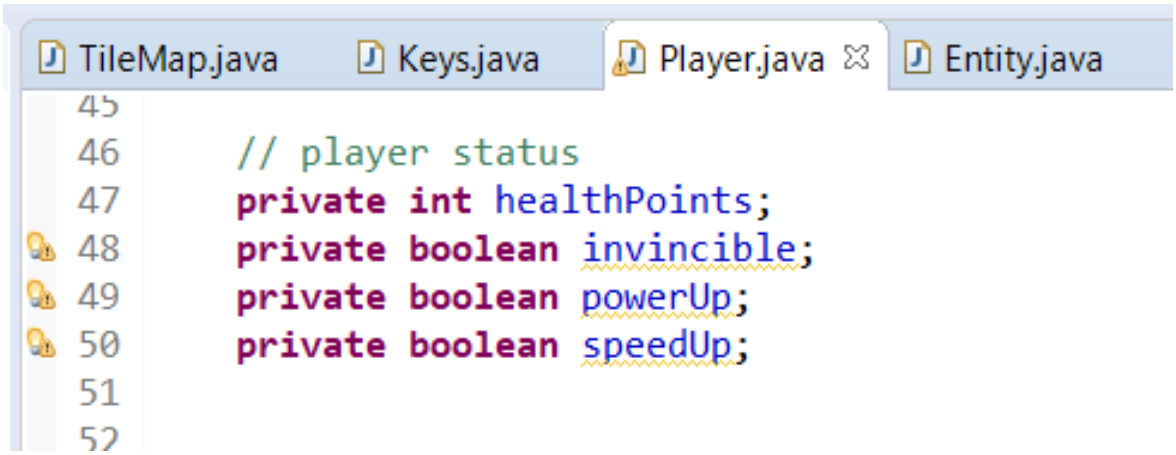
Name : Aimi Syahirah Sazali

Student ID: 20014176

Module: Software Maintenance

Coursework: Diamond Hunter

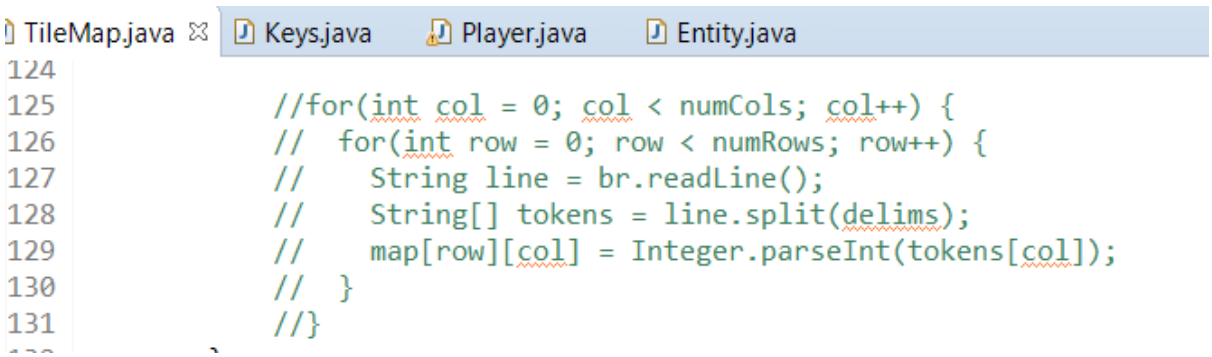
1. Dead Code



1(a)

A variable, parameter, field, method or class no longer used is called a dead code (“refactoring.guru”, n.d.). A dead code can be a result of changing the software requirements and making corrections without cleaning up the old code.

In 1(a), there are three unused variables in the class Player. The variables invincible, powerup and speedup are declared but not used in the rest of the class.



