Software Testing

Lab Assignments

March 24, 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 1-6 (including report)
Thomas	Exercises 7-11 (including report)

Table 1: Division of tasks amongst team members.

Verdeel taken & vul taakverdeling tabel aan.

3 Assignment 04 – State Machines

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 State Machines

"*Now that we have monsters that can move, we should adjust the Engine so that the monsters are indeed activated. We first create a regression test suite for the engine as it exists at the moment (i.e., without monster moves), and then extend it.*"

3.1.1 Exercise 01

> **Question:** Analyze the description of the state machine from Engine class as discussed in the design document `doc/pacman-design.txt` . Create a UML state diagram from this description. Note that the actual implementation of the state machine can be inspected in Engine class itself.

We added the JPacman design decisions to the end of the report for easy access, and completeness. See Section 3.3. Note design decision section ***D3. Engine State Diagram***. The **D3.x** design decisions explicitly specify a state machine and consequently a state diagram for *Engine* that is implicitly described by the use cases. We will derive the UML diagram from these descriptions. But we will make the distinction between statements about *Engine* design and *Game* design; **we prioritize the *Engine* class design** and mostly ignore the *Game* class design decisions that are mixed in the **D3.x** design decisions.

See Figure 1 for the *Engine* state diagram, and Table 2 for the short descriptions of each state and event.

Decision **D3.3.** *Move Player* states that "*For the state machine, we distinguish moves that keep the game in the playing state, and moves that cause a transition to one of the game over states.*" The subsequent sections **D3.3.1**, **D3.3.2** and **D3.3.3** break down a generic player move into three distinct types: *Keep on playing move*, *Winning move* and *Killing move*.

We will design the state machine diagram to match the *Engine* implementation as close as possible. This hopefully reduces the possibility of faults due to a mismatch between state machine and source code implementation. This should also make it easier to discover sneak paths in later exercises.

We specify six states and four events, which closely matches the actual *Engine* implementation. One difference is that the state machine events *PlayerMove* and *MonsterMove* are named differently in *Engine*, respectively `Engine.movePlayer(int, int)` and `Engine.moveMonster(int, int)` . The two other events map exactly to the *Engine* functions `Engine.start()` and `Engine.quit()` . The state machine states also map to existing *Engine* functions.

Representing our state machine using these four events requires using conditional **(guarded) transitions** to determine the nature of every move, i.e. is a *Player Move* or *Monster Move* of the type "killing", "winning" or "keep playing" move.

Add Output Actions to the table??? See [1] p94 (the page nr

States (6)	
Starting	(**D3.1, D3.2**) The initial state in which the game is waiting to be started. See `Engine.inStartingState()`.
Playing	(**D3.1, D3.3**) A state in which the player and the monsters are making moves. See `Engine.inPlayingState()`.
Halted	(**D3.1**) A state in which move making is temporarily suspended. See `Engine.inHaltedState()`.
Game Over	(**D3.1**) A super state containing two substates, Player Won and Player Died, representing the two ways in which the game can end. See `Engine.inGameOverState()`.
Player Win	(**D3.1**) A state representing the player won the game, so the game ended. See `Engine.inWonState()`.
Player Died	(**D3.1**) A state representing the player lost the game, so the game ended. See `Engine.inDiedState()`.
Events (5)	
Start	(**D3.2**) When in the *Starting* state and the game is initialized, then fire the *Start* event to transition to the *Playing state*. When in the *Halted* state, return to the *Playing* state and resume the game. Also note **D3.6 *Restart***, which states "*The game over state is a super state for either a player won or a player died state. The only action that can done is to restart the game, which requires an initialization of the underlying data structures.*". See `Engine.start()`, which also implements the restart functionality.
PlayerMove	(**D3.3, D3.3.x**) As long as the engine is in the playing state, the player can make moves. For the state machine, we distinguish moves that keep the game in the playing state (**D3.3.1** *KeepOnPlayingMove*), and moves that cause a transition to one of the game over states (**D3.3.2** *WinningMove* and **D3.3.3** *KillingMove*). See `Engine.movePlayer()`.
MonsterMove	(**D3.4**) 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. Consequently, a *MonsterMove* can represent a *KeepOnPlayingMove* or a *KillingMove*, but can not represent a *WinningMove*. See `Engine.moveMonster()`.
Quit	(**D3.5**) From the *Playing* state, the *Quit* event suspends the game and brings the engine into the *Halted* state. A *Start* event from the *Halted* state re-activates the game and brings the game back into the *Playing* state. See `Engine.quit()`.

