Software Testing Assignment 3: Decision Structures

Lars De Leeuw 20205693
Robbe Teughels 20211127

March 2025

Contents

1 DECISION STRUCTURES **3**

1.1 Exercise 1 . . . 3
1.2 Exercise 2 . . . 3
1.2.1 Move . . . 3
1.2.2 PlayerMove . . . 3
1.2.3 Guest . . . 4
1.2.4 Food . . . 4
1.2.5 Wall . . . 4
1.2.6 MovingGuest . . . 4
1.2.7 Monster . . . 4
1.2.8 Player . . . 4
1.3 Exercise 3 . . . 4
1.4 Exercise 4 . . . 6
1.4.1 PlayerMove . . . 6
1.4.2 Guest . . . 6
1.4.3 Food . . . 6
1.4.4 Wall . . . 6
1.4.5 MovingGuest . . . 6
1.4.6 Monster . . . 6
1.4.7 Player . . . 7
1.5 Exercise 5 . . . 7
1.6 Exercise 6 . . . 7
1.7 Exercise 7 . . . 9
1.8 Exercise 8 . . . 10

1 DECISION STRUCTURES

1.1 Exercise 1

Create a decision table following the style of Table 7.6 (Forgács) 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. - *Lars*

Conditions	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10	R11
Move remains within the borders.	Y	Y	Y	Y	Y	Y	Y	N	Y	N	Y
Move possible for guest type.	-	N	Y	N	Y	N	N	N	N	Y	Y
Guest type is player?	Y	Y	Y	-	-	N	N	-	-	-	-
Type of targetGuest is Player.	Y	N	N	N	N	Y	-	-	-	-	-
Type of targetGuest is Monster.	N	Y	N	N	N	N	-	-	-	-	-
Type of targetGuest is Food.	N	N	Y	N	N	N	N	-	-	-	-
Type of targetGuest is Wall.	N	N	N	Y	N	N	N	-	-	-	-
Type of targetGuest is Empty.	N	N	N	N	Y	N	N	-	-	-	-
Actions											
Move.			X		X						
Eat food.			X								
Kill player.						X					
Impossible.	X								X	X	X

1.2 Exercise 2

Run the current test suite and describe the coverage of Move, PlayerMove, Guest, and all Guest subclasses. - *Lars*

1.2.1 Move

The Move class has 69% line coverage and 56% branch coverage.

1.2.2 PlayerMove

The PlayerMove class extends the Move class. It has 69% line coverage and 50% branch coverage.

1.2.3 Guest

The Guest class has a 68% line coverage and 56% branch coverage.

1.2.4 Food

The Food class extends the Guest class. It has a 69% line coverage and 50% branch coverage.

1.2.5 Wall

The Wall class extends the Guest class. It has a 61% line coverage and 50% branch coverage.

1.2.6 MovingGuest

The MovingGuest abstract class extends the Guest abstract class. Its line coverage is 100% and it's branch coverage is not available. This is explained by the fact that the class only consists of an empty constructor and has no branching paths.

1.2.7 Monster

The Monster class extends the MovingGuest abstract class. It has a 18% line coverage and a 0% branch coverage. This class has no specific test cases yet which explains why there is almost no coverage.

1.2.8 Player

The Player class extends the MovingGuest abstract class. It has a 43% line coverage and a 23% branch coverage.

1.3 Exercise 3

> **Implement all entries in the decision table concerning player movements as JUnit test cases in PlayerMoveTest class. Since the player movement has been im- plemented already, start by testing these.** - *Lars*

