Software Testing Assignment 5: Mutation Testing

Lars De Leeuw 20205693
Robbe Teughels 20211127

April 2025

Contents

1 MUTATION TESTING

1.1 Exercise 1

> Use LittleDarwin to analyse the original version of JPacMan. LittleDarwin and its manual can be seen and downloaded from https://github.com/aliparsai/LittleDarwin. Explain the results. Now, use LittleDarwin to analyse your version of JPacMan. What differences can you see? - *Lars*

We ran LittleDarwin with the following command.

```
python3 -m littledarwin -m -b -p src/main/java/jpacman/model/ -t .
    --timeout=4 -c mvn,clean,test
```

We first ran LittleDarwin in the original JPacMan repository, the result of which can be found in table 1.1. Then in our version which contains modification made in function of the assignments, result of which can be found in table 1.1.

Comparing the two runs of LittleDarwin we first notice that a there is a noticeable amount of mutation tests run on our version. The original JPacMan only had 162 mutation tests whilst ours had 207. This makes sense given we expanded upon the original code base adding new features such as the undo button. Another difference is that our version has a higher overall mutation coverage percentage, taking the sum of the killed mutants and dividing by the sum of mutation tests run over the entire table we obtain a mutation coverage percentage of 81% for our version and 73% percent. Where the most notable jump in mutation coverage can be seen in the Player class which has a measly 20% mutation coverage in the original version and jumps up to 87.5% in our version.

Table 1: LittleDarwin Mutation Coverage Original JPacMan

Name	Percentage	Details
Board.java	83.3%	10/12
Cell.java	75.0%	12/16
Engine.java	65.6%	21/32
Food.java	75.0%	3/4
Game.java	79.5%	31/39
Guest.java	91.7%	11/12
Monster.java	0.0%	0/2
Move.java	80.6%	25/31
Player.java	20.0%	1/5
PlayerMove.java	42.9%	3/7
Wall.java	100.0%	2/2

Table 2: LittleDarwin Mutation Coverage Our Version

Name	Percentage	Details
Board.java	89.5%	17/19
Cell.java	56.5%	13/23
Engine.java	84.4%	27/32
Food.java	83.3%	5/6
Game.java	83.3%	40/48
Guest.java	91.7%	11/12
Monster.java	100.0%	4/4
MonsterMove.java	33.3%	1/3
Move.java	84.2%	32/38
Player.java	87.5%	7/8
PlayerMove.java	70.0%	7/10
Wall.java	100.0%	4/4

1.2 Exercise 2

> Repeat the previous exercise, but this time use branch coverage (with JaCoCo) instead of mutation coverage. Look for a class with 100% branch coverage and less than 100% mutation coverage. Why did this happen? Look for similar examples (interesting differences between the coverages of the two techniques) and explain the difference between two results. - *Lars*

When we look at the branch coverage provided by JaCoCo we cannot find a single class with 100% branch coverage, this is because JaCoCo is counting the branches which get created by the assert statements in the code. We tried several methods such as running without assertions enabled but no matter what we tried our JaCoCo kept including the assertion branches. We also cannot simply write more tests to obtain the 100% branch coverage because some of the branches are unreachable thanks to the enforcement of the assert statements. For example in the Cell class we have the methods getX() and getY() which start with checking the invariant, JaCoCo sees this as 2 branches, one where the invariant holds and one where it doesn't, but we are never able to call this method in the case the invariant doesn't hold because the invariant is enforced in the constructor and all other public methods which modify the Cell. As a compromise I tried increasing the branch coverage as high as I could and went from our previous JaCoCo branch coverage of 62% to 68%.

The Cell class now has our highest branch coverage according to JaCoCo, yet the mutation coverage is only 56.5%.

The Monster class has 100% mutation coverage but only 50% branch coverage. Looking at the mutations which were tried we only see mutations in the assert statements which will then evaluate to false and raise an AssertionError in the test-cases providing coverage for the Monster class. There are also only 4 mutants created for this class which makes a high mutation coverage easier to achieve.

Table 3: JaCoCo Branch Coverage Our JPacMan

Name	Percentage
Wall	50%
Game	63%
Monster	50%
Move	60%
Engine	64%
Food	60%
Cell	69%
MovingGuest	n/a
Player	50%
PlayerMove	50%
Guest	57%
Board	68%
MonsterMove	50%

1.3 Exercise 3

> Repeat the first exercises using PITest or Javalanche. Explain the difference in the results. - *Lars*

1.3.1 Installation & Use

On the LittleDarwin github page installation is shown to be as simple as pip installing the package, yet both my colleague and I ran in several difficulties regarding missing dependencies, version mismatches, needing a specific python version, ... The both of us using a different OS (Linux and Windows) didn't make resolving these issues any easier.

Installing PITest was in comparison a breeze, I simply added the following in the build/plugins section of our pom file:

```
      <plugin>
        <groupId>org.pitest</groupId>
        <artifactId>pitest-maven</artifactId>
        <version>1.19.1</version>
        <configuration>
          <targetClasses>
            <param>jpacman.*</param>
          </targetClasses>
          <targetTests>
            <param>jpacman.*</param>
```

```xml
          </targetTests>
          <threads>4</threads>
          <outputFormats>
            <param>HTML</param>
          </outputFormats>
          <failWhenNoMutations>false</failWhenNoMutations>
          <mutators>
            <mutator>DEFAULTS</mutator>
          </mutators>
          <argLine>-enableassertions</argLine>
        </configuration>
      </plugin>
```

When then executing the command:

```
mvn clean test org.pitest:pitest-maven:1.19.1:mutationCoverage
```

Revealed an error which was quickly pin-pointed to being due to the old JUnit 4 version we were using, so after updating to a supported JUnit 4 version (4.6+) it worked flawlessly!

