SENG275 – Lab 8
Due Wednesday March 24, 11:55 pm

Welcome to Lab 8. Throughout this document, some sections will be in Red, and others will be in **bold**. The sections in Red will be marked. The questions in **bold** will not be graded. However, you should give these questions some thought, as these are the sorts of questions that could appear on a quiz or midterm. This lab is worth 4% of your final grade.

Quick summary of what you need to do:

1. Write the Acceptance tests for the four scenarios on pages 7 and 8. Place your tests in a file named src/test/java/nl.tudelft.jpacman/acceptance/AcceptanceTest.java. Put any maps you create in the src/test/java/resources directory.

You're done!

System Testing

System or end-to-end testing involves automated testing of the entire, actual piece of software we're working with. While we sacrifice the controllability and observability of discrete unit tests, and we also sacrifice the simplicity and flexibility of mocking, we gain *realism* – we're testing the actual behaviour of the real program, as it would be delivered to the customer.

The system tests we will write today are similar in structure, but they differ in the nature of the scenarios we'll use.

Scenarios written by testers – Smoke Tests

We've encountered a system test before, in the form of a 'Smoke Test' (default-test/…/LauncherSmokeTest.java). This test ran a series of actions, checking for success after each, according to a pre-programmed scenario.

Take a look at the smoke test source code. If you had to write out what we asked pacman to do, it might look something like "Start the game, then move east, then west, then east 7 steps, then north 6, then west 2, then north 2. Then wait, then move west 10, then east 10. At some point during this pacman gets caught, so end the game". As you're reading this, you're probably wondering 'why this particular series of actions?' The answer is that this test is intended to convince the *tester* that the program is working; a smoke-test after all is intended to just exercise the program to see if it catastrophically fails. The actual behaviour performed during the test isn't as important as just doing *something*.

In other words, the scenario in a smoke test (and every other type of test we've written this semester) were specified by the tester, because they're intended to *verify* that the program is written correctly.

Scenarios written by users - Acceptance Tests

System tests are also used to *validate* the program – to ensure that the program meets the users' needs (in other words, that the correct program has been written). In these cases we call them Acceptance Tests – we want to know if the customer will *accept* the program as written.

Because it's sometimes difficult for customers to express their needs clearly, Agile methodology proposed the *User Story* – a short narrative in a set structure (this is a form of behaviour-driven development). These scenarios can be used as a guide to testing, and can help clarify when a user need has been met and when it hasn't (and thus, whether or not a program is *valid*.) In theory, if the user agrees that the *Acceptance Criteria* have been met, then they should agree that their needs have been met.

Here's the structure:

```
**Title** _one line describing the story_

**Narrative**

    As a [role]
     I want [feature]
    So that [benefit]

**Acceptance Criteria,** _presented as scenarios of the form:_

    Scenario 1: Title
    Given [context]
     and  [some more context]...
    When  [event]
    Then  [outcome]
     and  [another outcome]...
```

For example, consider a user who wants to be able to start the game using the start button. They might write their user story like this:

```
Title: Starting the game

Narrative

As a player
I want to be able to start the game
So that I can play

Acceptance Criteria

Scenario 1: Click the start button
Given the game isn't in progress
When the start button is clicked
Then the game should be in progress.
```

In this structure, the role is ‘player’, ‘starting the game’ is the feature requested, and the benefit is ‘playing the game’. For each scenario, if the test confirms that the outcome occurred, then the acceptance criteria has been met.

The test for this user story has already been written; you can find it in `default-test/…/integration/`**`StartupSystemTest.java`**.

Worked Example:

Let’s take a more complicated scenario.

```
Title: Move the Player

Narrative

As a player,
 I want to move my Pacman around on the board;
So that I can earn all points and win the game.

Acceptance Criteria:

Scenario 1: The player moves on empty square

Given the game has started,
 and  my Pacman is next to an empty square;
When  I press an arrow key towards that square;
Then  my Pacman can move to that square
 and  my points remain the same.
```

Specifying a custom map

Using the default board (in main/resources/board.txt) is a problem, since Pacman doesn’t start next to a square without a pellet. We’ll create a custom map for this test; we can store custom maps in **`test/resources`** as text files, and load them with launcher.withMapFile():

The file I’ll create for this test will look like this:

```
#####
#. P#
#####
```

I can save it as ‘../test/resources/moveNoPelletNoScoreIncrease.txt’ and load it in my test like this:

```
Launcher launcher = new Launcher();
launcher =
launcher.withMapFile("/moveNoPelletNoScoreIncrease.txt");
launcher.launch();
```

Note the forward-slash before the filename; this is required by the Java's resource loader class.