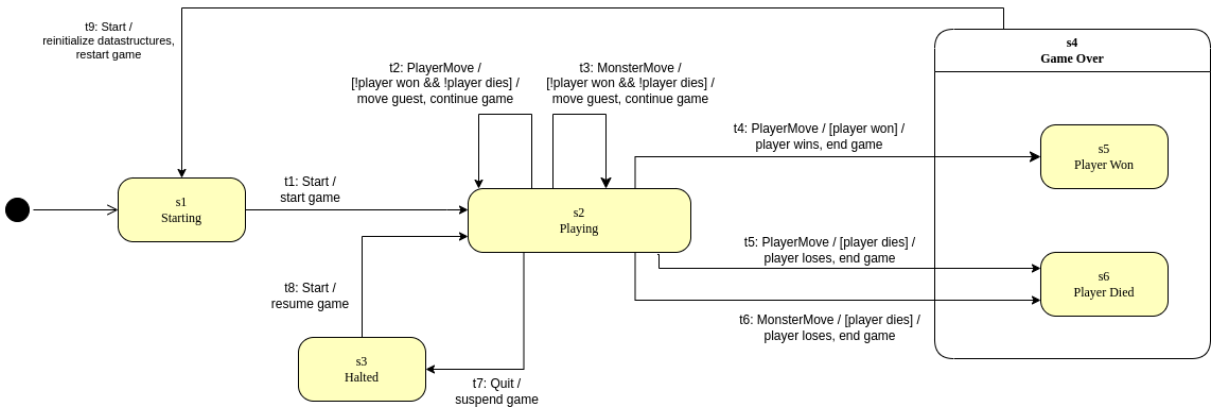


Figure 1: The state machine state diagram for the JPacman *Engine*.

3.1.2 Exercise 02

Question: Make a state-event transition table (see Forgács [1] Table 6.1 (see Figure 5), given Figure 6.2 (see Figure 6) as the state diagram, Table 6.3 (see Figure 7) is the resulting state-event table).

See Table 3 for the state-event table of the state machine in Figure 1. Every empty cell in the table reflects a corresponding triplet *(Start State, Input Event, End State)* that is an **undefined** transition w.r.t. the state machine.

A state-event table makes it clear which are the outgoing transitions of a given start state.

For brevity, let us abbreviate the transition conditions (guards):

- WIN : `player won`
- DIE : `player dies`
- CTU : `!player won && !player dies` (CTU stands for "continue")

	Input Event			
	Start	*PlayerMove*	*MonsterMove*	*Quit*
Start State	**End State / [Guard] / Action(s)**			
Starting	*Playing* / start game			
Playing		*Playing* / [CTU] / move guest, continue game *Player Won* / [WIN] / player wins, end game *Player Died* / [DIE] / player loses, end game	*Playing* / [CTU] / move guest, continue game *Player Died* / [DIE] / player loses, end game	*Halted* / suspend game
Halted	*Playing* / resume game			
Game Over	Starting / reinitialize datastructures, restart game			
Player Won				
Player Died				

Table 3: The state-event transition table for the state diagram in Figure 1.

3.1.3 Exercise 03

Question: Make a state-state transition table (see Forgács [1] Table 6.2, see Figure 5).

See Table 4 for the state-state table of the state machine in Figure 1. Every empty cell in the table reflects a corresponding triplet *(Start State, Input Event, End State)* that is an **undefined** transition w.r.t. the state machine.

A state-state table makes it easy to verify if two states are connected by any transitions at all, but finding all transitions that are triggered by a given event is more difficult.

For brevity, let us abbreviate the transition conditions (guards):

- WIN : `player won`
- DIE : `player dies`
- CTU : `!player won && !player dies` (CTU stands for "continue")

	End State					
	Starting	*Playing*	*Halted*	*Game Over*	*Player Won*	*Player Died*
Start State	**Event / [Guard] / Action(s)**					
Starting		*Start* / start game				
Playing		*Player-Move* / [CTU] / move guest, continue game *Monster Move* / [CTU] / move guest, continue game	*Quit* / suspend game		*Player-Move* / [WIN] / player wins, end game	*Player-Move* / [DIE] / player loses, end game *Monster-Move* / [DIE] / player loses, end game
Halted		*Start* / resume game				
Game Over	Start / reinitialize datastructures, restart game					
Player Won						
Player Died						

Table 4: The state-state transition table for the state diagram in Figure 1.

3.1.4 Exercise 04

Question: Create and describe a series of test case specifications that achieve transition coverage. Implement the test cases in JUnit in the *EngineTest* class.

See Codeblock 1 for the **transition coverage tests cases in java**. Note that the third test case, currently fails, since the integration of *MonsterMove* with *Engine* has not yet been implemented.

The *Engine* and JPacman program in general are stateful. We need to take this into account when defining the test cases, since the *Board* configuration of *Player*, *Monster* and other objects influences which test (move combinations) are feasible in practice. Note that we use the following map below in the test cases, where 0 is an empty cell,W is a *Wall*, F is *Food*, P is the *Player* and M is a *Monster*. To win, the player must move left once and then down once. To lose, the player must immediately move down, or the *Monster* adjacent to the *Player* must immediately move up.

```
0W0
FP0  <-- Player is at (x,y) = (1,1)
FM0  <-- Monster is at (x,y) = (1,2)
0WM
```

As per the textbook [1], "*State transition testing is measured mainly for **valid** transitions. There are many different test selection criteria regarding the transition-based method.*" The textbook advocates for two possible criteria that it views as a good balance between reliability and computational cost: the *all-transition-state* criterium and the *all-transition–transition* criterium.

However, to achieve basic transition coverage as the exercise asks simply satisfying the ***all-transitions*** criterion is enough; **we will specify a test suite that traverses every transition at least once**. Note that "*... satisfying the all-transition-state criterion also satisfies the all-transitions criterion.*" A more robust test suite could try satisfying the *all-transition-state* criterion and still solve this exercise.

