Mostafa Osama Mostafa Abdulrazic

015061

School of Computer Science

Software Engineering Year 2

Software Maintenance

Coursework One (1)

Word Count: 980 (Excluding front page, references page and titles)

Introduction:

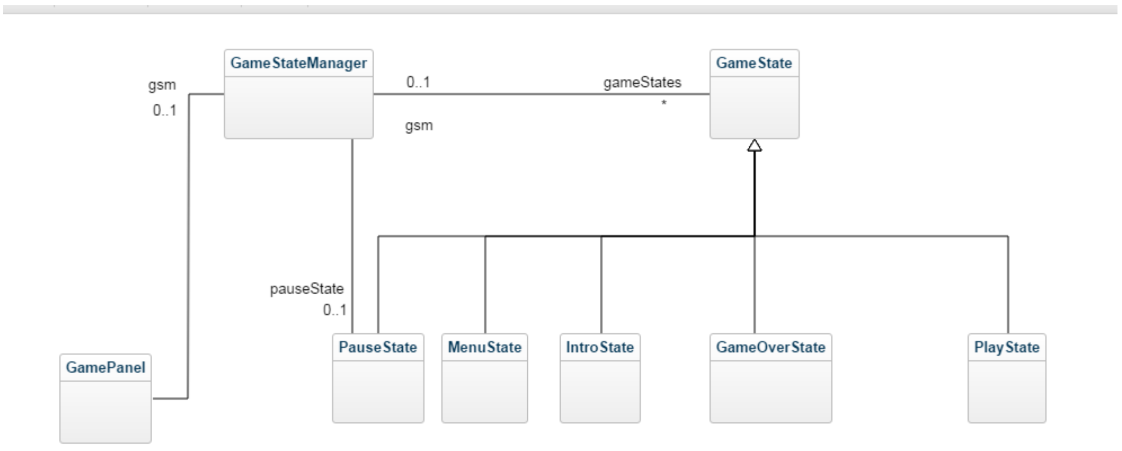
Diamond Hunter is a 2D adventure game that is written in pure Java. The main objective of the game is to collect the 15 diamonds scattered on the tile map in the least amount of time possible. The player controls a figure on the map that is controlled by the keyboard arrows. The player is supposed to move through the map and collect the diamonds while facing different obstacles along the way.

How the code works:

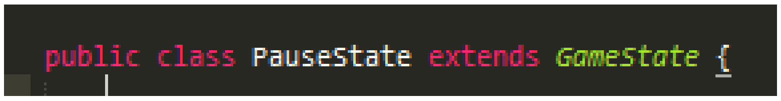
The source code is divided into six packages with each package containing a java source file with the definition of a class inside of it. Each package deals with a major portion of how the game works and all the six packages are interconnected with each to carry out the different functions of the game. For an instance, the Entity package contains all the java files which control the gameplay (picking up of items, diamonds and their positions, movement of player, etc.). The GameState packages contains the java files associated with different states in the game like in the introduction state and when the pause state for example. HUD packages holds one java file which contains the details for the Heads Up Display which is the display area for the player to see the vital statistics which in this case would be the diamond count, items occupied, etc.

The Main package is as the name suggest the main package that would load up a JFrame and put a game panel into it which would basically keep the game moving forward and grabs key events. The Manager package manages the data and content in the game, as well as the keys, music and game states. Finally, TileMap package contains the java files related to the map like loading the map and fixing the size, etc.

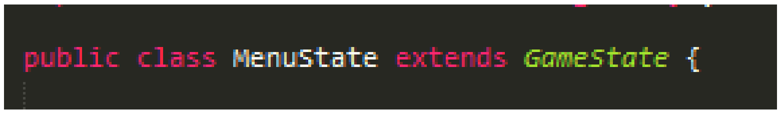
To further clarify how the classes within different game packages actually work together a simple high level class diagram has been put together to explain the relationships between the following game packages (GameState, Main, Main):