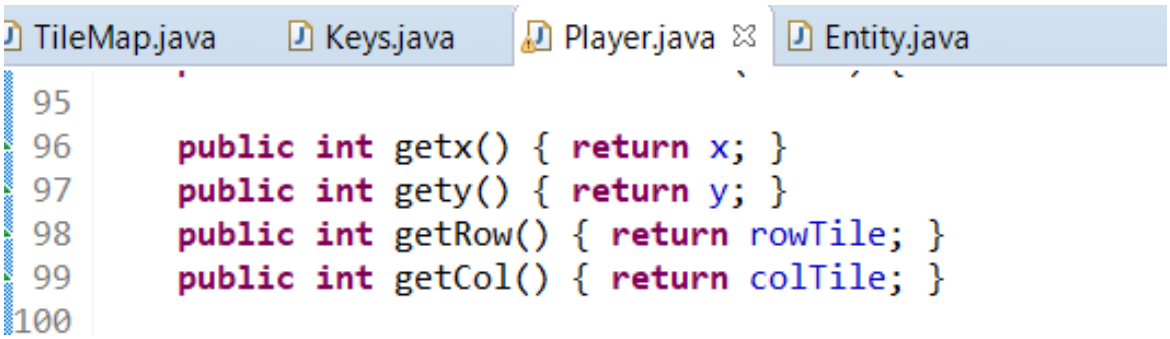
1(b)

A dead code can also be found in TileMap.java as shown in 1(b). This fragment of code is no longer used, thus put into a comment block instead of removed properly.

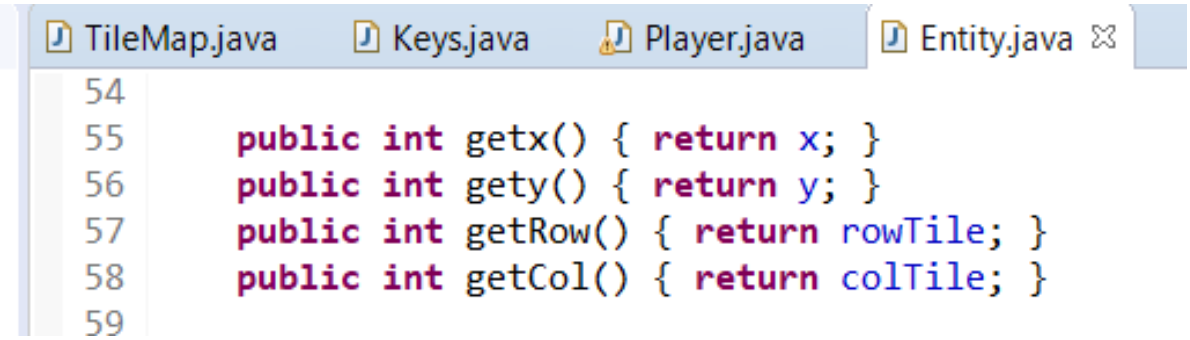
This can be solved by removing the lines 48, 49 and 50 in TileMap.java and lines 125 until 131 in Player.java to reduce the code size.

2. Duplicate Code

Duplicate Code occurs when fragments of code are similar but exists in different parts of the program (“refactoring.guru”, n.d.). This is mostly due to habitual behavior of the developer, or a lack of understanding of the system under maintenance, the problem or the solution (Kamiya, 2002).



2(a)



2(b)

In this example represented by 2(a) and 2(b), Entity is the base class and Player is the sub class, meaning that Player is inheriting the properties and methods of Entity. There is no need to declare the four methods again in the class Player as they are already

declared in Entity. Instead, this similar fragment of code in the class Player can be removed while the identical fragment remains in the class Entity. This will simplify the structure of the code and make it shorter, making the code cheaper to support.

3. Sequence of If Statements

TileMap.java | Keys.java | Player.java | Entity.java

```
30
31    public static int keySet(int i, boolean b) {
32        if(i == KeyEvent.VK_UP) keyState[K1] = b;
33        else if(i == KeyEvent.VK_LEFT) keyState[K2] = b;
34        else if(i == KeyEvent.VK_DOWN) keyState[K3] = b;
35        else if(i == KeyEvent.VK_RIGHT) keyState[K4] = b;
36        else if(i == KeyEvent.VK_SPACE) keyState[K5] = b;
37        else if(i == KeyEvent.VK_ENTER) keyState[K6] = b;
38        else if(i == KeyEvent.VK_ESCAPE) keyState[K7] = b;
39        else if(i == KeyEvent.VK_F1) keyState[K8] = b;
40        return 0;
41    }
```

3(a)

As shown in 3(a), there is a sequence of If statements in Keys.java that can be simplified into ‘switch’ as there are many ‘else if’ statements to improve code organization.

The changes made can be seen in 3(b).

