Assignment 4

Group 10

Exercise 1 - Final Product

Completion of Requirements

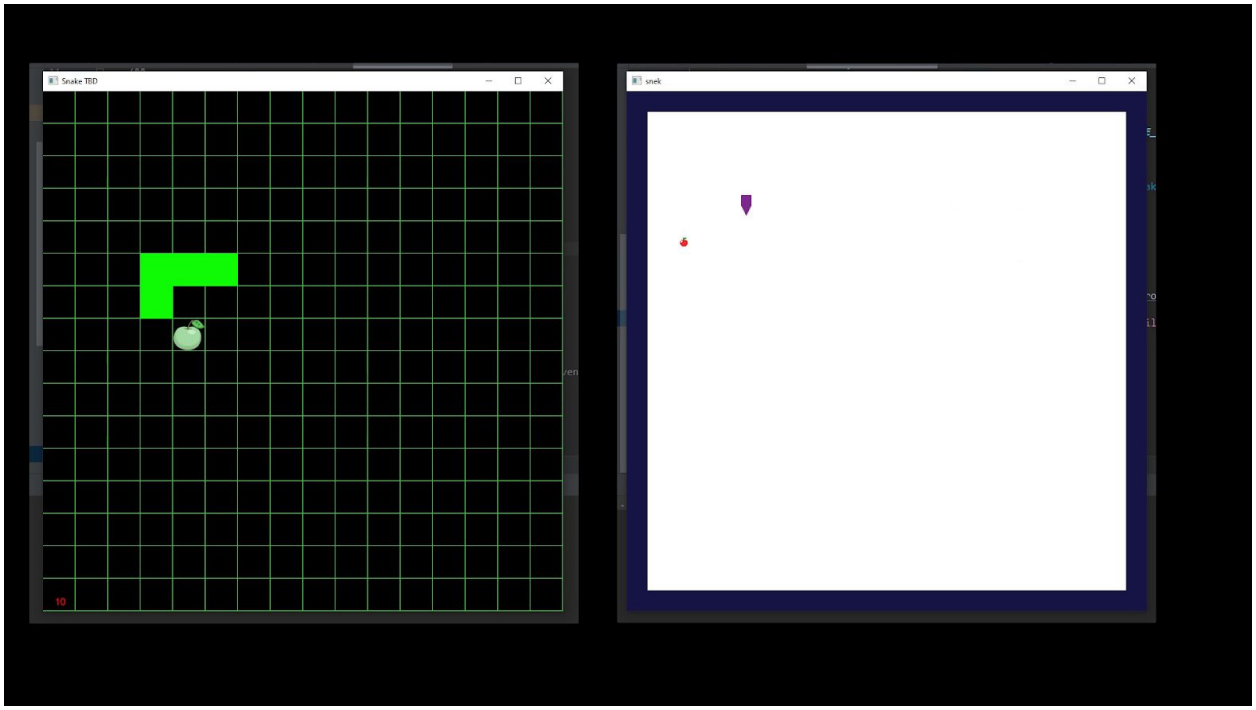
In our final product, all but one of the initial “Must” requirements are implemented. The requirement is: “The player shall be able to see their personal highscore (only one highest score);”. We decided to move this requirement to our Should list and ended up not implementing it in our final version of the game.
Then we worked on our Should requirements. We implemented more than half of them from the initial list and made some changes:

- “The game shall have 3 or more different snake skins from which the player can choose from;” we changed to “The snake shall have different skins for different levels of the game;” because we decided to implement the change snake skins together with changing levels.
- We added: “The game map shall have different tile textures (walls, obstacles, background) for different levels of the game;”, because we decided to change the appearance of the game for each level.
- “The interactive elements shall have textures (shall not be just solid color blocks);” was moved from Could to Should and implemented because we decided that it is a valuable feature.
- To the list of interactive elements we added “a power-up that increases the score of the game” because we decided to create a double score power-up and “a special food object that grows the snake twice and gives twice more points than the simple food object” because we decided to create a golden apple object with this effect;
- “The game shall automatically generate all interactive elements (food, downgrades, power-ups, obstacles, etc.) on the arena during the gameplay at a predefined frequency;” was changed to “The game shall automatically generate all edible interactive elements (food, downgrades and power-ups) on the arena during the gameplay at a predefined frequency;” because we decided to have the position of obstacles fixed throughout the gameplay.

