

SENG275 – Lab 3

Due Wednesday Feb 10, 5:00 pm

Technical note: You now have a fourth repository in your gitlab.csc.uvic.ca account, named Bowling.git.

Welcome to Lab 3. 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:

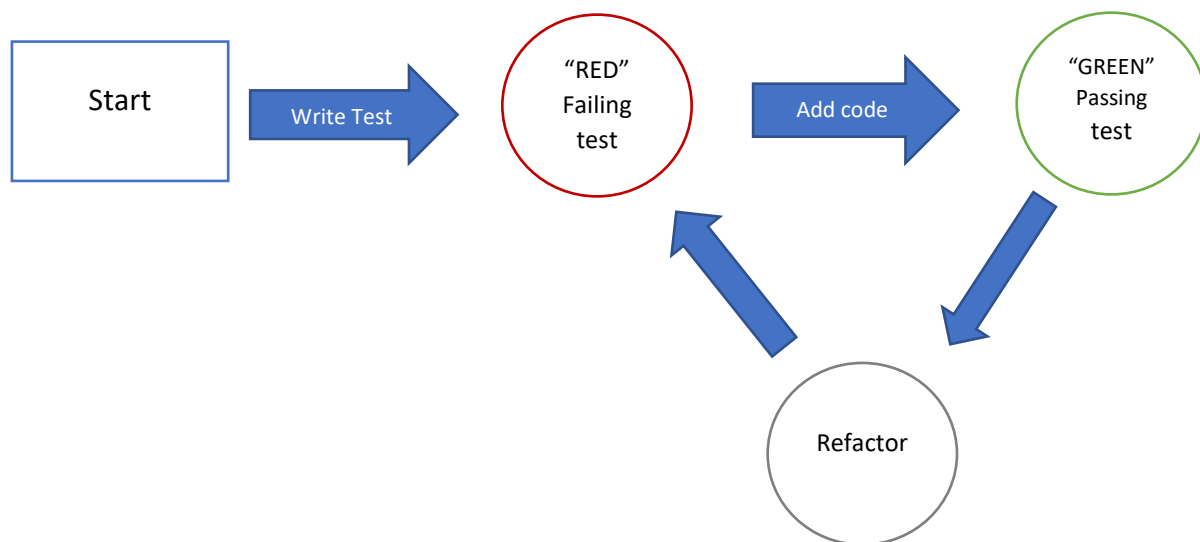
1. Using the Test Driven Development method, first create a test for the first bowling game score scenario on page 6. Then add functionality to the game to make the test pass. Finally refactor the resulting code. (worth 1%)
2. Using the Test Driven Development method, first create a test for the second bowling game score scenario on page 6. Then add functionality to the game to make the test pass. Finally refactor the resulting code. (worth 1%)
3. After running the JPacman Smoke Test, and inspecting the code for the Game, Unit, Board and Level classes, answer Exercises 3-6 on page 8 in your Labs/Lab_3/readme.txt file. (worth 2%)
4. Commit your changes to the Bowling and Lab_3 repositories, and push the commits back to gitlab.

You're done!

Test-Driven Development

We have, until now, been writing our tests *after* our application code is written. The first half of this lab will allow us to practice Test-Driven Development or TDD. This method follows the Red-Green-Refactor procedure:

- RED: before we implement a feature, we write a test for that feature. Of course, this test should fail – it won't pass until we actually write the code. Make sure the test fails before continuing!
- GREEN: write the MINIMUM amount of code necessary to make the test pass. Resist the urge to skip ahead – as soon as the test passes, move on to...
- REFACTOR: Adding code may have introduced duplication, a lack of clarity, or 'magic numbers' that could be replaced by named constants. Address these purely stylistic issues, then resume with the RED step.



Ten-Pin Bowling

The rules of bowling:

- A game of bowling consists of 10 frames, in which a bowler has two chances per frame to (rolling a bowling ball down a wooden lane) knock down all ten pins. If you want to learn (test) how bowling scoring works, take a look [here](#).

- If a bowler knocks down all the pins on the first chance, this is called a strike.
- When a strike is scored, add 10 plus the number of pins knocked down from their next two rolls.
- If a bowler knocks down all of the pins during their turn, this is called a spare.
- When a spare is scored, add 10 plus the number of pins knocked down from their next roll.
- Otherwise an open frame is scored for the number pins knocked down on both rolls. (An 'open frame' is any frame that does not end with a strike or a spare).
- The tenth frame is different – if the player scores a spare (they knock down all the pins in two throws) they get a third throw, then the game ends. If they get a strike on the first throw of the tenth frame, they get two more throws then the game ends.

One test has been written – it just checks that the test framework is hooked up correctly. Our job is to implement the above scoring rules using TDD.