PITest was also way faster than LittleDarwin requiring only a couple of seconds compared to the 7-10 minutes LittleDarwin required on my machine (on my collegues machine LittleDarwin took nearly 1 hour).

1.3.2 Results

Firstly we notice that the speed benefit of PITest does not come at the cost of amount of mutants tested. For the original JPacman model packages we ran 218 mutants compared to the 162 with LittleDarwin, our modified version of JPacman ran 272 mutants compared to the 207 with LittleDarwin. Do note that quality of the mutants is also if not more important than the sheer quantity of mutants.

The overall mutation coverage for the model package with PITest for the original version is 65% which is lower than the 73% of LittleDarwin. However looking at the test strength which only includes mutations which are actually covered by a test from the test suite we get 74% which is similar to LittleDarwin. For our version the mutation coverage is 81% which is the same as the 81% obtained with LittleDarwin, the test strenght is even higher at 85%.

Wall.java is the only class with 100% mutation coverage and this is only in our version, whilst with LittleDarwin we had 100% for both Wall.java and Monster.java. Looking closer at Monster.java we find that this class only has a 50% mutation coverage with PITest. Examining the survived mutants we find that the two survived mutants changed a constant false return value to true in meetPlayer and meetMonster respectively. This boolean signifies whether or not the monsters cell may be occupied when the Player or another Monster tries to move to it. I think these are good mutations showing that our tests can be improved thus the 100% given by LittleDarwin is misleading.

PlayerMove.java has a notably higher mutation coverage in both the original version and our modified than with LittleDarwin. In our modified version we have a single mutant holding us back from 100% mutation coverage so I took a closer look. The surviving mutant replaced the boolean expression for the invariant of PlayerMove with true, however we can never write a test to kill this mutant because to do so we would need to reach a state which can't be reached due to earlier assertions. Thus this mutant is actually an equivalent one leaving us with a 100% mutation coverage.

Table 4: PITest Mutation Coverage Original JPacMan

Name	Line Coverage	Mutation Coverage	Test Strength
Board.java	98% 40/41	80% 16/20	80% 16/20
Cell.java	100% 38/38	63% 12/19	63% 12/19
Engine.java	67% 54/81	54% 31/57	79% 31/39
Food.java	100% 16/16	75% 6/8	75% 6/8
Game.java	91% 94/103	66% 31/47	72% 31/43
Guest.java	100% 25/25	63% 5/8	63% 5/8
Monster.java	44% 4/9	33% 1/3	100% 1/1
Move.java	93% 56/60	76% 22/29	76% 22/29
Player.java	67% 20/30	58% 7/12	64% 7/11
PlayerMove.java	100% 27/27	77% 10/13	77% 10/13
Wall.java	100% 8/8	50% 1/2	50% 1/2

Table 5: PITest Mutation Coverage Our Version

Name	Line Coverage	Mutation Coverage	Test Strength
Board.java	98% 40/41	86% 24/28	86% 24/28
Cell.java	100% 47/47	85% 23/27	85% 23/27
Engine.java	76% 71/93	67% 42/63	84% 42/50
Food.java	100% 20/20	89% 8/9	89% 8/9
Game.java	97% 134/138	86% 49/57	88% 49/56
Guest.java	100% 25/25	63% 5/8	63% 5/8
Monster.java	100% 13/13	50% 2/4	50% 2/4
MonsterMove.java	100% 22/22	80% 8/10	80% 8/10
Move.java	100% 73/73	88% 29/33	88% 29/33
Player.java	100% 40/40	79% 11/14	79% 11/14
PlayerMove.java	100% 34/34	94% 15/16	94% 15/16
Wall.java	100% 12/12	100% 3/3	100% 3/3

1.3.3 Conclusion

I believe both tools can be of use in finding weak spots in test-suits, however if I would have to choose one it would overwhelmingly be PITest. There are several reasons for this. The first is just how much faster PITest was which allows for rapid feedback when improving the test-suite. The second is that I found the generated report to be much more comprehensible. Lastly I also prefer the types of mutations made by PITest compared to LittleDarwin. From what I saw LittleDarwin produced a lot more change == to != type of mutations whilst PITest seemed to create mutations which seem more useful and less granular like removing statements causing the side-effect of the function to be wrong. However I may be wrong in this as I did not look at every single mutant in detail in both LittleDarwin and PITest.

1.4 Exercise 4

> Find a killed mutant. Explain why, where, and how it was killed? - *Lars*

The following is a killed mutant of the Engine class, it modified the inPlayingState() method which is a logical conjuction of which is meant to only return true if the Engine state machine is exactly in one state and that state is the PlayingState. The mutation removed one of the logical negations causing the method to only return true if the state machine is in both the PlayerWon state AND the PlayingState which is functionally wrong for the inPlayingState() method.