We enumerate several paths through the state machine, so that when combined they provide 100% transition coverage. The **first path** will cover the majority of transitions.

$$\text{Starting} \xrightarrow[\text{Start}]{(1)} \text{Playing} \xrightarrow[\text{Quit}]{(2)} \text{Halted} \xrightarrow[\text{Start}]{(2)} \text{Playing} \xrightarrow[\text{PlayerMove}]{(3)} \text{Playing} \xrightarrow[\text{MonsterMove}]{(3)} \text{Playing} \xrightarrow[\text{PlayerMove}]{(4)} \text{Player Won} \xrightarrow[\text{Start}]{(7)} \text{Starting}$$

We need **two more paths** to cover the all transitions to the *Player Died* state, since it has no outgoing transitions.

$$\text{Starting} \xrightarrow{\text{Start}} \text{Playing} \xrightarrow[\text{PlayerMove}]{(5)} \text{Player Died}$$

$$\text{Starting} \xrightarrow{\text{Start}} \text{Playing} \xrightarrow[\text{MonsterMove}]{(6)} \text{Player Died}$$

We mark events in the paths to indicate when specific transitions are taken. This is to show the transition coverage explicitly. The event(s) marked by

- (1) covers the initial *Starting* to *Playing* transition, t_1.
- (2) cover the *Halted* cycle, transitions t_7, t_8.
- (3) cover the *Playing* self loops, transitions t_2, t_3.
- (4) covers the player winning the game, transition t_4.
- (5) covers the *Player* moving onto a *Monster*, transition t_5.
- (6) covers the *Monster* moving onto a *Player*, transition t_6.
- (7) covers restarting the game, transition t_9.

This covers all transitions specified in the state machine in **Figure 1**. Again, see **Codeblock 1** for the java test cases. See the note at the start of this exercise for more info.

```java
/** The long path that covers most transitions,
 * and ends in the player reaching the winning end state.
 */
@Test public void testLongAndWinning() {
    /* Starting --(Start)-> Playing */
    assertTrue(theEngine.inStartingState());
    theEngine.start();
    assertFalse(theEngine.inStartingState());
    assertTrue(theEngine.inPlayingState());

    /* Playing --(Quit)-> Halted */
    assertTrue(theEngine.inPlayingState());
    theEngine.quit();
    assertFalse(theEngine.inPlayingState());
    assertTrue(theEngine.inHaltedState());

    /* Halted --(Start)-> Playing */
    assertTrue(theEngine.inHaltedState());
    theEngine.start();
    assertFalse(theEngine.inHaltedState());
    assertTrue(theEngine.inPlayingState());

    /* Playing --(PlayerMove)-> Playing */
    assertTrue(theEngine.inPlayingState());
    theEngine.movePlayer(-1, 0); // dx = -1  means  LEFT
    assertTrue(theEngine.inPlayingState());

    /* Playing --(MonsterMove)-> Playing */
    assertTrue(theEngine.inPlayingState());
    theEngine.moveMonster(theMonster, 0, 1);
    assertTrue(theEngine.inPlayingState());

    /* Playing --(PlayerMove)-> Player Won */
    assertTrue(theEngine.inPlayingState());
    theEngine.movePlayer(0, 1); // dy = 1  means  DOWN
    assertFalse(theEngine.inPlayingState());
    assertTrue(theEngine.inWonState());
    assertTrue(theEngine.inGameOverState());

    /* Player Won --(Start)-> Starting */
    assertTrue(theEngine.inWonState());
    theEngine.start();  // Restart the game.
    assertFalse(theEngine.inWonState());
    assertTrue(theEngine.inStartingState());
}

/** A short path that ends in the player dying by moving onto a monster. */
@Test public void testPlayerDiesToMonster() {
    /* Starting --(Start)-> Playing */
    assertTrue(theEngine.inStartingState());
    theEngine.start();
    assertFalse(theEngine.inStartingState());
    assertTrue(theEngine.inPlayingState());

    /* Playing --(PlayerMove)-> Player Died */
    assertTrue(theEngine.inPlayingState());
    // Move the player into the monster.
    theEngine.movePlayer(0, 1); // dy = 1  means  DOWN
    assertFalse(theEngine.inPlayingState());
    assertTrue(theEngine.inDiedState());
    assertTrue(theEngine.inGameOverState());
}

/** A short path that ends in a monster killing the player by moving
 * onto them.
 */
@Test public void testMonsterKillsPlayer() {
    /* Starting --(Start)-> Playing */
    assertTrue(theEngine.inStartingState());
    theEngine.start();
    assertFalse(theEngine.inStartingState());
    assertTrue(theEngine.inPlayingState());

    /* Playing --(MonsterMove)-> Player Died */
    assertTrue(theEngine.inPlayingState());
    // Move the monster into the player.
    theEngine.moveMonster(theMonster, 0, -1); // dy = -1  means  UP
    assertFalse(theEngine.inPlayingState());
    assertTrue(theEngine.inDiedState());
    assertTrue(theEngine.inGameOverState());
}
```

Codeblock 1: The transition coverage test cases for *Engine*.

3.1.5 Exercise 05

Question: Finite state machine specifications are often incomplete. Omitted (state, stimulus) pairs are usually ignored. Identify test case specifications for all omitted pairs, and verify that these are indeed ignored, i.e., that the state machine does not secretly implement any sneak path. Show which events are allowed or not allowed in which state. Create a sneak path test suite in JUnit to cover these scenarios.

