Angel Quintanilla

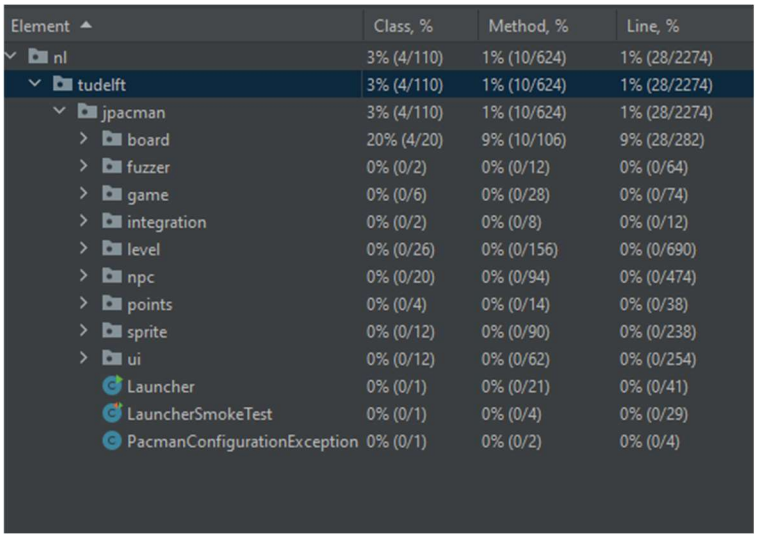
CS 472 – 1001

Professor Businge

2 February 2022

Link to Fork Repository : https://github.com/aq6476/cs472project-fork

Task 1:



Element	Class, %	Method, %	Line, %
nl	3% (4/110)	1% (10/624)	1% (28/2274)
tudelft	3% (4/110)	1% (10/624)	1% (28/2274)
jpacman	3% (4/110)	1% (10/624)	1% (28/2274)
board	20% (4/20)	9% (10/106)	9% (28/282)
fuzzer	0% (0/2)	0% (0/12)	0% (0/64)
game	0% (0/6)	0% (0/28)	0% (0/74)
integration	0% (0/2)	0% (0/8)	0% (0/12)
level	0% (0/26)	0% (0/156)	0% (0/690)
npc	0% (0/20)	0% (0/94)	0% (0/474)
points	0% (0/4)	0% (0/14)	0% (0/38)
sprite	0% (0/12)	0% (0/90)	0% (0/238)
ui	0% (0/12)	0% (0/62)	0% (0/254)
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)

Figure 1 : Coverage Test in the jpacman/Test directory when it is cloned

Task 2 : testIsAlive() Method:

This method essentially just creates a Player, and sets their status to be Alive. We test that isAlive() returns the correct value, which is “true” in this case. Below the chart is the impact on the code coverage of our testing environment.

```java
package level;

import nl.tudelft.jpacman.level.Player;
import nl.tudelft.jpacman.level.PlayerFactory;
import nl.tudelft.jpacman.sprite.PacManSprites;
import org.junit.jupiter.api.Test;

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

/**
 * General Test for the isAlive() method to make sure it
 * performs to expectations, returning the correct values
 * in its use cases
 */
no usages  new *
public class PlayerTest {
    /**
     * Does isAlive() return a correct alive status?
     */
    no usages  new *
    @Test
    void testIsAlive() {
        PacManSprites sprite = new PacManSprites();
        PlayerFactory fact = new PlayerFactory(sprite);
        Player player = fact.createPacMan();

        assertThat(player.isAlive()).isEqualTo( expected: true);


    }
}
```

