Software Testing and Quality Engineering

STQE Labwork, IN3205-II - Part 2

Giovanni Martina, #1129228
Cees-Willem Hofstede, #1272012

Exercise 33

In other to know what should happen when monsters and players try to move we have come up with the following decision table indicating the various possible conditions.

Player	T	T	T	T	T	T	T	F	F	F	F	F	F	F
CWB	T	-	F	T	T	T	-	-	F	-	T	T	T	T
COP	F	T	-	F	F	F	-	-	-	-	T	F	F	F
COM	F	-	-	F	F	T	-	-	-	T	F	F	F	F
COF<TFC	F	-	-	T	F	-	-	-	-	-	-	T	F	-
COF==TFC	F	-	-	F	T	-	-	-	-	-	-	F	T	-
COW	F	-	-	F	F	F	T	T	-	-	F	F	F	F
Out	AM	DM	DM	IFC	PW	PD	DM	DM	DM	DM	PD	DM	DM	AM

Abbreviations given in this table have the following meaning.

Player: Moving guest is a player

CWB: Destination cell is within borders
COP: Destination cell is occupied by another player
COM: Destination cell is occupied by a monster
COF<TFC: Destination cell is occupied by food and current food count is less than total food count
COF==TFC: Destination cell is occupied by the last available food item
COW: Destination cell is occupied by a wall
AM: Allow move
DM: Deny move
IFC: Increase current food count
PW: Player wins
PD: Player dies

Exercise 34

Given our decision table we can come up with the following test case specifications.

The validity and consequence of a move depends on the type of MovingGuest initiating the move. In this game only players and monsters can move. For a player a requested move should be denied when that player wants to move to a cell that is not within the defined borders of the board. Moving a player to a wall should likewise be denied and a possible occasion of moving a player to a cell occupied by another player should also be denied.

When moving a player to a cell within the defined borders of the board that is not occupied by any guest (food, monster or wall) the move should be allowed. When moving a player to cell within the defined borders of the board that is occupied by a food item the current food count should be increased while it has not reached the total food count yet. When moving a player to a cell within the defined borders that is occupied by the last food item available the player should win the game. When moving the player to call within the defined borders that is occupied by a monster the player should die.

A monster requesting a move to a cell defined outside the borders of the board should be denied that request. Likewise moves to cells containing other monsters or containing a wall should also be denied. A move by a monster to a cell within the borders should be allowed if the destination cell does not contain a wall or player. A move to a cell defined within borders of the board and containing a player should lead to the player dying and the monster taking its place in the cell. Finally a move by a monster to a cell containing a food item should be denied.

Excercise 35

Move

name	class, %	method, %	block, %	line, %
class Move	100% (1/1)	92% (12/13)	72% (244/341)	79% (43.2/55)
die (): void		0% (0/1)	0% (0/13)	0% (0/3)
apply (): void		100% (1/1)	65% (44/68)	74% (8.9/12)
playerDies (): boolean		100% (1/1)	67% (8/12)	78% (1.6/2)
precomputeEffects (): void		100% (1/1)	70% (50/71)	85% (11.9/14)
Move (MovingGuest, Cell): void		100% (1/1)	74% (48/65)	91% (11.8/13)
<static initializer>		100% (1/1)	75% (6/8)	75% (0.8/1)
moveDone (): boolean		100% (1/1)	76% (25/33)	72% (2.2/3)
movePossible (): boolean		100% (1/1)	81% (21/26)	75% (1.5/2)
withinBorder (): boolean		100% (1/1)	86% (6/7)	85% (0.8/1)
tryMoveToGuestPrecondition (Guest): boolean		100% (1/1)	92% (11/12)	91% (0.9/1)
moveInvariant (): boolean		100% (1/1)	95% (19/20)	95% (1/1)
getMovingGuest (): MovingGuest		100% (1/1)	100% (3/3)	100% (1/1)
initialized (): boolean		100% (1/1)	100% (3/3)	100% (1/1)

As can be seen in the table the die() method is not covered at all by the test suite. In the testsuite a move is actually made, but the only thing that is tested is that when a move is possible, the move will actually take place as expexted. This means that the move will always go towards an empty cell, hence die() could not occur from that move. Some other methods are only covered partially because of boolean checks that lead to the fact that some parts are not cover. For example a part for when the cell is occupied. This will not get covered since it is a precondition that the cell is empty.

PlayerMove

class PlayerMove	*100% (1/1)*	*100% (8/8)*	*72% (119/165)*	*84% (21.7/26)*
setFoodEaten (int): void		*100% (1/1)*	*58% (14/24)*	*78% (3.1/4)*
getFoodEaten (): int		*100% (1/1)*	*67% (8/12)*	*78% (1.6/2)*
getPlayer (): Player		*100% (1/1)*	*67% (8/12)*	*78% (1.6/2)*
tryMoveToGuest (Guest): boolean		*100% (1/1)*	*67% (10/15)*	*77% (1.5/2)*
apply (): void		*100% (1/1)*	*70% (37/53)*	*81% (6.5/8)*
<static initializer>		*100% (1/1)*	*75% (6/8)*	*75% (0.8/1)*
PlayerMove (Player, Cell): void		*100% (1/1)*	*82% (18/22)*	*97% (5.8/6)*
invariant (): boolean		*100% (1/1)*	*95% (18/19)*	*94% (0.9/1)*

All functions are tested but because of all the asserts and several shorthand boolean checks only some parts of the functions get covered.

Guest and subclasses
Guest is covered in the same way PlayerMove is. All the functions are covered, but because of the asserts and shorthand boolean checks some part of methods are not. The same holds for the Monster subclass with the exception the method meetPlayer() is never covered because in the testsuite the player will never bumb into a monster. For the exact same reason the same coverage conditions hold for the Player subclass with the addition of a method die() which is never covered for the exact same reason as above.