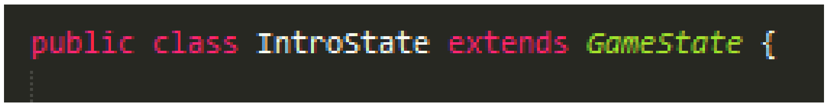
There are two types of relationships in the class diagram, generalization and association respectively. PauseState, MenuState, IntroState, GameOverState and PlayState are all inheriting the class GameState within their own class. Evidence from the code is shown below:



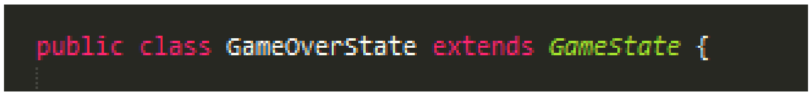
```
public class PauseState extends GameState {
```



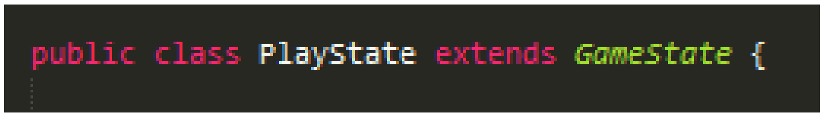
```
public class MenuState extends GameState {
```



```
public class IntroState extends GameState {
```



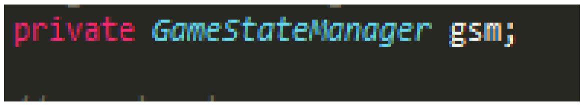
```
public class GameOverState extends GameState {
```



```
public class PlayState extends GameState {
```

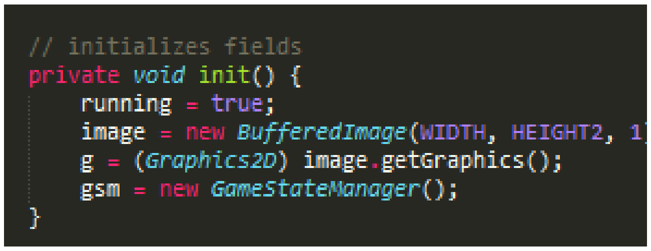
Generalization means that each of these class is part of its superclass which is GameState. Each of these classes is in fact a GameState.

Associations are on the other hand a weak coupling as the associated classes still remain independent of each other at some level. GamePanel is associated with GameStateManager because it has an object (gsm) of class type GameStateManager. Evidence in the code is provided below:



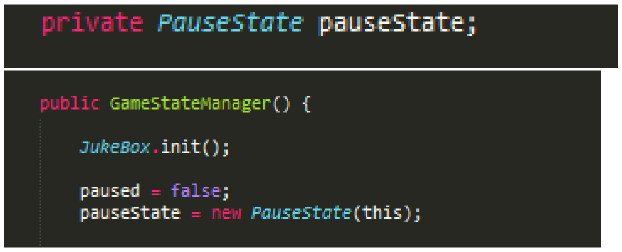
```java
private GameStateManager gsm;
```

The values (0..1) are specifying the minimum boundary and the maximum boundary. In the case where the method init() gets called then only one object would be initialized. Hence the maximum value is 1. However, if the code is not run then no object is initialized and hence the minimum value is 0. Snippet of the code showing the method where gsm would be initialized is below:



```java
// initializes fields
private void init() {
    running = true;
    image = new BufferedImage(WIDTH, HEIGHT2, 1
    g = (Graphics2D) image.getGraphics();
    gsm = new GameStateManager();
}
```

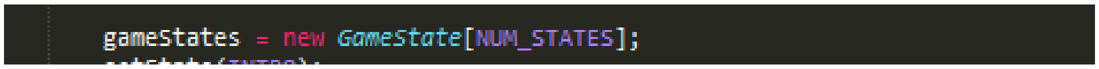
GameStateManager is associated with PauseState as it has an object of class type of PauseState in it. The minimum boundary is 0 and the maximum is 1 as the object is either not initialized if the code doesn’t run or initialized only once. Snippet of the code is below:



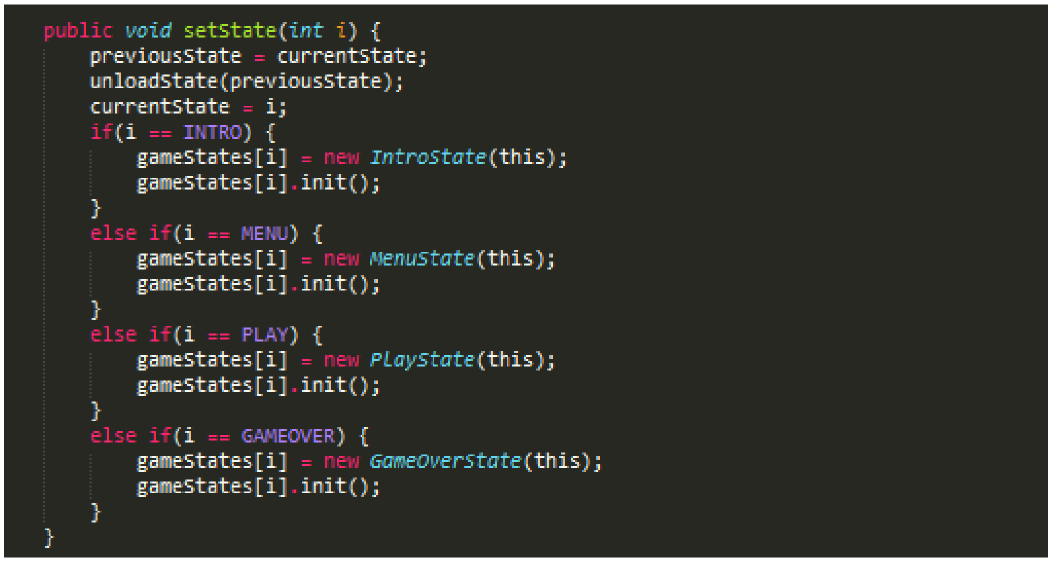
```java
private PauseState pauseState;
```

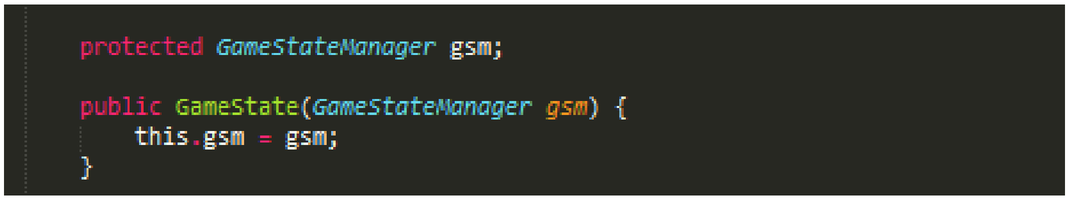
```java
public GameStateManager() {

    JukeBox.init();

    paused = false;
    pauseState = new PauseState(this);
```

GameStateManager and GameState have a bi-directional association between them where each class has an object of the other class type. GameState has a minimum boundary of 0 and a maximum of 1 for the object gsm because the object was only initialized once or none at all. However, the object gameStates of class type GameState has a range of * which means that it could be any number because in the code the object is an array and the index number could be changed to anything. Snippet of the code is below:



```java
    gameStates = new GameState[NUM_STATES];
```


```java
public void setState(int i) {
    previousState = currentState;
    unloadState(previousState);
    currentState = i;
    if(i == INTRO) {
        gameStates[i] = new IntroState(this);
        gameStates[i].init();
    }
    else if(i == MENU) {
        gameStates[i] = new MenuState(this);
        gameStates[i].init();
    }
    else if(i == PLAY) {
        gameStates[i] = new PlayState(this);
        gameStates[i].init();
    }
    else if(i == GAMEOVER) {
        gameStates[i] = new GameOverState(this);
        gameStates[i].init();
    }
}
```

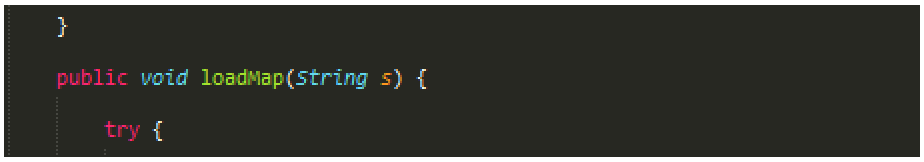

```java
protected GameStateManager gsm;

public GameState(GameStateManager gsm) {
    this.gsm = gsm;
}
```

