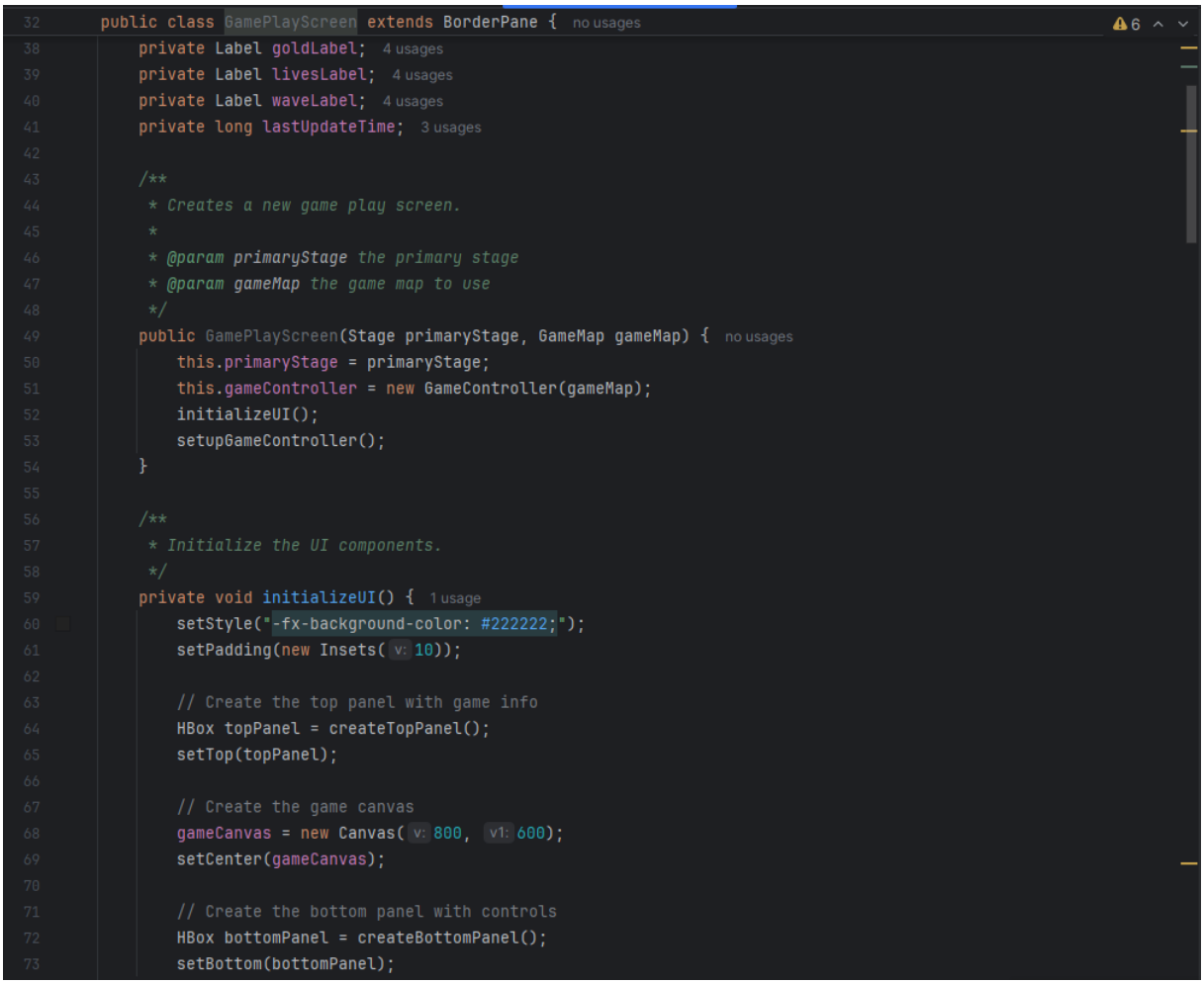
Usage of Software Patterns:

Now to specify, everything that has Controller at it's name, is exactly coded – implemented as controller pattern. To have a look:

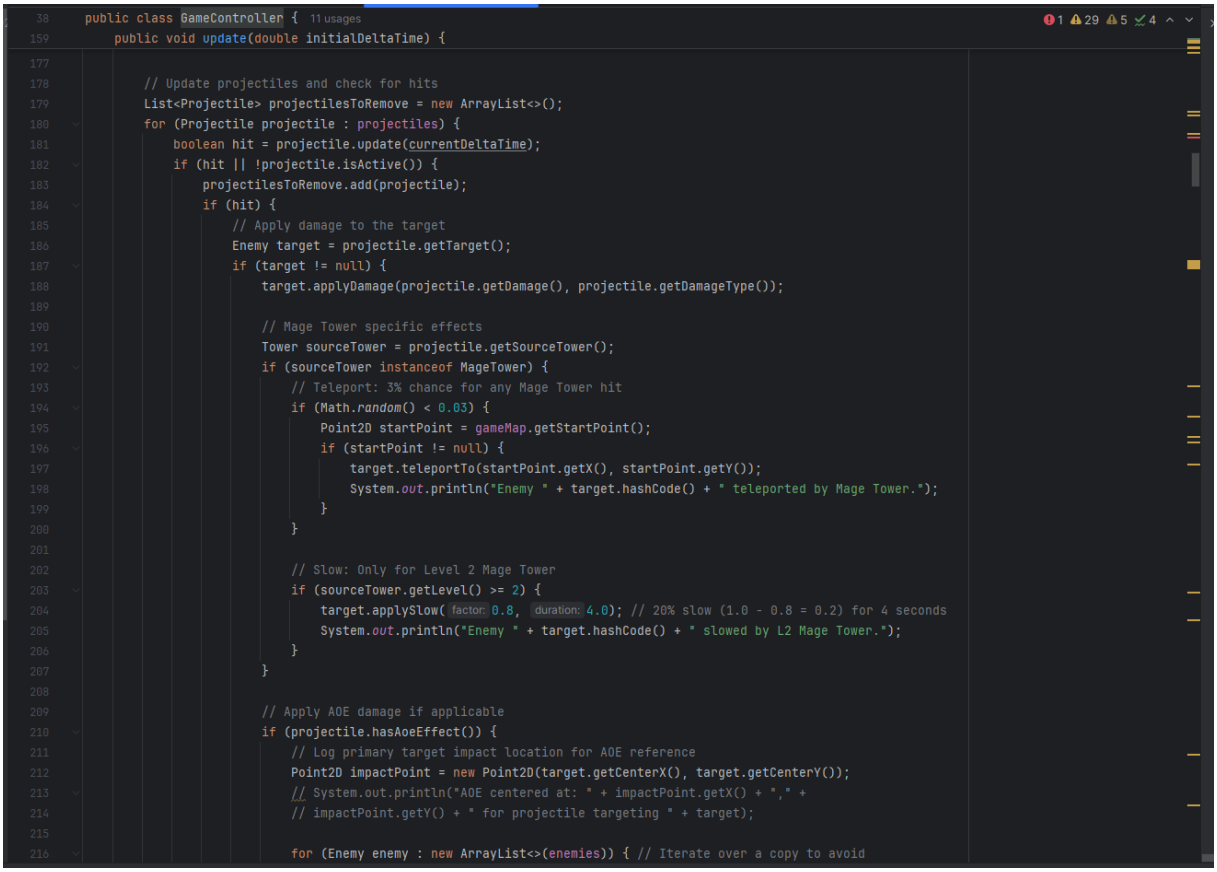
This is our GameSession



```
32  public class GamePlayScreen extends BorderPane {  no usages
38      private Label goldLabel;  4 usages
39      private Label livesLabel;  4 usages
40      private Label waveLabel;  4 usages
41      private long lastUpdateTime;  3 usages
42
43      /**
44       * Creates a new game play screen.
45       *
46       * @param primaryStage the primary stage
47       * @param gameMap the game map to use
48       */
49      public GamePlayScreen(Stage primaryStage, GameMap gameMap) {  no usages
50          this.primaryStage = primaryStage;
51          this.gameController = new GameController(gameMap);
52          initializeUI();
53          setupGameController();
54      }
55
56      /**
57       * Initialize the UI components.
58       */
59      private void initializeUI() {  1 usage
60          setStyle("-fx-background-color: #222222;");
61          setPadding(new Insets( v: 10));
62
63          // Create the top panel with game info
64          HBox topPanel = createTopPanel();
65          setTop(topPanel);
66
67          // Create the game canvas
68          gameCanvas = new Canvas( v: 800,  v1: 600);
69          setCenter(gameCanvas);
70
71          // Create the bottom panel with controls
72          HBox bottomPanel = createBottomPanel();
73          setBottom(bottomPanel);
```

GameSession's task is to initalize everything, as can be seen from the small part of the code attached, UI, and the GameController is booted here. Important for **model-view seperation**, and enchances it.

