Software Testing

Lab Assignments

March 17, 2025

Niels Van der Planken
s0191930@ad.ua.ac.be

Thomas Gueutal
s0195095@ad.ua.ac.be

Contents

1 Introduction

This report is a deliverable for one of the testing assignments for the course Software Testing (Master Computer Science 2001WETSWT) at the University of Antwerp in the academic year of 2024-2025.

As per the course description, "*The objective of the lab work of the Software Testing course is to help you learn how you can apply the various testing techniques and test design patterns as discussed during the lectures in practice. You will apply these techniques to a simple Pacman system written in Java. The amount of coding that needs to be done is relatively small: The focus is on testing.*".

This report was written in **Typst**.

2 Division of Tasks

The deliverables are as follows.

1. Code solutions to the tasks, located at **this** Github repo. Note that we add the source code at the end of an assignment as a separate release to the github repo.
2. This report, discussing the solutions and code.

We briefly discuss the division of tasks. We should strive for a half-half split of work between the two team members.

Member	Tasks
Niels	Exercises 1, 3, 4, 8 (including report)
Thomas	Exercises 2, 5, 6, 7 (including report)

Table 1: Division of tasks amongst team members.

3 Assignment 03 – Decision Structures

We repeat a note made at the start of the assignment file.

Important Note: Create an archive for JPacman system after performing all the exercises that require modifications to the files. Submit it along with your report. Refer to the "Assignments General" document for introductory information.

3.1 Exercise 1

Question: Create a decision table following the style of Table 7.6 (Forgács [1], or see Figure 10 in our report) indicating what should happen when a guest tries to occupy a new cell. Cases to be distinguished include whether or not the move remains within the borders, whether or not the move is possible based on the type of the moved object (player or monster), and the type of the (optional) guest occupying the other cell

See Table 2 for the decision table of what happens when a guest tries to occupy a new cell.

A player (or monster) can move into several different things: an empty space, a food token, a wall, another player or a monster. Depending on the situation, the move action is either allowed or blocked. The decision table gives an overview of these situations.

Important to note is that moving a player into the monster is a valid move – a valid use case – in the context of the specification. But, the code will say that actually performing the move is not possible; Codeblock 1 shows that a generic *Move* object that has marked the player as dead due to this particular move, `!playerDies()`, can not actually complete that movement. In other words, the game would end but the move would not be performed.

```java
/* Move.java */
public boolean movePossible() {
    assert initialized() : "run precompute first!";
    return withinBorder()
        && targetCellAvailable
        && !playerDies()
        && !moveDone();
}
```

Codeblock 1: The source code for `Move.movePossible()`.

Case	Within Borders?	Guest Type	Target Cell Occupied by	Move Allowed?	Action Taken
1	Yes	Player	Empty	Yes	Move successful
2	Yes	Player	Wall	No	Move blocked
3	Yes	Player	Food	Yes	Move & Consume Food
4	Yes	Player	Monster	Yes	Player Dies
5	Yes	Player	Another player	No	Move blocked
6	Yes	Monster	Empty	Yes	Move successful
7	Yes	Monster	Wall	No	Move blocked
8	Yes	Monster	Food	No	Move Blocked
9	Yes	Monster	Player	Yes	Attack player, player dies
10	Yes	Monster	Another Monster	No	Move blocked
11	No	Any	Any	No	Move blocked (out of bounds)

Table 2: Decision table for guest movement

3.2 Exercise 2

Question: Run the current test suite and describe the coverage of `Move, PlayerMove, Guest` and all `Guest` subclasses.

We observe the relevant class hierarchy subtree for *Move*, *PlayerMove* and *Guest* generated by `mvn site`, see Figure 1.

```
├── Move
|   └── PlayerMove
└── Guest
    ├── Food
    ├── MovingGuest
    |     ├── Monster
    |     └── Player
    └── Wall
```

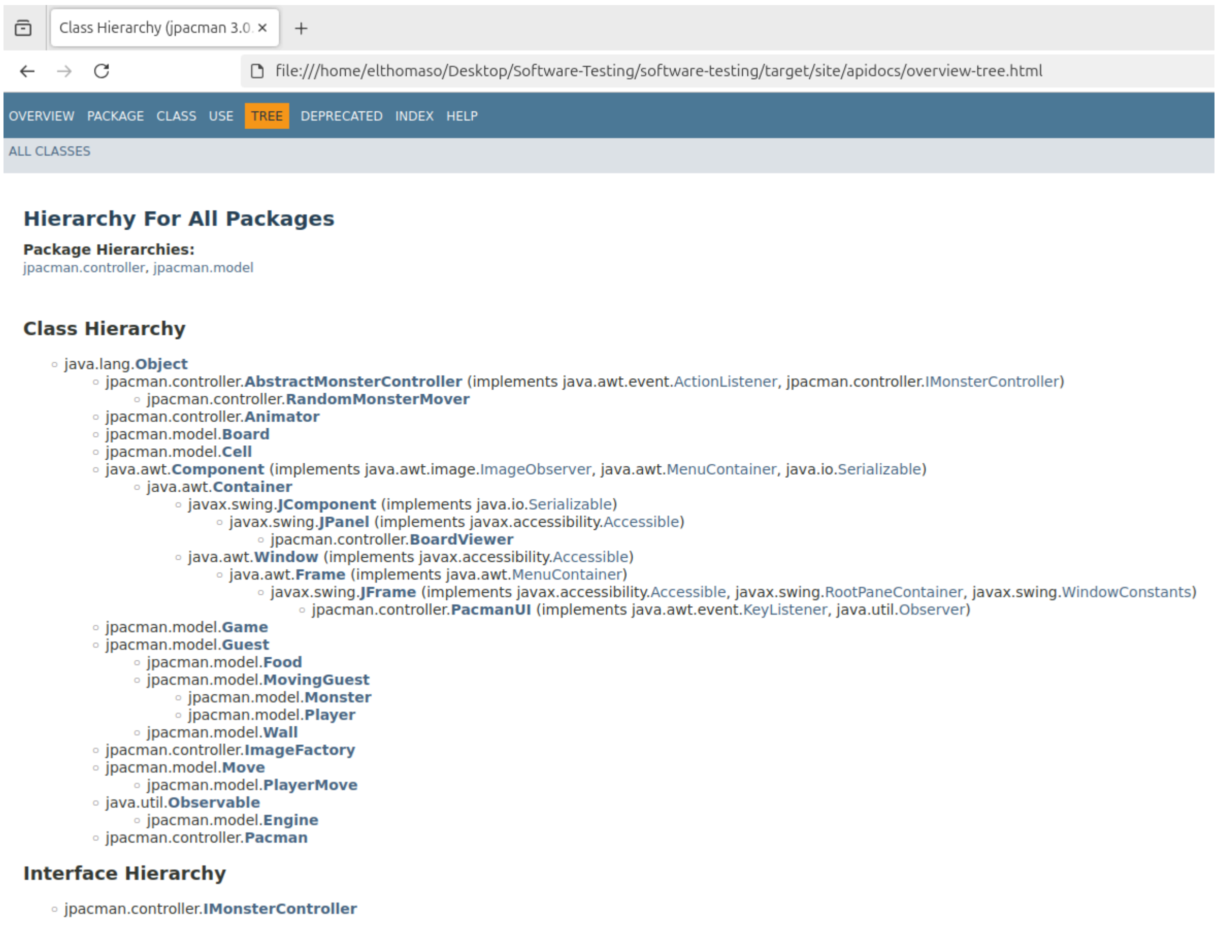
We summarize the major coverage percentages in Table 3. See Figure 2 for the `jpacman.model` subsection of the coverage report.

- Most subclasses of *Guest* have brief implementations, as seen by their low method count and line count amounts: *Food*, *MovingGuest*, *Wall* and *Monster*.
- The other classes of interest have more substantial implementations, as seen by their higher method count and line count amounts: *Move*, *PlayerMove*, *Guest* and *Player*.