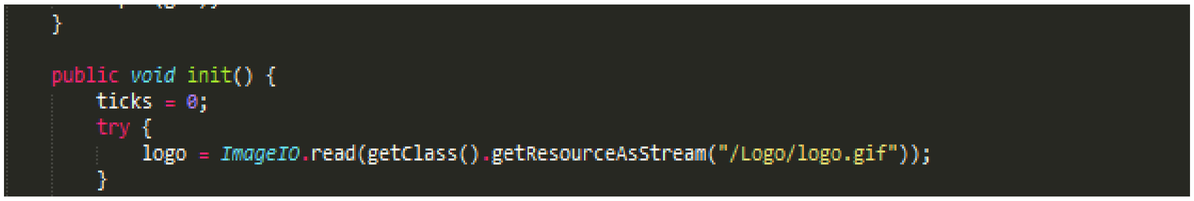
How can the code be improved?

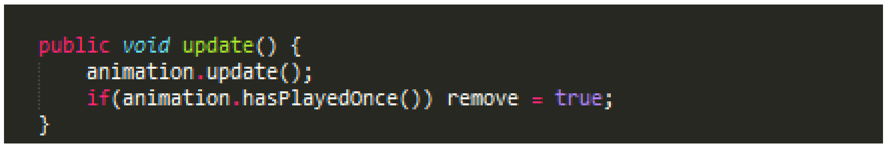
Below is a list of five examples within the code where the code could have been better.

1- Public methods contain no comments:

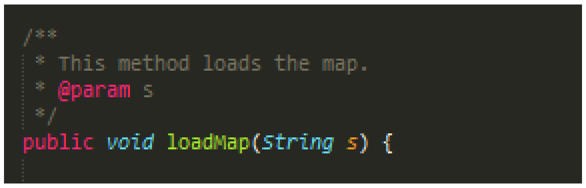
Below are a few screenshots from different java files lacking comment:


```java
}

public void loadMap(String s) {

    try {
```

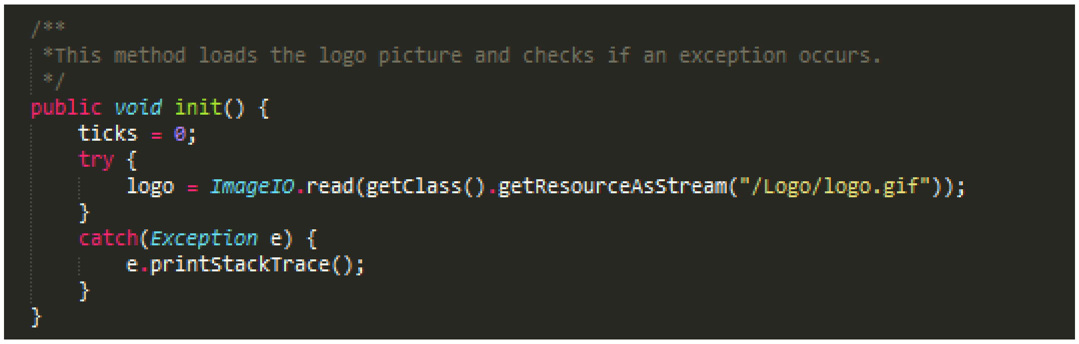

```java
}

public void init() {
    ticks = 0;
    try {
        logo = ImageIO.read(getClass().getResourceAsStream("/Logo/logo.gif"));
    }
```


```java
public void update() {
    animation.update();
    if(animation.hasPlayedOnce()) remove = true;
}
```