We didn’t implement anything from the “Could” or “Won’t” lists and didn’t change anything.

The updated list of requirements, with the parts that are done clearly marked can be found in our Wiki page - > Assignment1 -> Exercise 1 Requirements (updated)

Why we changed the map?



In the older version of the map (left) we were pretty limited with what textures and shape the cells took on. We'd manually draw out a grid pattern every iteration and have it be of a predefined color. Same goes for the snake, it was rendered as square shapes only.

But this new map (right) gave us the possibility to have our own textures for the map tiles & different textures and shapes for the snake itself. All this we could create on our own, which we also did. We have two different types of GameMaps. TiledGameMap and CustomGameMap .

Since TiledGameMap was initialized once at creation and uses libGdx's TiledMap class, it is very efficient when working with libGdx. With the help of an application called "Tiled" we can essentially draw out obstacles for our level (level 2 and 3 are examples of this).

The CustomGameMap was initially made in the hopes that we could randomly generate obstacles using some kind of placement function. Unfortunately we ran out of time and funding, but we still wanted to

show it, so our first level is a ‘default’ game map. This new versions of our map gave our game customizability, efficiency and more options to expand in the future to make our game even more sophisticated.

References

Images

The images we used in the game were selected from Google Images using the “Labeled for noncommercial reuse with modification” usage rights filter. Links to those images are provided in code comments in the places they were used. Other images that were used are:

- Background image: https://i.pinimg.com/originals/39/c3/cc/39c3cc0490fa3e939cdb3d0f69b154b6.jpg
- Leaderboard background image: https://wallpaperaccess.com/full/1170751.png
- Frog picture: https://img.favpng.com/8/17/6/pepe-the-frog-vector-graphics-pixel-art-clip-art-png-favpng-9HWYPrvNZJjTK20qQPNKUCy5y.jpg
- Mario mushroom: http://pngimg.com/uploads/mario/mario_PNG75.png

Tutorials

Here are links to tutorials that inspired parts of our code:

- Database: https://www.sqlitetutorial.net/sqlite-java/
- For map : https://www.youtube.com/playlist?list=PLrnO5Pu2zAHIKPZ8o14_FNIp9KVvwPNpn
- For states: https://youtu.be/24p1Mvx6KFg

Exercise 2 - Refactoring

Part 1

The thorough analysis of the software metrics, using CodeMR plugin for Intellij, showed that half our "modules" are well structured and have low or medium-low metrics scores that defined issues in the projects. In particular, those are *Models*, *Services* and *Utils*.

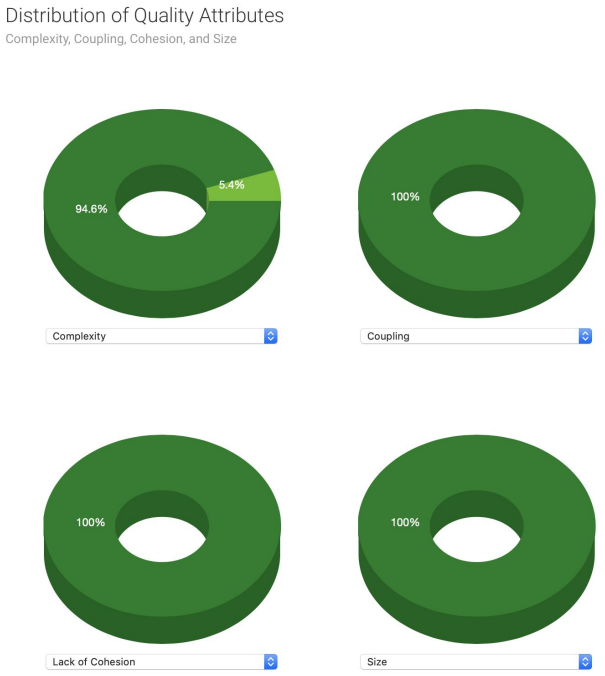


Figure 1: *Models* metrics.