TileMap.java | Keys.java | Player.java | Entity.java

```
29      public static int K8 = 7;
30
31      public static int keySet(int i, boolean b) {
32          switch (i) {
33          case KeyEvent.VK_UP:
34              keyState[K1] = b;
35              break;
36          case KeyEvent.VK_LEFT:
37              keyState[K2] = b;
38              break;
39          case KeyEvent.VK_DOWN:
40              keyState[K3] = b;
41              break;
42          case KeyEvent.VK_RIGHT:
43              keyState[K4] = b;
44              break;
45          case KeyEvent.VK_SPACE:
46              keyState[K5] = b;
47              break;
48          case KeyEvent.VK_ENTER:
49              keyState[K6] = b;
50              break;
51          case KeyEvent.VK_ESCAPE:
52              keyState[K7] = b;
53              break;
54          case KeyEvent.VK_F1:
55              keyState[K8] = b;
56              break;
57
58          }
59          return 0;
60      }
61
```

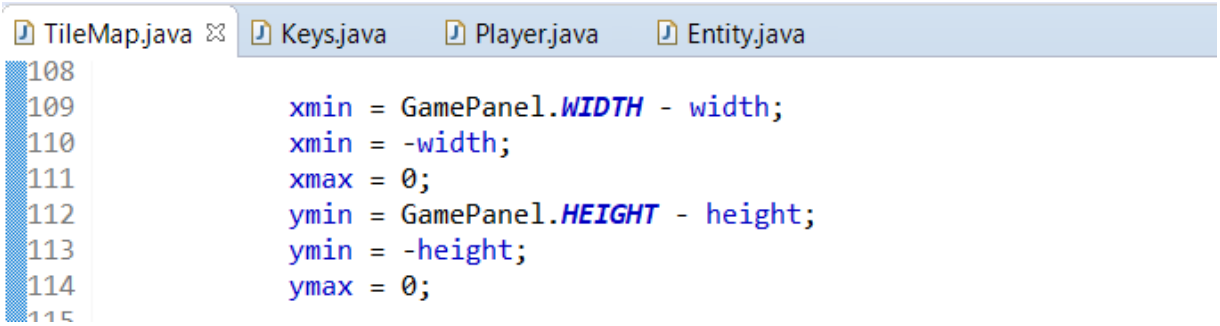
3(b)

4. Code Redundancy

Code redundancy is an example of code smells. Code smells are not easily detected by the compiler because they are different from syntax errors or any other errors. They cannot be identified because they do not stop the program from running but they are a symbol of bad program design (Bavota et al., 2015).

One example can be taken from TileMap.java whereby the variables ‘xmin’ and ‘ymin’ are assigned two times, as shown in 4(a). Since they are re-assigned, the first

assignments are not necessary. Therefore, lines 109 and 112 should be removed from the program.



4(a)

5. Long Method

A long method is a result of always adding something to a method but nothing is taken out. This is because it is often easier to add to an existing method rather than creating a new one (“refactoring.guru”, n.d.). When we keep adding lines to a method, it results in a long complex method that is hard to simplify.

From screenshots 5(a) to 5(d), in Player.java, we can see that the same fragment of code is repeated throughout the method update() in different ‘if’ statements. This causes the same method to contain too many lines.

To simplify this, the code fragments to ‘check if on water’ and ‘if going from land to water’ can be moved outside of the ‘if’ statements and placed in the beginning of the method update(), as shown in 5(e). This way, the fragment of code is only written once in the method instead of four times for four different ‘if’ statements. The lines of code for Player.java can be reduced by 36, making it easier to be understood and maintained.

TileMap.java | Keys.java | Entity.java | Player.java

```
154         // set animation
155         if(down) {
156
157             // check if on water
158             boolean current = onWater;
159             if(tileMap.getIndex(ydest / tileSize, xdest / tileSize) == 4) {
160                 onWater = true;
161             }
162             else {
163                 onWater = false;
164             }
165             // if going from land to water
166             if(!current && onWater) {
167                 JukeBox.play("splash");
168             }
```

5(a)

TileMap.java | Keys.java | Entity.java | Player.java

```
176         }
177         if(left) {
178
179             // check if on water
180             boolean current = onWater;
181             if(tileMap.getIndex(ydest / tileSize, xdest / tileSize) == 4) {
182                 onWater = true;
183             }
184             else {
185                 onWater = false;
186             }
187             // if going from land to water
188             if(!current && onWater) {
189                 JukeBox.play("splash");
190             }
```