With this mutation applied a total of 9 test-cases out of 19 for the Engine class fail. Successfully killing the mutant. The 9 test-cases are: StartGameKillRestartEatRoamWin, GameOverDiedSneakPath, GameOverWonSneakPath, GameOverLostHaltedPathUndo, GameOverWonHaltedPathUndo, StartingHaltedPathUndo, StartGameTakeBreakDie, PlayingSneakPath, HaltedSneakPath. All these test-cases use the Engine.start() method which enters the state-machine into the PlayingState, at the end of this method we now we should only be in the PlayingState and this is asserted with the invariant() but because the inPlayingState() now only returns true if it is both in the PlayingState AND the PlayerWon state the assertion fails causing the test-case to fail as well.

```
/* LittleDarwin generated order-1 mutant
mutant type: ConditionalOperatorDeletion
----> before:         && !theGame.playerDied() && !theGame.
    playerWon();
----> after:         && !theGame.playerDied() &&  theGame.playerWon
    ();
----> line number in original file: 48
----> mutated node: 868

...
```

```

/**
 * We're actually playing a game.
 * @return true iff game is still on.
 */
public synchronized  boolean inPlayingState() {
    return !starting && !halted && theGame.initialized()
    && !theGame.playerDied() &&  theGame.playerWon();
}
```

1.5 Exercise 5

> Take a survived mutant for class Engine, and write a test that kills it. Repeat the process until all survived mutants from the Engine class are covered. Rerun LittleDarwin to confirm. Note that you can run LittleDarwin for specific classes/files. This drastically cuts down on run time.

- *Lars*

The Engine class has 5 survived mutants, where 4 of them are equivalent mutations of the invariant code listed below. This piece of code simply checks whether or not we are in only one state of our state machine and our Game object is not null.

```
/**
* We can be in at most one of the Engine's states.
*
* @return True if we in exactly one state.
*/
protected boolean invariant() {
// beware of the xor:
// xor on odd nr of args also permits all args to be true.
// (see in3420 exam in 2003)
boolean oneStateOnly = inStartingState()
^ inPlayingState()
^ inHaltedState()
^ inDiedState()
^ inWonState()
&& !(inStartingState() && inPlayingState() && inHaltedState()
        && inDiedState() && inWonState());
return oneStateOnly && theGame != null;
}
```

1.5.1 Mutant 1

```
/* LittleDarwin generated order-1 mutant
mutant type: ConditionalOperatorReplacement
----> before:          return oneStateOnly && theGame != null;
----> after:         return oneStateOnly || theGame != null;
----> line number in original file: 99
----> mutated node: 704
*/
```

```

...

/**
 * We can be in at most one of the Engine's states.
 *
 * @return True if we in exactly one state.
 */
protected boolean invariant() {
    // beware of the xor:
    // xor on odd nr of args also permits all args to be true.
    // (see in3420 exam in 2003)
    boolean oneStateOnly = inStartingState()
    ^ inPlayingState()
    ^ inHaltedState()
    ^ inDiedState()
    ^ inWonState()
    && !(inStartingState() && inPlayingState() && inHaltedState()
            && inDiedState() && inWonState());
    return oneStateOnly || theGame != null;
}
```

This mutant changes the AND to OR at the return statement, given we have a properly implemented state machine the oneStateOnly will always be true when theGame is not null and will even crash in the case where theGame is null because some of the helper methods to determine the state access theGame and will cause null reference exceptions. Thus the OR is functionally the same as the original AND.

1.5.2 Mutant 2

```
/* LittleDarwin generated order-1 mutant
mutant type: ConditionalOperatorReplacement
----> before:         ^ inWonState() || !(inStartingState() &&
    inPlayingState()
----> after:          ^ inWonState() && !(inStartingState() &&
    inPlayingState()
----> line number in original file: 92
----> mutated node: 1313
*/

...

/**
 * We can be in at most one of the Engine's states.
 *
 * @return True if we in exactly one state.
 */
protected boolean invariant() {
    // beware of the xor:
    // xor on odd nr of args also permits all args to be true.
    // (see in3420 exam in 2003)
    boolean oneStateOnly = inStartingState()
    ^ inPlayingState()
    ^ inHaltedState()
    ^ inDiedState()
    ^ inWonState()
    || !(inStartingState() && inPlayingState() && inHaltedState()
            && inDiedState() && inWonState());
    return oneStateOnly && theGame != null;
}
```

This mutation again changes an AND to an OR, changing the meaning of oneStateOnly to the XOR over the in*State() methods returns true AND we are not in all states at the same time to the XOR over the in*State() methods returns true OR we are not in all states at the same time. Now again thanks to our proper state machine implementation where we can only ever be in one state we can never kill this mutant.

1.5.3 Mutant 3

```
/* LittleDarwin generated order-1 mutant
mutant type: ConditionalOperatorReplacement
----> before:          && inDiedState() && inWonState());
----> after:           || inDiedState() && inWonState());
----> line number in original file: 97
----> mutated node: 1576
*/

...

/**
 * We can be in at most one of the Engine's states.
 *
 * @return True if we in exactly one state.
 */
protected boolean invariant() {
    // beware of the xor:
    // xor on odd nr of args also permits all args to be true.
    // (see in3420 exam in 2003)
    boolean oneStateOnly = inStartingState()
    ^ inPlayingState()
    ^ inHaltedState()
    ^ inDiedState()
    ^ inWonState()
    && !(inStartingState() && inPlayingState() && inHaltedState()
            || inDiedState() && inWonState());
    return oneStateOnly && theGame != null;
}
```

This mutation also changes an AND to an OR, which again changes the meaning of the invariant slightly but due to our proper state machine implementation where we can only ever be in one state we can never kill this mutant.

