ASSIGNMENT 4

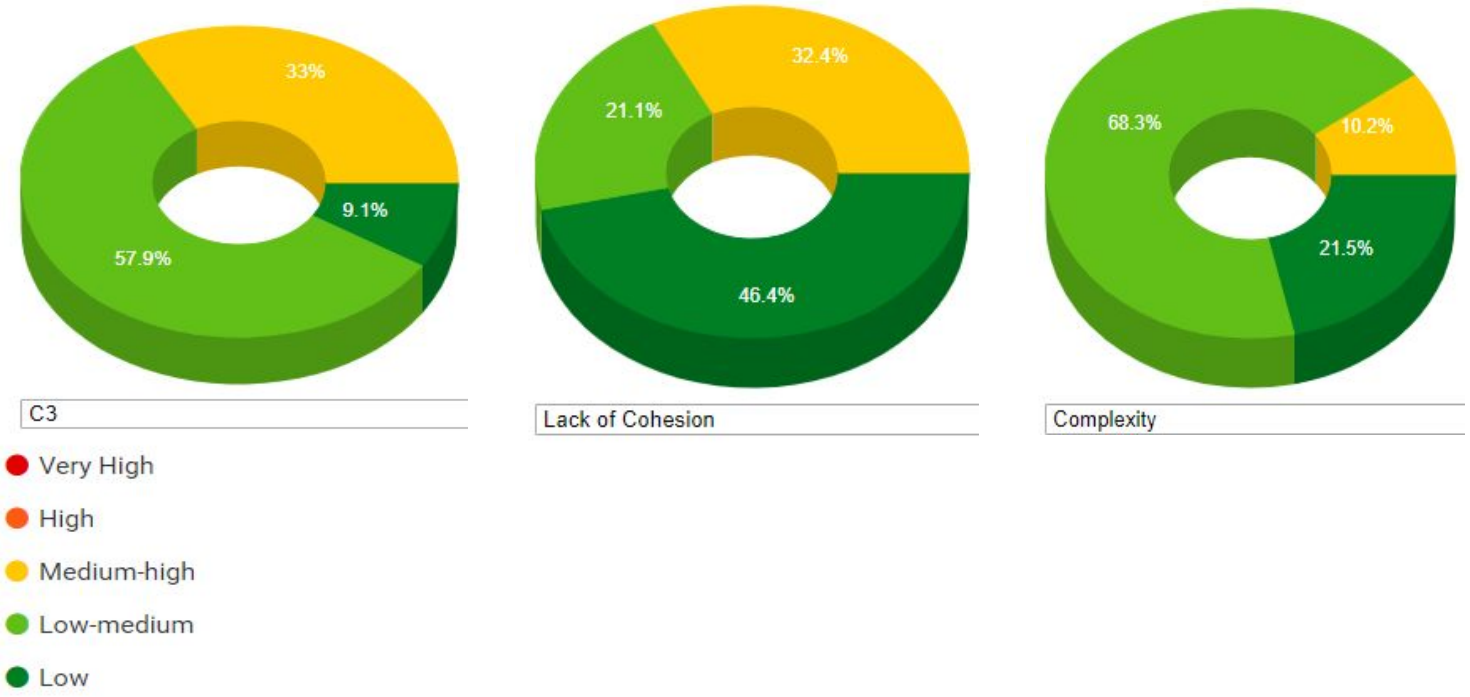
GROUP 8

24.01.2020

Exercise 2 - Refactoring

1.

We have chosen for 2 different metrics calculator, CodeMR which gives a visual representation of the calculated metrics and MetricsReloaded which is simplistic but makes it easy to identify which metrics are above a certain threshold. To identify potential problematic metrics we first evaluated which metrics we believed still needed to be improved on in certain classes. For example, in several of the Gui classes, we realized that there are methods which became too complex and needed to be refactored to improve maintainability. For class level metrics we realised that some of the classes are taking on too much responsibility and are not cohesive enough.



To identify the thresholds which allow us to identify these code smells we made use of the visual feedback provided by CodeMR. When a class has at least a threshold of Medium-High we considered refactoring that class and aimed to at least improve the method/class in an appropriate manner. However, these quality attributes did not always portray a code smell, for example, the launcher class (which is used to switch screens) is considered highly coupled. However, the launcher class needs to be passed around different Gui classes, if we were to decrease the coupling by constricting the Gui to a single or a small amount of classes then we would severely decrease the cohesion and maintainability of the Gui. Therefore we looked at the computed metrics on a case-by-case basis and decided for each individual class whether or not a refactoring was appropriate.

