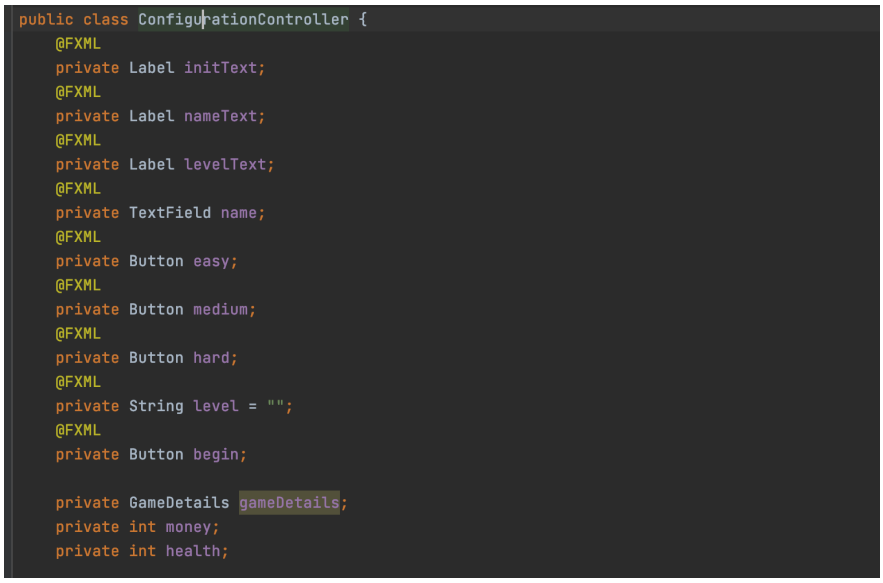
SOLID/GRASP Principle Write-Up

Controller (GRASP)



```
public class ConfigurationController {
    @FXML
    private Label initText;
    @FXML
    private Label nameText;
    @FXML
    private Label levelText;
    @FXML
    private TextField name;
    @FXML
    private Button easy;
    @FXML
    private Button medium;
    @FXML
    private Button hard;
    @FXML
    private String level = "";
    @FXML
    private Button begin;

    private GameDetails gameDetails;
    private int money;
    private int health;
```

```
@FXML
protected void onBegin(ActionEvent e) throws IOException {


    if (name.getText().isBlank() || name.getText().isEmpty()) {
        System.out.println("hi");
        Alert alert = new Alert(Alert.AlertType.ERROR,  s: "Please write a valid name");
        level = "";
        //alert.getDialogPane().setExpandableContent(new Label("Please write a valid name"));
        alert.showAndWait();
    } else if (level.equals("")) {
        Alert alert = new Alert(Alert.AlertType.ERROR,  s: "Please choose a valid level");
        //alert.getDialogPane().setExpandableContent(new Label("Please choose a valid level"));
        alert.showAndWait();
    } else {
        Alert alert = new Alert(
                Alert.AlertType.INFORMATION,
                 s: "Name: " + name.getText() + "\nLevel: " + level
        );
        alert.showAndWait();
        GameDetails old = StoreGame.getGameDetails();
        if (old != null) {
            gameDetails = new GameDetails( money: this.money + old.getExtraMoney(),
                    this.health, this.level, this.name.getText());
        } else {
            gameDetails = new GameDetails(this.money, this.health,
                    this.level, this.name.getText());
        }
        StoreGame.setGameDetails(gameDetails);
        gameScreen(e);
    }
}
```

The Configuration Controller responds to actions made to the UI and causes changes to the actual functionality. This is shown in the onBegin() method of the class. It delegates the responsibility of functionality changes to the GameDetails class rather than doing itself.

Creator (GRASP)

```java
public void putTower(Button backgroundButton, String type) {
    String[] id = backgroundButton.getId().split( regex: ",");
    int row = Integer.parseInt(id[0]);
    int col = Integer.parseInt(id[1]);
    Tower tower = new Tower(row, col, type);
    gameDetails.includeTower(tower);
    URL url = TowerDefenseApplication.class.getResource( name: "assets/images/BadTower.png");
    Image towerImg = new Image(String.valueOf(url));
    ImageView towerImgView = new ImageView();
    towerImgView.setImage(towerImg);
    //quit.setMaxSize(100, 100);
    towerImgView.setFitHeight(100);
    towerImgView.setFitWidth(92);
    towerImgView.setPreserveRatio(true);
    backgroundButton.setGraphic(towerImgView);

}
```

The LandScapeController class is a creator for the Tower class. It has the proper parameters that you are to pass into the tower (has the initializing data). The creator class in our game is also essentially the map of the whole game, and the towers are a necessary part of the map.