All (state, transition) pairs omitted from the state machine in Figure 1, which corresponds to all empty cells in the state-event table Table 3, are supposed to be ignored. We simplify Table 3 by replacing all transitions by a simple "x" when a given state-event pair is allowed, and leaving the cell blank when it is not allowed. See Table 5.

We keep the test cases very basic. We only test the state-event pairs without really caring about the guard conditions, since we do not have direct access to many private class members to for example force a player losing state without going through the *Engine*'s public interface.

See Codeblock 2 for the sneak path test cases. We implement one test case each for the states *Starting*, *Playing*, *Halted*, *Player Won* and *Player Died*. The state *Game Over* is a super state that is always reached when either of its sub states is reached. The general approach we took was to trigger events until we reach the required state of the test case, then then trigger all events that should be ignored and finally validate that those events are indeed ignored.

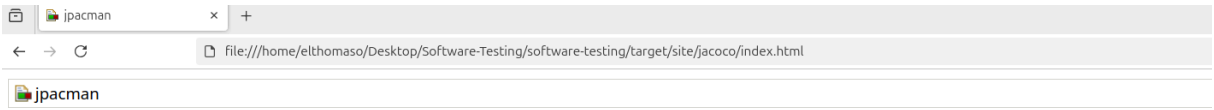
	Input Event			
	Start	*PlayerMove*	*MonsterMove*	*Quit*
Start State	**End State / [Guard] / Action(s)**			
Starting	x			
Playing		x	x	x
Halted	x			
Game Over	x			
Player Won	x			
Player Died	x			

Table 5: The allowed state-event pairs in the state-event table for the state diagram in Figure 1. If a cell (state, event) contains an "x", then that pair is allowed (is **not** ignored). All other pairs should be ignored.

3.1.6 Exercise 06

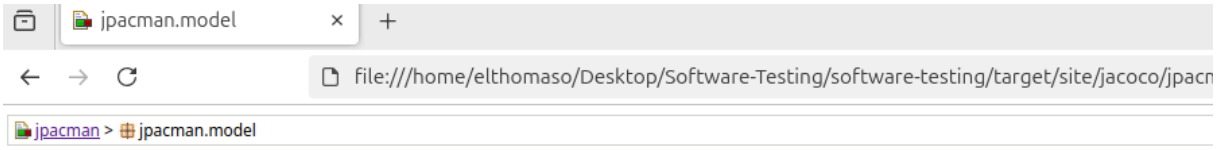
Question: Analyze the coverage of your test suite, report and fix any missing test cases. Extend the models, implementation, and test suite to cater for monster moves as well. Then re-analyze the coverage.

All the missing coverage is due to there being nog tests that falsify the assertion conditions of the tested methods in engine. All lines are covered.