Here's the test corresponding to this scenario:

```
private Launcher launcher;
private Game game;
private Player player;

@BeforeEach
public void before() {launcher = new Launcher();}

@AfterEach
public void after() {launcher.dispose();}

@Test
public void moveNoPelletNoScoreIncrease() {
    launcher = launcher.withMapFile("/moveNoPelletNoScoreIncrease.txt");
    launcher.launch();

    game = launcher.getGame();
    game.start();

    // check that we've loaded the correct board - not required but nice

    assertEquals(3, game.getLevel().getBoard().getHeight());
    assertEquals(5, game.getLevel().getBoard().getWidth());
    assertEquals(1, game.getLevel().remainingPellets());

    // given the game has started

    assertTrue(game.isInProgress());

    // and my pacman is next to an empty square

    assertTrue(game.getLevel().isAnyPlayerAlive());
    player = game.getPlayers().get(0);
    Square target = player.getSquare().getSquareAt(Direction.WEST);
    assertTrue(target.getOccupants().isEmpty());

    // when I press arrow key toward that square, my pacman can move to
    // that square and my points remain the same.

    assertTrue(target.isAccessibleTo(player));
    int pointsBeforeMoving = player.getScore();
    game.move(player, Direction.WEST);

    assertEquals(pointsBeforeMoving, player.getScore());
    assertEquals(player.getSquare(), target);
}
```

Note that our test design breaks our ‘one ACTION, one test’ rule. This is intended – system tests are intended to exercise the program by executing a series of actions. Acceptance tests follow a ‘one acceptance criteria, one test’ rule instead.

Buttons and Keypresses

Although the user stories referred to start buttons being clicked and keys being pressed, we need to remember that they are written from the user’s perspective – the user sees a button and clicks it, so that’s what goes into the story. The actual program, however, might not have been set up to allow us to click buttons or press keys programmatically. Thus we may need to interpret the story a little to find out how to follow it.

In the case of user input, we would check the ‘ui’ package in the main program to see if there was a way to ‘click’ the start button or ‘press a key’ with a function call; it’s possible that the JPacman developers would have anticipated this need and built ways to programmatically interact with the interface.

Instead, we find that the interface is generated at runtime using a system of method callbacks, and that ‘clicking the start button’ just amounts to calling game.start(). Similarly, each key is configured to call game.move() with a different Direction. Since there’s no way to make our tests more realistic than just calling those methods directly, that’s what we have to do.

This may be disappointing, but it may help to remember that a user story just describes what visible behaviour the user wants to see – what exactly happens behind the scenes to make it happen isn’t their concern, nor should it be.

Exercises:

Write acceptance tests for the following user stories, one per scenario. Put them in '…test/…/acceptance/AcceptanceTest.java' – you'll need to create a new 'acceptance' package alongside board, game, level and npc. Each of your tests should pass and fully satisfy the acceptance criteria – you don't need to add comments, but doing so will help the marker follow your work.

Hints:

- Remember you're testing the entire, actual game – you need to keep everything you've learned about JPacman in mind in order to predict the behaviour of the system. For example, why do you think it was necessary for the custom map for the worked example to contain a pellet on the left side, if pacman never moved onto it and we never tested for it? (remembering how game.start() works might help you here).
- Freely use the code provided for you in the smoke test, the StartupSystemTest and the worked example above.
- Some of the tests may require more general programming skills – you may need to write loops, use collections and maintain variables of different types in order to satisfy the acceptance criteria. For example, the Java *instanceof* operator may be helpful – `(something instanceof Pellet)` will evaluate to `true` if something is an object of type `Pellet`.
- Trying to start a game without a Pacman on the board will usually fail; include one even if you don't need it.
- Code you can use as a template is provided on the page following the scenarios.

```
Title: Suspend the game

Narrative

As a player,
 I want to be able to suspend the game;
So  that I can pause and do something else.

Acceptance Criteria

Scenario 1: Suspend the game.
Given the game has started;
When  the player clicks the "Stop" button;
Then  the game is not in progress.
```

Title: Move the player

Narrative

As a player,
I want to move my Pacman around on the board;
So that I can earn all points and win the game.

Acceptance Criteria

Scenario 2: The player consumes
Given the game has started,
and my Pacman is next to a square containing a pellet;
When I press an arrow key towards that square;
Then my Pacman can move to that square,
and I earn the points for the pellet,
and the pellet disappears from that square.

Scenario 3: The player is trapped
Given the game has started,
and my Pacman is surrounded by walls;
When I press an arrow key towards any direction;
Then my Pacman does not move.

Scenario 4: The player dies
Given the game has started,
and my Pacman is next to a cell containing a ghost;
When I press an arrow key towards that square;
Then my Pacman dies,
and the game is over.

```java
package nl.tudelft.jpacman.acceptance;

import nl.tudelft.jpacman.Launcher;
import nl.tudelft.jpacman.board.Direction;
import nl.tudelft.jpacman.board.Square;
import nl.tudelft.jpacman.board.Unit;
import nl.tudelft.jpacman.game.Game;
import nl.tudelft.jpacman.level.Pellet;
import nl.tudelft.jpacman.level.Player;
import org.junit.jupiter.api.AfterEach;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;

import static org.junit.jupiter.api.Assertions.*;

import static org.assertj.core.api.Assertions.assertThat;

public class AcceptanceTest {


    private Launcher launcher;
    private Game game;
    private Player player;

    /**
     * Start a launcher, which can display the user interface.
     */
    @BeforeEach
    public void before() {
        launcher = new Launcher();

    }

    /**
     * Close the user interface.
     */
    @AfterEach
    public void after() {
        launcher.dispose();
    }
}
```