The JUnit tests 1 contain the tests for the entries related to a PlayerMove which are not impossible.

```
1 @Test
2 public void testPlayerMoveToPlayer() {
3     var playerMove = createMove(playerCell);
4     assertTrue(playerMove.initialized());
5
6     assertFalse(playerMove.movePossible());
7     assertFalse(playerMove.playerDies());
8     assertTrue(playerMove.invariant());
```

```
}

@Test
public void testPlayerMoveToMonster() {
    var playerMove = createMove(monsterCell);
    assertTrue(playerMove.initialized());

    assertFalse(playerMove.movePossible());
    assertTrue(playerMove.playerDies());
    assertTrue(playerMove.invariant());
}

@Test
public void testPlayerMoveToFood(){
    var oldFoodEaten = thePlayer.getPointsEaten();
    var playerMove = createMove(foodCell);
    assertTrue(playerMove.initialized());
    var foodAmount = playerMove.getFoodEaten();

    assertTrue(playerMove.movePossible());
    assertTrue(playerMove.invariant());
    playerMove.apply();

    assertEquals(thePlayer.getLocation().getX(), foodCell.getX());
    assertEquals(thePlayer.getLocation().getY(), foodCell.getY());
    assertEquals(thePlayer.getPointsEaten(), oldFoodEaten +
    foodAmount);
    assertFalse(playerMove.playerDies());
    assertTrue(playerMove.invariant());
}

@Test
public void testPlayerMoveToWall() {
    var playerMove = createMove(wallCell);
    assertTrue(playerMove.initialized());

    assertFalse(playerMove.movePossible());
    assertFalse(playerMove.playerDies());
    assertTrue(playerMove.invariant());
}

@Test
public void testPlayerMoveToEmpty(){
    var oldFoodEaten = thePlayer.getPointsEaten();
    var playerMove = createMove(emptyCell);
    assertTrue(playerMove.initialized());

    assertTrue(playerMove.movePossible());
    assertTrue(playerMove.invariant());
    playerMove.apply();

    assertEquals(thePlayer.getLocation().getX(), emptyCell.getX());
    assertEquals(thePlayer.getLocation().getY(), emptyCell.getY());
    assertEquals(thePlayer.getPointsEaten(), oldFoodEaten);
    assertFalse(playerMove.playerDies());
    assertTrue(playerMove.invariant());
```

```
64 }
```

Listing 1: "JUnit tests for PlayerMoveTest."

1.4 Exercise 4

> **Re-run with coverage enabled, and re-assess the coverage.**
> - *Lars*

1.4.1 PlayerMove

The PlayerMove class extends the Move class. It has 69% line coverage and 50% branch coverage. The coverage remained the same.

1.4.2 Guest

The Guest class has a 68% line coverage and 56% branch coverage. The coverage remained the same.

1.4.3 Food

The Food class extends the Guest class. It has a 69% line coverage and 50% branch coverage. The coverage remained the same.

1.4.4 Wall

The Wall class extends the Guest class. It has a 61% line coverage and 50% branch coverage. The coverage remained the same.

1.4.5 MovingGuest

The MovingGuest abstract class extends the Guest abstract class. Its line coverage is 100% and it's branch coverage is not available. This is explained by the fact that the class only consists of an empty constructor and has no branching paths. The coverage remained the same.

1.4.6 Monster

The Monster class extends the MovingGuest abstract class. The line coverag

> **Explain the interplay between the abstract methods Guest.meetPlayer and Move.tryMoveToGuest and their implementations in Guest and Move subclasses.** - *Lars*

The Guest.meetPlayer method serves as a means to probe what would happen if the guest were to be the targetCell of a PlayerMove. If the player can displace the guest the method returns true.

The Move.tryMoveToGuest method utilizes double dispatch, meaning its behavior is not only determined by the base class of the source Guest but also the base class of the target Guest. The Guest.meetPlayer method is used in the concrete implementations of the Move.tryMoveToGuest method. e increased from 18% to 63% and the branch coverage from 0% to 50%.

1.4.7 Player

The Player class extends the MovingGuest abstract class. The line coverage increased from 43% to 56% the branch coverage from 23% to 38%.

1.5 Exercise 5

> **Explain the interplay between the abstract methods Guest.meetPlayer and Move.tryMoveToGuest and their implementations in Guest and Move subclasses.** - *Lars*

The Guest.meetPlayer method serves as a means to probe what would happen if the guest were to be the targetCell of a PlayerMove. If the player can displace the guest the method returns true.

The Move.tryMoveToGuest method utilizes double dispatch, meaning its behavior is not only determined by the base class of the source Guest but also the base class of the target Guest. The Guest.meetPlayer method is used in the concrete implementations of the Move.tryMoveToGuest method.