2.

Method Level Refactors

1 -

Before

Name	#MC
MoveSnake(float, float, Snake)	22

After

Name	#MC
MoveSnake(float, float, Snake)	15

Documentation

By applying the extract method refactor, we can see that the complexity of the MoveSnake method has decreased. We have moved the statements responsible for eating the snack and eating a Powerup to different methods. This will make it easier to understand the code but also this will increase the maintainability of the code, if a change is required, only the private method will have to be changed.

2 -

Before

Name	#MC
MoveSnake(float, float, Snake)	37

After

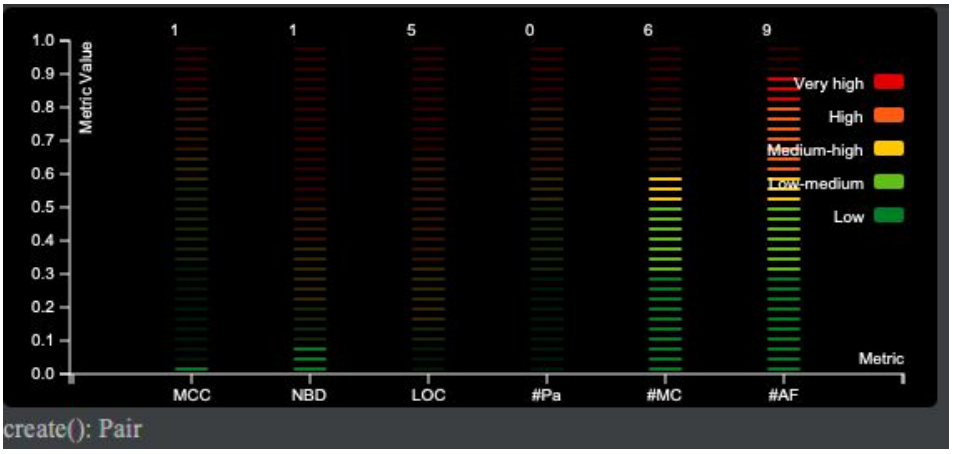
Name	#MC
MoveSnake(float, float, Snake)	13

Documentation

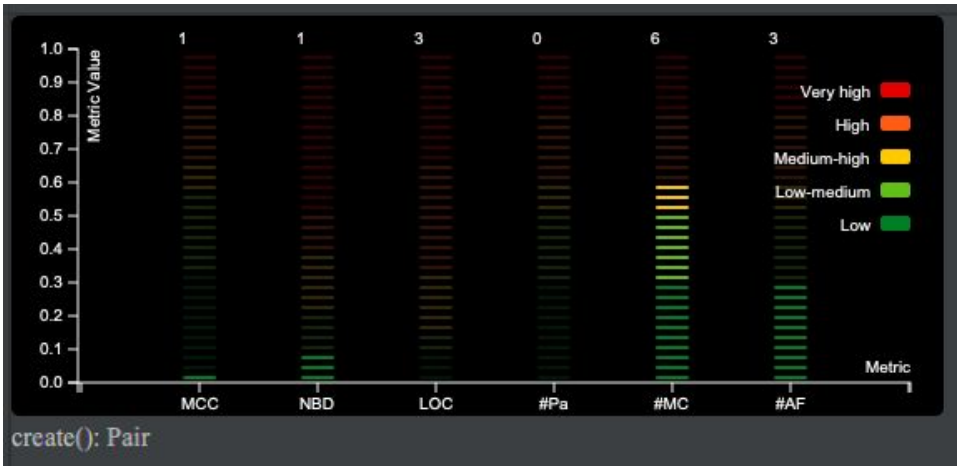
The method for creating the actors of the Gui contained several if-statements and therefore became too complex. By applying the extract method refactor, we can ensure that the method stays readable and doesn’t become too complex to understand. This will make the code easier to maintain.

