CMPT276: Assignment 4 Code Review

Brandon Chan 301558235 Aaron Lu 301462437

Code smells:

Brandon:

1: Guard.isCollidingWithPlayer confusing variable names:
In our Guard.Java file, we calculate the hitbox for the player and guard to see if they are colliding. However, in doing so I used confusing variable names that were unintuitive. I refactored the code using easier to understand variable names. An example of a confusing name is xLeftGuard and it was changed to leftSideGuardHitbox. This was pushed in commit labelled "code smell #1".

2: EnemyCollisionIntegrationTest Unsafe Code:
In our EnemeyCollisionIntegration test we use assertTrue to check that the new direction is different from the old direction assuring the guard has turned when a collision occurs. This code is unsafe as can be refactored for simplicity by using an assertNotEquals call instead. This was pushed in commit labelled "code smell #2".

Old call:
```
assertTrue(!(directions[i] == guard.direction));
```

New call:
```
assertNotEquals(directions[i], guard.direction);
```

3: PlayerObjectIntegrationTest
testPlayerBatteryCollision low documentation:

While testing our object-player collisions, we have a test for the player-battery collision which is different from other collisions as it requires you to re-initialize the SuperObject with a new battery. On the right you can see the added comments to the test. This was pushed in commit labelled "code smell #3".

```
@Test
public void testPlayerBatteryCollision() {
    // Set initial player score
    int initialScore = gp.score;

    //creating new super object to store battery
    gp.obj = new SuperObject[2];
    gp.obj[0] = new OBJ_Battery();

    //new SuperObject automatically used within .pickUpObject
    player.pickUpObject(i:0);

    // Verify that the player's score has increased
    assertEquals(initialScore + 100, gp.score);
}
```

4: PlayerMovementTest Lack of documentation

While testing player movement there is a lack of documentation showing how the test actually works. This can lead to other group members being confused on how the tests are set up or run. I updated the documentation with full explanation of how the test works as shown on the right. This was pushed in commit labelled "code smell #4".

```
@Test
public void testPlayerUpMovement() {
    int initialY = player.worldY; // accessing original player y coordinate
    keyH.upPressed = true; // simulating moving upwards
    player.update(); // updating player location with player.update
    int newY = player.worldY; // calculating new y coordinate
    assertEquals(initialY - player.speed, newY); //calculating if player's new y coordinate changed according to speed
}
```

Aaron:

5: **Lack Of Documentation:** TileManager.java
The methods getTileImage(), loadMap(), and TileManager() suffer from the lack of documentation. getTileImage() reads tile images from recourses and stores them in the Tile[] array. loadMap() Loads map data from our mapFile, reads the tile numbers and stores them in an array, and finally handles

exceptions. TileManager() (the constructor) sets the gamePanel to a pointer placeholder, creates a new instance of Tile, and loads the TileImage. This lack of documentation resulted in confusion when a partner needed to refer to one of the functions in the TileManager. Comments have now been added.

6: **Code Duplication:** keyHandlerTest.java
The keyHandlerTest is tasked with testing if our game, more specifically the keyHandler class (processes keyboard Inputs) responds properly to keyboard inputs. Before the code review, my test consisted of 4 nearly identical blocks of code (D key test seen below), testing the W,A,S,D keys . This is considered a code duplication. As a result, I have refactored this code with switch cases and a for loop that shuffles through the W, A, S, D keys and included a default condition where in the event the test was to fail, the default will assertTrue(false) and throw an error.

```
/**
 * Validates the correct implementation of the keyHandler by confirming the state of the keyPress has changed
 * dKeyPressEvent simulates a keypress, passes it to heyH.keyPressed, then assert checks if our instance of keyH has changed states
 * @param
 * @return
 */
@Test
public void dKeyPressed(){
GamePanel gameTest = new GamePanel();
KeyHandler keyH = new KeyHandler(gameTest);
gameTest.currentGameState = gameState.playing;

// SIMULATE a keypress event via KeyEvent, then sending it to keyPressed
KeyEvent dKeyPressEvent = new KeyEvent(new JFrame(), KeyEvent.KEY_PRESSED, System.currentTimeMillis(), 0, KeyEvent.VK_D, KeyEvent.CHAR_UNDEFINED);
keyH.keyPressed(dKeyPressEvent);

// CHECK if the 'rightPressed' flag is set to true
assertTrue(keyH.rightPressed);
}
```

7: **Confusing Variable Names:** TileManager.java
Our game utilizes a camera system that moves as the player moves, giving the user a small view into the characters surroundings. As a result, we have 2 sets of rows and columns. One row/col set is what the player sees (row, col) and the other set is the entire map (used for locating where the player is on the map). As a result, our variable naming when initially implementing the code was confusing which resulted in one of our teammates accidentally using the camera view row and col as the world, which broke the game. We never got around to changing it to a more clear name, so with this code review, I have.

```
//World Settings
public final int fullMapCol = 50;
public final int fullMapRow = 50;
```

8: **Unsafe Code:** ScoreSystemTest.java
When implementing the tests of our ScoreSystem, I overlooked the closure of our game state. As the tests are running back to back and the game variables (game state, score, lives, points) are manually modified to simulate a certain event to trigger tests, it's best to reset each instance to prevent conflicts within the tests. To do this, I manually set the state of the game to close after our tests have been complete (last line of code). This allows the subsequent test to start in a clean state.

```
public class ScoreSystemTest {

    /**
     * Validates the correct implementation of the gameOverState when the score falls below 0
     * pickUpObject: Switches currentGameState
     * @param
     * @return
     */
    @Test
    public void scoreBelowZeroTest (){
        //INITIALIZE GamePanel
        GamePanel gpT = new GamePanel();
        gpT.setupGame();

        //SET Score to < 0 (Illustrating a Negative Score)
        gpT.score = -2;

        //TESTING if the game will send currentGameState to gameOverState
        gpT.player.pickUpObject(i:7);
        assertEquals(gameState.gameOver,gpT.currentGameState);
        gpT.setcurrentGameState(gameState.gameOver);
    }
```