Most classes have **most**, if not all, of their **lines covered**. The test suite mainly lacks in proper branch coverage. Since line coverage is high, this means the missed instructions are mostly contained in the missed branches. So, **increasing branch coverage will proportionally increase instruction coverage in this case**. In other words, except for in *Monster* and *Player*, there are no branches that if covered will lead to a (large) increase in number of lines covered, which would consequently increase instruction coverage (greatly), so the low hanging fruit for increasing instruction coverage has already been picked. Only the painstaking coverage of all branches can gradually increase instruction coverage.

Class	Instruction Coverage	Branch Coverage	Missed Methods / #Methods	Missed Lines / #Lines
Move	70%	56%	1 / 13	4 / 60
PlayerMove	70%	50%	0 / 8	0 / 27
Guest	66%	53%	0 / 6	0 / 25
Food	70%	50%	0 / 7	0 / 16
MovingGuest	100%	n/a	0 / 1	0 / 2
Monster	18%	0%	1 / 4	5 / 9
Player	43%	23%	2 / 12	10 / 30
Wall	61%	50%	0 / 4	0 / 8

Table 3: The coverage results **before** extending the *PlayerMoveTest* suite.


Class Hierarchy (jpacman 3.0
file:///home/elthomaso/Desktop/Software-Testing/software-testing/target/site/apidocs/overview-tree.html
OVERVIEW PACKAGE CLASS USE TREE DEPRECATED INDEX HELP
ALL CLASSES

Hierarchy For All Packages

Package Hierarchies:
jpacman.controller, jpacman.model

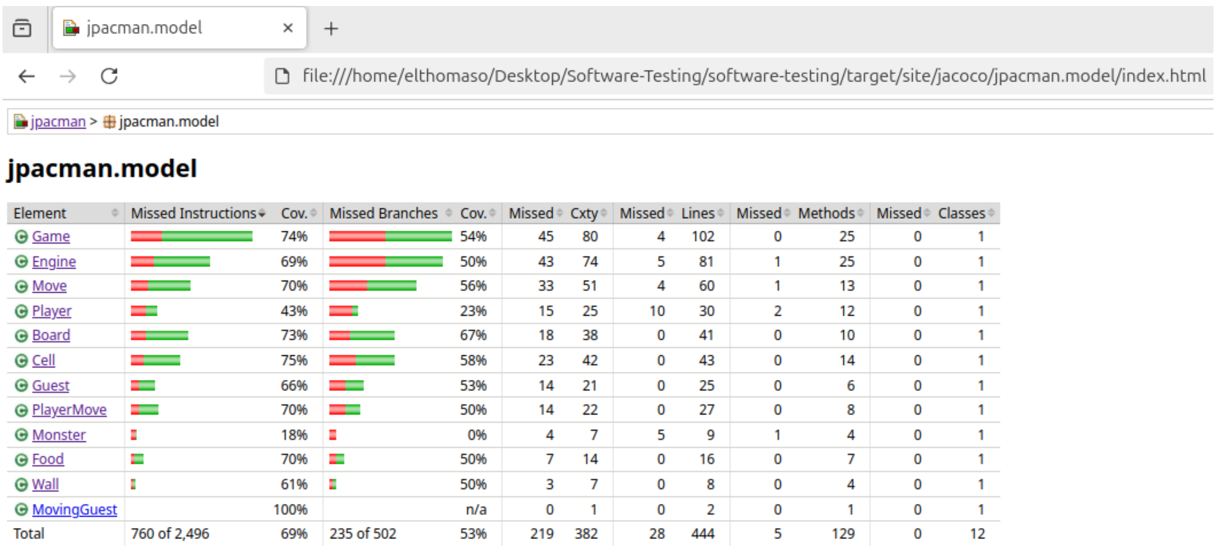
Class Hierarchy

- java.lang.**Object**
  - jpacman.controller.**AbstractMonsterController** (implements java.awt.event.ActionListener, jpacman.controller.IMonsterController)
    - jpacman.controller.**RandomMonsterMover**
  - jpacman.controller.**Animator**
  - jpacman.model.**Board**
  - jpacman.model.**Cell**
  - java.awt.**Component** (implements java.awt.image.ImageObserver, java.awt.MenuContainer, java.io.Serializable)
    - java.awt.**Container**
      - javax.swing.**JComponent** (implements java.io.Serializable)
        - javax.swing.**JPanel** (implements javax.accessibility.Accessible)
          - jpacman.controller.**BoardViewer**
      - java.awt.**Window** (implements javax.accessibility.Accessible)
        - java.awt.**Frame** (implements java.awt.MenuContainer)
          - javax.swing.**JFrame** (implements javax.accessibility.Accessible, javax.swing.RootPaneContainer, javax.swing.WindowConstants)
            - jpacman.controller.**PacmanUI** (implements java.awt.event.KeyListener, java.util.Observer)
  - jpacman.model.**Game**
  - jpacman.model.**Guest**
    - jpacman.model.**Food**
    - jpacman.model.**MovingGuest**
      - jpacman.model.**Monster**
      - jpacman.model.**Player**
    - jpacman.model.**Wall**
  - jpacman.controller.**ImageFactory**
  - jpacman.model.**Move**
    - jpacman.model.**PlayerMove**
  - java.util.**Observable**
    - jpacman.model.**Engine**
  - jpacman.controller.**Pacman**

Interface Hierarchy

- jpacman.controller.**IMonsterController**

Figure 1: The `mvn site` class hierarchy.



jpacman.model

file:///home/elthomaso/Desktop/Software-Testing/software-testing/target/site/jacoco/jpacman.model/index.html

jpacman > jpacman.model

jpacman.model

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
Game		74%		54%	45	80	4	102	0	25	0	1
Engine		69%		50%	43	74	5	81	1	25	0	1
Move		70%		56%	33	51	4	60	1	13	0	1
Player		43%		23%	15	25	10	30	2	12	0	1
Board		73%		67%	18	38	0	41	0	10	0	1
Cell		75%		58%	23	42	0	43	0	14	0	1
Guest		66%		53%	14	21	0	25	0	6	0	1
PlayerMove		70%		50%	14	22	0	27	0	8	0	1
Monster		18%		0%	4	7	5	9	1	4	0	1
Food		70%		50%	7	14	0	16	0	7	0	1
Wall		61%		50%	3	7	0	8	0	4	0	1
MovingGuest		100%		n/a	0	1	0	2	0	1	0	1
Total	760 of 2,496	69%	235 of 502	53%	219	382	28	444	5	129	0	12

Figure 2: The `mvn site` coverage for *Move*, *PlayerMove*, *Guest* and all *Guest* subclasses.

3.3 Exercise 3

Question 3: Implement all entries in the decision table concerning player movements as JUnit test cases in `PlayerMoveTest` class. Since the player movement has been implemented already, start by testing these.

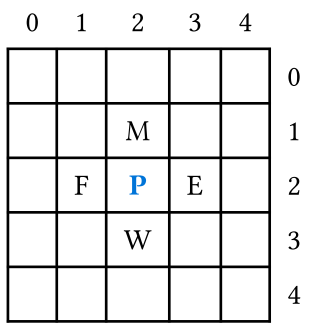
See Codeblock 10 for the source code of the implemented *PlayerMoveTest* test cases.

The implemented cases are case **1 to 5** and case **11**.

We created a set up environment for the tests where we create a `5x5` board and select some dedicated cells to be operated on. All the cells are adjacent to one another:

Consider the (x, y) position of each Guest type on the Board grid.

- `(2, 1)` : monster cell **(M)**
- `(2, 2)` : player cell **(P)**
- `(2, 3)` : wall cell **(W)**
- `(3, 2)` : empty cell **(E)**
- `(1, 2)` : food cell **(F)**
- `null` : out-of-bounds cell



With every test we use the `movePossible()` method to see if the outcome is equal to the expectations. In the test where the player moves into the food object, we also check that the food gets eaten properly and that the player receives a point for eating the food. All the tests can be found in the file `src/test/java/jpacman/model/PlayerMoveTest.java`.

