Tarik Resimovic
CS 472 (Fall 2023)
GitHub Repo: https://github.com/tresimovic/jpacman
Unit Testing Report

Task 2

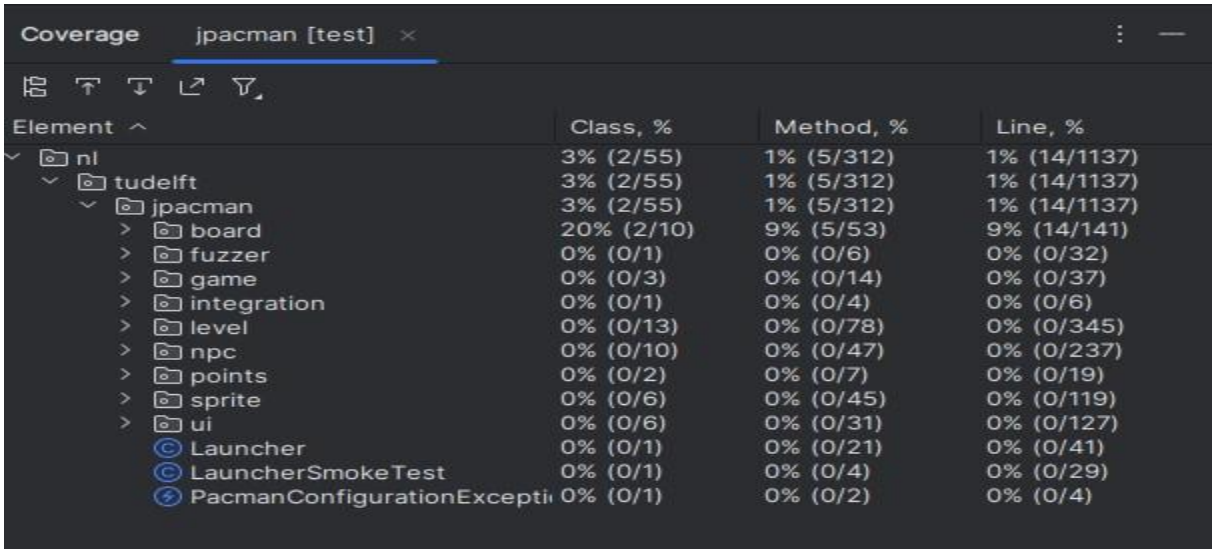

Fig. 1: IntelliJ coverage analysis before any unit tests were added.

The screenshot above shows the coverage analysis using Gradle in IntelliJ before any additional unit tests were added to the project. This is simply the coverage obtained from using the tests already provided by default in the JPacman repository. Figure 2 below is a screenshot of the first new unit test added, `PlayerTest.java`. This is the unit test provided by the professor and tests the `isAlive()` method of the `Player` class in the `level` package. Figure 3 shows the resulting increase in code coverage once this first test is added to the test suite.

```
public class PlayerTest {
    1 usage
    private static final PacManSprites SPRITE_STORE = new PacManSprites();
    1 usage
    private PlayerFactory factory = new PlayerFactory(SPRITE_STORE);
    1 usage
    private Player player = factory.createPacMan();

    Tarik Resimovic
    @Test
    void testAlive() { assertThat(player.isAlive()).isEqualTo( expected: true); }
}
```

Fig. 2: PlayerTest.java

Element	Class, %	Method, %	Line, %
nl	16% (9/55)	9% (30/312)	8% (95/1153)
tudelft	16% (9/55)	9% (30/312)	8% (95/1153)
jpacman	16% (9/55)	9% (30/312)	8% (95/1153)
board	20% (2/10)	9% (5/53)	9% (14/141)
fuzzer	0% (0/1)	0% (0/6)	0% (0/32)
game	0% (0/3)	0% (0/14)	0% (0/37)
integration	0% (0/1)	0% (0/4)	0% (0/6)
level	15% (2/13)	6% (5/78)	3% (13/350)
npc	0% (0/10)	0% (0/47)	0% (0/237)
points	0% (0/2)	0% (0/7)	0% (0/19)
sprite	83% (5/6)	44% (20/45)	52% (68/130)
ui	0% (0/6)	0% (0/31)	0% (0/127)
Launcher	0% (0/1)	0% (0/21)	0% (0/41)
LauncherSmokeTest	0% (0/1)	0% (0/4)	0% (0/29)
PacmanConfigurationExceptio	0% (0/1)	0% (0/2)	0% (0/4)