5(b)

TileMap.java | Keys.java | Entity.java | Player.java

```
198         }
199         if(right) {
200
201             // check if on water
202             boolean current = onWater;
203             if(tileMap.getIndex(ydest / tileSize, xdest / tileSize) == 4) {
204                 onWater = true;
205             }
206             else {
207                 onWater = false;
208             }
209             // if going from land to water
210             if(!current && onWater) {
211                 JukeBox.play("splash");
212             }
213
```

5(c)

TileMap.java | Keys.java | Entity.java | Player.java

```
221         if(up) {
222
223             // check if on water
224             boolean current = onWater;
225             if(tileMap.getIndex(ydest / tileSize, xdest / tileSize) == 4) {
226                 onWater = true;
227             }
228             else {
229                 onWater = false;
230             }
231             // if going from land to water
232             if(!current && onWater) {
233                 JukeBox.play("splash");
234             }
```

5(d)

TileMap.java | Keys.java | Entity.java | *Player.java

```
144                 JukeBox.play("tilechange");
145             }
146         }
147     }
148
149     public void update() {
150
151         ticks++;
152         boolean current = onWater;
153
154         if(tileMap.getIndex(ydest / tileSize, xdest / tileSize) == 4) {
155             onWater = true;
156         }
157         else {
158             onWater = false;
159         }
160         // if going from land to water
161         if(!current && onWater) {
162             JukeBox.play("splash");
163         }
164
```

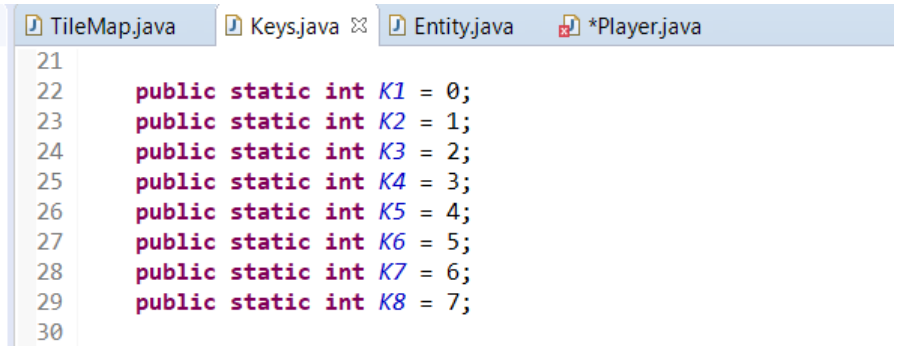
5(e)

6. Missing Comments and Indentation

The use of comments in a programming language makes code more understandable and maintainable (Singh, 2017). Without good code comments, there is no explanation of why things are done. Therefore, there should be minimal comments to understand the purpose of a fragment of code.

For example, in 6(a), there are no comments that help us to understand the purpose of line 22 to 29 in Keys.java. It is not explained why all the variables K1 to K8 are first assigned as 0 to 7, making this fragment of code unintuitive.

There should be at least one comment that states why the variables are assigned from 0 to 7.

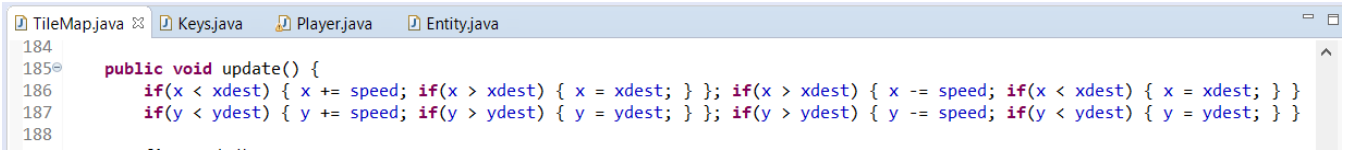


6(a)

The lack of indentations can also be an obstacle when it comes to understanding the code. A good code should be properly structured. This is helpful to understand where a block of code starts and where it ends, so that the logic of the code becomes evident and straightforward (Singh, 2017).