To show that even though we have something named GameSession, respecting the **Controller Pattern**, we have a great separation of tasks between our controllers, to give one example, gameController's one of many task is to see whether the projectiles reach the enemy, it handles **decision logic**, **coordinates actions,** and handles the inputs from the **GameSession.**



```
public class GameController { 11 usages
    public void update(double initialDeltaTime) {

        // Update projectiles and check for hits
        List<Projectile> projectilesToRemove = new ArrayList<>();
        for (Projectile projectile : projectiles) {
            boolean hit = projectile.update(currentDeltaTime);
            if (hit || !projectile.isActive()) {
                projectilesToRemove.add(projectile);
                if (hit) {
                    // Apply damage to the target
                    Enemy target = projectile.getTarget();
                    if (target != null) {
                        target.applyDamage(projectile.getDamage(), projectile.getDamageType());

                        // Mage Tower specific effects
                        Tower sourceTower = projectile.getSourceTower();
                        if (sourceTower instanceof MageTower) {
                            // Teleport: 3% chance for any Mage Tower hit
                            if (Math.random() < 0.03) {
                                Point2D startPoint = gameMap.getStartPoint();
                                if (startPoint != null) {
                                    target.teleportTo(startPoint.getX(), startPoint.getY());
                                    System.out.println("Enemy " + target.hashCode() + " teleported by Mage Tower.");
                                }
                            }

                            // Slow: Only for Level 2 Mage Tower
                            if (sourceTower.getLevel() >= 2) {
                                target.applySlow( factor: 0.8,  duration: 4.0); // 20% slow (1.0 - 0.8 = 0.2) for 4 seconds
                                System.out.println("Enemy " + target.hashCode() + " slowed by L2 Mage Tower.");
                            }
                        }

                        // Apply AOE damage if applicable
                        if (projectile.hasAoeEffect()) {
                            // Log primary target impact location for AOE reference
                            Point2D impactPoint = new Point2D(target.getCenterX(), target.getCenterY());
                            // System.out.println("AOE centered at: " + impactPoint.getX() + "," +
                            // impactPoint.getY() + " for projectile targeting " + target);

                            for (Enemy enemy : new ArrayList<>(enemies)) { // Iterate over a copy to avoid
```

To give an example on one of our listeners

OptionsScreen:

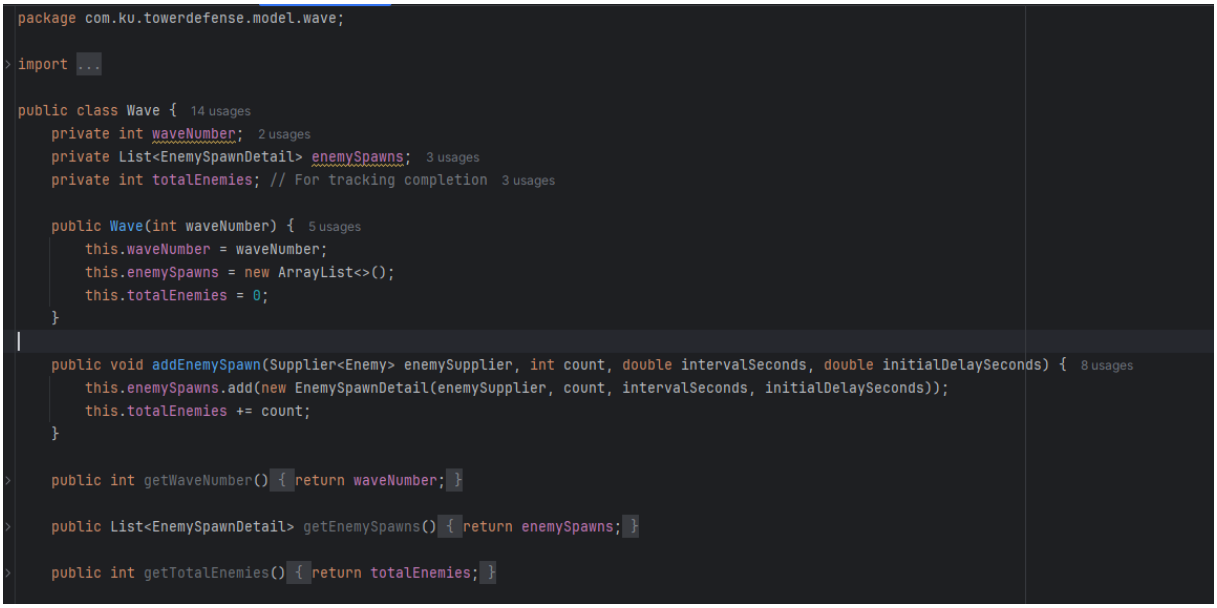
```java
public class OptionsScreen extends BorderPane {  4 usages
    }

    /**
     * Add a number option with slider to the parent VBox (option board).
     *
     * @param parentBoard  the VBox board to add to
     * @param labelText    the option label
     * @param initialValue the initial value
     * @param min          the minimum value
     * @param max          the maximum value
     * @param setter       the setter method to call when value changes
     */
    private void addNumberOption(VBox parentBoard, String labelText, int initialValue, int min, int max,  28 usages
            SliderChangeListener setter) {
        Label optionLabel = new Label(labelText);
        optionLabel.getStyleClass().add("options-label");

        TextField valueField = new TextField(String.valueOf(initialValue));
        valueField.setPrefWidth(60); // Original, will scale
        valueField.getStyleClass().add("options-value-field");

        Slider slider = new Slider(min, max, initialValue);
        slider.setShowTickMarks(true);
        slider.setShowTickLabels(true);
        slider.setMajorTickUnit(Math.max(1, (max - min) / 4.0)); // Original
        slider.setBlockIncrement(1);
        slider.getStyleClass().add("options-slider");

        slider.valueProperty().addListener((obs, oldValue, newValue) -> {
            int rounded = (int) Math.round(newValue.doubleValue());
            valueField.setText(String.valueOf(rounded));
            setter.setValue(rounded);
        });

        valueField.setOnAction(e -> {
            try {
                int value = Integer.parseInt(valueField.getText());
                if (value >= min && value <= max) {
                    slider.setValue(value);
                    setter.setValue(value);
                } else {
```

As can be seen, bunches of buttons to form up a value file, where the game will boot according to these, completely selected by user before game starts, sole reason for this class is to listen for inputs from the user.

```java
    @FunctionalInterface  1 usage
    private interface SliderChangeListener {
        void setValue(int value);  2 usages
    }
}
```

Finally, one example class where we employed factory pattern:

Wave.java:



```java
package com.ku.towerdefense.model.wave;

import ...

public class Wave {  14 usages
    private int waveNumber;  2 usages
    private List<EnemySpawnDetail> enemySpawns;  3 usages
    private int totalEnemies; // For tracking completion  3 usages

    public Wave(int waveNumber) {  5 usages
        this.waveNumber = waveNumber;
        this.enemySpawns = new ArrayList<>();
        this.totalEnemies = 0;
    }

    public void addEnemySpawn(Supplier<Enemy> enemySupplier, int count, double intervalSeconds, double initialDelaySeconds) {  8 usages
        this.enemySpawns.add(new EnemySpawnDetail(enemySupplier, count, intervalSeconds, initialDelaySeconds));
        this.totalEnemies += count;
    }

    public int getWaveNumber() { return waveNumber; }

    public List<EnemySpawnDetail> getEnemySpawns() { return enemySpawns; }

    public int getTotalEnemies() { return totalEnemies; }
```

This class is where we spawn our enemies, which are classes of their own. This design uses the **factory method concept via Supplier<Enemy>** to defer and abstract the creation of enemy instances.