1.5.4 Mutant 4

```
/* LittleDarwin generated order-1 mutant
mutant type: ConditionalOperatorReplacement
```

```
----> before:        && !(inStartingState() && inPlayingState() &&
    inHaltedState()
----> after:         && !(inStartingState() && inPlayingState() ||
    inHaltedState()
----> line number in original file: 97
----> mutated node: 1602
*/

...

/**
 * We can be in at most one of the Engine's states.
 *
 * @return True if we in exactly one state.
 */
protected boolean invariant() {
    // beware of the xor:
    // xor on odd nr of args also permits all args to be true.
    // (see in3420 exam in 2003)
    boolean oneStateOnly = inStartingState()
    ^ inPlayingState()
    ^ inHaltedState()
    ^ inDiedState()
    ^ inWonState()
    && !(inStartingState() && inPlayingState() || inHaltedState()
            && inDiedState() && inWonState());
    return oneStateOnly && theGame != null;
}
```

This mutation also changes an AND to an OR, which again changes the meaning of the invariant slightly but due to our proper state machine implementation where we can only ever be in one state we can never kill this mutant.

1.5.5 Mutant 5

The fifth surviving mutant is shown below, this mutation is also an equivalent one. This mutation changes an logical OR to an AND in the inStartingState() method. In theory this would allow the method to potentially return true when it is not supposed to when the starting variable is set to true (which can only be if the state machine is in the StartingState) and if the player died (only possible in the PlayerDiedState) XOR the player won (only possible in the PlayerWonState). The player can never die until the game has exited the StartingState and entered the PlayingState because the Engine does not allow moves in any other state. The player can also never enter the PlayerWonState before the PlayingState even when we try to manually remove food from a properly initialized map. Because of the proper implementation of the state machine of the Engine class we can thus never be in more than 1 state and thus we can never kill this mutant.

```
/* LittleDarwin generated order-1 mutant
mutant type: ConditionalOperatorReplacement
----> before:         return starting && !(theGame.playerDied() ||
    theGame.playerWon());
```

```
----> after:          return starting && !(theGame.playerDied() &&
    theGame.playerWon());
----> line number in original file: 39
----> mutated node: 1292

*/

...

/**
 * The game has been set up, and is just waiting to get started.
 * @return true iff game is starting.
 */
public synchronized boolean inStartingState() {
    return starting && !(theGame.playerDied() && theGame.playerWon
    ());
}
```

1.6 Exercise 6

> Can you find an example of an equivalent mutant? How do you know (proof) it is equivalent?

- *Lars*

For the Engine class mutant 4 1.5.4 and 5 1.5.5 from the previous exercise are not only equivalent to each other but also to the non-mutated code. The relevant non-mutated code comes down to:

```
inStartingState() && inPlayingState() && inHaltedState() &&
    inDiedState() && inWonState()
```

Whilst the mutations look like the following:

```
inStartingState() && inPlayingState() || inHaltedState() &&
    inDiedState() && inWonState()
```

```
inStartingState() && inPlayingState() && inHaltedState() ||
    inDiedState() && inWonState()
```

The Engine class in our implementation is a state-machine which can only ever be in 1 state at a time. The non-mutated code can logically only ever evaluate to true if the state-machine is in all states at once, which cannot occur due to our correct implementation. The first mutation can evaluate to true if the state machine is in all states but also if the state machine is in:

- StartingState AND Playing State AND DiedState AND WonState
- StartingState AND Halted State AND DiedState AND WonState
- Playing State AND Halted State AND DiedState AND WonState
- Halted State AND DiedState AND WonState

Which are 4 more ways to evaluate to true compared to the non-mutated code however all of these require the state machine to be in more than one state which is not possible due to our correct implementation. The second mutation is similar also requiring the state machine to be in more than one state for the other ways to evaluate to true.

1.7 Exercise 7

> Make a mutation-adequate test suite for the model package of JPac-Man. How many tests did you have to write? (Count them!) How many equivalent mutants did you find? Explain why each of them is equivalent. Note that you can run LittleDarwin for specific classes/files. This drastically cuts down on run time.

- *Robbe*

1.7.1 Board

```java
1      /**
2       * A board's invariant is simply that both the width and the height are not
3       * negative.
4       *
5       * @return True iff widht and height nonnegative.
6       */
7      protected boolean invariant() {
8          return width >= 0 || height >= 0;
9      }
```

Listing 1: "Equivalent mutation 1 of Board"

This mutant changes the && to || on line 8. The creation of the board only allows values greater than or equal to 0; the modification of these parameters is fully prohibited. This gives always a true evaluation of the left and right sides of the equation, resulting in a true outcome for both && and ||. This is an equivalent mutation.

```java
1      /**
2       * Check that each cell has a correct link to this board. This function
3       * could be part of the invariant, but checking it each time is considered
4       * too expensive, which is why it is offered as a separate function.
5       *
6       * @return True iff the cell/board association is consistent
7       */
8      protected boolean consistentBoardCellAssociation() {
9          boolean result = true;
10         for (Cell[] row : cellAt) {
11             for (Cell c : row) {
12                 result = result || c.getBoard().equals(this);
13             }
14         }
15         return result;
16     }
```

Listing 2: "Equivalent mutation 2 of Board"

This mutant changes the && to || on line 12. From the creation of the board, adding a cell to the board gives a cell assigned to that board without any means to change the board in a given cell or change a cell of the board. This gives always a true evaluation of the right side of the equation. Therefore, as the previous, this an equivalent mutation.