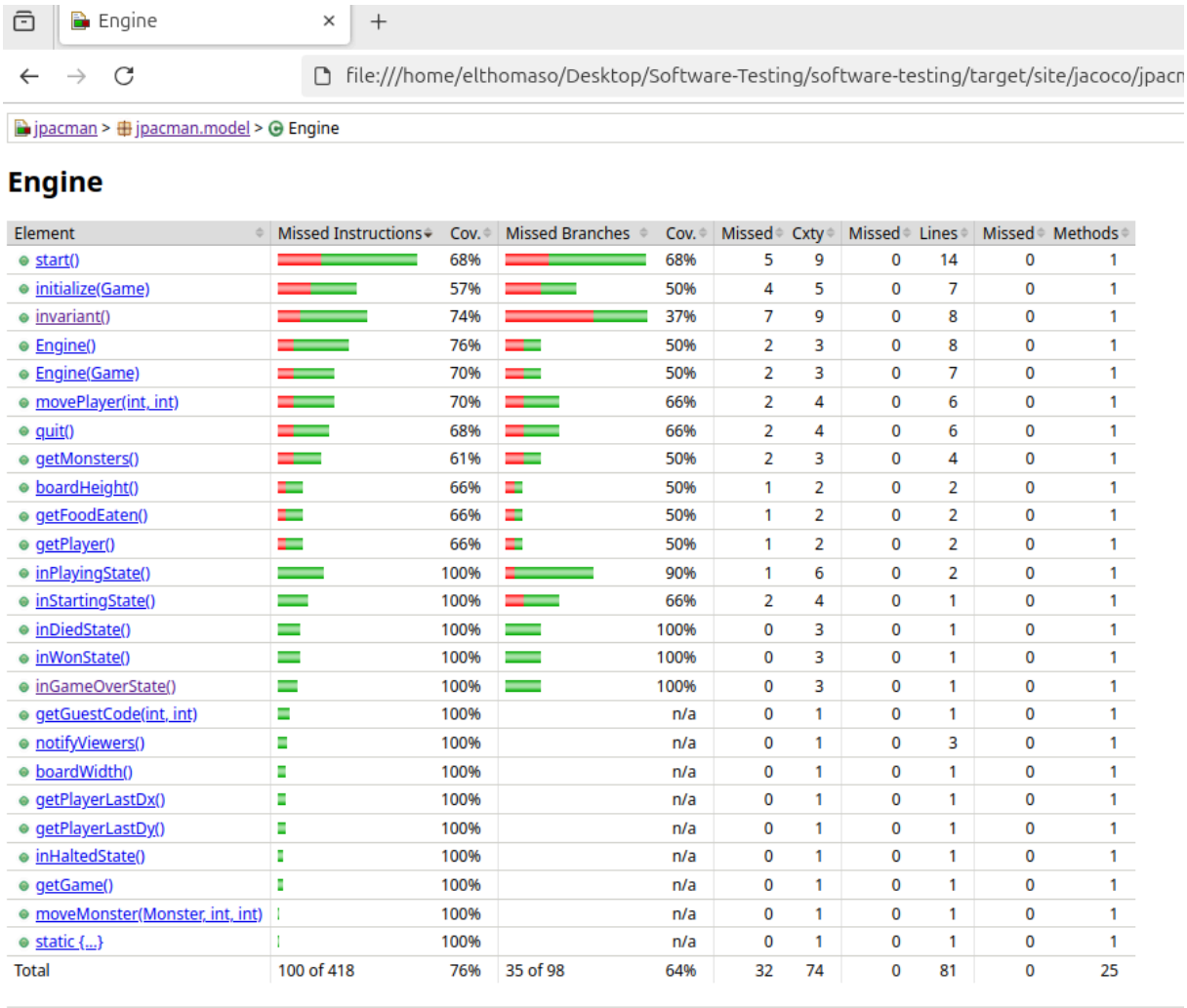
jpacman

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
jpacman.model		73%		59%	220	420	1	490	0	140	0	13
jpacman.controller		80%		56%	87	179	40	353	9	77	0	14
Total	1,048 of 4,369	76%	309 of 753	58%	307	599	41	843	9	217	0	27



jpacman.model

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
Game		77%		61%	41	80	1	102	0	25	0	1
Engine		76%		64%	32	74	0	81	0	25	0	1
Move		74%		63%	32	58	0	72	0	14	0	1
Board		73%		67%	18	38	0	41	0	10	0	1
Cell		75%		60%	22	42	0	43	0	14	0	1
Player		66%		50%	16	29	0	35	0	13	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	739 of 2,754	73%	223 of 556	59%	220	420	1	490	0	140	0	13



3.2 Undo Button

"*We will extend JPacMan with a working undo button. By pressing undo while playing or after just having died, the user can undo his last move, as well as any monster moves that occurred after it, after which the game will enter the halted state, from where it can press undo again. Since this is a testing course, your primary focus should be on the way in which you test this extension. In case you are running out of time, you are advised to specify test cases without working implementation, instead of an implementation without test cases.*"

3.2.1 Exercise 07

Question: Using the style of `doc/pacman-requirements.txt` document, provide a use case capturing the undo requirement. Think of possible situations that can occur, and describe the desired behavior. Summarize your design decisions in the style of the `doc/pacman-design.txt` document.

Use Case Name: Undo Last Move

Actors: Player, JPacMan Game System

Description: The player can undo their last move, including any monster moves that occured afterward. Upon pressing the undo button, the game reverts to the previous state before the last player move was made

Preconditions:

- The game is in a playable state or a halted state (after the player's death).
- The player has made at least one move.

Main flow:

1. The player presses the undo button.
2. The system checks the stack of previous moves stored in the `Game` class.
3. The system reverts the player's last move and any subsequent monster moves.
4. The game enters the halted state.
5. The player may press the undo again or continue playing

Alternative Flow:

- If the player has not made any moves, the undo button is disabled or ignored.

Post-conditions:

- The game is reverted to the previous valid state before the last player move.
- The game is halted

Design Decisions:

- Use of a stack in the `Game` class to track moves.
- The `Move` class should implement an `undo()` method to facilitate move reversal.
- The undo button is integrated into the user interface.

3.2.2 Exercise 08

> **Question:** *Move* class implements the Command pattern, which was invented to facilitate undo. Write test cases for a `Move.undo` method, and extend *Move* class accordingly.

There are several test cases implemented or thought about.

Test Case 1: Undo Single Player Move

- In this test case, we set up the environment such that we have a `Move` object representing a single player move. Then we call the `Move.undo()` method. The expected outcome is that the move is reversed successfully, restoring the previous state.
- The test was implemented and ran successfully, important to note is that this was implemented in the `PlayerMoveTest` since this test required a player.

Test Case 2: Undo Eating fruit Move

- For this case, a player moves into a fruit object and therefor eats it, gaining points. When we call the `Move.undo()` method, the expected outcome is that the fruit reappears and that a food-point is deducted of the score.
- This test was also implemented in the `PlayerMoveTest.java` testclass and ran without errors.

Test Case 3: Undo without Moves

- We set up the environment such that no moves are made. When we call the `Move.undo()`, the expected outcome is that nothing happens
- This case was not implemented because in our `undo()` method for this exercise, there is an assertion that `previousLocation` cannot be `null`. If we want to undo a move, without a move have taken place, then this assertion will fail.

Test Case 4: Undo multiple Moves

- In this environment we have 2 moves that are applied, then we undo both moves and expect to be back at the begin state.

Test Case 5: Undo a deadly move **Test Case 6:** Ensure that the game state is set to *halted* after an undo operation. **Test Case 7:** Undo on an already halted game state **Test Case 8:** Ensure that undo operations work correctly when the game is restarted or reset.