Exercise 36

The test specifications as given previously have been added to PlayerMoveTest as JUnit test cases. Testing the impossibility of a player move has been achieved by checking whether or not an assertion error is generated and whether the move was actually performed. Other tests done are checking if eating food leads to the food count increasing by one and if eating all food leads to winning the game. We also check on the impossibility of moving into walls and dying when meeting monsters and on allowing moves to empty cells.

```
@Test
/**
 * Test moving a player to a cell outside the borders of the board
 */
public void testMovingOutsideBorders() {
   if (TestUtils.assertionsEnabled()) {
          boolean failureGenerated;
          boolean movePossible = false;
          try {
                 Board board = getTheGame().getBoard();
                 Cell cellOutside = board.getCell(board.getWidth() + 1, board.getHeight() + 1);
                 PlayerMove move = createMove(cellOutside);
                 movePossible = move.movePossible();
                 failureGenerated = false;
          }
          catch (AssertionError ae) {
                 failureGenerated = true;
          }

          assertTrue(failureGenerated);
          assertFalse(movePossible);
   }
}
```

```java
@Test
/*
 * A move to a wall should not be possible
 */
public void testMoveToWall() {
    PlayerMove move = createMove(getWallCell());
    assertFalse(move.movePossible());
    assertFalse(move.playerDies());
    assertTrue(move.invariant());
}

@Test
/*
 * A move to a monster should kill me
 */
public void testMoveToMonster() {
    PlayerMove move = createMove(getMonsterCell());
    assertFalse(move.movePossible());
    assertTrue(move.playerDies());
    assertTrue(move.invariant());
}

@Test
/*
 * A move to an empty cell should be ok
 */
public void testMoveToEmptyCell() {
    PlayerMove move = createMove(getEmptyCell());
    assertTrue(move.movePossible());
    assertFalse(move.playerDies());
    move.apply();
    assertTrue(move.moveDone());
    assertTrue(move.invariant());
}

@Test
/*
 * Test that eating food should increase my score
 */
public void testEatingFood() {
    PlayerMove move = createMove(getFoodCell());
    int food = getThePlayer().getPointsEaten();
    assertEquals(0, food);
    assertTrue(move.movePossible());
    assertEquals(1, move.getFoodEaten());
    move.apply();
    assertTrue(move.moveDone());
    assertEquals(food + 1, getThePlayer().getPointsEaten());
    assertTrue(move.invariant());
}
```

```
    @Test
    /*
     * Test winning the game by eating all food
     */
    public void testWinTheGame() {
       Cell theOtherFoodCell = getTheGame().getBoard().getCell(0, 2);
       boolean isFood = theOtherFoodCell.getInhabitant() instanceof Food;
       assertTrue(isFood);
       PlayerMove move = createMove(getTheGame().getBoard().getCell(0, 2));
       assertTrue(move.movePossible());
       move.apply();
       assertTrue(move.moveDone());
       assertTrue(move.invariant());
       move = createMove(getFoodCell());
       assertTrue(move.movePossible());
       move.apply();
       assertTrue(move.moveDone());
       assertTrue(getTheGame().gameOver());
       assertTrue(getTheGame().playerWon());
       assertFalse(getTheGame().playerDied());
       assertTrue(move.invariant());
    }
```

Exercise 37

In the Move class method coverage of die() has been raised to 100% due to the addition of the test cases in PlayerMoveTest. This makes sense as we now let the player meet a monster in our tests. The coverage of PlayerMove and Guest has not changed at all with the addition of the test cases given in exercise 36. For Guest subclasses Wall, Food and Monster the meetPlayer() method is now being covered but for Player the methods meetPlayer() and die() are not being covered. The method meetPlayer() in Player is not being covered seeing as there is no support for multiplayer games.

Exercise 38

The abstract super class Guest has a method called meetPlayer() that children of Guest must implement in order to indicate whether it is possible for a Player to move to the current Guest's position and take its place. Children implementations of Guest should modify the state of the Move object passed as an argument to indicate the possible effects of the Player meeting the current Guest.

A Move represents a movement from one cell to another initiated by the Guest in the original location. The effects of the move depend on both the source Guest and possible destination Guest if any. When a Move is initiated by a Player. A call to tryMoveToGuest() gets forwarded to the meetPlayer() implementations of the children of Guest. Depending on the child different Move states and actions are possible. In the case of Wall the move is disallowed, in the case of Food, the points are increased and the move is allowed. In the case of Monster the move is disallowed and the state changes to indicate the player dies. In the case of another Player being the Guest at the destination cell the move is also disallowed. This last case should never happen though as it is currently impossible to have more than one Player.

Exercise 39

```java
package jpacman.model;

import static org.junit.Assert.assertEquals;
import static org.junit.Assert.assertFalse;
import static org.junit.Assert.assertNotSame;
import static org.junit.Assert.assertTrue;
import jpacman.TestUtils;

import org.junit.Test;

/**
 * Specialize the general MoveTest test suite to one
 * that is tailored to MonsterMoves.
s */
public class MonsterMoveTest extends MoveTest {

       /**
     * The move the monster would like to make.
     */
       private MonsterMove aMonsterMove;

    /**
     * Create a move object that will be tested.
     *  @see jpacman.model.MoveTest#createMove(jpacman.model.Cell)
     *  @param target The cell to be occupied by the move.
     *  @return The move to be tested.
     */
       @Override
       protected MonsterMove createMove(Cell target) {
              aMonsterMove = new MonsterMove(getTheMonster(), target);
        return aMonsterMove;
       }

       /*
        * test that the player will in fact die when a monster meets a player.
        */
       @Test
       public void playerDies() {
              Cell playerCell = getPlayerCell();
              MonsterMove monsterMove = createMove(playerCell);
              assertEquals(getThePlayer(), playerCell.getInhabitant());
              assertEquals(getTheMonster(), monsterMove.getMonster());
              assertTrue(monsterMove.playerDies());
              assertTrue(monsterMove.invariant());
       }

       @Test
       public void allowMove() {
              Cell emptyCell = getEmptyCell();
              MonsterMove monsterMove = createMove(emptyCell);
              assertTrue(monsterMove.movePossible());
              monsterMove.apply();
              assertTrue(monsterMove.moveDone());
              assertTrue(monsterMove.invariant());
       }

       @Test
       public void denyMoveWhenWall() {
              Cell wallCell = getWallCell();
              MonsterMove monsterMove = createMove(wallCell);
              assertFalse(monsterMove.movePossible());
              assertTrue(monsterMove.invariant());
       }
```

```
        @Test
        public void denyMoveWhenMonster() {
                Cell monsterCell = getMonsterCell();
                MonsterMove monsterMove = createMove(monsterCell);
                assertFalse(monsterMove.movePossible());
                assertTrue(monsterMove.invariant());
        }

    @Test
    /**
     * Test moving a monster to a cell outside the borders of the board
     */
    public void testMovingOutsideBorders() {
        if (TestUtils.assertionsEnabled()) {
                boolean failureGenerated;
                boolean movePossible = false;
                try {
                        Board board = getTheGame().getBoard();
                        Cell cellOutside = board.getCell(board.getWidth() + 1, board.getHeight() + 1);
                        MonsterMove move = createMove(cellOutside);
                        movePossible = move.movePossible();
                        failureGenerated = false;
                }
                catch (AssertionError ae) {
                        failureGenerated = true;
                }

                assertTrue(failureGenerated);
                assertFalse(movePossible);
        }
    }
}
```