1.7.2 Cell

```
@Test(expected = AssertionError.class)
public void testInvalidCellCreation() {
    try {
        Cell cell = new Cell(20, 20, aBoard);
    } catch (Exception e) {
        assert false;
    }
}

@Test
public void testAdjacentNotZeroLocationX() {
    Cell cell = new Cell(1, 1, aBoard);
    Cell other = new Cell(2, 1, aBoard);
    assertTrue(cell.adjacent(other));
}

@Test
public void testAdjacentNotZeroLocationY() {
    Cell cell = new Cell(1, 1, aBoard);
    Cell other = new Cell(1, 2, aBoard);
    assertTrue(cell.adjacent(other));
}

@Test(expected = AssertionError.class)
public void testAdjacentDifferentBoard() {
    try {
        Cell cell = new Cell(1, 1, aBoard);
        Board otherBoard = new Board(width, height);
        Cell other = new Cell(2, 1, otherBoard);
        cell.adjacent(other);
    } catch (Exception e) {
        assert false;
    }
}

@Test(expected = AssertionError.class)
public void testFreeWhileOccupied() {
    try {
        Cell cell = new Cell(1, 1, aBoard);
        Player player = new Player();
        player.occupy(cell);
        cell.free();
    } catch (Exception e) {
        assert false;
    }
}
```

Listing 3: "Added testcases of Cell"

With 5 additional tests, we have eliminated all 10 surviving 5.7.Mutants.

1.7.3 Engine

```
    public synchronized boolean inStartingState() {
        return starting && !(theGame.playerDied() && theGame.
    playerWon());
    }
```

Listing 4: "Equivalent mutation 1 of Engine"

Because of the proper definition and implementation of the Engine class, it is impossible to be in two states at a given time. This tells us that only 1 of the parameters can be true. Allowing an equivalent mutation to change the inner || to &&.

```
    protected boolean invariant1() {
        // beware of the xor:
        // xor on odd nr of args also permits all args to be true.
        // (see in3420 exam in 2003)
        boolean oneStateOnly = inStartingState()
        ^ inPlayingState()
        ^ inHaltedState()
        ^ inDiedState()
        ^ inWonState()
        && !(inStartingState() && inPlayingState() && inHaltedState
    ()
                && inDiedState() && inWonState());
        return oneStateOnly || theGame != null;
    }
    protected boolean invariant2() {
        boolean oneStateOnly = inStartingState()
        ^ inPlayingState()
        ^ inHaltedState()
        ^ inDiedState()
        ^ inWonState()
        || !(inStartingState() && inPlayingState() && inHaltedState
    ()
            && inDiedState() && inWonState());
        return oneStateOnly && theGame != null;
    }
    protected boolean invariant3() {
        boolean oneStateOnly = inStartingState()
        ^ inPlayingState()
        ^ inHaltedState()
        ^ inDiedState()
        ^ inWonState()
        && !(inStartingState() && inPlayingState() && inHaltedState
    ()
            || inDiedState() && inWonState());
        return oneStateOnly && theGame != null;
    }
    protected boolean invariant4() {
        boolean oneStateOnly = inStartingState()
        ^ inPlayingState()
        ^ inHaltedState()
        ^ inDiedState()
        ^ inWonState()
        && !(inStartingState() && inPlayingState() || inHaltedState
    ()
            && inDiedState() && inWonState());
        return oneStateOnly && theGame != null;
    }
```

Listing 5: "Equivalent mutation 2-5 of Engine"

Here, the same type of mutations occurs. Only 1 parameter can be true, allowing for many equivalent mutations in this section.

1.7.4 Food

```
@Test(expected = AssertionError.class)
public void testCreateInvalidFood() {
    try {
        Food food = new Food(-1);
    } catch (Exception e) {
        assert false;
    }
}
```

Listing 6: "Added testcase of Food"

1 additional test, to eliminate 1 mutation.

1.7.5 Game

```
    /**
     * Adds the move to the stack
     * @param move the move executed
     */
    protected void pushMoveStack(Move move) {
        assert movedStack != null || !movedStack.isEmpty();
        movedStack.lastElement().add(move);
        assert movedStack.lastElement().contains(move);
    }
```

```
    /**
     * Pop the movedStack
     * @return A list containing all moves happened after the
    segment created.
     */
    protected Vector<Move> popMoveStack() {
        assert movedStack != null || !movedStack.isEmpty();
        var retval = movedStack.removeLast();
        if (movedStack.isEmpty()) {
            nextSegMoveStack();
        }
        assert !movedStack.isEmpty();
        return retval;
    }
```

Listing 7: "Equivalent mutation 1-2 of Game"