Changes in the Move Class

In this exercise it was not necessary yet to use a stack to keep track of previous game states. That is an adaption for the next exercise. For this exercise however, it was important to knew the previous game

state since we needed to implement the `Move.undo()` method.
To keep track, two variables are introduced in the `Move` class: `previousLocation` and `previousGuest`. The first one stores the current location of the guest that's about to be moved. The second one checks if the space where we are moving to is empty or not. If it is not empty it, it stores the guest of that space. This will later be removed or adapted when we change to using a stack for the undo method.
Another important change is to the `Move.apply()` method. The actual moving happens in this method, meaning that all the information needed to undo a move can be retrieved here. When a move is applied, the two variables mentioned above are updated.

3.2.3 Exercise 09

> **Question:** It is probably easiest to keep track of a stack of moves in *Game* class. Write test cases that ensure proper pushing of moves made, and popping of moves that must be undone, and provide the underlying implementation.

The implementation of the stack in the `Game` class was done as follows. We used the Stack class from Java itself and made a variable `moveStack` that initialized when the game gets initialized. Then, when in the `Game` class the method `applyMove` gets called, the move (if possible) will be pushed to the stack. When a move needs to be undone, we pop the latest move from the stack and use the implemented `Move.undo()` from previous exercise. Important to note is that this is not specific to a player-move. Meaning that every "undo" is whatever move was made last: a player move or a monster move.
Another important thing to mention is that we have not yet implemented the "halt" functionality after the undo method is called. This shall be fixed in a later session.

Test cases for this exercise are the following:
Test Case 1: Popping on an empty stack

- When no moves have been made, popping the stack should not be allowed

Test Case 2: Pushing and Popping a player move to an empty cell
Test Case 3: Pushing and Popping a player move to a food cell
Test Case 4: Pushing and Popping a monster move
Test Case 5: pushing and popping more than 1 move

3.2.4 Exercise 10

> **Question:** Extend the JPacMan state machine diagram, the corresponding test cases, and *Engine* implementation to cater for the new undo event. Extend the JPacMan user interface with a new undo button which activates the corresponding functionality in *Engine* class.

3.2.5 Exercise 11

> **Question:** Given your experience with adding an undo button to JPacMan, reflect on the design of JPacMan. Describe refactorings that you found necessary, or suggest improvements to the design and code base.

It was quite an experience. When we first started implementing the Undo method in exercise 9, we wanted to store the last location and guest that was moved. We needed to think: is the undo move for a player the same as an undo move for a monster? If not, what is the difference? Well, the difference is in how they handle moving into a food cell. When a player bumps into a food cell, it eats the fruit and gets a point, however when a monster wants to occupy a cell, the move gets denied. We needed to take into account that if we undo a player eating a food token, the points gotten for that food token needed to be subtracted of that players score.
It also took us some time figuring out how exactly food and score was processed in the move class, since

there isn't just a line that says `score = score + food_points` or something alike. It takes quite the route: when the `Game` calls `movePlayer()`, the `PlayerMove()` constructor is called in which the `Move.preComputeEffects()` method is being called. Here it checks wether the new cell is empty or not. If the cell is not empty, the general `TryMoveToGuest` method is called. Then the appropriate guest method is called upon. In the case of a player eating food, the method `food.meetplayer()` is called and there eventually the score gets updated.
A suggestion would be to put this train somewhere in the comments, so it's easier to follow the flow of certain functionality.

3.3 JPacman Design Decisions

Below, we attach the JPacman design documentation located at `doc/pacman-design.txt` for completeness.

Arie van Deursen, CWI & TUD, 2005-2008

D0. JPacman Design. This document contains the most important design decisions for the Pacman implementation. These decisions deal with the package structure, the key classes contained in each package, design patterns used (most notably MVC and Observer), and the design of the underlying state machine.

D1. Package Structure Pacman's architecture is setup according to the model-view controller architecture. The model contains non-gui classes for the key pacman elements, such as the board itself and cell guests such as the player, monsters, and food. The viewer contains a graphical representation of the board. The controller contains the top level gui element as well as a controller to initiate monster moves. Since in Java Swing the GUI controllers are integrated with the user interface, the actual implementation has merged the controller and viewer package into one package called "jpacman.controller". All model classes are contained in a package called "jpacman.model".

D2. Model Structure The model itself contains classes for the key concepts of a JPacman game. These include:

D2.1. The Board itself, which consists of a series of Cells.

D2.2. Guests that can inhabit a cell. Guests can be static such as Wall and Food, or a MovingGuest, for either the Player or a Monster.

D2.3. Moves themselves: since making a move is fairly complex (depending on the type of the mover and the occupier of the target cell), a separate abstraction is used distinguishing a PlayerMove and a MonsterMove. The corresponding decision table is implemented by simulating a double dispatch mechanism in Java. A Move object can figure out whether a move is possible at all, what the moves' effect will be (food that may be eaten, the player may die), and it can actually conduct the move. The Move class implements the Command design pattern (Gamma et al).

D2.4. The Game itself having associations with the actual Board, Monsters, and the Player object, and which keeps track of the total amount of food in the game.

D2.5. The top level Engine that keeps track of the internal state of the game. The engine also adopts the Subject role of the Observer design pattern and offers a Facade for inspecting the state of the Game.

D3. Engine State Diagram Implicit in the uses cases is a state machine indicating which user interaction is possible. This machine is implemented in a separate class, the Engine. (Strictly speaking, the Engine could be merged with the Game class. The tradeoff here is between separating the state machine from the rest of the functionality at the cost of having an extra class that has to do some additional forwarding.) The states and events that are allowed are discussed below.