Exercise 40

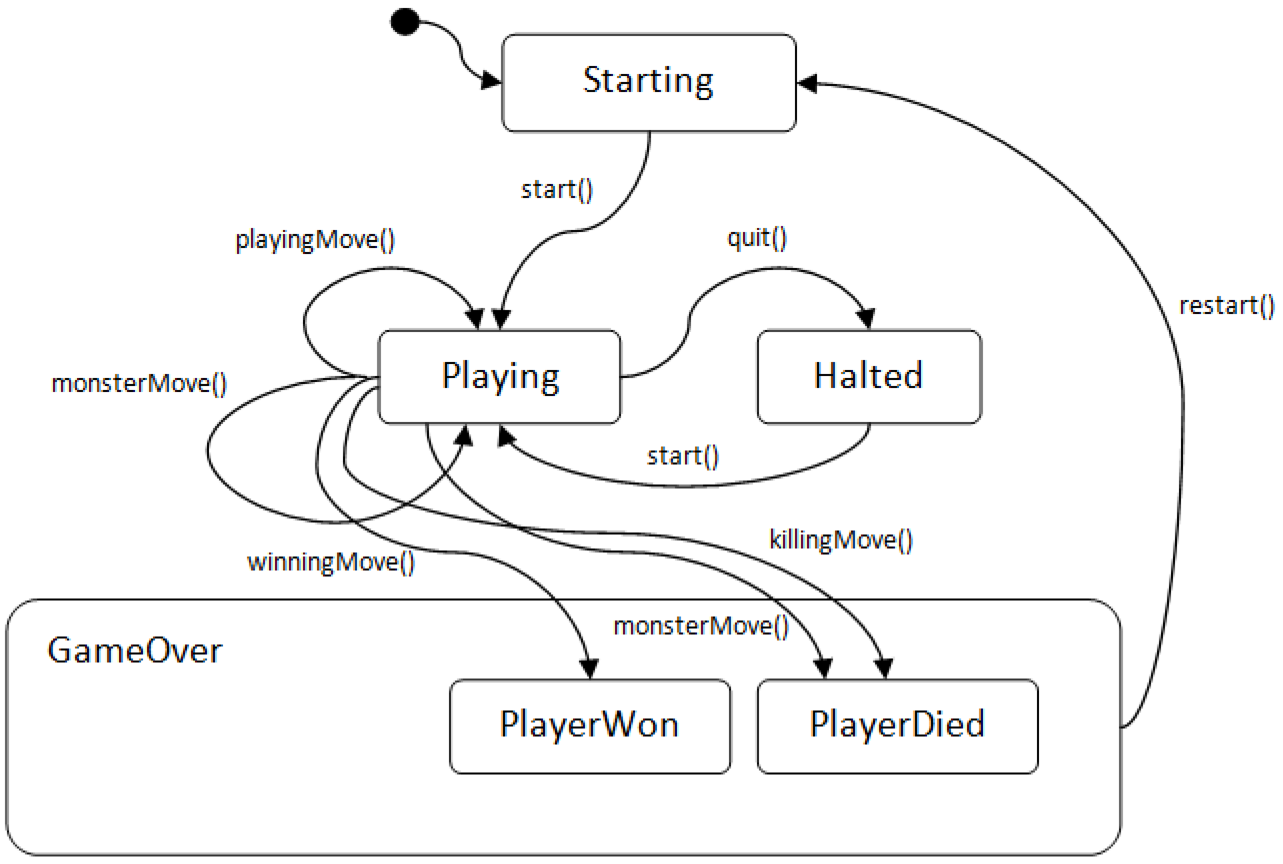
```
package jpacman.model;

/**
 * Class to represent the effects of moving a monster.
 */
public class MonsterMove extends Move {

        /**
     * The player wishing to move.
     */
    private Monster theMonster;

    /**
     * Create a move for the given monster to a given target cell.
     *
     * @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 false at all times, since the monsters cannot move over occupied cells
     * @see jpacman.model.Move#tryMoveToGuest(jpacman.model.Guest)
     */
    @Override
    protected boolean tryMoveToGuest(Guest targetGuest) {
        assert tryMoveToGuestPrecondition(targetGuest)
            : "percolated precondition";
        if (targetGuest.guestType() == Guest.PLAYER_TYPE) {
              die();
        }
        return false;
    }

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

Exercise 41

If this question is about coverage of the MonsterMoveTest (a bit unclear) than only the outofborders test could be thrown out since this will never be the case in the current design. However, we may extend the game later on that does require this possibility and so, the test as well.

Exercise 42

After reading through the description of the state machine we've created the following UML state diagram of the Engine class.



Several move() methods can be distinguished in this diagram. We have monsterMove() to indicate Monster movement. This can result in no change from the Playing state or in a change to the PlayerDied state if a Monster meets a Player. Secondly we have playingMove() to indicate Player movement that won't result in the Player dying or winning the game. Thirdly and fourthly we have winningMove() to indicate a Move by the Player that will result in him winning the game and killingMove() when a Player tries to move to a cell occupied by a Monster, resulting in his death.

Exercise 43

The following state transitions achieve transition coverage because by following the sequences we manage to traverse every possible state transition.

Starting->Playing->Playing->PlayerWon->Starting
Starting->Playing->Halted->Playing
Starting->Playing->Playing->PlayerDied
Starting->Playing->Playing->PlayerDied->Starting

Now transforming these to test case specifications we achieve the following. It's evident that after the method calls we need to assert the correct Engine state.

Test Case 1:

- start()
- playingMove()
- winningMove()
- restart()

We start the game, let the player perform a playing move, let the player perform a winning move and then restart the game.

Test Case 2:

- start()
- quit()
- start()

We start the game, then halt the game and start the game again.

Test Case 3:

- start()
- playingMove()
- killingMove()

We start the game, let the player perform a playing move, let the player perform a killing move by meeting a monster.

Test Case 4:

- start()
- monsterMove()
- monsterMove()
- restart()

We start the game, let the monster perform a monster move, let the monster perform a monster move that kills the player by meeting him and then restart the game.

Exercise 44

The test cases as obtained from the previous UML state diagram have been added to EngineTest as the following JUnit tests.