For case 11, note that the *Move* class constructor, see Codeblock 3, considers a its target *Cell* location *toCell* to be out-of-bounds of the *Board* iff. `toCell == null`. This is supported by the fact that the constructor explicitly asserts that *toCell* is allowed to be `null`, and that the *Move*'s target *Cell* member *to* is considered to be within the *Board* borders iff. `to != null`, see `Move.withinBorder()` in Codeblock 3. This test case checks if a player can move out of bounds (a cell outside the board). To test this, we simply pass `null` to the *Cell* constructor and test `Move.movePossible()` as normal.

For case 5, our initial approach was to create another player and have it occupy the empty cell, then test if the move was prohibited or not. The code for the initial approach can be found in Codeblock 2. This approach however led to an error message, seen in Figure 3, telling us that 2 players are not allowed in the `Move` class. Therefore we adapted the test such that the player tries to move to itself. This move should not be possible and thus will be blocked.

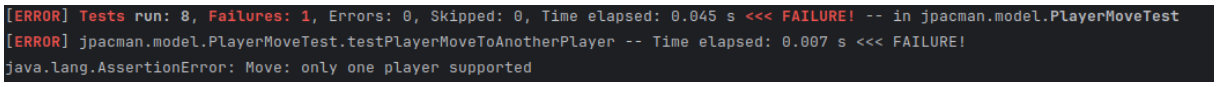
```java
@Test
public void testPlayerMoveToAnotherPlayer(){
    // for this test we put the player and another player next to each
    // other
    // therefor we let anotherPlayer occupy the empty cell
    anotherPlayer.occupy(emptyCell);
    aPlayerMove = new PlayerMove(player, emptyCell);
    assertFalse(aPlayerMove.movePossible());
    anotherPlayer.deoccupy();
}
```

Codeblock 2: The original *anotherPlayer* test

```java
/* Move.java */
public Move(MovingGuest fromGuest, Cell toCell) {
    assert fromGuest != null;
    assert fromGuest.getLocation() != null;
    assert toCell == null
    || fromGuest.getLocation().getBoard() == toCell.getBoard();
    this.mover = fromGuest;
    this.to = toCell;
    assert moveInvariant() : "Move invariant invalid";
}

/* Move.java */
private boolean withinBorder() {
    return to != null;
}
```

Codeblock 3: The *Move* class constructor and `Move.withinBorder()` source code.



```
[ERROR] Tests run: 8, Failures: 1, Errors: 0, Skipped: 0, Time elapsed: 0.045 s <<< FAILURE! -- in jpacman.model.PlayerMoveTest
[ERROR] jpacman.model.PlayerMoveTest.testPlayerMoveToAnotherPlayer -- Time elapsed: 0.007 s <<< FAILURE!
java.lang.AssertionError: Move: only one player supported
```

Figure 3: Error message *anotherPlayer* move test

3.4 Exercise 4

Question 4: Re-run with coverage enabled, and re-asses the coverage.

See **Figure 4** for the updated coverage report. **Table 4** Summarizes the important, relevant coverage numbers. This can be compared with **Table 3**. When we look at the coverage now, we see some positive changes. The coverage of some classes increased:

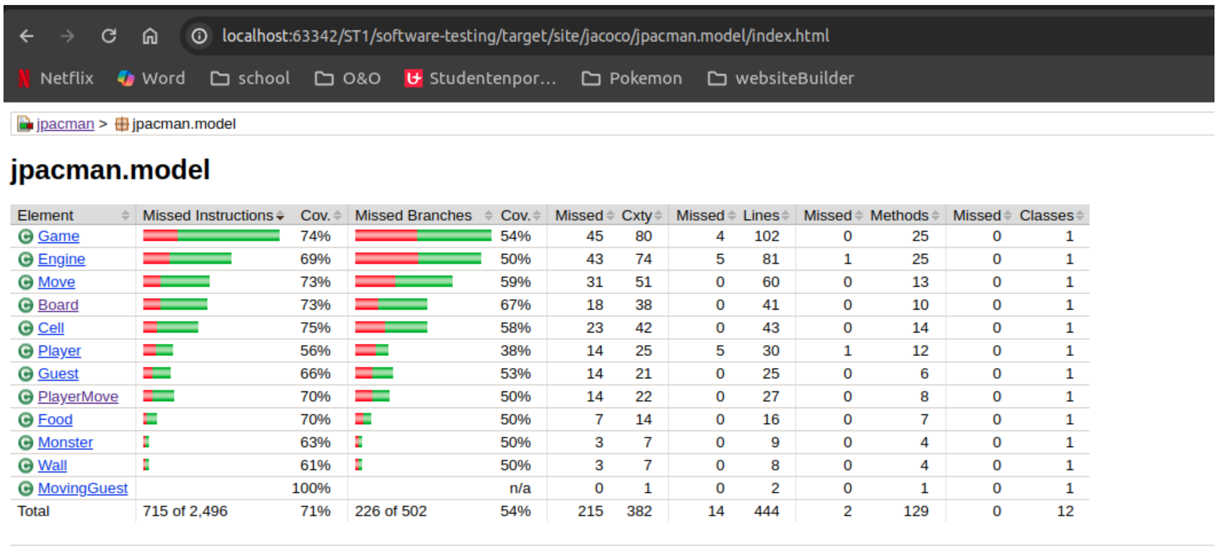
- `Move`
- `Player`
- `Food`
- `Monster`

The coverage increase is because we use those *Guest* subclasses as helpers in the `PlayerMoveTest` tests. We call some specific methods, such as `movePossible()`, which were not used before, leading to an increase in the coverage. Important to note is that the `PlayerMove` coverage did not improve. This is because our decision table covers specific execution paths and specific scenarios.

The goal of the decision table is to ensure all possible execution paths are covered. This approach greatly favors line coverage over instruction or branch coverage. As such, these paths and scenarios mainly result in the calling of methods that were already tested by other unit tests. Since the JaCoCo coverage report favors instruction and branch coverage, we see no improvement to the displayed coverage rates as a result of adding the `PlayerMoveTest` tests.

Class	Instruction Coverage	Branch Coverage	Missed Methods / #Methods	Missed Lines / #Lines
Move	73%	59%	0 / 13	0 / 60
PlayerMove	70%	50%	0 / 8	0 / 27
Guest	66%	53%	0 / 6	0 / 25
Food	70%	50%	0 / 7	0 / 16
MovingGuest	100%	n/a	0 / 1	0 / 2
Monster	63%	50%	0 / 4	0 / 9
Player	56%	38%	1 / 12	5 / 30
Wall	61%	50%	0 / 4	0 / 8

Table 4: The coverage results **after** extending the *PlayerMoveTest* suite.



localhost:63342/ST1/software-testing/target/site/jacoco/jpacman.model/index.html
Netflix
Word
school
O&O
Studentenpor...
Pokemon
websiteBuilder

jpacman > jpacman.model

jpacman.model

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
Game		74%		54%	45	80	4	102	0	25	0	1
Engine		69%		50%	43	74	5	81	1	25	0	1
Move		73%		59%	31	51	0	60	0	13	0	1
Board		73%		67%	18	38	0	41	0	10	0	1
Cell		75%		58%	23	42	0	43	0	14	0	1
Player		56%		38%	14	25	5	30	1	12	0	1
Guest		66%		53%	14	21	0	25	0	6	0	1
PlayerMove		70%		50%	14	22	0	27	0	8	0	1
Food		70%		50%	7	14	0	16	0	7	0	1
Monster		63%		50%	3	7	0	9	0	4	0	1
Wall		61%		50%	3	7	0	8	0	4	0	1
MovingGuest		100%		n/a	0	1	0	2	0	1	0	1
Total	715 of 2,496	71%	226 of 502	54%	215	382	14	444	2	129	0	12

Figure 4: The `mvn site` coverage for *Move*, *PlayerMove*, *Guest* and all *Guest* subclasses after implementing PlayerMoveTest.