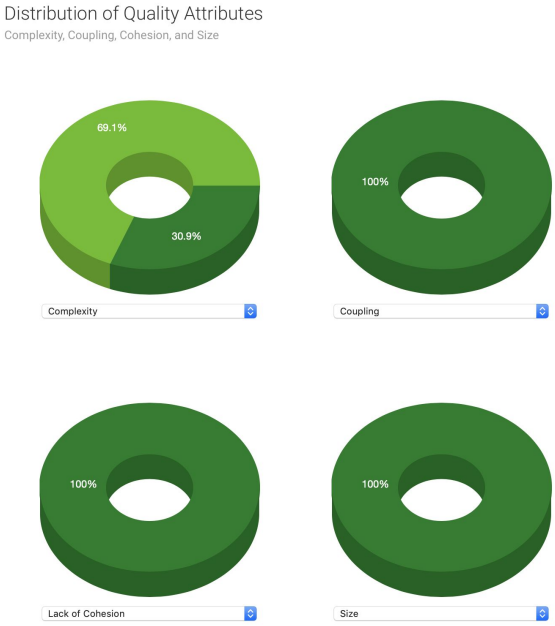


Figure 2: *Services* metrics.

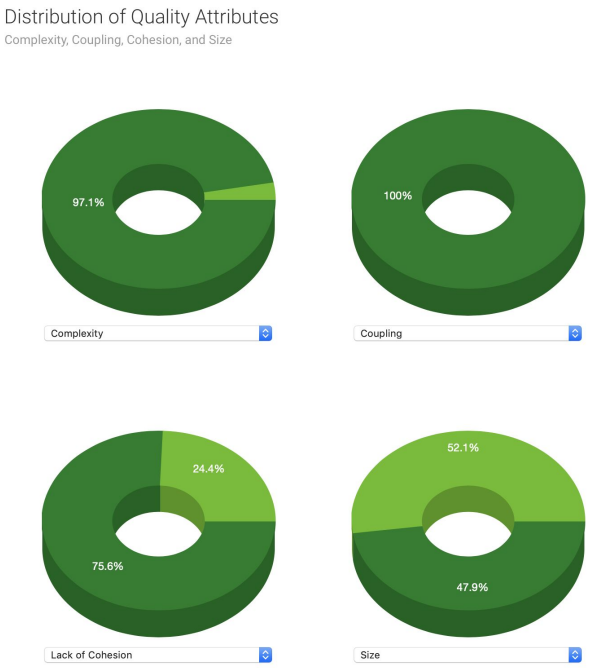


Figure 3: *Utils* metrics.

Other modules, however, breach threshold in one or multiple quality attributes. In the tool we used, they are identified as medium quality or worse. Most of the cases, the problem is in the lack of cohesion among methods. These classes have low level of cohesion, which means that the content of those classes have small relation between each other, which in our case is measured based on parameters the methods have.

More detailed analysis showed that *Entities* module contains only one problematic class, while all other suffy the quality threshold. In particular, this class is *SnakeBody*, and the problem is again in the lack of cohesion.

ID	CLASS	COUPLING	COMPLEXITY	LACK OF COHESION	SIZE	LOC	COMPLEXITY	COUPLING	LACK OF COHESION	SIZE
1	SnakeBody					96	low-medium	low-medium	medium-high	low-medium

Figure 4: SnakeBody Detailed Metrics.

The module States has leaks among multiple classes and multiple attributes, with prevailing problem in coupling.

Figure 5: Module *States* metrics.

And finally, the module *World* has also multiple leaks, prevailing in LOC again.

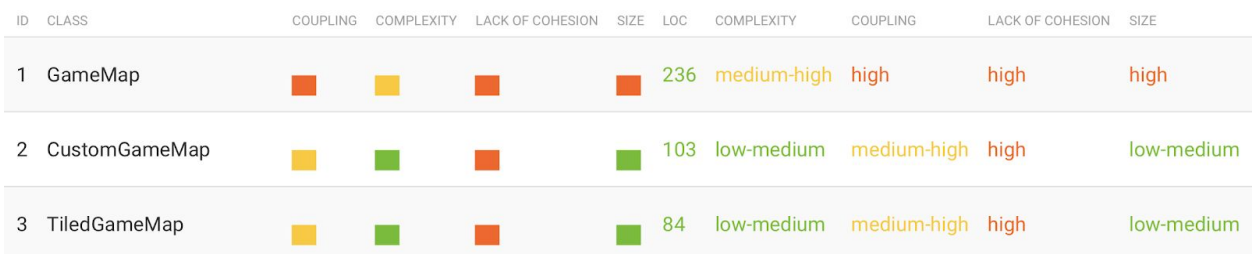


Figure 6: Module *World* metrics.