1.6 Exercise 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.** - *Robbe*

The implementation is similar to the PlayerMove. A new abstract method "meetMonster" is defined in Guest for specialized behavior. The usage is the same as the meetPlayer method from exercise 1.5.

```
1  public class MonsterMove extends Move {
2
3      /**
4       * The monster wishing to move.
5       */
6      private Monster theMonster;
7
8      /**
9       * Create a move for the given monster to a given target cell.
10      *
```

```
     * @param monster
     *            the Monster to be moved
     * @param newCell
     *            the target location.
     * @see jpacman.model.Move
     */
    public MonsterMove(Monster monster, Cell newCell) {
        // preconditions checked in super method,
        // and cannot be repeated here ("super(...)" must be 1st
    stat.).
        super(monster, newCell);
        theMonster = monster;
        precomputeEffects();
        assert invariant();
    }

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

    /**
     * Attempt to move the monster towards a target guest.
     * @param targetGuest The guest that the monster will meet.
     * @return true if the move is possible, false otherwise.
     * @see Move#tryMoveToGuest(Guest)
     */
    @Override
    protected boolean tryMoveToGuest(Guest targetGuest) {
        assert tryMoveToGuestPrecondition(targetGuest)
            : "percolated precondition";
        return targetGuest.meetMonster(this);
    }

    /**
     * Return the monster initiating this move.
     *
     * @return The moving monster.
     */
    public Monster getMonster() {
        assert invariant();
        return theMonster;
    }

    /**
     * Actually apply the move, assuming it is possible.
     */
    @Override
    public void apply() {
        assert invariant();
        assert movePossible();
```

```
66          super.apply();
67          assert invariant();
68      }
69  }
```

Listing 2: "Implemenation of MonsterMove"

1.7 Exercise 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. - *Robbe*

In MonsterMoveTest, we have implemented R4 - R8. These test check for all possible combinations that we can expect to happen for a MonsterMove. This results in a 70% line coverage and 50% branch coverage for the MonsterMove class.

```
1   @Test
2   public void testMonsterMoveToPlayer() {
3       var monsterMove = createMove(playerCell);
4       assertTrue(monsterMove.initialized());
5
6       assertFalse(monsterMove.movePossible());
7       assertTrue(monsterMove.playerDies());
8       assertTrue(monsterMove.invariant());
9   }
10
11  @Test
12  public void testMonsterMoveToMonster() {
13      var monsterMove = createMove(monsterCell);
14      assertTrue(monsterMove.initialized());
15
16      assertFalse(monsterMove.movePossible());
17      assertFalse(monsterMove.playerDies());
18      assertTrue(monsterMove.invariant());
19  }
20
21  @Test
22  public void testMonsterMoveToFood() {
23      var monsterMove = createMove(foodCell);
24      assertTrue(monsterMove.initialized());
25
26      assertFalse(monsterMove.movePossible());
27      assertFalse(monsterMove.playerDies());
28      assertTrue(monsterMove.invariant());
29  }
30
31  @Test
32  public void testMonsterMoveToWall() {
33      var monsterMove = createMove(wallCell);
34      assertTrue(monsterMove.initialized());
35
36      assertFalse(monsterMove.movePossible());
```

```
37      assertFalse(monsterMove.playerDies());
38      assertTrue(monsterMove.invariant());
39  }
40
41  @Test
42  public void testMonsterMoveToEmpty() {
43      var monsterMove = createMove(emptyCell);
44      assertTrue(monsterMove.initialized());
45
46      assertTrue(monsterMove.movePossible());
47      assertTrue(monsterMove.invariant());
48      monsterMove.apply();
49
50      assertEquals(theMonster.getLocation().getX(), emptyCell.getX())
        ;
51      assertEquals(theMonster.getLocation().getY(), emptyCell.getY())
        ;
52
53      assertFalse(monsterMove.playerDies());
54      assertTrue(monsterMove.invariant());
55  }
```

Listing 3: "Implemenation of MonsterMoveTest"

1.8 Exercise 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? - *Robbe*

Looking at a detailed coverage report, the uncovered parts are those that are impossible to occur. These are the pre- and post-conditions that are not tested to fail. These are included in the decision table at R8 -R11. Implementing these should result in 100% branch and 100% line coverage. We currently have 5 tests and 1 duplicate, This results in 8 total tests.