3.5 Exercise 5

Question 5: Explain the interplay between the abstract methods `Guest.meetPlayer` and `Move.tryMoveToGuest` and their implementations in `Guest` and `Move` subclasses.

First, we show simplified source code for the `Move.tryMoveToGuest(Guest)` and `Guest.meetPlayer(PlayerMove)` methods, see **Codeblock 5**. We repeat the class hierarchy for the classes of interest.

```
├── Move
|   └── PlayerMove
└── Guest
    ├── Food
    ├── MovingGuest
```

```
|       ├── Monster
|       └── Player
└── Wall
```

The abstract method `Move.tryMoveToGuest(Guest)` states that it "*implements a double dispatch: the actual behavior depends on both the subclass of the source (Move) and the target guest (Guest).*" So, by definition, an implementation of `tryMoveToGuest` calls an implementation of the abstract method `Guest.meetPlayer(PlayerMove)`. The `meetPlayer` implementation then determines if a player could move onto the cell occupied by this concrete guest object, and propagates that decision upwards to the caller, `tryMoveToGuest`, to let the concrete *PlayerMove* interpret and act on the movement decision. Finally, the call to `Move.tryMoveToGuest(Guest)` returns whether the movement was successful or not.

So, concrete `Move.tryMoveToGuest(Guest)` implementations define the movement validation logic, while the concrete `Guest.meetPlayer(PlayerMove)` implementation defines how an (un)successful move should be handled.

To make this more concrete, we include some pseudocode that clarifies the "call stack" that shows when `Move.tryMoveToGuest(Guest)` would normally be called during the execution of the pacman game, see **Codeblock 4**.

```
// Construct objects until the call stack of interest is triggered.
Cell dst = ...; // The movement destination.
Player ply = new Player(); // The Player that wants to move.
PlayerMove move = new PlayerMove(ply, dst);
--> move.precomputeEffects() // PlayerMove constructor induces the call stack.
    --> move.tryMoveToGuest(guest);
        --> guest.meetPlayer(move); // Perform double dispatch.
            --> ... // Implementation details of `Guest.meetPlayer(PlayerMove)`
            <-- // Make the movement decision (true / false).
        <-- // Propagate the movement decision upwards.
    <-- // The PlayerMove acted on the movement decision.
<-- // End of "call stack"
```

Codeblock 4: Pseudocode for a "call stack" when calling `Move.tryMoveToGuest(Guest)`.

```java
/* Move.java */
protected abstract boolean
tryMoveToGuest(Guest targetGuest);

/* PlayerMove.java */
@Override
protected boolean
tryMoveToGuest(Guest targetGuest) {
    ...
    return targetGuest.meetPlayer(this);
}

/* Guest.java */
protected abstract boolean
meetPlayer(PlayerMove aMove);

/* Food.java, Monster.java, Player.java, Wall.java */
@Override
public boolean
meetPlayer(PlayerMove aMove) {
    ...
}
```

Codeblock 5: Simplified source code for `Move.tryMoveToGuest(Guest)` and `Guest.meetPlayer(PlayerMove)` .

3.6 Exercise 6

Question 6: Implement a monster move in the same style as a player move. Add a `MonsterMove` class, place it correctly in the inheritance hierarchy, and implement the required methods. Make sure you add or update appropriate invariants as well as pre- and post-conditions wherever possible, and implement them using assertions.

3.6.1 MonsterMove Specification

We consult the `doc/pacman-design.txt` and `doc/pacman-requirements.txt` documentation for the requirements of a `MonsterMove` . See Figure 5 where we reference (inline) the relevant design decisions and use cases.

1. "***Moves to food** are not possible.*"
2. "*If a monster attempts to move to the **cell occupied by the player**, the player dies and the game enters the player died state.*"
3. "*The **engine** remains in the Playing state if one of the monsters makes a non killing move.*"
4. Use cases UC5 and UC6 describe entry and exit conditions, plus behavior for …
    - a generic monster move.
    - a monster meeting (moving onto) a player specifically

3.6.2 Implementation Details – MonsterMove

This subsection discusses the ***MonsterMove* class implementation details**. The `Guest.meetMonster(MonsterMove)` implementations will be discussed in the next section.

We start by copying the `PlayerMove.java` file and renaming all mentions of "player" to "monster", which results in an incomplete `MonsterMove` class.

Next, we remove all "food" related aspects (member variables and methods, etc.) because a monster cannot move to any *Cell* occupied by a *Food* guest and does not concern itself with food in general;

"*Moves to food are not possible.*" This gives us the following (see **Codeblock 6**) barebones *MonsterMove* class API.

```java
public class MonsterMove extends Move {
    /* Class member variables. */
    private Monster theMonster;

    /* Class methods. */
    public MonsterMove(Monster monster, Cell newCell) { ... }
    public boolean invariant() { ... }
    public Monster getMonster() { ... }
    @Override public void apply() { ... }

    @Override protected boolean tryMoveToGuest(Guest targetGuest) {
        ...
        /* Call "Guest.meetMonster(MonsterMove)"! */
        return targetGuest.meetMonster(this);
    }
}
```

Codeblock 6: The barebones *MonsterMove* class API.

Now we discuss the implementation details of the methods listed in **Codeblock 6**.

First, some of the methods added to *MonsterMove* are identical to their counterparts in *PlayerMove*, and we only have to do some variable renaming to fit the "monster" theme.

- The constructor `MonsterMove.MonsterMove(Monster, Cell)` remains very similar, since it delegates its implementation details to `Move.precomputeEffects()` and `MonsterMove.invariant()`.
- The method `MonsterMove.tryMoveToGuest(Guest)` remains very similar, since it is part of the double dispatch pattern, and it delegates its implementation details to `Move.tryMoveToGuestPrecondition(Guest)` and `Guest.meetMonster(MonsterMove)`.
- `MonsterMove.getMonster()` is a simple getter.

Second, the method `MonsterMove.invariant()` only differs from `PlayerMove.invariant()` in that it removes the food related sub-condition, see **Codeblock 7**. This ties into the removing of all food related code from *MonsterMove* mentioned previously.

```java
/* MonsterMove.java */
public boolean invariant() {
    return moveInvariant() &&
           theMonster != null &&
           getMovingGuest().equals(theMonster);
}

/* PlayerMove.java */
public boolean invariant() {
    return moveInvariant() &&
           foodEaten >= 0 &&    // Only the player cares about food.
           thePlayer != null &&
           getMovingGuest().equals(thePlayer);
}
```

Codeblock 7: A comparison of the *MonsterMove* and *PlayerMove* invariant methods.

Lastly, `MonsterMove.apply()` simply calls `Move.apply()` in addition to the relevant pre- and post-conditions asserts that were also part of `PlayerMove.apply()`.

```java
/* MonsterMove.java */
@Override
public void apply() {
    assert invariant();
    assert movePossible();
    super.apply();
    assert invariant();
}
```

3.6.3 Implementation Details – meetMonster

The meaning of the `Guest.meetMonster(MonsterMove)` method is that a *Monster* wants to move into a *Cell* that is currently occupied by the callee *Guest* object that implements the called `meetMonster` method.

Similar to the simplified *PlayerMove* source code structure in **Section 3.5**, **Codeblock 5** we express a similar simplified code structure in **Codeblock 8**. The changes compared to the *PlayerMove* code block are as follows.

- **We add an abstract `Guest.meetMonster(MonsterMove)` method to the *Guest* class.**
- We add concrete `meetMonster(MonsterMove)` implementations to the all concrete subclasses of *Guest*: *Food*, *Monster*, *Player* and *Wall*.