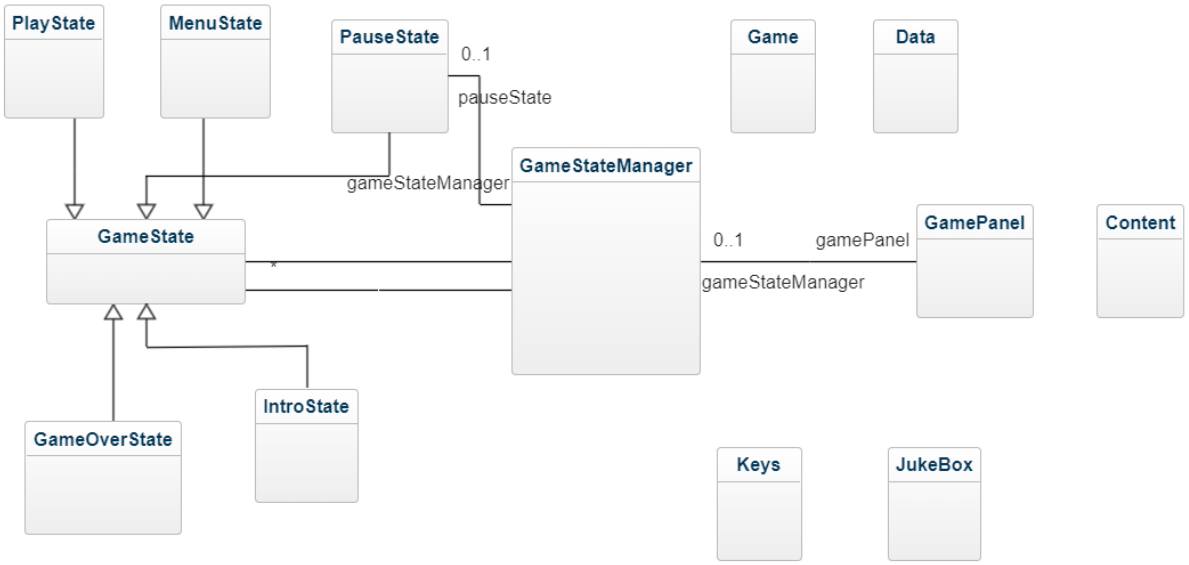
An example of the lack of indentations can be seen in 6(b). The large amount of ‘if’ statements in TileMap.java line 186 and 187 are difficult to understand due to the way it is written. The ‘if’ statements are written in a way that it is difficult to tell whether they are nested loops or separate loops. This can seem like the variables ‘xdest’ and ‘ydest’ are assigned multiple times, which can result in code redundancy.

To simplify this code, we can break up the if statements into multiple lines, making it easy to see all of the nested loops.



6(b)

High Level Class Diagram



GameState is the parent class for five classes, PlayState, MenuState, PauseState, IntroState and GameOverState. They have a “is-a” relationship whereby all the five classes inherits the properties and methods from GameState.

GameStateManager has a 0 to many relationship with GameState. GameState has a 0 to 1 relationship with GameStateManager.

GameStateManager has a 0 to 1 relationship with PauseState.

GamePanel has a 0 to 1 relationship with GameStateManager.

References

Ibrahim, R., Ahmed, M., Nayak, R. (2018). Reducing redundancy of test cases generation using code smell detection and refactoring. Retrieved from https://www.sciencedirect.com/science/article/pii/S1319157818300296

Farrell, J. (2001) Make bad code good. Retrieved from https://www.javaworld.com/article/2075129/make-bad-code-good.html

Singh, N. (2017) Good Code vs Bad Code. Retrieved from https://medium.com/better-programming/good-code-vs-bad-code-35624b4e91bc

Nikishaev, A. (2017) 13 Simple Rules for Good Coding. Retrieved from https://hackernoon.com/few-simple-rules-for-good-coding-my-15-years-experience-96cb29d4acd9

Refactoring. Retrieved from https://sourcemaking.com/refactoring

Endenburg, M. (2016). How code duplication impacts software maintainability. Retrieved from https://www.linkedin.com/pulse/how-code-duplication-impacts-software-maintainability-endenburg

Aladdin, M. (2018). Write clean code and get rid of code smells with real life examples. Retrieved from https://codeburst.io/write-clean-code-and-get-rid-of-code-smells-aea271f30318