Guided Example

Take a look at the `createFrames()` method in `BowlingGame.java`. The program is already partially complete – in a real TDD environment we'd be starting from scratch. Acquaint yourself with the following variables:

- The **rolls[]** array contains the number of pins knocked down by each roll.
- The **frames** ArrayList contains the score per frame. `createFrames` populates this ArrayList.
- The **frameRolls** ArrayList will hold a series of integers, which will add up to the score for this frame.

Our goal (for the section of the method between the two comments) is to write code that will correctly read integers from the **rolls** array, and add them to the **frameRolls** arraylist, depending on what's going on in the game.

Challenge: Write a test that passes only when the program correctly scores a player who throws ‘all gutter balls’ – they score zero on every roll. This is the following scenario:

Frame	1		2		3		4		5		6		7		8		9		10		
Roll	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	
Score	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
Frame Score	0		0		0		0		0		0		0		0		0		0		
Total Score																				0	

First, write the initial, RED test for this scenario:

```
@Test
public void AllGutterBallsScores0() {
    BowlingGame game = new BowlingGame(new int[] {0,0, 0,0, 0,0, 0,0, 0,0,
0,0, 0,0, 0,0, 0,0, 0,0});
    int result = game.score();
    assertEquals(0, result);
}
```

This test fails, since between the comments we have a placeholder that makes every frame score -1 (this is artificial, to make this initial test fail). Normally, before you add any code, the first test you write will fail.

Let’s make the MINIMAL change required to make the test pass. Change the line between the comments:

```
frameRolls.add(0);
```

Resist the temptation to add more code. Instead, re-run your tests, and note that they are all GREEN. At this point, we could refactor, but we haven’t added any code that needs it, so we’ll write our next RED test.

Challenge: Write a test that passes only when the program correctly scores a player who hits one pin, and misses every other one. This scenario looks something like this:

Frame	1		2		3		4		5		6		7		8		9		10	
Roll	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
Score	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0
Frame Score	0		0		0		0		0		0		1		0		0		0	

Total Score		1
-------------	--	---

Again, write the initial, RED test for this scenario:

```
@Test
public void onePinOnly() {
    BowlingGame game = new BowlingGame(new int[] {0,0, 0,0, 0,0, 0,0, 0,0,
0,0, 1,0, 0,0, 0,0, 0,0});
    int result = game.score();
    assertEquals(1, result);
}
```

Run the test, and note that it is RED. Now we return to BowlingGame.java, and again make the MINIMUM change required to make the test pass.

```
frameRolls.add(rolls[rollIndex]);
rollIndex++;
```

Unfortunately, this basic change doesn't pass – the for-loop we're in goes through ten frames, but we have 20 rolls in our scenario. We need to remember we're trying to produce the total score for a single frame, so we need to consider both rolls in the frame instead:

```
frameRolls.add(rolls[rollIndex]);
frameRolls.add(rolls[rollIndex + 1]);
rollIndex += 2;
```

Now our test passes, and we're ready for the next RED test.

TDD Exercises:

Your code DOES NOT have to worry about invalid or nonsensical inputs. When creating your tests, use the calculator at bowlinggenius.com to explore different scenarios and determine the expected score.

1. Implement spares into the program. Remember to write a failing test first! After you've written the code that causes the test to pass, remember to refactor if appropriate. The following scenario will work, or create your own:

Frame	1		2		3		4		5		6		7		8		9		10		
Roll	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	
Score	4	4	4	4	4	4	4	4	5	5	4	4	4	4	4	4	4	4	4	4	
Frame Score	8		8		8		8		14		8		8		8		8		8		
Total Score																				86	

2. Implement strikes into the program. Remember to write a failing test first! After you've written the code that causes the test to pass, remember to refactor if appropriate. The following scenario will work, or create your own:

Frame	1		2		3		4		5		6		7		8		9		10		
Roll	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	
Score	4	4	4	4	4	4	4	4	4	4	4	4	10		4	4	4	4	4	4	
Frame Score	8		8		8		8		8		8		18		8		8		8		
Total Score																				90	

Hints:

- A spare (getting all 10 pins down in one frame) means that the frame will score 10, plus whatever the next roll knocks down.
- A strike (getting all 10 pins down in the first roll of a frame) means that the frame will score 10, plus whatever the next two rolls knock down.
- Note that after a strike is rolled, the next roll begins the next frame – a frame with a strike in it only has one roll.

When you're done, push the repository back to gitlab.