See **Table 5** for the detailed (concrete) implementations of the `Guest.meetMonster(MonsterMove)` methods in all concrete *Guest* subclasses. They should match the specification described in the referenced documentation in **Figure 5**.

Note that the concrete `meetMonster` implementations all return `false`; a *Cell* object supports at most one *Guest* object at a time, so a concrete *Guest* can never actually move into an occupied *Cell*. Rather, **the state of the *MonsterMove* should express the required side effects of the move**. Furthermore, only the `Player.meetMonster(MonsterMove)` implementation performs any actual operations on the passed *MonsterMove* object: the player must die due to the touch between *Monster* and *Player*. All other implementations only block the movement and imply nothing else.

Each concrete *Guest* subclass does call the most appropriate invariant function in its pre- and post-condition asserts.

The **important requirement** that the player is alive as a pre-condition for a generic monster move, should be fulfilled by the yet-to-be-implemented *Game* method that constructs the *MonsterMove* object. This is because in general, the `MonsterMove.tryMoveToGuest(Guest)` and `Guest.meetMonster(MonsterMove)` methods do ***not*** have access to the player object(s) in the game, so they would not be able to validate this requirement. As such, this use case requirement must be validated higher in the call stack.

```java
/* Move.java */
protected abstract boolean
tryMoveToGuest(Guest targetGuest);

/* MonsterMove.java */
@Override
protected boolean
tryMoveToGuest(Guest targetGuest) {
    ...
    return targetGuest.meetPlayer(this);
}

/* Guest.java */
protected abstract boolean
meetMonster(MonsterMove aMove);

/* Food.java, Monster.java, Player.java, Wall.java */
@Override
public boolean
meetMonster(MonsterMove aMove) {
    ...
}
```

Codeblock 8: Simplified source code for `Move.tryMoveToGuest(Guest)` and `Guest.meetMonster(MonsterMove)`.

```java
/* Food.java
 * Monster moves onto Food.
 */
@Override
protected boolean
meetMonster(MonsterMove theMove) {
    // Food invariant required.
    assert foodInvariant();
    assert theMove != null;
    assert !theMove.initialized();
    return false;
}
```

```java
/* Wall.java
 * Monster moves onto Wall.
 */
@Override
public boolean
meetMonster(MonsterMove aMove) {
    // Guest invariant suffices.
    assert guestInvariant();
    assert aMove != null;
    assert !aMove.initialized();
    return false;
}
```

```java
/* Player.java
 * Monster moves onto Player.
 */
@Override
protected boolean
meetMonster(MonsterMove theMove) {
    // Player invariant required.
    assert playerInvariant();
    assert theMove != null;
    assert !theMove.initialized();
    theMove.die();
    return false;
}
```

```java
/* Monster.java
 * Monster moves onto Monster.
 */
@Override
protected boolean
meetMonster(MonsterMove theMove) {
    // Guest invariant suffices.
    assert guestInvariant();
    assert theMove != null;
    assert !theMove.initialized();
    return false;
}
```

Table 5: The detailed implementations of the concrete `Guest.meetMonster(MonsterMove)` methods.

3.6.4 Referenced Documentation

D3.3.3. Killing move: If a move (by a player) is possible and causes the player to die (hitting upon a monster), the machine makes a transition to the Player Died state.

D3.4. Monster move: A move by a Monster is similar to a player move, except that moves to food are not possible. If a monster attempts to move to the cell occupied by the player, the player dies and the game enters the player died state. The engine remains in the Playing state if one of the monsters makes a non killing move. Monster moves are triggered by a monster controller generating random moves at a specified interval, which runs in a separate thread.

UC5: Simple monster move use case.

Actor: monster

1. Entry condition: The player is alive and playing
2. At regular intervals (every 50 milliseconds) one of the monsters makes a random move (up, down, left, right) If the target cell is empty and not beyond the border, the move is possible. If the target cell is a wall or food element the move will not be possible.

UC6: Monster meets player use case.

Actor: monster

1. Entry condition: The player is alive and playing
2. One of the monsters is to be moved to the cell occupied by the player.
3. The player dies and this game is over.

Figure 5: Monster move design descriptions and use cases.

3.7 Exercise 7

Question 7: Introduce a `MonsterMoveTest` class to implement the test cases related to monster moves. You will probably want to extend `MoveTest` for this. Verify the test coverage for this class.

See Codeblock 9 for the source code of the implemented *MonsterMoveTest* test cases.

We implement the test class in `src/test/java/jpacman/model/MonsterMoveTest.java`. As a start, we can copy the `PlayerMoveTest.java` test code and adapt it to the *MonsterMove* class.

The implemented cases are cases **6 to 11** listed in Table 2; we implement all remaining test cases in the decision table defined at the start.

We created a set up environment for the tests where we create a `5x5` board and select some dedicated cells to be operated on. All the cells are adjacent to one another:

Consider the (x, y) position of each Guest type on the Board grid.

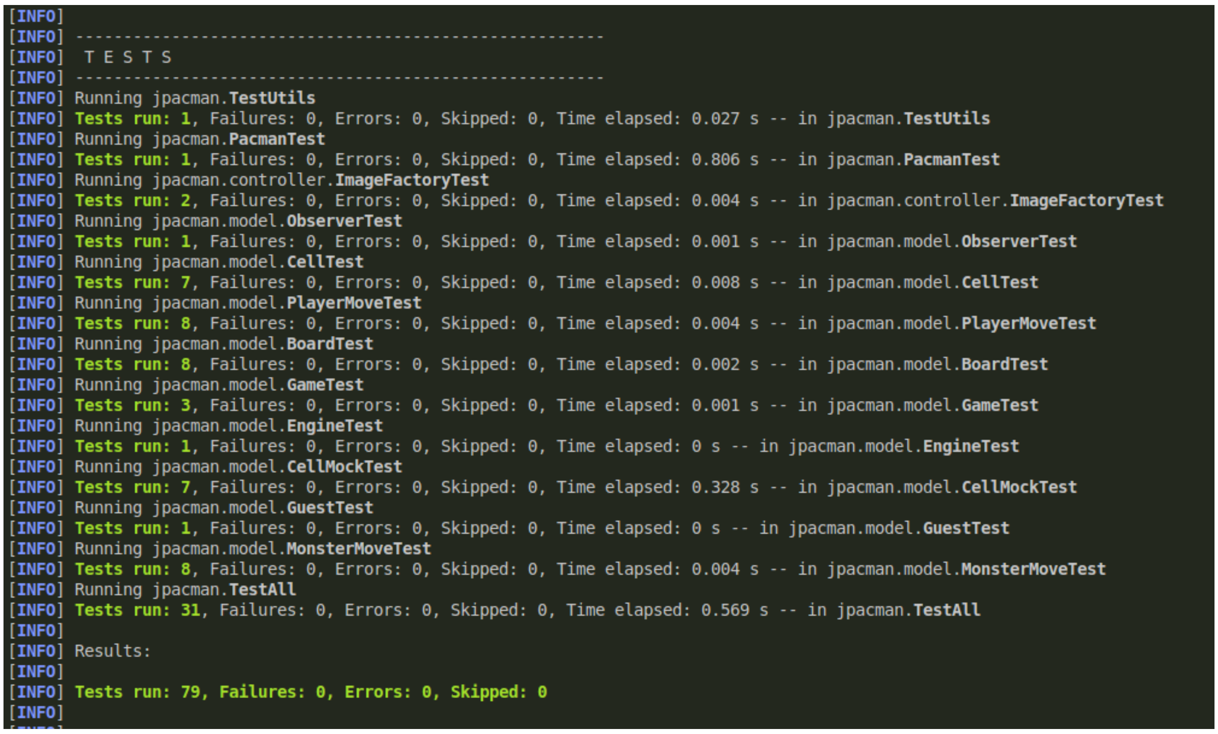
- `(2, 1)` : player cell **(P)**
- `(2, 2)` : monster cell **(M)**
- `(2, 3)` : wall cell **(W)**
- `(3, 2)` : empty cell **(E)**
- `(1, 2)` : food cell **(F)**
- `null` : out-of-bounds cell

0	1	2	3	4	
					0
		P			1
	F	M	E		2
		W			3
					4

With every test we use the `movePossible()` method to see if the outcome is equal to the expectations. In the test where the monster moves into the food object, the move should not be possible.