The mutations change the && to || on line 6 and 15. The evaluation of both sides is True after ‘initialize‘ is called. As this happens in the construction of this instance, it is impossible to reach these parts of the code with a different evaluation. Removing an element of the stack ensures that there is always an element present ofter the opperation, guaranteeing true evaluation of the right side. Resulting in 2 equivalent mutations.

```java
@Test
public void testFinishGame() {
    for (int i = 0; i < theGame.boardWidth() * theGame.boardHeight
    (); i++) {
        var cell = theGame.getBoard().getCell( i% theGame.
    boardWidth(), i/theGame.boardHeight());
        var inhab = cell.getInhabitant();
        if (inhab instanceof Food) {
            var move = new PlayerMove(thePlayer, cell);
            move.apply();
        }
    }
    assertTrue(theGame.gameOver());
    assertTrue(theGame.playerWon());
}

@Test
public void testMorePointsEatenThanTotal() {
    theGame.getPlayer().eat(999);
    assertFalse(theGame.invariant());
}

@Test(expected = AssertionError.class)
public void testInvalidCustomMap() {
    var map = new String[]{"WWW", "WWW"};
    var game = new Game(map);
}
```

Listing 8: ”Added testcases of Game”

3 additional tests, eliminating 6 mutations.

1.7.6 Guest

```java
@Test
public void testDeoccupyFromNull() {
    var guest = new Player();
    assertNull(guest.getLocation());
    assertNull(guest.deoccupy());
}
```

Listing 9: ”Added testcase of Guest”

1 additional test, eliminating 1 mutation.

1.7.7 Monster Move

```java
    /**
     * Verify that the monster/mover equal
     * and non-null.
     *
     * @return true iff the invariant holds.
     */
    public boolean invariant1() {
        return moveInvariant() || theMonster != null &&
    getMovingGuest().equals(theMonster);
    }
    public boolean invariant2() {
        return moveInvariant() && theMonster != null ||
    getMovingGuest().equals(theMonster);
    }
```

Listing 10: "Equivalent mutation 1-3 of Monster Move"

The three parts of the expression will always return True. This gives us the equivalent mutations that switches the && int ||.

1.7.8 Move

```
    /**
     * Check that the guest to be moved indeed occupies a cell.
     * Furthermore, moves that cause the player to die are not
    possible.
     *
     * @return True iff the mover occupies a cell.
     */
    public boolean moveInvariant1() {
        return mover != null
            || fromCell != null
            && mover.getLocation() != null
            && (!initialized || !(movePossible() && playerDies));
    }
    public boolean moveInvariant2() {
        return mover != null
                && fromCell != null
                || mover.getLocation() != null
                && (!initialized || !(movePossible() && playerDies)
    );
    }
    public boolean moveInvariant3() {
        return mover != null
                && fromCell != null
                && mover.getLocation() != null
                || (!initialized || !(movePossible() && playerDies)
    );
    }
```

Listing 11: "Equivalent mutation 1-3 of Move"

The 4 sub-expressions in this function will always evaluate True once the constructor is called. All attemts to make any of these False, results in an assertion in the constructor itself, not reaching this part of the code. As these are always True, the && can be changed to || without changing the outcome of the entire expression. Resulting in 3 equivalent mutations.

```
@Test(expected = AssertionError.class)
public void testCreateMoveToNull() {
    aMove = createMove(null);
}
```

```java

@Test
public void testTryMoveToGuestPreconditionGuestNull() {
    aMove = createMove(emptyCell);
    assertFalse(aMove.tryMoveToGuestPrecondition(null));
}

@Test
public void testTryMoveToGuestPreconditionAlreadyInit() {
    aMove = createMove(theFood.getLocation());
    assertFalse(aMove.tryMoveToGuestPrecondition(theFood));
}
```

Listing 12: "Added testcases of Move"

3 additional tests, to eliminate 3 mutations.

1.7.9 Player

```java
@Test(expected = AssertionError.class)
public void testTotalNegativeFood() {
    try {
        Player p = new Player();
        p.eat(-1);
    } catch (Exception e) {
        assert false;
    }
}
```

Listing 13: "Added testcase of Player"

1 additional test, to eliminate 1 mutation.

1.7.10 PlayerMove

```java
    /**
     * Verify that the food eaten remains non negative, the player/
    mover equal
     * and non-null.
     *
     * @return true iff the invariant holds.
     */
    public boolean invariant1() {
        return moveInvariant() || foodEaten >= 0 && thePlayer !=
    null
        && getMovingGuest().equals(thePlayer);
    }
    public boolean invariant2() {
        return moveInvariant() && foodEaten >= 0 || thePlayer !=
    null
                && getMovingGuest().equals(thePlayer);
    }
    public boolean invariant3() {
        return moveInvariant() && foodEaten >= 0 && thePlayer !=
    null
                || getMovingGuest().equals(thePlayer);
    }
```

Listing 14: "Equivalent mutation 1-3 of PlayerMove"

The 4 sub-expressions will always evaluate True for a valid instance. As the creation of an instance requires a valid instance of Move, the move variant, $thePlayer \neq null$ and $getMovingGuest().equals(thePlayer)$ will always be True. The foodEaten can only be modified when moving to a Food guest. As these can only have a positive foodvalue, foodEaten will also be positive. This allows for the && to be replaced by ||, resulting in 3 equivalent mutations.

1.7.11 Recap

In total, we have added 14 test cases.

1.8 Exercise 8

> Make a mutation-adequate test suite for the controller package of JPacMan. How many tests did you have to write? (Count them!) How many equivalent mutants did you find? Explain why each of them is equivalent.