Figure 2 : Code of test used to test the isAlive() method

Element	Class, %	Method, %	Line, %
nl	16% (18...	9% (60/624)	8% (190/23...
tudelft	16% (18...	9% (60/624)	8% (190/23...
jpacman	16% (18...	9% (60/624)	8% (190/23...
board	20% (4/...	9% (10/106)	9% (28/282)
fuzzer	0% (0/2)	0% (0/12)	0% (0/64)
game	0% (0/6)	0% (0/28)	0% (0/74)
integration	0% (0/2)	0% (0/8)	0% (0/12)
level	15% (4/...	6% (10/156)	3% (26/700)
npc	0% (0/20)	0% (0/94)	0% (0/474)
points	0% (0/4)	0% (0/14)	0% (0/38)
sprite	83% (10...	44% (40/90)	52% (136/2...
ui	0% (0/12)	0% (0/62)	0% (0/254)
Launcher	0% (0/1)	0% (0/21)	0% (0/41)
LauncherSmokeTest	0% (0/1)	0% (0/4)	0% (0/29)
PacmanConfiguratic	0% (0/1)	0% (0/2)	0% (0/4)

Figure 3 : Coverage after adding tests for the IsAlive() method

Task 2.1 : Add 3 More Method Tests

Method 1: java/nl/tudelft/jpacman/level/LevelFactory.java – CreatePellet()

What the code accomplishes is complete a series of constructors to eventually create a Pellet object. To verify that this pellet object returned by CreatePellot() matches the criteria we need, we assert to make sure its width and sprite are equal to the correct values associated with them.

Code:

```
package level;

import nl.tudelft.jpacman.level.LevelFactory;
import nl.tudelft.jpacman.level.Pellet;
import nl.tudelft.jpacman.npc.ghost.GhostFactory;
import nl.tudelft.jpacman.points.DefaultPointCalculator;
import nl.tudelft.jpacman.sprite.PacManSprites;
import org.junit.jupiter.api.Test;

import static org.assertj.core.api.Assertions.assertThat;
no usages  new *
public class LevelFactoryTest {


    no usages  new *
    @Test
    void testLevelFactoryPelletGeneration() {
        //create a level factory
        PacManSprites sprites = new PacManSprites();
        GhostFactory ghostFactory = new GhostFactory(sprites);
        DefaultPointCalculator pointCalc = new DefaultPointCalculator();
        LevelFactory lfact = new LevelFactory(sprites, ghostFactory, pointCalc);

        //use the level factory to create a pellet
        Pellet p = lfact.createPellet();

        //check if pellet matches the correct values, seeing as LevelFactory's
        //PELLET_VALUE parameter in the constructor is private, we just use 10 instead
        assertThat(p.getValue()).isEqualTo( expected: 10);
        assertThat(p.getSprite()).isEqualTo(sprites.getPelletSprite());
    }


}
```

Figure 4 : Testing implementation if the CreatePellet() method performs to expectations

Coverage Results:

nl	23% (26/110)	12% (78/624)	10% (244/2328)
tudelft	23% (26/110)	12% (78/624)	10% (244/2328)
jpacman	23% (26/110)	12% (78/624)	10% (244/2328)
board	20% (4/20)	9% (10/106)	9% (28/282)
fuzzer	0% (0/2)	0% (0/12)	0% (0/64)
game	0% (0/6)	0% (0/28)	0% (0/74)
integration	0% (0/2)	0% (0/8)	0% (0/12)
level	30% (8/26)	14% (22/156)	8% (62/706)
CollisionInteractionMap	0% (0/2)	0% (0/9)	0% (0/41)
CollisionMap	100% (0/0)	100% (0/0)	100% (0/0)
DefaultPlayerInteractionMap	0% (0/1)	0% (0/5)	0% (0/13)
Level	0% (0/2)	0% (0/17)	0% (0/113)
LevelFactory	50% (1/2)	28% (2/7)	24% (7/29)
LevelTest	0% (0/1)	0% (0/9)	0% (0/30)
MapParser	0% (0/1)	0% (0/10)	0% (0/71)
Pellet	100% (1/1)	100% (3/3)	100% (6/6)
Player	100% (1/1)	37% (3/8)	54% (13/24)
PlayerCollisions	0% (0/1)	0% (0/7)	0% (0/21)
PlayerFactory	100% (1/1)	100% (3/3)	100% (5/5)
npc	10% (2/20)	2% (2/94)	1% (6/486)
points	50% (2/4)	0% (0/14)	4% (2/42)
sprite	83% (10/12)	48% (44/90)	56% (146/260)
ui	0% (0/12)	0% (0/62)	0% (0/254)
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)