```java
   /**
    * Test Case 1: starting-playing-playing-playerwon-starting
    * as obtained from UML state diagram
    */
   @Test
   public void testStateTransition1() {
       assertTrue(theEngine.inStartingState());
       theEngine.start();
       assertTrue(theEngine.inPlayingState());
       theEngine.movePlayer(-1, 0); //eat food left
       assertTrue(theEngine.inPlayingState());
       theEngine.movePlayer(0, 1); //eat food down
       assertTrue(theEngine.inGameOverState());
       assertTrue(theEngine.inWonState());
       theEngine.start();
       assertTrue(theEngine.inStartingState());
       assertTrue(theEngine.invariant());
   }

   /**
    * Test Case 2: starting-playing-halted-playing
    * as obtained from UML state diagram
    */
   @Test
   public void testStateTransition2() {
       assertTrue(theEngine.inStartingState());
       theEngine.start();
       assertTrue(theEngine.inPlayingState());
       theEngine.quit();
       assertTrue(theEngine.inHaltedState());
       theEngine.start();
       assertTrue(theEngine.inPlayingState());
       assertTrue(theEngine.invariant());
   }

   /**
    * Test Case 3: starting-playing-playing-playerdied
    * as obtained from UML state diagram
    */
   @Test
   public void testStateTransition3() {
       assertTrue(theEngine.inStartingState());
       theEngine.start();
       assertTrue(theEngine.inPlayingState());
       theEngine.movePlayer(1, 0); //go to the empty cell to the right
       assertTrue(theEngine.inPlayingState());
       theEngine.movePlayer(-1, 1); //go to a monster
       assertTrue(theEngine.inGameOverState());
       assertTrue(theEngine.inDiedState());
       assertTrue(theEngine.invariant());
   }

   /**
    * Test Case 4: starting-playing-playing-playerdied-starting
    * as obtained from UML state diagram
    */
   @Test
   public void testStateTransition4() {
       assertTrue(theEngine.inStartingState());
       theEngine.start();
       assertTrue(theEngine.inPlayingState());
```

```
        theEngine.moveMonster(getTheMonster(), 1, 0); //monster goes to empty cell
        assertTrue(theEngine.inPlayingState());
        theEngine.moveMonster(getTheMonster(), -1, -1); //monster goes to player
        assertTrue(theEngine.inGameOverState());
        assertTrue(theEngine.inDiedState());
        theEngine.start();
        assertTrue(theEngine.inStartingState());
        assertTrue(theEngine.invariant());
    }
```

The last test fails due to the incomplete implementation of monster movement in the application.

Exercise 45

Using transition coverage we can come up with the following state sequences that identify omitted pairs.

Starting->Halted->Starting
Starting->GameOver->Playing
Starting->Playing->Starting
Starting->Halted->GameOver->Halted

Using method calls we can transform these to the following test case specifications.

Test Case 1:

- quit()
- start()

Test Case 2:

- killingMove()
- start()

Test Case 3:

- start()
- start()

Test Case 4:

- quit()
- killingMove()
- quit()

Now in order to verify that no sneak path is implemented we look at the implementation of Engine. In Test Case 1 we immediately enter the Halted state from the Starting state by issuing quit(). From the implementation it's clear that this is impossible because quit() verifies that we are currently in the Playing state in order to alter the current state. Entering the Starting state from the Halted state is also not possible because issuing quit() from the Halted state only sets halted to false (meaning we don't explicitly enter the Starting state).

Issuing a killingMove() from the Starting state in Test Case 2 is not possible due to a check that verifies when performing a move that we are currently in the Playing state. Issuing a start() from the GameOver

states causes us to return to the Starting state and not the Playing state.
In Test Case 3 issuing a start() from the Playing state has no effect whatsoever so it's impossible to go from Playing to Starting.

Finally in Test Case 4 like in Test Case 1 it's again impossible to issue quit() and have that change our state from Starting to Halted. Performing a move from the Halted state is impossible as well due to the check to verify we are currently in the Playing state. Changing from GameOver to Halted is impossible because issuing quit() verifies that we are currently in the Playing state.

Exercise 46

The following JUnit tests have been added to EngineTest.