- *Robbe*

1.8.1 AbstractMonsterController

```java
@Test(expected = AssertionError.class)
public void testMonsterMoverEngineNull() {
    createController(null);
}
```

Listing 15: "Added testcase of AbstractMonsterController"

1 additional test, eliminating 2 mutation.

1.8.2 BoardViewer

```java
@Test
public void testSerialVersionUnchanged() {
    assertEquals(-4976741292570616918L, BoardViewer.
    serialVersionUID);
}

@Test
public void testAnimationSequence() {
    int count = theBoardViewer.getAnimationCount();
    while (count != 0) {
        theBoardViewer.nextAnimation();
        count = theBoardViewer.getAnimationCount();
    } /* Start at the beginning of the sequence */

    int count2;
    for (int i = 0; i < theImageFactory.monsterAnimationCount() *
    theImageFactory.playerAnimationCount() -1; i++) {
        theBoardViewer.nextAnimation();
        count2 = theBoardViewer.getAnimationCount();
        assertEquals(count + 1, count2);
        count = count2;
    }
    theBoardViewer.nextAnimation();
    assertEquals(0, theBoardViewer.getAnimationCount());
}
```

Listing 16: "Added testcases of BoardViewer"

2 additional tests, eliminating 3 mutations, leaving 28 surviving mutations, possibly equivalent.

This class heavily relays on JPanel to display graphics. Just setting up the test for testAnimationSequence, we had to unprivatize some fields to have the test not hard-coded (ImageFactory for the amount of frames instead of magic number 12) and extending the class with a getAnimationCount() to be able to check values. Continuing with the creation of graphical elements, we were unable to write useful test as almost everything is hidden within the libraries. This class is not cost-effective to write automated tests for.

1.8.3 ImageFactory

```
    /**
     * Invariant that may be a bit expensive to compute all the
    time,
     * so it is selectively invoked.
     * @return true iff invariant holds and all images are non-null
    .
     */
    public boolean invariant() {
        boolean result = monsterImage != null;
        result = result && monsterImage.length > 0;
        for (int i = 0; i < monsterImage.length; i++) {
            result = result && monsterImage[i] != null;
        }
        result = result && playerImage != null;
        result = result && playerImage[0] != null;
        for (int i = 0; i < playerImage.length; i++) {
            for (int j = 0; j < playerImage[i].length; j++) {
                result = result && playerImage[i][j] != null;
            }
        }
        return result;
    }
```

Listing 17: "Equivalent mutations 1-8 of ImageFactory"

To create a valid ImageFactory instance, all files must be present and correctly loaded. Otherwise, an exception is thrown. This means that when we can call the invariant, all checks $! = null$ will be true. All modifications for the traversal of the images and && operations, will be equivalent, as long as the 'for loops' remain finite. This gives us a total of 8 equivalent mutations.

```
@Test
public void testPlayerDefault() {
    assertNotNull(imf.player(0,0,0));
}

```

```java
@Test
public void testMonsterAnimation() {
    Image m1 = imf.monster(0);
    Image m2 = imf.monster(1);
    Image m3 = imf.monster(2);
    assertNotSame(m1, m2);
    assertNotSame(m2, m3);
}
```

Listing 18: "Added testcases of ImageFactory"

2 additional tests, eliminating 3 mutations.

1.8.4 Pacman

```java
    /**
     * Instance variables that can't be null.
     * @return True iff selected instance variables all aren't null
    .
     */
    protected boolean invariant1() {
        return theEngine != null || monsterTicker != null &&
    theViewer != null;
    }
    protected boolean invariant2() {
        return theEngine != null && monsterTicker != null ||
    theViewer != null;
    }
```

Listing 19: "Equivalent mutations 1-2 of Pacman"

Constructing an instance requires a not null engine and not null monster ticker (or will be created). A new viewer will also be created. This means that the 3 not null checks will always succeed when the object is created, without any means of changing it. This gives us 2 equivalent mutations to change the && int ||.

```java
private void testMove(Runnable move, int dx, int dy) {
    pacman.start();
    var old_location = pacman.getEngine().getPlayer().getLocation()
    ;
    move.run();
    assertEquals(dx, pacman.getEngine().getPlayerLastDx());
    assertEquals(dy, pacman.getEngine().getPlayerLastDy());
    var location = pacman.getEngine().getPlayer().getLocation();
    assertEquals(old_location.getX() + dx, location.getX());
    assertEquals(old_location.getY() + dy, location.getY());
}

@Test public void testPlayerMoveLeft() {
    testMove(() -> pacman.left(),-1, 0);
}

@Test public void testPlayerMoveRight() {
    testMove(() -> pacman.right(),1, 0);
}

@Test public void testPlayerMoveUp() {
    testMove(() -> pacman.up(),0, -1);
}

@Test public void testPlayerMoveDown() {
    testMove(() -> pacman.down(),0, 1);
}

@Test
public void testMainWithArgsPrintsWarning() {
    PrintStream originalErr = System.err;
    ByteArrayOutputStream errContent = new ByteArrayOutputStream();
    System.setErr(new PrintStream(errContent));
    try {
        Pacman.main(new String[]{"ignored-argument"});
        String output = errContent.toString();
        assertFalse(output.isEmpty());
    } catch (IOException e) {
        assertTrue(false);
    } finally {
        System.setErr(originalErr);
    }
}
```

Listing 20: "Added testcases of Pacman"

5 additional tests, eliminating 3 mutations. Some extra tests where added for completion.

