Testing Lab Tasks 2.1 & 3

Task 2.1:

The three methods I implemented tests for within Level.java are:
addObserver(), removeObserver(), and **remainingPellets()**

The class coverage is now at 23% after the introduction.

Element	Class, %	Method, %	Line, %
nl	23% (13/55)	12% (39/306)	12% (143/1147)
tudelft	23% (13/55)	12% (39/306)	12% (143/1147)
jpacman	23% (13/55)	12% (39/306)	12% (143/1147)
board	40% (4/10)	13% (7/53)	11% (16/142)
fuzzer	0% (0/1)	0% (0/6)	0% (0/32)
game	0% (0/3)	0% (0/14)	0% (0/37)
integration	0% (0/1)	0% (0/4)	0% (0/6)
level	30% (4/13)	16% (12/72)	17% (59/343)
npc	0% (0/10)	0% (0/47)	0% (0/237)
points	0% (0/2)	0% (0/7)	0% (0/19)
sprite	83% (5/6)	44% (20/45)	52% (68/130)
ui	0% (0/6)	0% (0/31)	0% (0/127)
Launcher	0% (0/1)	0% (0/21)	0% (0/41)
LauncherSmokeTest	0% (0/1)	0% (0/4)	0% (0/29)
PacmanConfigurationException	0% (0/1)	0% (0/2)	0% (0/4)

Details of addObserver()

The testAddObserver() method checks if an observer is successfully added to the Level object. It does this by creating a mocked observer, adding it to the level, and then using reflection to verify its existence in a private field named "observers."

```
@Test
public void testAddObserver() throws NoSuchFieldException, IllegalAccessException {
    Level.LevelObserver mockObserver = mock(Level.LevelObserver.class);

    level.addObserver(mockObserver);

    // Access the private field using reflection
    Field field = Level.class.getDeclaredField( name: "observers");
    field.setAccessible(true);
    Set<Level.LevelObserver> actualObservers = (Set<Level.LevelObserver>) field.get(level);

    assertTrue(actualObservers.contains(mockObserver));
}
```

removeObserver()

The testRemoveObserver() method ensures that an added observer can be removed from the Level. It first makes sure of the observer's existence then removes it, using reflection again to confirm its absence.

```
@Test
public void testRemoveObserver() throws NoSuchFieldException, IllegalAccessException {
    Level.LevelObserver mockObserver = mock(Level.LevelObserver.class);

    level.addObserver(mockObserver);

    // Access the private field using reflection
    Field field = Level.class.getDeclaredField( name: "observers");
    field.setAccessible(true);
    Set<Level.LevelObserver> actualObservers = (Set<Level.LevelObserver>) field.get(level);

    assertTrue(actualObservers.contains(mockObserver));

    // Use the remove observer method
    level.removeObserver(mockObserver);

    assertFalse(actualObservers.contains(mockObserver));
}
```

remainingPellets()

The testRemainingPellets() method tests the accuracy of the count of remaining game pellets. It sets up a mock 2x2 board, places 3 mock pellets on it, and verifies that the method of the Level class correctly holds: that there are 3 pellets left.

```
@Test
public void testRemainingPellets() {
    int width = 2;
    int height = 2;
    int expectedPellets = 3;

    Square[][] squares = new Square[width][height];
    for (int x = 0; x < width; x++) {
        for (int y = 0; y < height; y++) {
            squares[x][y] = mock(Square.class);
            when(squares[x][y].getOccupants()).thenReturn(Collections.emptyList());
        }
    }

    // Place 3 pellets on the board
    when(squares[0][0].getOccupants()).thenReturn(Collections.singletonList(mock(Pellet.class)));
    when(squares[0][1].getOccupants()).thenReturn(Collections.singletonList(mock(Pellet.class)));
    when(squares[1][0].getOccupants()).thenReturn(Collections.singletonList(mock(Pellet.class)));

    when(mockBoard.getWidth()).thenReturn(width);
    when(mockBoard.getHeight()).thenReturn(height);
    for (int x = 0; x < width; x++) {
        for (int y = 0; y < height; y++) {
            when(mockBoard.squareAt(x, y)).thenReturn(squares[x][y]);
        }
    }

    assertEquals(expectedPellets, level.remainingPellets(), message: "Number of pellets should match expected"
}
```

Task 3:

Q: Are the coverage results from `JaCoCo` similar to the ones you got from `IntelliJ` in the last task? Why so or why not?

The coverage results from JaCoCo are higher by a little over 30% than the built-in IntelliJ extension for code coverage. I assume the results are different as JaCoCo includes branch coverage and appears to go into more detail than what is possible with line coverage.

Q: Did you find helpful the source code visualization from `JaCoCo` on uncovered branches?

Yes, this gives a more specific view of what was missed in testing for uncovered branches.

Q: Which visualization did you prefer and why? `IntelliJ`'s coverage window or `JaCoCo`'s report?

Personally, I prefer the seamless integration of IntelliJ's coverage over the external nature of JaCoCo's reports. However, it does appear that JaCoCo is a more visually distinct, helpful tool for more granular detail.