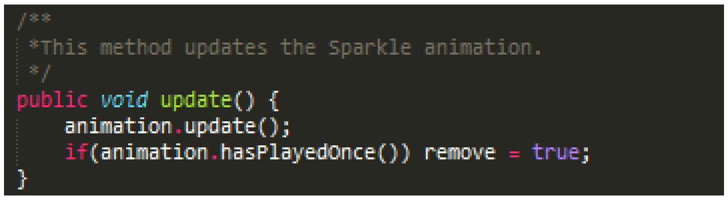
In order to improve the code, Javadoc comments have been inserted. Javadoc comments are important for the code authors to know what the code should do. It also helps the readers reading the code afterwards. **(Farrell, 2001)**



```java
/**
 * This method loads the map.
 * @param s
 */
public void loadMap(String s) {
```



```java
/**
 *This method loads the logo picture and checks if an exception occurs.
 */
public void init() {
    ticks = 0;
    try {
        logo = ImageIO.read(getClass().getResourceAsStream("/Logo/logo.gif"));
    }
    catch(Exception e) {
        e.printStackTrace();
    }
}
```

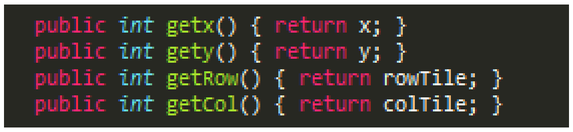


```java
/**
 *This method updates the Sparkle animation.
 */
public void update() {
    animation.update();
    if(animation.hasPlayedOnce()) remove = true;
}
```

2- Superclass variables declared again in subclass:

Player.java is the subclass of Entity.java, and it has four methods that are already inherited from the superclass being declared again into it. Since declaring those methods again is just extra code then deleting that part of the code would just save more space and free more memory.

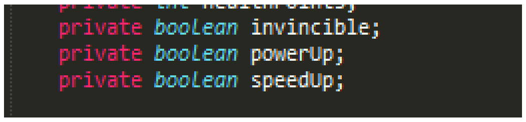
These are the four methods that were declared again in the subclass:



```java
public int getx() { return x; }
public int gety() { return y; }
public int getRow() { return rowTile; }
public int getCol() { return colTile; }
```

3- Unused variables:

Below are three variables that were never used:

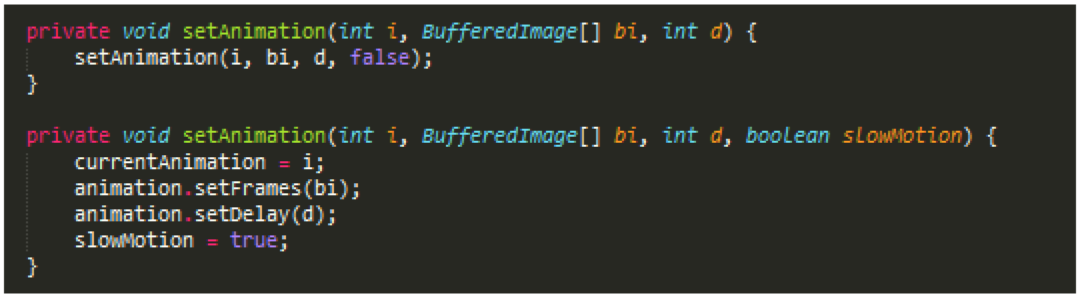


```java
private boolean invincible;
private boolean powerUp;
private boolean speedUp;
```

Deleting those unused variables would help reduce code size as well as simplify the support. **(Dead Code, n.d.)**

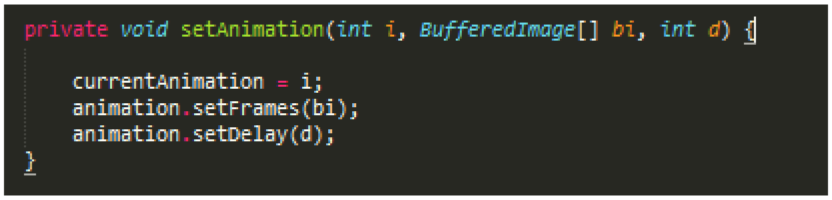
4- Extra method:

In the Player.java code there are two methods called as SetAnimation with the first one taking three parameters and the second one taking four. However, the only job of the first method is to put the same three variables into the second one and just add false into the fourth parameter. Taking note that the fourth parameter(slowMotion) is not being used anywhere and it's always set to false in the first method and then changed to true in the second one. This simply causes more memory usage and a more complex code.



```java
private void setAnimation(int i, BufferedImage[] bi, int d) {
    setAnimation(i, bi, d, false);
}

private void setAnimation(int i, BufferedImage[] bi, int d, boolean slowMotion) {
    currentAnimation = i;
    animation.setFrames(bi);
    animation.setDelay(d);
    slowMotion = true;
}
```