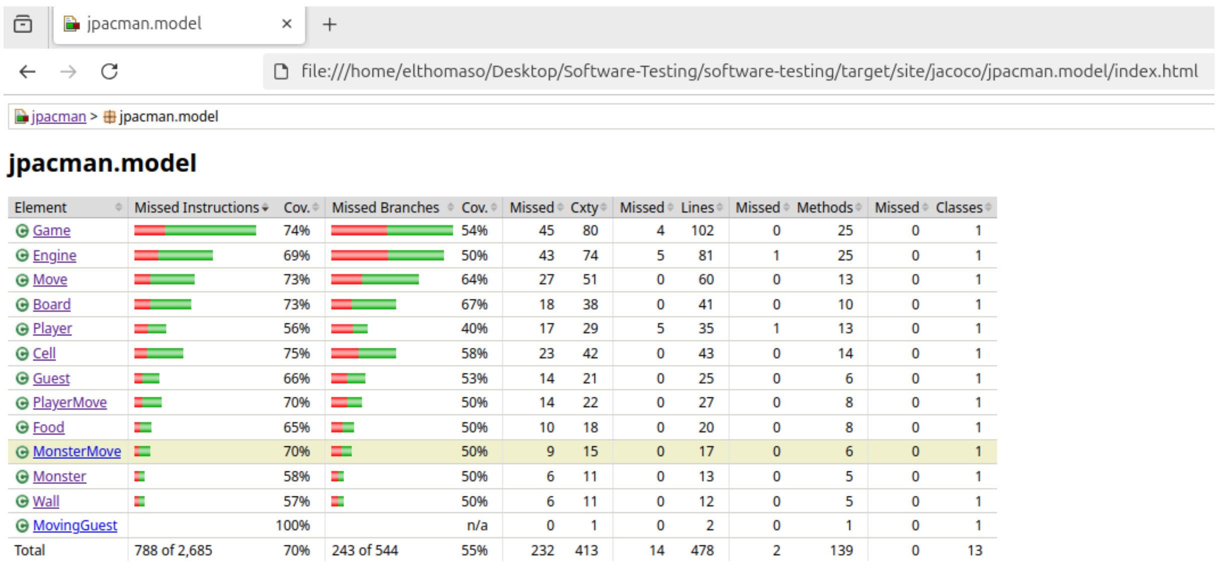
We verify the results of adding *MonsterMoveTest* to the testing suite. Figure 6 show the results of running `mvn test`, which includes 8 test cases in *MonsterMoveTest*. Figure 7 Show the coverage results. The ***PlayerMove* and *MonsterMove* classes both have the same high-level coverage ratios**: an instruction coverage of 70% and a branch coverage of 50%. The *PlayerMove* and *MonsterMove* classes both inherit from the abstract *Move* class, and their implementations are very similar. Since the *PlayerMoveTest* and *MonsterMoveTest* test classes also implement very similar test cases, based on the decision table Table 2, so their coverage ratios are very similar as well.

The coverage report shows that all lines and methods are covered. The problem is again branch coverage; many instructions are not covered because they live inside of uncovered branches that are part of a logical expression. This is supported by the low branch coverage vs instruction coverage in Figure 8.



```
[INFO]
[INFO] -------------------------------------------------------
[INFO]  T E S T S
[INFO] -------------------------------------------------------
[INFO] Running jpacman.TestUtils
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.027 s -- in jpacman.TestUtils
[INFO] Running jpacman.PacmanTest
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.806 s -- in jpacman.PacmanTest
[INFO] Running jpacman.controller.ImageFactoryTest
[INFO] Tests run: 2, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.004 s -- in jpacman.controller.ImageFactoryTest
[INFO] Running jpacman.model.ObserverTest
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.001 s -- in jpacman.model.ObserverTest
[INFO] Running jpacman.model.CellTest
[INFO] Tests run: 7, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.008 s -- in jpacman.model.CellTest
[INFO] Running jpacman.model.PlayerMoveTest
[INFO] Tests run: 8, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.004 s -- in jpacman.model.PlayerMoveTest
[INFO] Running jpacman.model.BoardTest
[INFO] Tests run: 8, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.002 s -- in jpacman.model.BoardTest
[INFO] Running jpacman.model.GameTest
[INFO] Tests run: 3, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.001 s -- in jpacman.model.GameTest
[INFO] Running jpacman.model.EngineTest
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0 s -- in jpacman.model.EngineTest
[INFO] Running jpacman.model.CellMockTest
[INFO] Tests run: 7, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.328 s -- in jpacman.model.CellMockTest
[INFO] Running jpacman.model.GuestTest
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0 s -- in jpacman.model.GuestTest
[INFO] Running jpacman.model.MonsterMoveTest
[INFO] Tests run: 8, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.004 s -- in jpacman.model.MonsterMoveTest
[INFO] Running jpacman.TestAll
[INFO] Tests run: 31, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.569 s -- in jpacman.TestAll
[INFO]
[INFO] Results:
[INFO]
[INFO] Tests run: 79, Failures: 0, Errors: 0, Skipped: 0
[INFO]
```

Figure 6: The testing results of running the test suite, including *MonsterMoveTest*

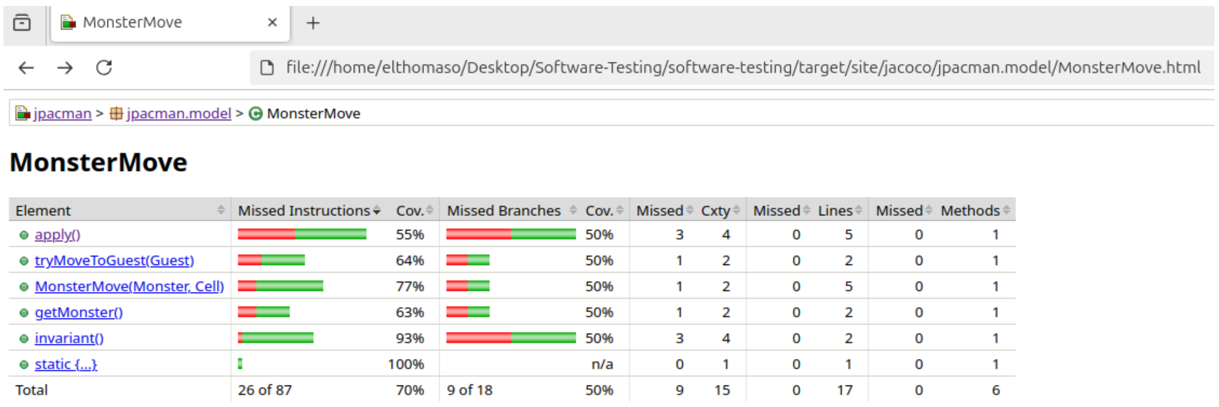


jpacman.model
file:///home/elthomaso/Desktop/Software-Testing/software-testing/target/site/jacoco/jpacman.model/index.html
jpacman > jpacman.model

jpacman.model

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
Game		74%		54%	45	80	4	102	0	25	0	1
Engine		69%		50%	43	74	5	81	1	25	0	1
Move		73%		64%	27	51	0	60	0	13	0	1
Board		73%		67%	18	38	0	41	0	10	0	1
Player		56%		40%	17	29	5	35	1	13	0	1
Cell		75%		58%	23	42	0	43	0	14	0	1
Guest		66%		53%	14	21	0	25	0	6	0	1
PlayerMove		70%		50%	14	22	0	27	0	8	0	1
Food		65%		50%	10	18	0	20	0	8	0	1
MonsterMove		70%		50%	9	15	0	17	0	6	0	1
Monster		58%		50%	6	11	0	13	0	5	0	1
Wall		57%		50%	6	11	0	12	0	5	0	1
MovingGuest		100%		n/a	0	1	0	2	0	1	0	1
Total	788 of 2,685	70%	243 of 544	55%	232	413	14	478	2	139	0	13

