Code-Review

Haruka Mibuchi, Gurinder Bhogal, Asifiwe Julio Patrick

We noticed that the GameFrame class was very large and was being used for more than it was planned to be used for. The GameFrame class is meant to keep track of all the various screens our games can have and switch between them depending on the state of the game. But the class ended up doing extra work such as storing a reference to many other classes and accessing their methods for other classes. Most of our refactorings are focused on reducing the size and workload of the GameFrame class. This was a success as we were able to take the GameFrame class from 257 lines to 168 lines.

Our 1st refactoring was to make the GameFrame class itself a singleton class (commit `ddbfa1be`) . This was because the GameFrame was being passed into the constructor of many classes, so they could call methods from other classes through it. Doing this cleaned up many classes as they no longer needed to store the instance of the GameFrame class and could request for it whenever they wanted. Doing only this refactoring cut the time it takes to build our game into a quarter of the original time.

```
public static GameFrame getInstance() {
        if (instance == null)
                new GameFrame();

        return instance;

}
```

Our 2nd Refactoring was to make the InputHandler class into a singleton (commit `b3812a93`). This cleaned up the code in the Celebrity class as it needed to strore the inputhandler after getting it from the GameFrame.

```
private InputHandler() {
        instance = this;
}

public static InputHandler getInstance() {
        if (instance == null) {
                new InputHandler();
        }
        return instance;
}
```

```
public class Celebrity extends MoveableEntity {
        InputHandler inpHandler;

        public Celebrity(int x, int y) {
                super(x, y);
                this.inpHandler = GameFrame.getInstance().getInputHandler();
                this.speed = 4;
                this.direction = Direction.NONE;
                loadImage();
@@ -67,7 +64,7 @@ public class Celebrity extends MoveableEntity {
         * its permissable
         */
        public void update() {
                direction = inpHandler.getDirection();
                direction = InputHandler.getInstance().getDirection();
                loadImage();
```

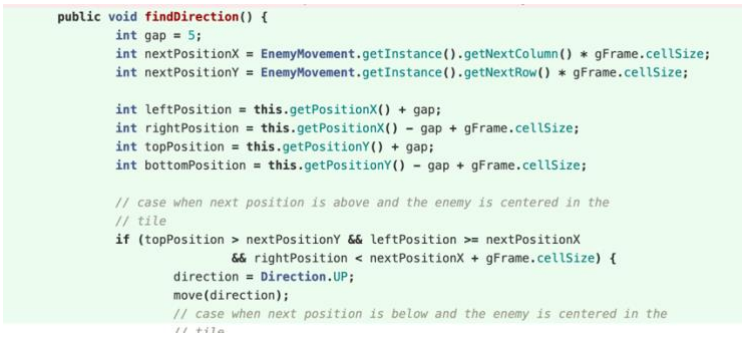
Our 3rd Refactoring was to create a method to calculate the top, bottom, left, and right rows/columns of our entities (commit `81b2be67`). While writing our code we had just written the code for calculating each of these values every single time in each method. This led to a lot of duplicate code which needed to be cleaned up. We already had a Point class which was used to keep track of which pixel an entity was at. We expanded that class to also calculate the columns/rows for the entities.

```
public void checkMapCollision(Celebrity character) {
        // find the column(s)/row(s) the entity is in
        int gap = 5;
        int leftColumn = (character.getPositionX() + gap) / gFrame.cellSize;
        int rightColumn = (character.getPositionX() - gap + gFrame.cellSize) / gFrame.cellSize;
        int topRow = (character.getPositionY() + gap) / gFrame.cellSize;
        int bottomRow = (character.getPositionY() - gap + gFrame.cellSize) / gFrame.cellSize;
        int leftColumn = character.getLeftColumn();
        int rightColumn = character.getRightColumn();
        int topRow = character.getTopRow();
        int bottomRow = character.getBottomRow();
```

Our 4th refactoring was to make the GamePanel as well as all other panels into a singleton class (commit `73c75e6c` and `f9f7baea`). This reduced the load placed on the GameFrame class since any time another class wished to access the GamePanel it would need to go through the GameFrame first. As well as improved the readability of the code.

```
private static GameFrame instance = null;
private Thread thread;
private InputHandler inpHandler;
private StartPanel startPanel;
private PausedPanel pausedPanel;
private WinPanel winPanel;
private LosePanel losePanel;
private CardLayout cardLayout;
private Score score;
private boolean paused;
@@ -41,10 +37,6 @@ public class GameFrame extends JFrame implements Runnable {
        instance = this;
        inpHandler = InputHandler.getInstance();
        cardLayout = new CardLayout();
        startPanel = new StartPanel();
        pausedPanel = new PausedPanel();
        winPanel = new WinPanel();
        losePanel = new LosePanel();
        score = new Score();
        this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        this.setSize(screenWidth, screenHeight); // 46 columns 27 rows
```

Our 5th refactoring was to split the update method in the Paparazzi class into two methods (commit `f29d2e36`). We noticed that this method was doing way more than it should. It would make calls to the EnemyMovement class and the CollisionFinder class then calculate the next direction for the enemy to move. We created a new method to handle finding the next direction.



```
public void findDirection() {
        int gap = 5;
        int nextPositionX = EnemyMovement.getInstance().getNextColumn() * gFrame.cellSize;
        int nextPositionY = EnemyMovement.getInstance().getNextRow() * gFrame.cellSize;

        int leftPosition = this.getPositionX() + gap;
        int rightPosition = this.getPositionX() - gap + gFrame.cellSize;
        int topPosition = this.getPositionY() + gap;
        int bottomPosition = this.getPositionY() - gap + gFrame.cellSize;

        // case when next position is above and the enemy is centered in the
        // tile
        if (topPosition > nextPositionY && leftPosition >= nextPositionX
                        && rightPosition < nextPositionX + gFrame.cellSize) {
                direction = Direction.UP;
                move(direction);
                // case when next position is below and the enemy is centered in the
```

Our 6th refactoring was to make the EnemyMovement and CollisionFinder class follow a singleton pattern (commit `714f75fd` and `cb5584f7`). These classes were being stored in the GameFrame which meant any other class had to go through the GameFrame to access them. Making them singletons reduced the load of the GameFrame class as well as improved code readability.

Our 7th refactoring was to clean-up some calculations done by the update and findDirection methods of the Paparazzi class (commit `c03093a7`). These calculations had primitive data like the gap. We created methods in the point class to clean up the code for the findDirection method and took advantage of our 3rd refactoring to clean up the code in the update method.

```
public void findDirection() {
        int gap = 5;
        int nextPositionX = EnemyMovement.getInstance().getNextColumn() * gFrame.cellSize;
        int nextPositionY = EnemyMovement.getInstance().getNextRow() * gFrame.cellSize;

        int leftPosition = this.getPositionX() + gap;
        int rightPosition = this.getPositionX() - gap + gFrame.cellSize;
        int topPosition = this.getPositionY() + gap;
        int bottomPosition = this.getPositionY() - gap + gFrame.cellSize;
        int leftPosition = this.getHitBoxLeft();
        int rightPosition = this.getHitBoxRight();
        int topPosition = this.getHitBoxTop();
        int bottomPosition = this.getHitBoxBottom();
```

Our 8th refactoring was to make the Score class follow a singleton Design pattern (commit `208c20a3`). The score was being changed through the GameFrame which is not our intended use for the GameFrame. But the GameFrame was previously the link between classes, so we needed to change the score there. By making the Score a singleton class any class can change the score directly.