1.8.5 PacmanUI

```java
@Test
public void testAllKeypressesHandeled() {;
    var engine = pacman.getEngine();
    assertTrue(engine.inStartingState());

    pacmanUi.keyPressed(keyMap.get("VK_S"));
    assertTrue(engine.inPlayingState());
    var location1 = engine.getPlayer().getLocation();

    pacmanUi.keyPressed(keyMap.get("VK_LEFT"));
    var location2 = engine.getPlayer().getLocation();
    assertEquals(location1.getX() - 1, location2.getX());

    pacmanUi.keyPressed(keyMap.get("VK_UP"));
    location1 = engine.getPlayer().getLocation();
    assertEquals(location2.getY(), location1.getY() + 1);

    pacmanUi.keyPressed(keyMap.get("VK_RIGHT"));
    location2 = engine.getPlayer().getLocation();
    assertEquals(location1.getX() + 1, location2.getX());

    pacmanUi.keyPressed(keyMap.get("VK_DOWN"));
    location1 = engine.getPlayer().getLocation();
    assertEquals(location2.getY(), location1.getY() - 1);

    pacmanUi.keyPressed(keyMap.get("VK_E"));
    assertTrue(engine.inHaltedState());

    pacmanUi.keyPressed(keyMap.get("VK_S"));
    assertFalse(engine.inHaltedState());

    pacmanUi.keyPressed(keyMap.get("VK_Q"));
    assertTrue(engine.inHaltedState());
}

@Test
public void testDisplaySize() {
    pacmanUi.display();
    int expectedHeight = pacmanUi.getBoardViewer().windowHeight() +
     2 * 40;
    int actualHeight = pacmanUi.getHeight();
    assertEquals(expectedHeight, actualHeight);
}

@Test
public void testSerialVersionUnchanged() {
    assertEquals(-5947037932193718 3L, PacmanUI.serialVersionUID);
}
```

Listing 21: "Added testcases of PacmanUI"

2 additional tests, eliminating 10 mutations.

1.8.6 RandomMonsterMover

```
    /**
     * Actually conduct a random move in the underlying engine.
     *
     * @see jpacman.controller.IMonsterController#doTick()
     */
    public void doTick() {
        Monster theMonster = getRandomMonster();

        int dx = 0;
        int dy = 0;

        int dir = getRandomizer().nextInt(Direction.values().length
    );
        Direction d = Direction.values()[dir];
        switch(d) {
        case UP:
            dy = -1;
            break;
        case DOWN:
            dy = 1;
            break;
        case LEFT:
            dx = -1;
            break;
        case RIGHT:
            dx = 1;
            break;
        default:
            assert false;
        }

```

```
31          assert dy >= -1 && dy <= 1;
32          assert
33          Math.abs(dx) == 1 && dy == 0
34          ||
35          Math.abs(dy) == 1 && dx == 0;
36
37          getEngine().moveMonster(theMonster, dx, dy);
38      }
```

Listing 22: "Equivalent mutations 1-3 of RandomMonsterMover"

We can run this function with 4 different outcomes, (assuming 1 monster). This is influenced by a randomizer with a seed. In all possible cases, dx is changed (into 1 or -1) or dy is changed (into 1 or -1), the other will be 0. This will be asserted before returning by || the different possibilities. In this implementation, the && can be changed into || on lines 31, 33, and 35. These mutations will have the same result for the different possible combinations of dx and dy. giving 3 equivalent mutations. Combining line 31-35 into $assertMath.abs(dx) + assertMath.abs(dy) == 1$ is a cleaner approach for this problem without those mutations.

```
@Test
public void testGenerateAllRandomDirections() throws Exception {
    Field randomizerField = AbstractMonsterController.class.getDeclaredField("randomizer");
    randomizerField.setAccessible(true);
    Random random = (Random) randomizerField.get(null);
    random.setSeed(0L);
    /* Seed specially selected to have all 4 directions within the first 4 doTick() */
    pacman.start();
    var controller = new RandomMonsterMover(pacman.getEngine());
    controller.start();
    var monster = pacman.getEngine().getMonsters().get(0);
    var location = monster.getLocation();
    /* Right, Left, Up, Down */
    for (int i = 0; i < 4; i++) {
        controller.doTick();
    }
    assertEquals(location, monster.getLocation());
}
```

Listing 23: "Added testcase of RandomMonsterMover"

1 additional test, eliminating 10 mutations.

1.8.7 Recap

In total, we have added 13 test cases.

1.9 Exercise 9

What are the upsides and downsides of mutation testing? Explain your argument.

- *Lars*

I'll begin with the downsides because these are most obvious.

- **Computational Overhead:**
  Every mutant potentially triggers the entire test-suite to be run which scales badly for larger projects.

- **False Positives:**
  Depending on how mutation testing is handled the amount false positive surviving mutants can become non-trivial. Manually verifying whether or not a mutant is a false positive quickly becomes cumbersome and is not always that easy.

However mutation testing also has several upsides, if used properly it can discover weaknesses in the test-suite improving the overall test quality. Another upside is its automatic nature, once setup properly it can continue to provide value with little to no effort providing another metric for your test-suite which complements the common line coverage and branch coverage nicely.