Figure 7: The code coverage results of running the test suite, including *MonsterMoveTest*


MonsterMove

file:///home/elthomaso/Desktop/Software-Testing/software-testing/target/site/jacoco/jpacman.model/MonsterMove.html

jpacman > jpacman.model > MonsterMove

MonsterMove

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods
apply()		55%		50%	3	4	0	5	0	1
tryMoveToGuest(Guest)		64%		50%	1	2	0	2	0	1
MonsterMove(Monster, Cell)		77%		50%	1	2	0	5	0	1
getMonster()		63%		50%	1	2	0	2	0	1
invariant()		93%		50%	3	4	0	2	0	1
static {...}		100%		n/a	0	1	0	1	0	1
Total	26 of 87	70%	9 of 18	50%	9	15	0	17	0	6

Figure 8: The code coverage results of running the test suite, specifically for *MonsterMove*

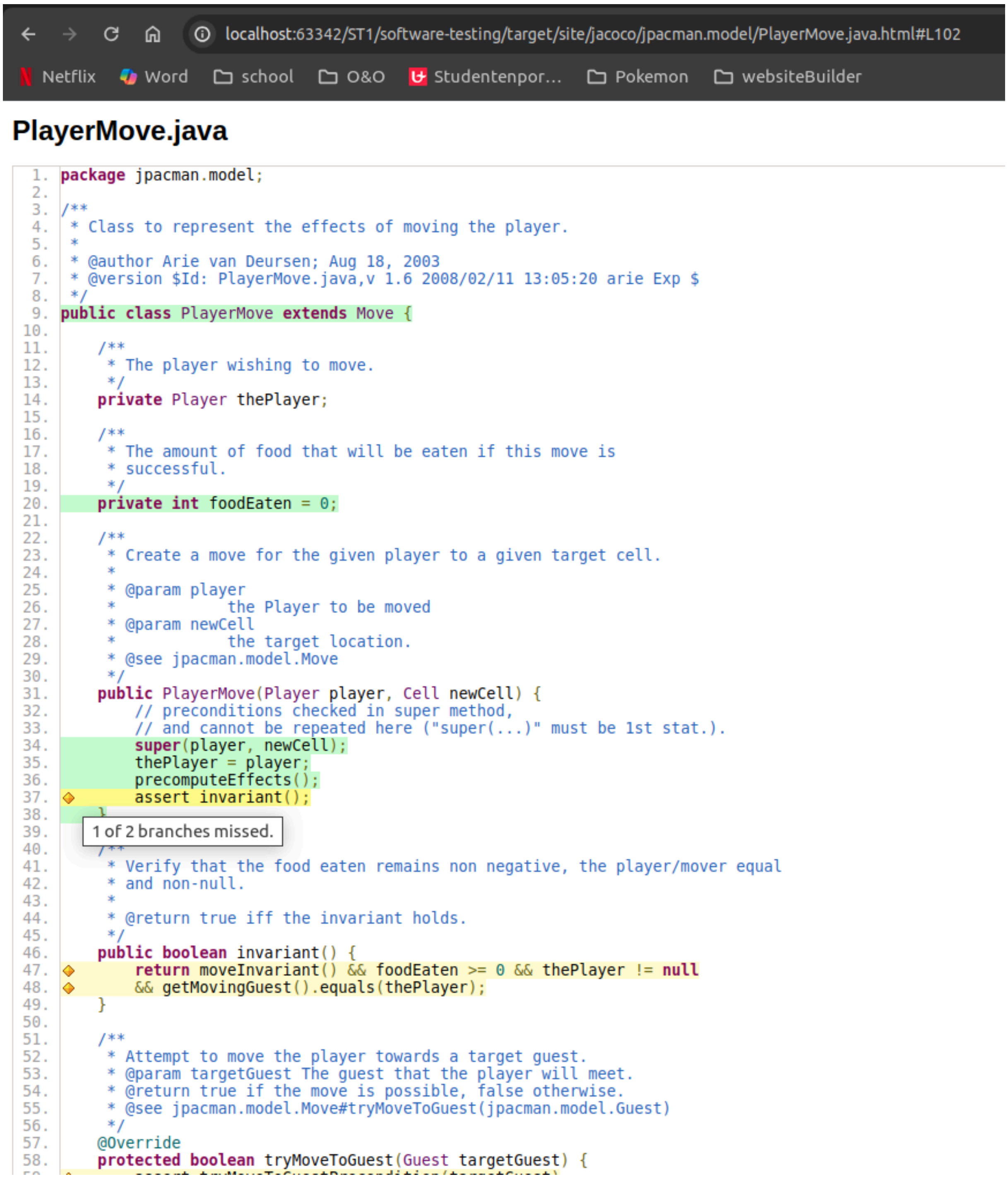
3.8 Exercise 8

Question 8: How many tests in your decision table would you need to get 100% coverage of the relevant moving methods? Why do you need the remaining test cases?

Our decision table is already fully implemented. If we look at the coverage, we notice that we don't get 100% on the relevant moving methods, see Figure 7. Further inspection shows us that especially the `assert` statements in every method are not fully covered w.r.t. instruction and branch coverage, see for example Figure 9. In every one of the test cases that we implemented in JUnit, we always made all Java assertion conditions true. In other words, we did not write tests that tried to fail the pre- and post-condition assertions of the methods under test. This is exactly why we don't have 100% coverage: not all branches of the assertion conditions were exhaustively covered.

If we would want to reach 100% coverage, we should implement some "faults". An example of those "faults" is trying to apply a *Move* without having done the initialization, which will make the statement `assert initialized(); // line 175 in Move.java` false, which will boost our coverage percentage.

Since the decision table from Exercise 1 focuses on player movements and cell occupation, there is an underlying assumption that relevant objects are correctly initialized. Checking whether or not a player has all the correct attributes is outside the scope of this decision table.


localhost:63342/ST1/software-testing/target/site/jacoco/jpacman.model/PlayerMove.java.html#L102
Netflix
Word
school
O&O
Studentenpor...
Pokemon
websiteBuilder

PlayerMove.java

```java
package jpacman.model;

/**
 * Class to represent the effects of moving the player.
 *
 * @author Arie van Deursen; Aug 18, 2003
 * @version $Id: PlayerMove.java,v 1.6 2008/02/11 13:05:20 arie Exp $
 */
public class PlayerMove extends Move {

    /**
     * The player wishing to move.
     */
    private Player thePlayer;

    /**
     * The amount of food that will be eaten if this move is
     * successful.
     */
    private int foodEaten = 0;

    /**
     * Create a move for the given player to a given target cell.
     *
     * @param player
     *            the Player to be moved
     * @param newCell
     *            the target location.
     * @see jpacman.model.Move
     */
    public PlayerMove(Player player, Cell newCell) {
        // preconditions checked in super method,
        // and cannot be repeated here ("super(...)" must be 1st stat.).
        super(player, newCell);
        thePlayer = player;
        precomputeEffects();
        assert invariant();
    }

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

    /**
     * Attempt to move the player towards a target guest.
     * @param targetGuest The guest that the player will meet.
     * @return true if the move is possible, false otherwise.
     * @see jpacman.model.Move#tryMoveToGuest(jpacman.model.Guest)
     */
    @Override
    protected boolean tryMoveToGuest(Guest targetGuest) {
```

1 of 2 branches missed.

Figure 9: Coverage inspection of the *PlayerMove* class.