D3.1. States The states distinguished include: **Starting: initial state in which the game is waiting to be started.** Playing: state in which the player and the monsters are making moves. **Halted: state in**

which move making is temporarily suspended. Game over: super state containing two substates, Player Won and Player Died, representing the two ways in which the game can end.

D3.2. Start The initial state is the Starting state. The start event causes a transition to the Playing state.

D3.3. Move Player As long as the engine is in the playing state, the player can make moves. A player can move to an immediately adjacent cell, corresponding to an up, down, left or right move. Moves to adjacent, empty cells are possible. Other moves are impossible and are ignored. For the state machine, we distinguish moves that keep the game in the playing state, and moves that cause a transition to one of the game over states.

D3.3.1. Keep on playing move If a move is possible, does not cause the player to die, and results in a number of points for the player less than the total amount of food, the move can be applied, and the machine remains in the Playing state.

D3.3.2. Winning move If a move is possible, does not cause the player to die, and results in a total number of points for the player equal to the total amount of food in the game, the machine makes a transition to the Player Won state.

D3.3.3. Killing move If a move 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.

D3.5. Suspension From the playing state, the quit event suspends the game and brings the engine into the halted state. A start event from the halted state re-activates the game and brings the game back into the playing state.

D3.6. Restart The game over state is a super state for either a player won or a player died state. The only action that can done is to restart the game, which requires an initialization of the underlying data structures.

D4. Graphical User Interface The GUI consists of the elements discussed below.

D4.1. The Board Viewer, which is a Java Swing JPanel used to represent all the cells on the board, with specific representations for the various types of inhabitants. For moving guests (the player or the monsters) special gif pictures are used, whereas for static guests (wall or food) colored squares are used. The board viewer implements the observer role, and is thus notified by the engine about changes occuring on the board. Images are obtained from a (stateful) image factory, which given a sequence number obtain the next animation image to be displayed.

D4.2. The Pacman User Interface, which is a Java Swing frame adding buttons and other interactive user elements to the board viewer. The corresponding class is capable of listening to mouse and keyboard events, and maps these to the appropriate calls to the model classes. The user interface also displays the status and the amount of food eaten – to do so the observer pattern is used again. The GUI also provides an exit button, which allows the user to exit the game at any time, independent of the state of the engine.

D4.3. Controllers: The main class of the application is the Pacman class, which launches the Pacman User Interface as well as an Engine and connects the two. Moreover it launches a MonsterController for generating monster moves. The concrete MonsterController provided presently is the Random Monster Mover, which at every tick picks a random monster and lets it make a random move. It also launches a separate Animator thread, which is used to trigger the next animation.

D5. Build Time Architecture. The directory structure of the pacman project follows the recommendations as used in standard maven projects. The distribution includes a build file required for ant 1.6.5, and project files as required by Eclipse 3.2, making it easy to extend pacman using any of these tools. Pacman makes use of the generics and attributes of Java 1.5. Its test suite is based on JUnit 4, but is compatible with older versions of JUnit. Test coverage is measured using the open source Emma coverage analyzer, as well as its Eclipse counter part eclemma.

Table 6.1 State–event notation for a state transition table

	$event_1$	...	$event_N$
start state	next state / action	...	next state / action
s_1	s_i / a_k	...	–
...	...	...	...
s_n	s_j / a_l	...	s_p / a_q

,

Table 6.2 State–state notation for a state transition table

	s_1	...	s_n
states	event / action	...	event / action
s_1	e_i / a_k	...	–
...	...	...	...
s_n	e_j / a_l	...	e_p / a_q

,

Figure 5: Forgacs [1] table 6.1 and 6.2.

Figure 6.2 State diagram for the 'RoboDog' example

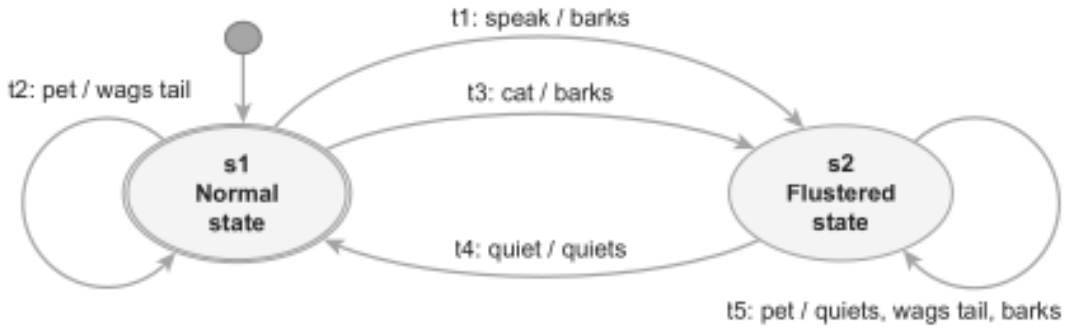


Figure 6: Forgacs [1] figure 6.2.