Fig. 3: Coverage increase from PlayerTest.java

For Task 2.1, the first unit test I created was `GhostFactoryTest.java` which tested the `createBlinky()` method of the `GhostFactory` class in the `npc.ghost` package. It also utilized, and therefore tested, the `hasSquare()` method of the `Unit` class in the `board` package. Figure 4 shows the unit test and Figure 5 shows the increase in code coverage as a result of adding this test to the test suite.

```
public class GhostFactoryTest {
    1 usage
    private static final PacManSprites sprites = new PacManSprites();
    1 usage
    private GhostFactory factory = new GhostFactory(sprites);

    Tarik Resimovic
    @Test
    void createGhosts() {
        Ghost ghost = factory.createBlinky();
        assertThat(ghost.hasSquare()).isEqualTo( expected: false);
    }
}
```

Fig. 4: GhostFactoryTest.java

Element	Class, %	Method, %	Line, %
nl	23% (13/55)	12% (38/312)	10% (116/1159)
tudelft	23% (13/55)	12% (38/312)	10% (116/1159)
jpacman	23% (13/55)	12% (38/312)	10% (116/1159)
board	20% (2/10)	11% (6/53)	10% (15/141)
fuzzer	0% (0/1)	0% (0/6)	0% (0/32)
game	0% (0/3)	0% (0/14)	0% (0/37)
integration	0% (0/1)	0% (0/4)	0% (0/6)
level	15% (2/13)	6% (5/78)	3% (13/350)
npc	40% (4/10)	12% (6/47)	6% (17/243)
points	0% (0/2)	0% (0/7)	0% (0/19)
sprite	83% (5/6)	46% (21/45)	54% (71/130)
ui	0% (0/6)	0% (0/31)	0% (0/127)
Launcher	0% (0/1)	0% (0/21)	0% (0/41)
LauncherSmokeTest	0% (0/1)	0% (0/4)	0% (0/29)
PacmanConfigurationExceptio	0% (0/1)	0% (0/2)	0% (0/4)

Fig. 5: Coverage increase from GhostFactoryTest.java

The second unit test I created was `LevelFactoryTest.java` which tested the `createGhost()` method in the `LevelFactory` class of the `level` package. Figure 6 shows the unit test and Figure 7 shows the increase in code coverage as a result of including this test.

```
public class LevelFactoryTest {
    2 usages
    private static final PacManSprites sprites = new PacManSprites();
    1 usage
    private GhostFactory ghostfactory = new GhostFactory(sprites);
    1 usage
    private final PointCalculator pointcalculator = null;
    1 usage
    private LevelFactory levelfactory = new LevelFactory(sprites, ghostfactory, pointcalculator);

    Tarik Resimovic
    @Test
    void testGhostCreator() {
        Ghost ghost = levelfactory.createGhost();
        assertThat(ghost.hasSquare()).isEqualTo( expected: false);
    }
}
```

Fig. 6: LevelFactoryTest.java

Element	Class, %	Method, %	Line, %
nl	25% (14/55)	12% (40/312)	10% (126/1161)
tudelft	25% (14/55)	12% (40/312)	10% (126/1161)
jpacman	25% (14/55)	12% (40/312)	10% (126/1161)
board	20% (2/10)	11% (6/53)	10% (15/141)
fuzzer	0% (0/1)	0% (0/6)	0% (0/32)
game	0% (0/3)	0% (0/14)	0% (0/37)
integration	0% (0/1)	0% (0/4)	0% (0/6)
level	23% (3/13)	8% (7/78)	6% (23/352)
npc	40% (4/10)	12% (6/47)	6% (17/243)
points	0% (0/2)	0% (0/7)	0% (0/19)
sprite	83% (5/6)	46% (21/45)	54% (71/130)
ui	0% (0/6)	0% (0/31)	0% (0/127)
Launcher	0% (0/1)	0% (0/21)	0% (0/41)
LauncherSmokeTest	0% (0/1)	0% (0/4)	0% (0/29)
PacmanConfigurationExceptio	0% (0/1)	0% (0/2)	0% (0/4)