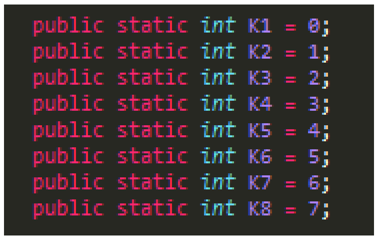
This could be easily fixed by combining both methods into one method like this:



```java
private void setAnimation(int i, BufferedImage[] bi, int d) {

    currentAnimation = i;
    animation.setFrames(bi);
    animation.setDelay(d);
}
```

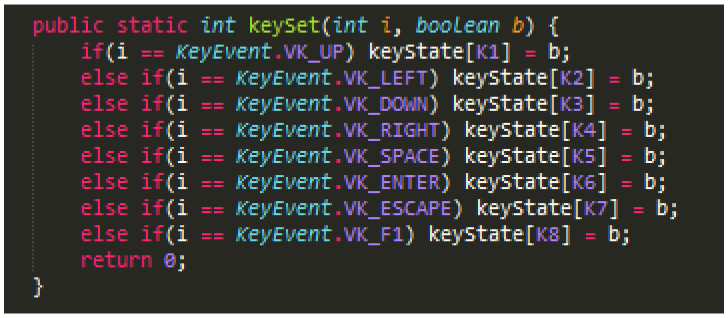
5- Inconvenient variable naming:

In Keys.java there are eight declared variables from K1 to K8. With the lack of comments, it's really hard to understand what those variables are used for.



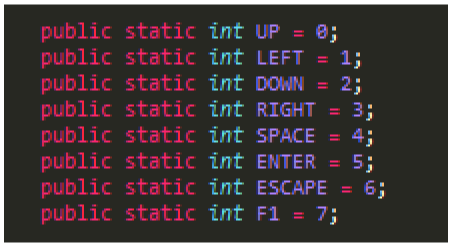
```java
public static int K1 = 0;
public static int K2 = 1;
public static int K3 = 2;
public static int K4 = 3;
public static int K5 = 4;
public static int K6 = 5;
public static int K7 = 6;
public static int K8 = 7;
```

However, after observation of this code, the variables functions were understood:



```
public static int keySet(int i, boolean b) {
    if(i == KeyEvent.VK_UP) keyState[K1] = b;
    else if(i == KeyEvent.VK_LEFT) keyState[K2] = b;
    else if(i == KeyEvent.VK_DOWN) keyState[K3] = b;
    else if(i == KeyEvent.VK_RIGHT) keyState[K4] = b;
    else if(i == KeyEvent.VK_SPACE) keyState[K5] = b;
    else if(i == KeyEvent.VK_ENTER) keyState[K6] = b;
    else if(i == KeyEvent.VK_ESCAPE) keyState[K7] = b;
    else if(i == KeyEvent.VK_F1) keyState[K8] = b;
    return 0;
}
```

In this example, K1 represents the UP button. Improving this code by renaming the variables K1 to K8 to their respective buttons will help in reducing the need for extensive comments as well as simplifying the code for reading. **(GoodVariableNames, 2014)**



```
public static int UP = 0;
public static int LEFT = 1;
public static int DOWN = 2;
public static int RIGHT = 3;
public static int SPACE = 4;
public static int ENTER = 5;
public static int ESCAPE = 6;
public static int F1 = 7;
```

References:

Farrell, D. J. (2001, March 23). *Make bad code good*. Retrieved from JavaWorld: http://www.javaworld.com/article/2075129/testing-debugging/make-bad-code-good.html?page=3

Dead Code. (n.d.). Retrieved from Sourcemaking: https://sourcemaking.com/refactoring/smells/dead-code

GoodVariableNames. (2014, December 12). Retrieved from Wiki: http://wiki.c2.com/?GoodVariableNames