3 -

Before



After

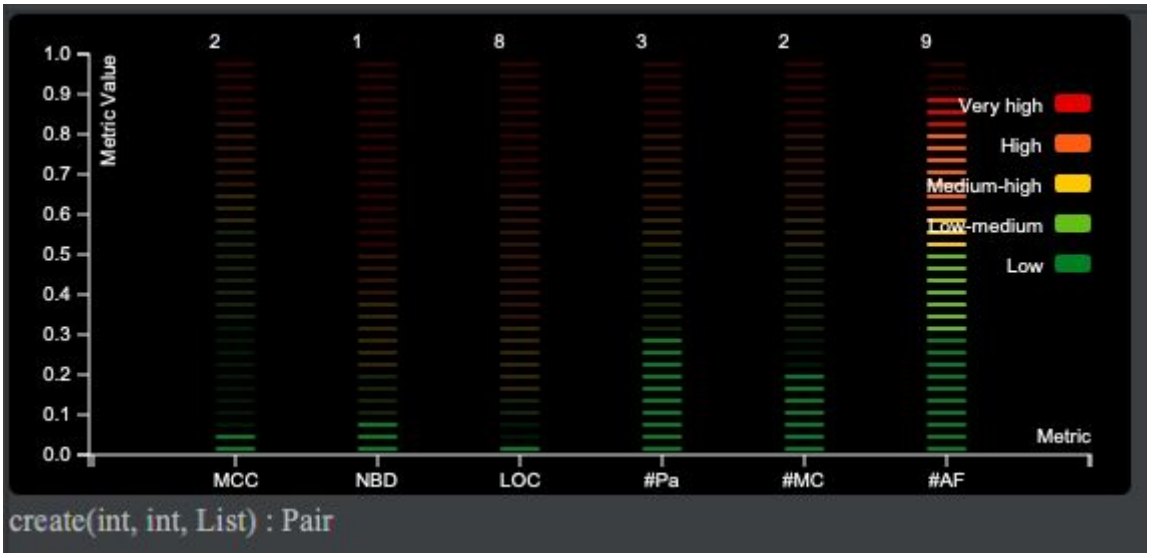


Documentation

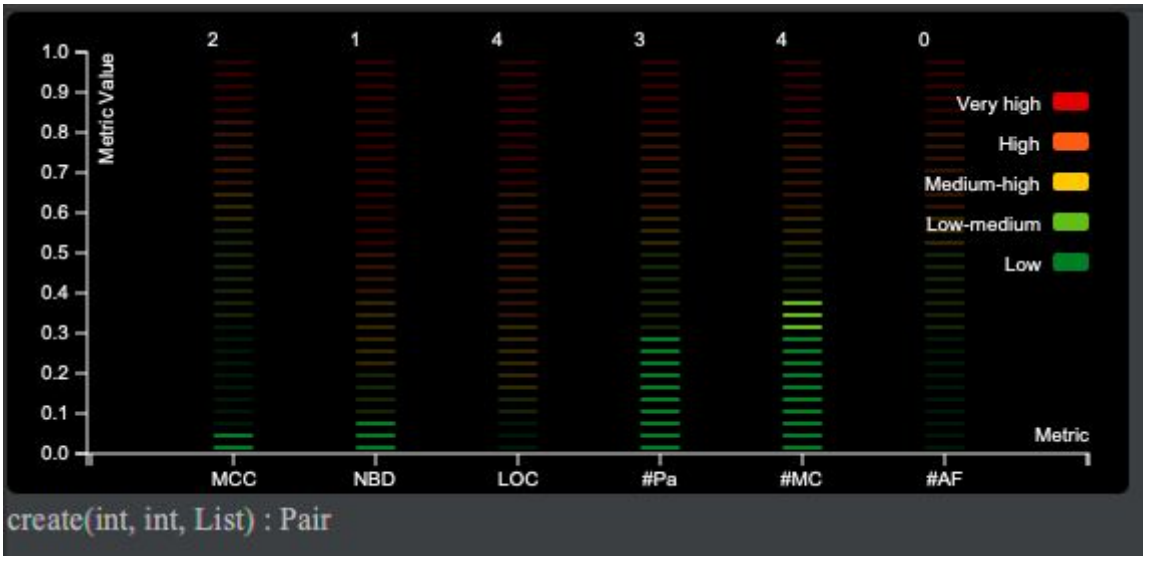
The method create() had high number of accessed fields, as we were calling the same method twice and doing the same operation (minus and division). We've applied extract method refactoring technique, we made an extra method for calculating the x and y value of the square.