Fig. 7: Coverage increase from LevelFactoryTest.java

The third unit test I created was `PelletTest.java` which tested the `getValue()` method of the `Pellet` class in the package `level`. It also uses and tests `getPelletSprite()` of the `PacManSprites` class in the package `sprite`. Figure 8 shows this unit test and Figure 9 shows the increase in code coverage as a result of it.

```java
public class PelletTest {
    1 usage
    private final PacManSprites sprites = new PacManSprites();
    1 usage
    private final Sprite sprite = sprites.getPelletSprite();
    1 usage
    private Pellet pellet = new Pellet( points: 250, sprite);

    Tarik Resimovic
    @Test
    void pelletTest() {
        int pelletValue = pellet.getValue();
        assertThat( actual: pelletValue == 250).isEqualTo( expected: true);
    }
}
```

Fig. 8: PelletTest.java

Element	Class, %	Method, %	Line, %
nl	27% (15/55)	13% (43/312)	11% (132/1162)
tudelft	27% (15/55)	13% (43/312)	11% (132/1162)
jpacman	27% (15/55)	13% (43/312)	11% (132/1162)
board	20% (2/10)	11% (6/53)	10% (15/141)
fuzzer	0% (0/1)	0% (0/6)	0% (0/32)
game	0% (0/3)	0% (0/14)	0% (0/37)
integration	0% (0/1)	0% (0/4)	0% (0/6)
level	30% (4/13)	11% (9/78)	7% (28/353)
npc	40% (4/10)	12% (6/47)	6% (17/243)
points	0% (0/2)	0% (0/7)	0% (0/19)
sprite	83% (5/6)	48% (22/45)	55% (72/130)
ui	0% (0/6)	0% (0/31)	0% (0/127)
Launcher	0% (0/1)	0% (0/21)	0% (0/41)
LauncherSmokeTest	0% (0/1)	0% (0/4)	0% (0/29)
PacmanConfigurationExceptio	0% (0/1)	0% (0/2)	0% (0/4)

Fig. 9: Coverage increase from PelletTest.java

Task 3

jpacman

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
nl.tudelft.jpacman.level		67%		57%	74	155	103	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		88%		62%	29	70	10	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	1,203 of 4,694	74%	290 of 637	54%	291	590	227	1,039	51	268	6	47

Fig. 10: JaCoCo coverage report

- **Are the coverage results from JaCoCo similar to the ones you got from IntelliJ in the last task? Why or why not?**

  The results from the two sources seem different. They seem to count different total numbers of certain elements of the source code. For example, IntelliJ lists 1,162 lines of code whereas JaCoCo lists 1,039. It also seems that JaCoCo reports much higher code coverage than what IntelliJ reports.

- **Did you find helpful the source code visualization from JaCoCo on uncovered branches?**

  Yes, it was helpful in pinpointing exactly what parts of the code were only partially tested or completely untested.

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

  I preferred IntelliJ's for initial coverage analysis because it was simpler and easier to read. However, I think JaCoCo's report is more helpful when you need to pinpoint exactly what parts of code in a specific section still need to be tested. So if you do extensive testing and are maybe missing only a few lines or branches left to be tested, JaCoCo is more useful than IntelliJ in showing what exactly is left to be tested.

End of report for tasks 2 and 3. This is the version that is uploaded to GitHub since tasks 4 and 5 do not require uploading anything to GitHub. The full version of the report including tasks 4 and 5 will be uploaded to Canvas.