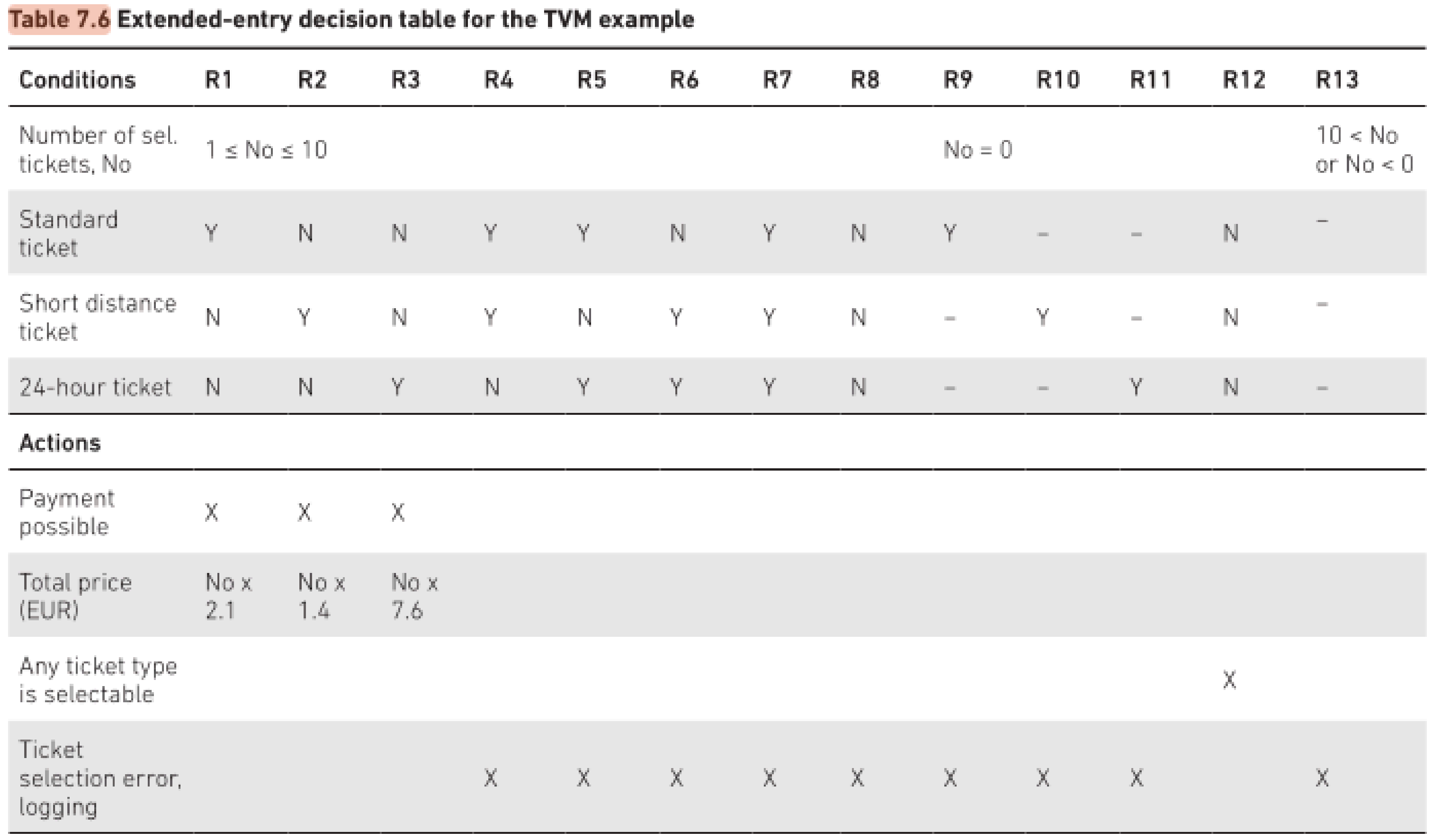


Table 7.6 Extended-entry decision table for the TVM example

Conditions	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10	R11	R12	R13
Number of sel. tickets, No	1 ≤ No ≤ 10								No = 0				10 < No or No < 0
Standard ticket	Y	N	N	Y	Y	N	Y	N	Y	–	–	N	–
Short distance ticket	N	Y	N	Y	N	Y	Y	N	–	Y	–	N	–
24-hour ticket	N	N	Y	N	Y	Y	Y	N	–	–	Y	N	–
Actions													
Payment possible	X	X	X										
Total price (EUR)	No x 2.1	No x 1.4	No x 7.6										
Any ticket type is selectable												X	
Ticket selection error, logging				X	X	X	X	X	X	X	X		X

Figure 10: The table referenced in Exercise 1.

```java
/**
 * Test to see if moving to an empty cell is allowed
 */
@Test
public void testMonsterMoveToEmptyCell() {
    aMonsterMove = new MonsterMove(monster, emptyCell);
    assertTrue(aMonsterMove.movePossible());
}

/**
 * Test to see if moving into a wall is not allowed
 */
@Test
public void testMonsterMoveToWall(){
    aMonsterMove = new MonsterMove(monster, wallCell);
    assertFalse(aMonsterMove.movePossible());
}

/**
 * Test to see if moving into a food token is not allowed.
 */
@Test
public void testMonsterMoveToFood(){
    aMonsterMove = new MonsterMove(monster, foodCell);
    assertFalse(aMonsterMove.movePossible());
}

@Test
public void testMonsterMoveToAnotherMonster(){
    aMonsterMove = new MonsterMove(monster, monsterCell);
    assertFalse(aMonsterMove.movePossible());
}

@Test
public void testMonsterMoveToPlayer(){
    aMonsterMove = new MonsterMove(monster, playerCell);
    assertTrue(aMonsterMove.playerDies());
    assertFalse(aMonsterMove.movePossible());
}

@Test
public void testMonsterMoveOutOfBounds(){
    // A NULL Cell represents an out-of-bounds Cell w.r.t. the Board.
    aMonsterMove = new MonsterMove(monster, null);
    assertFalse(aMonsterMove.movePossible());
}
```

Codeblock 9: The *MonsterMoveTest* test cases, extracted from the class so that it fits on one page in the report.

```java
/**
 * Test to see if moving to an empty cell is allowed
 */
@Test
public void testPlayerMoveToEmptyCell() {
    aPlayerMove = new PlayerMove(player, emptyCell);
    assertTrue(aPlayerMove.movePossible());
}

/**
 * Test to see if moving into a wall is not allowed
 */
@Test
public void testPlayerMoveToWall(){
    aPlayerMove = new PlayerMove(player, wallCell);
    assertFalse(aPlayerMove.movePossible());
}

/**
 * Test to see if moving into a food token is allowed and that the food gets
eaten properly
 */
@Test
public void testPlayerMoveToFood(){
    aPlayerMove = new PlayerMove(player, foodCell);
    assertTrue(aPlayerMove.movePossible());
    // apply the move
    aPlayerMove.apply();
    // check that the food is eaten and that we got a point
    assertEquals(1, player.getPointsEaten());
}

@Test
public void testPlayerMoveToMonster(){
    aPlayerMove = new PlayerMove(player, monsterCell);
    assertTrue(aPlayerMove.playerDies());
    assertFalse(aPlayerMove.movePossible());
}

@Test
public void testPlayerMoveToAnotherPlayer(){
    aPlayerMove = new PlayerMove(player, playerCell);
    assertFalse(aPlayerMove.movePossible());
}

@Test
public void testPlayerMoveOutOfBounds(){
    // A NULL Cell represents an out-of-bounds Cell w.r.t. the Board.
    aPlayerMove = new PlayerMove(player, null);
    assertFalse(aPlayerMove.movePossible());
}
```

Codeblock 10: The *PlayerMoveTest* test cases, extracted from the class so that it fits on one page in the report.

Referenties

[1] A. K. István Forgács, *Practical Test Design: Selection of traditional and automated test design techniques.*