Figure 5: Results of CreatePellet() Test

Method 2: java/nl/tudelft/jpacman/level/Player.java - Set Killer Method

This code checks if when setting the killer, the abstract Blinky class object is still recognized as the same data type , rather than having a runtime error of having the data type be casted, or returned as another Ghost. We check this as Ghost is a purely abstract class that has all other ghosts be derived from it.

```java
package level;

import nl.tudelft.jpacman.board.Direction;
import nl.tudelft.jpacman.level.Player;
import nl.tudelft.jpacman.level.PlayerFactory;
import nl.tudelft.jpacman.npc.ghost.Blinky;
import nl.tudelft.jpacman.sprite.PacManSprites;
import nl.tudelft.jpacman.sprite.Sprite;
import org.junit.jupiter.api.Test;

import java.util.Map;

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

/**
 * General Test for the isAlive() method to make sure it
 * performs to expectations, returning the correct values
 * in its use cases
 */
no usages  new *
public class PlayerTest {


    /*
        Check if when setting the killer the abstract class is still recognized as a
        "Blinky" unit at runtime instead of another type of Ghost. This is key for debugging
        as the unit class is abstract and type errors could be made via runtime
     */
    no usages  new *
    @Test
    void TestSetKiller() {
        PacManSprites sprite = new PacManSprites();
        PlayerFactory fact = new PlayerFactory(sprite);
        Player player = fact.createPacMan();
        Map<Direction, Sprite> spriteMap = null;
        Blinky killer = new Blinky(spriteMap);
        player.setKiller(killer);

        //check if abstract class at runtime is of the the same type ; Check if its also a blinky
        assertThat(player.getKiller().getClass()).isEqualTo(killer.getClass());

    }
```

Figure 4 : Testing implementation if the SetKiller() method performs to expectations

Coverage Results:

Element	Class, %	Method, %	Line, %
nl	27% (30/110)	13% (86/624)	11% (264/2286)
tudelft	27% (30/110)	13% (86/624)	11% (264/2286)
jpacman	27% (30/110)	13% (86/624)	11% (264/2286)
board	20% (4/20)	7% (8/106)	9% (26/272)
fuzzer	0% (0/2)	0% (0/12)	0% (0/64)
game	0% (0/6)	0% (0/28)	0% (0/74)
integration	0% (0/2)	0% (0/8)	0% (0/12)
level	30% (8/26)	16% (26/156)	10% (68/674)
CollisionInteractionMap	0% (0/2)	0% (0/9)	0% (0/41)
CollisionMap	100% (0/0)	100% (0/0)	100% (0/0)
DefaultPlayerInteractionMap	0% (0/1)	0% (0/5)	0% (0/13)
Level	0% (0/2)	0% (0/17)	0% (0/101)
LevelFactory	50% (1/2)	28% (2/7)	24% (7/29)
LevelTest	0% (0/1)	0% (0/9)	0% (0/30)
MapParser	0% (0/1)	0% (0/10)	0% (0/67)
Pellet	100% (1/1)	100% (3/3)	100% (6/6)
Player	100% (1/1)	62% (5/8)	66% (16/24)
PlayerCollisions	0% (0/1)	0% (0/7)	0% (0/21)
PlayerFactory	100% (1/1)	100% (3/3)	100% (5/5)
npc	30% (6/20)	8% (8/94)	4% (22/486)
points	50% (2/4)	0% (0/14)	4% (2/42)
sprite	83% (10/12)	48% (44/90)	56% (146/260)
ui	0% (0/12)	0% (0/62)	0% (0/254)
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)

Figure 6: Results of SetKiller() test

Method 3: java/nl/tudelft/jpacman/sprite/SpriteStore.java – loadSprite() method.