Smoke Test

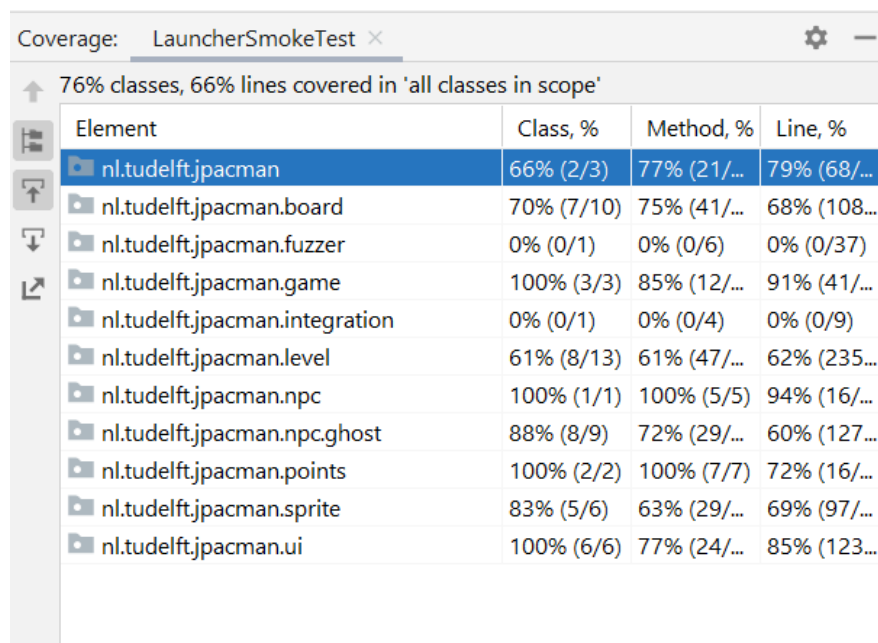
A Smoke Test is a simple system test that exercises minimal system behavior. If we just start the system, and we “see smoke” (i.e., not even the smoke test passes), there is no point in moving to the next step of the software development cycle.

JPacman has a simple smoke test provided already – find it at `default-test > java > nl.tudelft.jpacman > LauncherSmokeTest`. Run it and observe the results.

The smoke test has clearly done something – it has created a Game, created Pacman and some ghosts and moved them around on a board a little. Clearly some parts of the game were involved, but how effective was the smoke test? Did it run only a tiny fraction of the program, or did it execute a large proportion of the code? Were there parts it left out?

IntelliJ includes code coverage diagnostic tools to help answer these questions. Select `LauncherSmokeTest`, then click on Run from the menu, and choose ‘Run LauncherSmokeTest with Coverage’.

Note the ‘Coverage’ window that appears in the upper right.



Coverage: LauncherSmokeTest			
76% classes, 66% lines covered in 'all classes in scope'			
Element	Class, %	Method, %	Line, %
nl.tudelft.jpacman	66% (2/3)	77% (21/...	79% (68/...
nl.tudelft.jpacman.board	70% (7/10)	75% (41/...	68% (108...
nl.tudelft.jpacman.fuzzer	0% (0/1)	0% (0/6)	0% (0/37)
nl.tudelft.jpacman.game	100% (3/3)	85% (12/...	91% (41/...
nl.tudelft.jpacman.integration	0% (0/1)	0% (0/4)	0% (0/9)
nl.tudelft.jpacman.level	61% (8/13)	61% (47/...	62% (235...
nl.tudelft.jpacman.npc	100% (1/1)	100% (5/5)	94% (16/...
nl.tudelft.jpacman.npc.ghost	88% (8/9)	72% (29/...	60% (127...
nl.tudelft.jpacman.points	100% (2/2)	100% (7/7)	72% (16/...
nl.tudelft.jpacman.sprite	83% (5/6)	63% (29/...	69% (97/...
nl.tudelft.jpacman.ui	100% (6/6)	77% (24/...	85% (123...

With this display, you can evaluate whether or not the smoke test has evenly tested the various classes in the project, or has been focused on one narrow area of the code. You can extend this analysis to individual methods, and even see which particular lines of code were executed

during the test (code that is executed will be marked green in the source code display). This is useful (for example) to see if only one branch of a conditional is taken during your test, and if perhaps another test needs to be written to cover the other possibility.

If you find the coverage report annoying to navigate, click on the icon with the arrow pointing to the upper right (Generate coverage report). This will allow you to save an HTML-formatted report on your hard drive, which you can read with a web browser.

Exercises:

Answer the following conceptual questions in the **Lab_3/readme.txt** file in your Labs repository. 100 words max for each.

3. Execute the smoke test, with coverage enabled. Name 2 classes that are not well-tested. Take a look at those classes, and speculate as to their purpose. (0.5%)
4. Is the move() method in the game.Game class covered by the smoke test? (0.5%)
5. Change the getDeltaX() method in board.Direction, so that it returns deltaY instead. Re-run the smoke test and note the resulting error. Is the error message you see helpful in diagnosing the problem, or is there not enough context to figure out what went wrong? Would you prefer a smoke test or a unit test if you had to fix the problem from scratch? (0.5%)

Make sure you undo the change to getDeltaX() before moving on to the next question.

6. Skim over the Game, Unit, Board and Level classes and explain (max 100 words) how it appears these four classes are related to each other – what role does each play? What does each represent? (0.5%)

Commit and push your answers back to the Labs repository on gitlab.