Table 6.3 State–event notation for the 'RoboDog' example with undefined transitions

	input: speak	input: quiet	input: pet	input: cat
start state	end state / action	end state / action	end state / action	end state / action
normal	flustered / barks	–	normal / wags tail	flustered / barks
flustered	–	normal / quiets	flustered / quiets, wags tail, barks	–

Figure 7: Forgacs [1] table 6.3.

```
/** Test possible sneak paths for the Starting state. */
@Test public void testSneakStarting() {
    /* Starting --(PlayerMove)-> IGNORE */
    assertTrue(theEngine.inStartingState());
    theEngine.movePlayer(0, 1); // dy = 1  means  DOWN
    assertTrue(theEngine.inStartingState());

    /* Starting --(MonsterMove)-> IGNORE */
    assertTrue(theEngine.inStartingState());
    theEngine.moveMonster(theMonster, 0, -1); // dy = -1  means  UP
    assertTrue(theEngine.inStartingState());

    /* Starting --(Quit)-> IGNORE */
    assertTrue(theEngine.inStartingState());
    theEngine.quit();
    assertTrue(theEngine.inStartingState());
}

/** Test possible sneak paths for the Playing state. */
@Test public void testSneakPlaying() {
    /* Starting --(Start)-> Playing */
    assertTrue(theEngine.inStartingState());
    theEngine.start();
    assertFalse(theEngine.inStartingState());
    assertTrue(theEngine.inPlayingState());

    /* Playing --(Start)-> IGNORE */
    assertTrue(theEngine.inPlayingState());
    theEngine.start();
    assertTrue(theEngine.inPlayingState());
}

/** Test possible sneak paths for the Halted state. */
@Test public void testSneakHalted() {
    /* Starting --(Start)-> Playing */
    assertTrue(theEngine.inStartingState());
    theEngine.start();
    assertFalse(theEngine.inStartingState());
    assertTrue(theEngine.inPlayingState());

    /* Playing --(Quit)-> Halted */
    assertTrue(theEngine.inPlayingState());
    theEngine.quit();
    assertFalse(theEngine.inPlayingState());
    assertTrue(theEngine.inHaltedState());

    /* Halted --(PlayerMove)-> IGNORE */
    assertTrue(theEngine.inHaltedState());
    theEngine.movePlayer(0, 1); // dy = 1  means  DOWN
    assertTrue(theEngine.inHaltedState());

    /* Halted --(MonsterMove)-> IGNORE */
    assertTrue(theEngine.inHaltedState());
    theEngine.moveMonster(theMonster, 0, 1);
    assertTrue(theEngine.inHaltedState());

    /* Halted --(Quit)-> IGNORE */
    assertTrue(theEngine.inHaltedState());
    theEngine.quit();
    assertTrue(theEngine.inHaltedState());
}

/** Test possible sneak paths for the Player Won state. */
@Test public void testSneakPlayerWon() {
    /* Starting --(Start)-> Playing */
    assertTrue(theEngine.inStartingState());
    theEngine.start();
    assertFalse(theEngine.inStartingState());
    assertTrue(theEngine.inPlayingState());

    /* Playing --(PlayerMove)-> Playing */
    assertTrue(theEngine.inPlayingState());
    theEngine.movePlayer(-1, 0); // dx = -1  means  LEFT
    assertTrue(theEngine.inPlayingState());

    /* Playing --(PlayerMove)-> Player Won */
    assertTrue(theEngine.inPlayingState());
    theEngine.movePlayer(0, 1); // dy = 1  means  DOWN
    assertFalse(theEngine.inPlayingState());
    assertTrue(theEngine.inWonState());
    assertTrue(theEngine.inGameOverState());

    /* Player Won --(Quit)-> IGNORE */
    assertTrue(theEngine.inWonState());
    theEngine.quit();
    assertTrue(theEngine.inWonState());

    /* Player Won --(PlayerMove)-> IGNORE */
    assertTrue(theEngine.inWonState());
    theEngine.movePlayer(0, -1); // dy = -1  means  UP
    assertTrue(theEngine.inWonState());

    /* Player Won --(MonsterMove)-> IGNORE */
    assertTrue(theEngine.inWonState());
    theEngine.moveMonster(theMonster, 0, -1); // dy = -1  means  UP
    assertTrue(theEngine.inWonState());
}

/** Test possible sneak paths for the Player Died state. */
@Test public void testSneakPlayerDied() {
    /* Starting --(Start)-> Playing */
    assertTrue(theEngine.inStartingState());
    theEngine.start();
    assertFalse(theEngine.inStartingState());
    assertTrue(theEngine.inPlayingState());

    /* Playing --(PlayerMove)-> Player Died */
    assertTrue(theEngine.inPlayingState());
    theEngine.movePlayer(0, 1); // dx = 1  means  DOWN
    assertFalse(theEngine.inPlayingState());
    assertTrue(theEngine.inDiedState());

    /* Player Died --(Quit)-> IGNORE */
    assertTrue(theEngine.inDiedState());
    theEngine.quit();
    assertTrue(theEngine.inDiedState());

    /* Player Died --(PlayerMove)-> IGNORE */
    assertTrue(theEngine.inDiedState());
    theEngine.movePlayer(0, -1); // dy = -1  means  UP
    assertTrue(theEngine.inDiedState());

    /* Player Died --(MonsterMove)-> IGNORE */
    assertTrue(theEngine.inDiedState());
    theEngine.moveMonster(theMonster, 0, -1); // dy = -1  means  UP
    assertTrue(theEngine.inDiedState());
}
```

Codeblock 2: The sneak path test cases. You will have to zoom in a bit, sorry for that.

References

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