Another tool that helped us to improve our code quality unexpectedly ended up being the Jacoco coverage report. The leaks and fixes are shown later in part 2.

The final tool that helped us to reduce code-smell was the built-in Intellij tool that discovers duplicate code. About it later in part 2 as well.

Part 2

1., 2. SnakeBody

Method-wise: *updateBodyPartsPosition* and *renderSnake*.

The first refactored methods did not come up from software metrics from CodeMR, but rather from Test Coverage ones. Although the test coverage usually is not explicitly used for refactoring purposes, during its analysis on what tests were missing for complete coverage, it helped us to figure out the leaking place in the code itself. So, it was not about missing tests, but rather code quality.

```
/**
 * Updates the position of each body part.
 */
@SuppressWarnings("PMD.DataflowAnomalyAnalysis")
public void updateBodyPartsPosition(Coordinate coordinate) {
    if (bodyParts.size() > 0) {
        for (BodyPart bp : bodyParts) {
            int currX = bp.getCoordinate().getCoordinateX();
            int currY = bp.getCoordinate().getCoordinateY();
            bp.updateBodyPartPos(coordinate);
            coordinate = new Coordinate(currX, currY);
        }
    }
}
```

```
/**
 * First renders the head of the snake as a rectangle.
 * Then loops through the bodyparts and renders those.
 *
 * @param shapeRenderer - ShapeRenderer object
 */
@SuppressWarnings("PMD.DataflowAnomalyAnalysis")
public void renderSnake(ShapeRenderer shapeRenderer) {
    shapeRenderer.begin(ShapeRenderer.ShapeType.Filled);
    shapeRenderer.setColor(new Color(Color.GREEN));
    int x = this.headCoord.getCoordinateX();
    int y = this.headCoord.getCoordinateY();
    shapeRenderer.rect(x, y, CELL_SIZE, CELL_SIZE);
    if (bodyParts.size() > 0) {
        for (BodyPart bp : bodyParts) {
            Coordinate bodyCoord = bp.getCoordinate();
            shapeRenderer.rect(bodyCoord.getCoordinateX(), bodyCoord.getCoordinateY(),
                    CELL_SIZE, CELL_SIZE);
        }
    }
    shapeRenderer.end();
}
```

The figures above shows that in those 2 methods, 1 branch in both is missing from coverage. After investigating it further and also receiving some comments on this from TA, we figured out that it is not only impossible to write test that would cover this branch, but it would be meaningless. Thus, it was decided to remove these checks. This resulted in increased coverage metrics and improved code quality.

```java
    /**
     * Updates the position of each body part.
     * @param coordinate coordinate location next body part.
     */
    @SuppressWarnings("PMD.DataflowAnomalyAnalysis")
    public void updateBodyPartsPosition(Coordinate coordinate) {
        for (BodyPart bp : bodyParts) {
            int currX = bp.getCoordinate().getCoordinateX();
            int currY = bp.getCoordinate().getCoordinateY();
            bp.updateBodyPartPos(coordinate);
            coordinate = new Coordinate(currX, currY);
        }
    }
```

```java
for (int i = 1; i < getBodyParts().size(); i++) {
    BodyPart part = getBodyParts().get(i);
    Coordinate curr = part.getCoordinate();
    int a = curr.getCoordinateX();
    int b = curr.getCoordinateY();
    batch.draw(textureRegions[0][1],
            a * Sizes.TILE_PIXELS,
            b * Sizes.TILE_PIXELS,
            (float) Sizes.TILE_PIXELS / 2, (float) Sizes.TILE_PIXELS / 2,
            Sizes.TILE_PIXELS, Sizes.TILE_PIXELS, 1, 1,
            rot, true);
}
```

We count these for 2 different method-wise refactorings, since they were 2 methods, although in the same class.