```
    /**
     * Test Sneak Path: starting-halted-starting
     */
    @Test
    public void testSneakPath1() {
        assertTrue(theEngine.inStartingState());
        theEngine.quit();
        assertFalse(theEngine.inHaltedState());

        //go to playing then halted
        theEngine.start();
        assertTrue(theEngine.inPlayingState());
        theEngine.quit();
        assertTrue(theEngine.inHaltedState());

        //now go from halted to starting
        theEngine.start();
        assertFalse(theEngine.inStartingState());
        assertTrue(theEngine.invariant());
    }

    /**
     * Test Sneak Path: starting-gameover-playing
     */
    @Test
    public void testSneakPath2() {
        assertTrue(theEngine.inStartingState());
        theEngine.movePlayer(0, 1); //move player to monster and die
        assertFalse(theEngine.inGameOverState());

        //go to playing, kill player then go to playing again
        theEngine.start();
        assertTrue(theEngine.inPlayingState());
        theEngine.movePlayer(0, 1); //move player to monster and die
        assertTrue(theEngine.inGameOverState());
        theEngine.start();
        assertFalse(theEngine.inPlayingState());
        assertTrue(theEngine.invariant());
    }

    /**
     * Test Sneak Path: starting-playing-starting
     */
    @Test
    public void testSneakPath3() {
        assertTrue(theEngine.inStartingState());
        theEngine.start();
        assertTrue(theEngine.inPlayingState());
        theEngine.start();
        assertFalse(theEngine.inStartingState());
        assertTrue(theEngine.invariant());
    }

    /**
     * Test Sneak Path: starting-halted-gameover-halted
     */
    @Test
    public void testSneakPath4() {
        assertTrue(theEngine.inStartingState());
        theEngine.quit();
        assertFalse(theEngine.inHaltedState());

        //go to playing then halted
        theEngine.start();
        assertTrue(theEngine.inPlayingState());
        theEngine.quit();
        assertTrue(theEngine.inHaltedState());

        //go to gameover from halted
        theEngine.movePlayer(0, 1); //move player to monster and die
        assertFalse(theEngine.inGameOverState());

        //go to halted from gameover
        theEngine.start();
        assertTrue(theEngine.inPlayingState());
        theEngine.movePlayer(0, 1); //move player to monster and die
        assertTrue(theEngine.inGameOverState());
        theEngine.quit();
        assertFalse(theEngine.inHaltedState());
        assertTrue(theEngine.invariant());
    }
```

Exercise 47

After checking the coverage results after the addition of these sneak path tests it becomes clear that Engine has good coverage (minimum of 65% due to asserts). There is one test that fails because we still have no implementation of moving monsters.

Also it's not entirely clear if we need to verify coverage of the entire test suite. If so we come across occasions where methods still have 0% coverage. These are included in the Player, Cell, Board and GameTestCase classes. The meetPlayer() method of Player has 0% coverage as it is impossible to test this without multiplayer support. The toString() methods of Cell and Board are untested but we can test them by adding JUnit tests to CellTest and BoardTest respectively. Having done so these methods reach 100% method coverage. The getTheFood() method of GameTestCase has 0% method coverage, we add a simple getter test to PlayerMoveTest that covers this code and we manage to reach 100% method coverage for getTheFood() by doing so.

Finally, GameLoadException has 0% coverage as currently only hardcoded maps are used and it's impossible to load maps from external files.

Added to CellTest:

```
    @Test
    public void testToString() {
        String cell = "0@[1,1]";
        assertEquals(cell, centerCell.toString());
    }
```

Added to BoardTest:

```
    @Test
    public void testToString() {
       String board = "00000\n00000\n00000\n00000\n00000\n00000\n00000\n00000\n00000\n00000\n";
       assertEquals(board, theBoard.toString());
    }
```

Modified PlayerMoveTest:

```
    /**
     * Simple test of a few getters.
     */
    @Test
    public void testSimpleGetters() {
        PlayerMove playerMove = new PlayerMove(getThePlayer(), getFoodCell());
        assertEquals(getThePlayer(), playerMove.getPlayer());
        assertTrue(playerMove.movePossible());
        assertFalse(playerMove.playerDies());
        assertEquals(1, playerMove.getFoodEaten());
        assertEquals(getTheFood(), getFoodCell().getInhabitant());
        assertTrue(playerMove.invariant());
    }
```

Exercise 48

Engine and Game have been modified to allow for Monster moves.

Modified Game:

```
    /**
     * Move a monster to offsets (x+dx,y+dy). If the move is not possible
     * (non empty cell, beyond borders), the move is not carried out.
     * Precondition: initialized and game is not over yet
     * Postcondition: if move possible, move was performed and game updated to
     * reflect the new situation
     * @param m
     *        The monster to move
     * @param dx
     *        Horizontal movement
     * @param dy
     *        Vertical movement
     */
    void moveMonster(Monster m, int dx, int dy) {
       assert invariant();
       assert !gameOver();
       assert m != null;
       Cell targetCell =
              m.getLocation().cellAtOffset(dx, dy);
       MonsterMove move = new MonsterMove(m, targetCell);
       applyMove(move);
       assert invariant();
    }
```

Modified Engine:

```
    /**
     * Try to move the given monster along a given offset.
     *
     * @param monster
     *            The monster to be moved
     * @param dx
     *            Horizontal offset
     * @param dy
     *            Vertical offset
     */
    public synchronized void moveMonster(Monster monster, int dx, int dy) {
        assert invariant();
        assert monster != null;
        if (inPlayingState()) {
               theGame.moveMonster(monster, dx, dy);
               notifyViewers();
        }
        assert invariant();
    }
```

Exercise 49

After the actual implementation of monster movement has been added to Game and Engine all JUnit tests in EngineTest pass and have 100% method coverage. The rest of the classes continue with the same coverage as before.

Exercise 50

UC10: undo after dying.
Actor: player
1. The player dies.
2. The user presses the Undo button.
3. The moment the Undo button is pressed, the game enters the halted state, the player is brought back to life. and sets the movement 1 step back.

UC11: undo when halted.
Actor: player
1. The game is in the halted state
2. The user presses the Undo button.
3. The moment the Undo button is pressed, sets the movement 1 step back.
4. If the last move involved eating a food item, decrease the food count and place a food item on the cell where the player came from.

UC12: undo while playing.
Actor: player
1. the player is alive and playing
2. The user presses the Undo button.
3. The moment the Undo button is pressed, the game enters the halted state and sets all the movements 1 step back.
4. If the last move involved eating a food item, decrease the food count and place a food item on the cell where the player came from.

UC13: monster undo.
Actor: monster
1. The user presses the Undo button.
2. The moment the Undo button is pressed, the movement is set x steps back. The amount of steps back the monsters move depends on the player's last move before the Undo button was pressed. If for example a monster has performed three moves after the player's last move before Undo was pressed, that monster would need to take three steps back to be placed at his original location since the player's last move.

Exercise 51

D7. The Undo Functionality.
The Undo button causes the game to enter the halted state, when not already halted. We need to keep track of a Player's move and of Monster moves. We can do this by using a stack to keep track of all the moves performed during gameplay. Each time the player presses the keyboard and a valid move has been performed we push the move onto the stack. Whenever one of the monsters performs a valid move we push its move onto the stack as well. Now suppose we are currently in playing mode. Whenever the undo button gets pressed we keep popping all moves from the stack until we've popped exactly one player move as well. This results in all monster moves performed after the last player move getting undone and the last player move getting undone as well. Now the game has been halted and pressing the button again will again keep popping all moves until we've popped exactly one player move from the stack. When the stack is empty no more moves can be undone. If some action has been achieved during a move (dying, increasing score) we revert those actions accordingly.

Exercise 52

Test cases in MoveTest.java:

```java
    /**
     * Test the undo functionality.
     */
    @Test public void testUndo() {
        aMove = createMove(getEmptyCell());
        MovingGuest mover = aMove.getMovingGuest();
        Cell location1 = mover.getLocation();
        aMove.apply();
        Cell location2 = mover.getLocation();
        assertNotSame(location1, location2);
        aMove.undo();
        Cell location3 = mover.getLocation();
        assertEquals(location1, location3);
    }
```

Test cases in PlayerMoveTest.java:

```
/**
 * test undo move when food was eaten
 */
@Test public void testFoodUndo() {
   PlayerMove move = createMove(getFoodCell());
   int food1 = getThePlayer().getPointsEaten();
   assertTrue(move.movePossible());
   move.apply();
   int food2 = food1 + 1;
   assertEquals(food2, getThePlayer().getPointsEaten());
   move.undo();
   assertEquals(food1, getThePlayer().getPointsEaten());
}
```

Changes and additions applied to Move.java:

```
  /**
 * The cell the mover comes from.
 */
private Cell from = null;

/**
 * The guest who is possibly at the destination.
 */
private Guest toGuest = null;

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.from = fromGuest.getLocation();
    this.to = toCell;
    this.toGuest = toCell.getInhabitant();
    assert moveInvariant() : "Move invariant invalid";
}

/**
 * Obtain the guest at the destination of the move or null if no such guest.
 *
 * @return The guest at the destination
 */
protected Guest getGuestAtDestination() {
   return toGuest;
}

/**
 * Undo this move.
 */
protected void undo() {
   assert initialized();
   assert moveDone();
   mover.deoccupy();
    mover.occupy(from);
    if (getGuestAtDestination() != null) {
           getGuestAtDestination().occupy(to);
    }
    //old cell should revert to original occupant
    assert mover.getLocation().getX() == from.getX();
    assert mover.getLocation().getY() == from.getY();
    assert getGuestAtDestination() == to.getInhabitant();
```

```
         assert initialized();
    }
```

New test added to MonsterMoveTest.java

```
        @Test
        /**
         * Test a move by a monster to an empty cell
         * the monster should move to the cell, undo the move
         * the monster should have returned to its original cell
         */
        public void testUndoMonsterMove() {
                Cell emptyCell = getEmptyCell();
                aMonsterMove = createMove(emptyCell);
                assertTrue(aMonsterMove.movePossible());
                Cell original = aMonsterMove.getMonster().getLocation();
                assertNotSame(original, emptyCell);
                aMonsterMove.apply();
                assertEquals(emptyCell, aMonsterMove.getMonster().getLocation());
                aMonsterMove.undo();
                assertEquals(original, aMonsterMove.getMonster().getLocation());
        }
```

Changes applied to PlayerMove.java:

```
    @Override
    protected void undo() {
        assert invariant();
         assert moveDone();
         super.undo();
         if (getGuestAtDestination() != null
                     && getGuestAtDestination().guestType() == Guest.FOOD_TYPE) {
                Food eaten = (Food) getGuestAtDestination();
                getPlayer().eat(-eaten.getPoints());
         }
         assert invariant();
    }
```

Exercise 53

We've added the following JUnit test to GameTest:

```
    /**
     * Test if the underlying stack in game that tracks moves is pushing
     * and popping moves in the correct order
     */
    @Test
    public void testUndoStack() {
        PlayerMove move1 = new PlayerMove(getThePlayer(), getFoodCell());
        PlayerMove move2 = new PlayerMove(getThePlayer(), getEmptyCell());
        getTheGame().persistMove(move1);
        getTheGame().persistMove(move2);
        PlayerMove performed = (PlayerMove) getTheGame().getMostRecentMove();
        assertEquals(performed, move2);
        assertNotSame(performed, move1);
        performed = (PlayerMove) getTheGame().getMostRecentMove();
        assertEquals(performed, move1);
    }
```

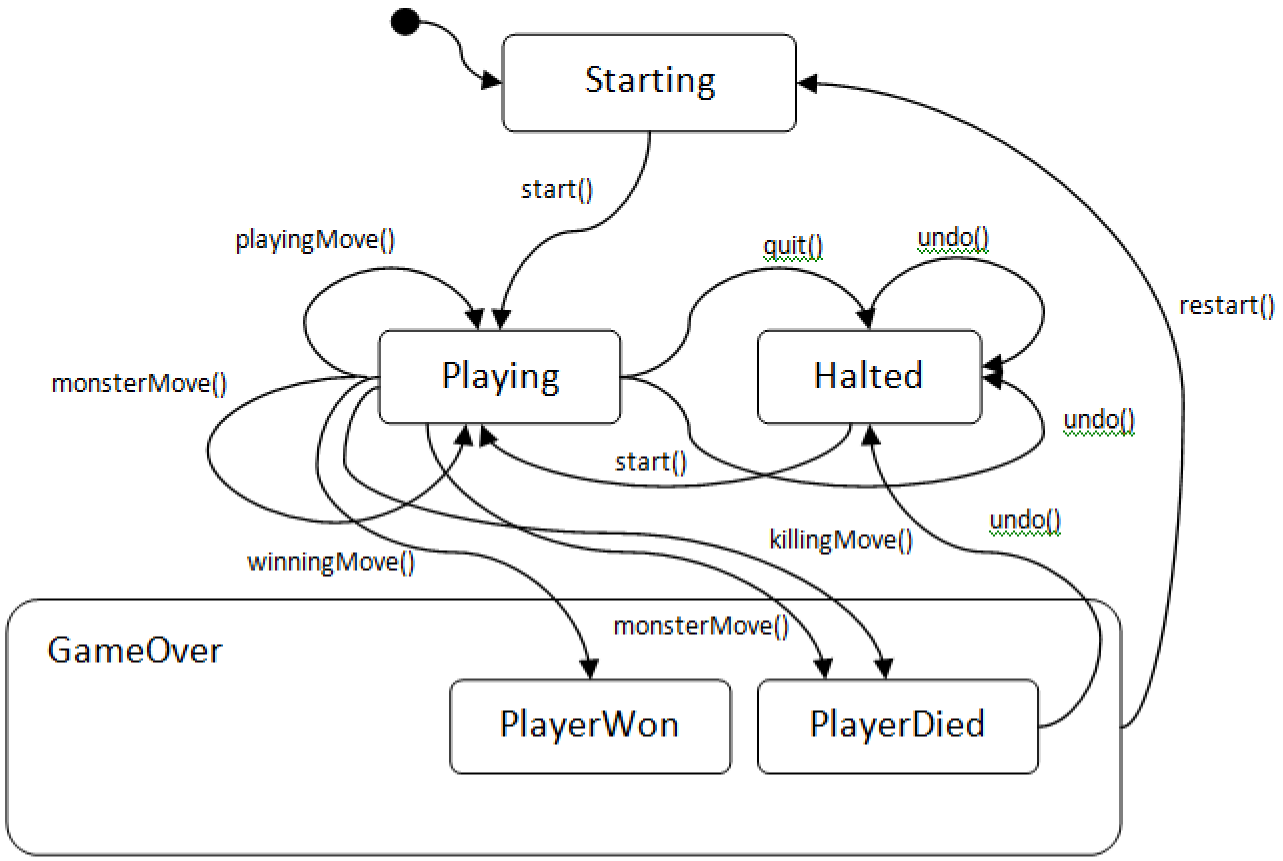
And it passes with the following changes applied to Game:

```
    import java.util.Stack;

    /**
     * The stack of all moves done in the game.
     */
    private Stack<Move> theStack = null;




  /**
   * Set the fields of the game to their initial values.     *
   * @throws GameLoadException if the game can't be loaded
   *         (in which case default values are used).
   */
  void initialize() throws GameLoadException {
    theStack = new Stack<Move>();
     if (theMap == null) {
         try {
             theMap = (new GameLoader()).obtainMap();
         } catch (GameLoadException gle) {
             // switch to default world map
             // (which should always load correctly).
             theMap = GameLoader.DEFAULT_WORLD_MAP;
             loadWorld(theMap);
             // inform outside world of switch to new map.
             throw gle;
         }
     }
     loadWorld(theMap);
     assert invariant();
  }

  /**
   * Check whether all relevant fields have been initialized.
   *
   * @return true iff initialization has completed successfully.
   */
  public boolean initialized() {
      return theBoard != null
          && thePlayer != null
          && theStack != null
          && monsters != null
          && totalPoints >= 0;
  }

  /**
   * Every move made in the game should be pushed onto the stack.
   * @param m The move to save
   */
  protected void persistMove(Move m) {
    assert invariant();
    assert m != null;
    theStack.push(m);
    assert invariant();
  }
```

```
    /**
     * The most recent move applied can be requested via this method.
     * pre-condition: we have at least one recent move saved
     * @return the most recent move applied
     */
    protected Move getMostRecentMove() {
        assert invariant();
        assert !theStack.isEmpty();
        return theStack.pop();
    }

    /**
     * Have there been any moves done by either a monster or the player?
     * @return true iff moves have been performed by some MovingGuest
     */
    protected boolean hasMoves() {
        assert invariant();
        return !theStack.isEmpty();
    }
```

Exercise 54

We can enter the Halted state from either the Playing or PlayerDied state by pressing the undo button. In the Halted state by pressing undo we remain in the Halted state. Therefore our UML state diagram previously given changes to the following.



We've added the following tests to EngineTest and most of them fail due to no existing implementation of undo functionality in Engine.

```
/**
 * Test undo simple player move
 */
@Test
public void testUndoPlayerMove() {
   theEngine.start();
   assertTrue(theEngine.inPlayingState());
   int x = getThePlayer().getLocation().getX();
   int y = getThePlayer().getLocation().getY();
   theEngine.movePlayer(1, 1); //move to empty cell next to me
   assertEquals(x+1, getThePlayer().getLocation().getX());
   assertEquals(y+1, getThePlayer().getLocation().getY());
   theEngine.undo(); //undo last move
   assertEquals(x, getThePlayer().getLocation().getX());
   assertEquals(y, getThePlayer().getLocation().getY());
   assertTrue(theEngine.inHaltedState());
}

/**
 * Test undoing after quitting
 */
@Test
public void testUndoAfterQuit() {
   theEngine.start();
   assertTrue(theEngine.inPlayingState());
   theEngine.quit();
   assertTrue(theEngine.inHaltedState());
   theEngine.undo();
   assertTrue(theEngine.inHaltedState());
}