4 -

Before



After



Documentation

The method create(int, int, List) had high number of accessed fields as we were calculating for the new x and y values for two squares; normal and coloured.

We've applied extract method technique, made two separate methods for calculating the values.

5 -

Before

```
switch (max) {
    case DOWN:
        game.moveDown(player);
        break;
    case LEFT:
        game.moveLeft(player);
        break;
    case RIGHT:
        game.moveRight(player);
        break;
    default:
        game.moveUp(player);
        break;
}
```

method	ev(G)	iv(G)	v(G)
snake.gui.gamescreens.SinglePlayerGameScreen.move(Snake,Direction)	2	3	6

After

```
public abstract void moveDirection(Game game, Snake player);
```

method	ev(G)	iv(G)	v(G)
snake.gui.gamescreens.SinglePlayerGameScreen.move(Snake,Direction)	1	2	2

Documentation

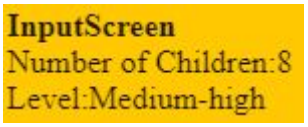
The method for moving the player based on the input of the user contained a switch statement. This is a violation of the Open-Closed principle since each time a new condition is created, the developer needs to update the switch statement (e.g. if we wanted to add diagonal directions). By using polymorphism we avoid this issue and we can ensure that each of the Direction classes implements this behaviour individually.

This refactoring also reduces the cyclomatic complexity of the method, making it less complex and easier to understand for the developers.

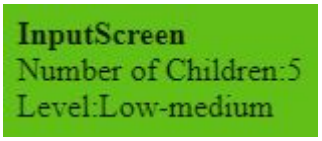
Class Level Refactors

1-

Before



After



Documentation

By applying the extract class + move method refactoring we reduce the number of children. By moving the methods which are required for authentication into a seperate class. Furthermore, the cohesion of the InputScreen has also increased. This places less responsibility on the InputScreen class and the refactor ensures that the maintainability of the code is improved.

2 -

Before

After

Documentation

The Snake class had a number of variables which were not related to the snake but rather to another class, which we called the Player class. By extracting a class, we increase the cohesion of the snake while also making the class understandable. The class also becomes more user-friendly and less complex therefore improving the maintainability.

3 -

Before

class	WMC	OCavg
snake.EmailSender	11	1,10

After

class	WMC	OCavg
snake.EmailSender	7	1,17

Documentation

The EmailSender has too much responsibility, especially considering that it was both responsible for creating and sending the message. By creating an email creator, we decouple the email sender from the task of creating the message. This puts less strain on the EmailSender class and makes sure that it remains reusable. By extracting this class, we also ensure that our code remains testable and is less prone to errors.

4 -

Before

class	CBO	DIT	LCOM
snake.gui.StyleUtility	9	1	16

After

class	CBO	DIT	LCOM
snake.gui.StyleUtility	9	1	8

Documentation

The style utility class contained a lot of calls to the same method of the libgdx framework. This lead to code duplication which decreased the maintainability of the code. By using the extract method refactor, we ensure that the class implements a single responsibility and is therefore more cohesive while also avoiding code duplication. In the metrics we can see that the LCOM metric has improved, indicating that the cohesion between methods has improved as well.

This change has also drastically reduced the number of fields of the class. Keeping everything in a hashmap rather than in separate fields makes sure that the class does not become too long and hard to interpret. Furthermore, by using a hashmap there cannot be more than 1 instance of one of the elements which reduces the amount of space necessary to run the application.

5 -

Before

class	RFC	WMC
snake.gui.gamescreens.SinglePlayerGameScreen	72	41
snake.gui.gamescreens.MultiPlayerGameScreen	14	7
Total		**48**
Average	43,00	24,00

After

class	RFC	WMC
snake.gui.gamescreens.SinglePlayerGameScreen	68	28
snake.gui.gamescreens.MultiPlayerGameScreen	12	6
Total		**34**
Average	40,00	17,00

Documentation

The SinglePlayerGameScreen had too much responsibility, it was rendering the game screen while also checking for the user input in terms of directional keys. By dividing the functionality into two classes, the SingePlayerGameScreen class for the actual rendering and the InputReader class for the reading of input, we ensure that the single responsibility principle is followed.

In terms of metrics, we can see that the Response for Class metric has decreased indicating that the class has become less complex. The WMC has also decreased indicating a further decrease in the complexity of the class and showing that the class takes on less responsibility indicating better separation.