3., 4. States

Class-wise

The built-in Intellij tool has helped us to identify other unpleasant spots. In the States module, there were multiple duplicates of code.

```
/**
 * This dialog box is shown when the user wants to start the game.
 */
public void gameRulesDialog() {
    Skin uiSkin = new Skin(Gdx.files.internal( path: "assets/cloud-form/skin/cloud-form-ui.json"));
    Dialog dialog = new Dialog( title: "Rules", uiSkin, windowStyleName: "dialog") {
        public void result(Object obj) {
            System.out.println("result " + obj);
        }
    };
    dialog.text("Use 'WASD' to move the snake.\n"
            + "Eat food to grow your snake.\n"
            + "Game will end when you either hit yourself or the wall.\n"
            + "Press p to pause the game.\n"
            + "Press q to quit the game.\n"
            + "Enjoy :) ");
    dialog.button( text: "OK", object: true);
    dialog.show(stage);
}
```

```
@Override
public void render(SpriteBatch batch) {
    Gdx.gl.glClearColor( red: 0, green: 0, blue: 0, alpha: 1);
    Gdx.gl.glClear(GL20.GL_COLOR_BUFFER_BIT);
    batch.begin();
    stage.act();
    stage.getBatch().begin();
    stage.getBatch().draw(background, x: 0, y: 0, width: 800, height: 800);
    stage.getBatch().end();
    stage.draw();
    batch.end();
}
```

```
/**
 * Sets title of login screen.
 */
private void initTitle() {
    BitmapFont bitmapFont = new BitmapFont(Gdx.files.internal( path: "assets/font.fnt"));
    Label.LabelStyle labelStyle = new Label.LabelStyle(bitmapFont,
            new Color( r: 0, g: 255, b: 0, a: 1));
    title = new Label( text: "Lil' Snake", labelStyle);
    title.setSize( width: 600, height: 120);
    title.setPosition( x: 100, y: 550);
    title.setFontScale(3);
    title.setAlignment(Align.center);
    stage.addActor(title);
}
```

Multiple classes were affected. The solution was to extract those methods into separate utility objects *GameRulesDialog* and *RendererHandler* that would be accessed when needed.

```
@Override
public void render(SpriteBatch batch) {
    RendererHandler.render(batch, stage, background);
}
```

Example of usage of a new extracted utility method.

For gameRulesDialog, there were 2 classes: *MenuState* and *PausedState*.
For initTitle method, there were 2 classes: *LoginState* and *SignUpState*.
For render, all of the states had this duplicated code.

After refactoring, we reduced the size of the classes that were affected by the duplicate code. Of course, we increased maintainability, because if the rule needs to be changed, not it is changed only in one class. The warning due to duplicate code has disappeared after the refactoring.

We counted this refactoring as for 2 different cases, for *gameRulesDialog+render* and *initTitle+render*, because these are two different groups of classes that required refactoring, although the cause is similar. Notice, we didn't count *render* refactoring into 3rd case, because it was present in both groups.

5. Lines of Code

Since we did not have much to refactor, we reduced the length of some methods.

SignUpState

Originally:

InitSignUp()	59 lines of code

Split into :

InitSignUpUsername()	10 lines of code
InitSignUpPassword()	11 lines of code
InitSignUpButton()	28 lines of code

LoginState

Originally:

initLogin(): 47 lines of code

Split into:

initLogin(): 9 lines of code

initButtons(): 25 lines of code

initPasswordfield(): 13 lines of code

In a few classes, some skins or labels were made several times in the methods, so an attribute was made to avoid this:

SignUpState and LoginState

Since methods were split, private attributes were made for the username TextField and the password TextField so that in other methods the password and username were accessible.

A certain labelstyle for a pink small label was made several times throughout the methods, thus one private attribute was made for this labelstyle.

MenuState

Skin with same texture was made in several methods, thus a private attribute made for the skin, so it did not have to be made in several methods.

GameOverState

A labelstyle for a green big label was made several times throughout the methods, thus one private attribute was made for this labelstyle.