/**
 * Test undo after player dies by meeting monster
 */
@Test
public void testUndoDieMove() {
   theEngine.start();
   assertTrue(theEngine.inPlayingState());
   theEngine.movePlayer(0,1);
   assertFalse(getThePlayer().living());
   assertTrue(theEngine.inGameOverState());
   assertTrue(getTheGame().playerDied());
   theEngine.undo();
   assertTrue(getThePlayer().living());
   assertFalse(getTheGame().playerDied());
   assertTrue(theEngine.inHaltedState());
   assertTrue(theEngine.invariant());
}

/**
 * Test undo after monster kills player
 */
@Test
public void testUndoAfterMonsterDie() {
   theEngine.start();
   assertTrue(theEngine.inPlayingState());
   theEngine.moveMonster(getTheMonster(), 0, -1);
   assertFalse(getThePlayer().living());
   assertTrue(theEngine.inGameOverState());
   assertTrue(getTheGame().playerDied());
   theEngine.undo();
   assertTrue(theEngine.inHaltedState());
   assertFalse(getTheGame().playerDied());
```

```
        assertTrue(getThePlayer().living());
        assertTrue(theEngine.invariant());
    }

    /**
     * Test undo when no moves have been done
     */
    @Test
    public void testUndoNotAllowed() {
        theEngine.start();
        assertTrue(theEngine.inPlayingState());
        theEngine.undo();
        assertFalse(theEngine.inHaltedState()); //no moves done yet, so cant undo anything
    }

    /**
     * Test many monster moves undone
     */
    @Test
    public void testUndoManyMonsterMoves() {
        theEngine.start();
        assertTrue(theEngine.inPlayingState());
        int x = getThePlayer().getLocation().getX();
        int y = getThePlayer().getLocation().getY();
        theEngine.movePlayer(1, -1);
        assertEquals(x+1, getThePlayer().getLocation().getX());
        assertEquals(y-1, getThePlayer().getLocation().getY());
        int mx = getTheMonster().getLocation().getX();
        int my = getTheMonster().getLocation().getY();
        theEngine.moveMonster(getTheMonster(), 1, 0);
        theEngine.moveMonster(getTheMonster(), 0, -1);
        theEngine.moveMonster(getTheMonster(), -1, 0);
        theEngine.undo();
        assertTrue(theEngine.inHaltedState());
        assertTrue(getThePlayer().living());
        assertEquals(x, getThePlayer().getLocation().getX());
        assertEquals(y, getThePlayer().getLocation().getY());
        assertEquals(mx, getTheMonster().getLocation().getX());
        assertEquals(my, getTheMonster().getLocation().getY());
    }

    /**
     * Test undo score
     */
    @Test
    public void testUndoScoring() {
        theEngine.start();
        assertTrue(theEngine.inPlayingState());
        int score = getThePlayer().getPointsEaten();
        theEngine.movePlayer(-1, 0); //eat some food
        assertEquals(score+1, getThePlayer().getPointsEaten());
        theEngine.undo();
        assertEquals(score, getThePlayer().getPointsEaten());
    }
```

We now need to add the implementation to Engine and fix the other classes in order to make these tests pass. The changes are as follows.

Changes added to Game.java:

```
    /**
     * We want to revive the player from his death.
     */
    protected void revive() {
       assert invariant();
       assert playerDied();
       assert gameOver();
       assert !playerWon();
       getPlayer().live();
       assert invariant();
       assert !playerDied();
       assert !gameOver();
    }

    /**
     * Actually apply the given move, if it is possible.
     * @param move The move to be made.
     */
    private void applyMove(Move move) {
        assert move != null;
        assert invariant();
        assert !gameOver();
        if (move.movePossible()) {
            move.apply();
            persistMove(move);
            assert move.moveDone();
            assert !playerDied() : "move possible => not killed";
        } else {
            if (move.playerDies()) {
                assert !playerWon() : "you can't win by dying";
                getPlayer().die();
                assert playerDied();
            }
        }
        assert invariant();
    }
```

Changes added to Engine.java:

```
    /**
     * Undo the last step the player made. If the player died, revive.
     * Game enters halted state
     */
    public synchronized void undo() {
       assert invariant();
       if (inPlayingState() || inHaltedState()) { //just undo all moves until undoing a player move
              if (theGame.hasMoves() && !inHaltedState()) {
                     halted = true;
              }

              while (theGame.hasMoves()) {
                     Move m = theGame.getMostRecentMove();
                     m.undo();

                     if (m instanceof PlayerMove) {
                            break;
                     }
              }
       }
       else if (inDiedState()) { //revive the player
              theGame.revive();
              halted = true;
       }
       notifyViewers();
```

```
        assert invariant();
    }
```

Changes added to Player.java:

```
    /**
     * The player has been revived by an undo.
     */
    protected void live() {
        assert playerInvariant();
        assert !living();
        alive = true;
        assert playerInvariant();
        assert living();
    }
```

With these added changes all tests now pass.

Exercise 55

A new button has been added to the window and clicking on it activates the underlying undo functionality. Changes added to the code include the following.

In Pacman.java:

```
    /**
     * Undo the last player moves and monster moves leading up to that move.
     */
    public void undo() {
       assert invariant();
       monsterTicker.stop();
       theEngine.undo();
       theAnimator.stop();
       assert invariant();
    }
```

In PacmanUI.java:

```
        JButton undoButton = new JButton("Undo");
        undoButton.addActionListener(new ActionListener() {
               public void actionPerformed(ActionEvent e) {
                      getController().undo();
                      requestFocusInWindow();
               }
        });

        JPanel buttonPanel = new JPanel();
        buttonPanel.add(startButton);
        buttonPanel.add(undoButton);
        buttonPanel.add(quitButton);
        buttonPanel.add(exitButton);
```

Exercise 56

The implementation of the undo functionality was quite smooth indicating a well-thought out overall design. It's nice to have a game to work with for an assignment in a class about software testing but one could argue that the level of object-orientedness of JPacman could be considered a bit over-engineered. Especially the double-dispatching used for moves can be problematic to wrap your head around. Adding a move could mean having all possible classes that apply to the move needing implementations of how to handle the move. Maybe some other methodology could be used here to better effect.