For this method we make sure that the the loadSprite() method returns an actual sprite object, instead of a null value or an empty sprite. We give the loadSprite function a determined path to the first sprite of a death animation. We assert that our dimensions are greater than zero as well.

```java
package sprite;

import nl.tudelft.jpacman.sprite.EmptySprite;
import nl.tudelft.jpacman.sprite.Sprite;
import nl.tudelft.jpacman.sprite.SpriteStore;
import org.junit.jupiter.api.Test;

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

public class spriteTest {
    @Test
    void verify_load_sprite(){
        EmptySprite emptySprite = new EmptySprite();
        SpriteStore store = new SpriteStore();
        Sprite sprite = null;

        //load a specific sprite - Pacman First frame of death animation -
        //and make sure it is neither null, or an empty sprite
        try {
            sprite = store.loadSprite( resource: "/sprite/dead.png");
        }
        catch(java.io.IOException e){
            sprite = null;
        }
        assertThat(sprite).isNotNull();
        assertThat(sprite.getHeight()).isGreaterThan( other: 0);
        assertThat(sprite.getWidth()).isGreaterThan( other: 0);



    }
}
```

Figure 7 : Testing implementation if the LoadSprite() method performs to expectations

Coverage Results:

nl	27% (30/110)	14% (88/624)	11% (266/2328)
tudelft	27% (30/110)	14% (88/624)	11% (266/2328)
jpacman	27% (30/110)	14% (88/624)	11% (266/2328)
board	20% (4/20)	9% (10/106)	9% (28/282)
fuzzer	0% (0/2)	0% (0/12)	0% (0/64)
game	0% (0/6)	0% (0/28)	0% (0/74)
integration	0% (0/2)	0% (0/8)	0% (0/12)
level	30% (8/26)	16% (26/156)	9% (68/706)
npc	30% (6/20)	8% (8/94)	4% (22/486)
points	50% (2/4)	0% (0/14)	4% (2/42)
sprite	83% (10/12)	48% (44/90)	56% (146/260)
ui	0% (0/12)	0% (0/62)	0% (0/254)
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)

Figure 7: Results of LoadSprite() Test

Task 3 : JaCoCo Coverage Report :

jpacman

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
nl.tudelft.jpacman.level		67%		57%	73	155	103	344	20	69	4	12
nl.tudelft.jpacman.npc.ghost		71%		55%	56	105	43	181	5	34	0	8
nl.tudelft.jpacman.ui		77%		47%	54	86	21	144	7	31	0	6
default		0%		0%	12	12	21	21	5	5	1	1
nl.tudelft.jpacman.board		86%		58%	44	93	2	110	0	40	0	7
nl.tudelft.jpacman.sprite		86%		59%	30	70	11	113	5	38	0	5
nl.tudelft.jpacman		69%		25%	12	30	18	52	6	24	1	2
nl.tudelft.jpacman.points		60%		75%	1	11	5	21	0	9	0	2
nl.tudelft.jpacman.game		87%		60%	10	24	4	45	2	14	0	3
nl.tudelft.jpacman.npc		100%		n/a	0	4	0	8	0	4	0	1
Total	1,210 of 4,694	74%	293 of 637	54%	292	590	228	1,039	50	268	6	47

Questions:

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

The results were greatly different, as the JaCoCo coverage was about 74% for total instructions, while the analysis from the ones that were showcased by IntelliJ gave a coverage of about 11% for total lines.

- Did you find helpful the source code visualization from JaCoCo on uncovered branches?

Yes, the source code visualization was very helpful, and is a great perspective to see which tests should be worked on, while which others can be considered more or less complete .

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

I think the IntelliJ coverage is more convenient to see an immediate effect on coverage, specifically because it's built into the interface. As for features, I have to state that the JaCoCo coverage report gives a lot more insight and easily pinpoints which conditions one would have to take when building test cases.