Single Responsibility Principle (SOLID)

```
public class BuyTower {
    @FXML
    private Label shopText;
    @FXML
    private Button quit;

    private Image quitImg;
    private Image badImg;
    private Image normalImg;
    private Image eliteImg;
```

```
    protected void purchaseElite(ActionEvent e) throws java.io.IOException {

        if (this.currentMoney < elitecost) {
            Alert alert = new Alert(Alert.AlertType.ERROR, s: "Insufficient Funds!");
            alert.showAndWait();
        } else {
            this.currentMoney -= elitecost;
            this.gameDetails.setMoney(this.currentMoney);
            this.onGameScreen(e);
            gameDetails.setImage("elite");
        }
    }

    @FXML
    protected void purchaseNormal(ActionEvent e) throws java.io.IOException {

        if (this.currentMoney < normalcost) {
            Alert alert = new Alert(Alert.AlertType.ERROR, s: "Insufficient Funds!");
            alert.showAndWait();
        } else {
            this.currentMoney -= normalcost;
            this.gameDetails.setMoney(this.currentMoney);
            this.onGameScreen(e);
            gameDetails.setImage("normal");
        }
    }

    @FXML
    protected void purchaseBad(ActionEvent e) throws java.io.IOException {


        if (this.currentMoney < badcost) {
            Alert alert = new Alert(Alert.AlertType.ERROR, s: "Insufficient Funds!");
            alert.showAndWait();
        } else {
```

The BuyTower class has only one responsibility: to buy towers from the shop. It allows us to separate concerns from the rest of the shop, so that we don't have to worry about things like upgrading towers within this class.

Open/Closed Principle (SOLID)

```java
package com.example.game;

public interface Tower {
    void place(int row, int col);
}
```

```java
public class EliteTower implements Tower{
    private String level;
    private float range;
    private float damagePerSecond;
    private String type;
    private int cost;

    GameDetails gameDetails = StoreGame.getGameDetails();

    public EliteTower(String level, float range, float damagePerSecond, String type, int cost) {
        this.level = level;
        this.range = range;
        this.damagePerSecond = damagePerSecond;
        this.type = type;
        this.cost = cost;
    }

    @Override
    public void place(int row, int col) {
        gameDetails.setLocation(row, col);
    }
```

```java
public class NormalTower implements Tower{
    private String level;
    private float range;
    private float damagePerSecond;
    private String type;
    private int cost;

    GameDetails gameDetails = StoreGame.getGameDetails();

    public NormalTower(String level, float range, float damagePerSecond, String type, int cost) {
        this.level = level;
        this.range = range;
        this.damagePerSecond = damagePerSecond;
        this.type = type;
        this.cost = cost;
    }

    @Override
    public void place(int row, int col) {
        gameDetails.setLocation(row, col);
    }
```

```java
public class BadTower implements Tower{
    private String level;
    private float range;
    private float damagePerSecond;
    private String type;
    private int cost;

    GameDetails gameDetails = StoreGame.getGameDetails();

    public BadTower(String level, float range, float damagePerSecond, String type, int cost) {
        this.level = level;
        this.range = range;
        this.damagePerSecond = damagePerSecond;
        this.type = type;
        this.cost = cost;
    }

    @Override
    public void place(int row, int col) {
        gameDetails.setLocation(row, col);
    }
```

In the latest version of our Tower implementation, we made it an interface that only has one method and then we implement the interface with the three tower types. This way, you can't modify the Tower code but you can extend it and give it different functionality based on its type, and the base tower functionality will remain the same even if one of the towers doesn't work properly.

Pure Fabrication (GRASP)

```java
package com.example.game;

public class StoreGame {
    private static GameDetails gameDetails = null;

    public static GameDetails getGameDetails() { return gameDetails; }

    public static void setGameDetails(GameDetails game) {
        StoreGame.gameDetails = game;
    }
}
```

In previous implementations of our app, the game details were not being correctly updated as new instances of it were being created each time the shop was opened. As such, we created a new class whose sole purpose is to store persistent data without it being reinitialized each time an action is performed in the game. That makes this class a pure fabrication.