection()` was utilized to determine if it was in fact east additionally the direction was set to north and `unit.getDirection()` was performed once more to determine if the new direction was in fact north. The new test coverage after implementing this unit test was:

Element ^	Class, %	Method, %	Line, %
nl	16% (18/110)	10% (68/624)	8% (202/2306)
tudelft	16% (18/110)	10% (68/624)	8% (202/2306)
jpacman	16% (18/110)	10% (68/624)	8% (202/2306)
board	20% (4/20)	13% (14/106)	12% (34/282)
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	15% (4/26)	8% (14/156)	4% (32/700)
npc	0% (0/20)	0% (0/94)	0% (0/474)
points	0% (0/4)	0% (0/14)	0% (0/38)
sprite	83% (10/12)	44% (40/90)	52% (136/260)
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 results from this unit test saw an increase of the board method coverage percentage to 13% as well as the line percentage to 12%.

3. Unit.hasSquare()

```
@Test
void testHasSquare()
{
    assertThat(ThePlayer.hasSquare()).isNotEqualTo( other: null);
}
```

One last method that a unit test was created for was the Unit.hasSquare(). Again, a Player was instantiated since the Player class extends the Unit class. The player that was instantiated was given a square. The has square method was compared to if the value was null or not. After this unit test the total test coverage was:

Element ^	Class, %	Method, %	Line, %
nl	16% (18/110)	11% (70/624)	8% (204/2306)
tudelft	16% (18/110)	11% (70/624)	8% (204/2306)
jpacman	16% (18/110)	11% (70/624)	8% (204/2306)
board	20% (4/20)	15% (16/106)	12% (36/282)
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	15% (4/26)	8% (14/156)	4% (32/700)
npc	0% (0/20)	0% (0/94)	0% (0/474)
points	0% (0/4)	0% (0/14)	0% (0/38)
sprite	83% (10/12)	44% (40/90)	52% (136/260)
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 test coverage in summary saw an increase of the board method to 15% and line percentage to 8%.

Task 3

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 test coverage results for JaCoCo appears to have a higher test coverage rate from the IntelliJ coverage results. This can be seen from the screenshot below:

jpacman

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed Cxty	Missed Lines	Missed Methods	Missed Classes
nl.tudelft.jpacman.level		67%		57%	74 155	104 344	21 69	4 12
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
default		0%		0%	12 12	21 21	5 5	1 1
nl.tudelft.jpacman.board		86%		59%	43 93	2 110	0 40	0 7
nl.tudelft.jpacman.sprite		86%		59%	30 70	11 113	5 38	0 5
nl.tudelft.jpacman		69%		25%	12 30	18 52	6 24	1 2
nl.tudelft.jpacman.points		60%		75%	1 11	5 21	0 9	0 2
nl.tudelft.jpacman.game		87%		60%	10 24	4 45	2 14	0 3
nl.tudelft.jpacman.npc		100%		n/a	0 4	0 8	0 4	0 1
Total		1,212 of 4,694		292 of 637	292 590	229 1,039	51 268	6 47

The results are likely not similar due to a different implementation of methods used for code coverage. One notable factor was the number of classes differed between the two test coverages. For example, IntelliJ did not factor in BoardFactory.Wall as a class but JaCoCo did.

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

The visualization of green to red really does make it easier to see what areas are completely lacking. In addition to this the missed branches are an important factor that would be incredibly helpful to determining total coverage. Having the potential missed branches highlight makes it easier to create higher quality unit tests.

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

JaCoCo's quick percentage graphs make it easy to see where code may not have been covered. In addition to this the missed branches code highlight for JaCoCo is very useful in ensuring high quality unit tests. I would favor JaCoCo due to this however if I needed a quick report, I would go with IntelliJ's coverage test. While it may not provide the missed branch and code highlighting it can give you a quick idea of what code you have covered. In the end both are useful in their own way providing different benefit versus the others.