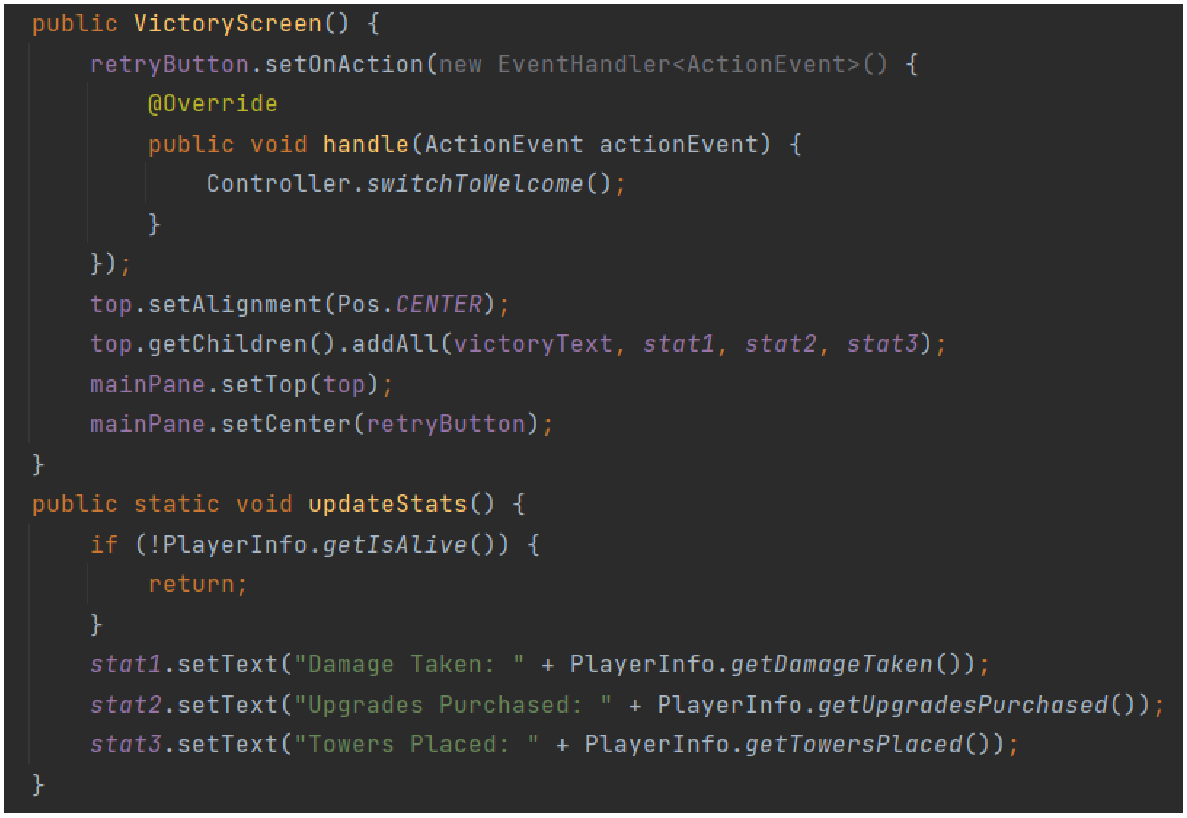
Information Expert: An example of the GRASP principle Information Expert can be seen in the upgrade() methods in RedTower.java, BlueTower.java, and GreenTower.java. This code follows this principle because instead of trying to manually upgrade towers when handling the upgrade button getting pressed in the MainGame class, we are assigning the upgrade responsibility for each type of tower to their respective tower classes. We are doing this because these classes have the relevant information, such as damage, slowMultiplier, and healthIncrement, that is necessary to fulfill our upgrade responsibility.

Single Responsibility Principle: The VictoryScreen class has the single responsiblity of managing the UI for the victory screen. For example, the class must display the stats tracked over the game, but it does not ever modify these stats or even store them itself. It instead just gets the numbers from the PlayerInfo class and updates the display text accordingly. It must also restart the game once the button is clicked. In this case, the VictoryScreen class does not handle any of the logical parts of resetting the game, and all it does is switch the UI screen to the welcome screen. The only reason for the VictoryScreen class to change would be to make a change to the UI on the victory screen.



```java
public VictoryScreen() {
    retryButton.setOnAction(new EventHandler<ActionEvent>() {
        @Override
        public void handle(ActionEvent actionEvent) {
            Controller.switchToWelcome();
        }
    });
    top.setAlignment(Pos.CENTER);
    top.getChildren().addAll(victoryText, stat1, stat2, stat3);
    mainPane.setTop(top);
    mainPane.setCenter(retryButton);
}
public static void updateStats() {
    if (!PlayerInfo.getIsAlive()) {
        return;
    }
    stat1.setText("Damage Taken: " + PlayerInfo.getDamageTaken());
    stat2.setText("Upgrades Purchased: " + PlayerInfo.getUpgradesPurchased());
    stat3.setText("Towers Placed: " + PlayerInfo.getTowersPlaced());
}
```

Polymorphism: An example of the GRASP principle, polymorphism, being followed can be seen with the enemies. We have an enemy abstract class that OrangeEnemy.java, PurpleEnemy.java, YellowEnemy.java, and FinalBoss.java extend from. These different enemies have different stats and functionalities. For instance, only the final boss is larger than a normal enemy, so it is the only class that has a setSize() method. This effectively reduced coupling between the different enemies since the other enemies would have no use for the setSize() method. If we wanted to add other special behaviors later on, we could easily do so by making new enemies with those behaviors or just adding them to the existing types of enemies.

Open/Closed Principle: An example of the SOLID principle, Open/Closed principle can be seen in the implementation of Tower and its various child classes. This shows this principle because instead of adding to the existing code we created new classes and connected them through inheritance. This allows for the core of the Tower class to not be able to be broken because new functionalities and implementations will be in an extension of that class. We have set up RedTower, YellowTower, and BlueTower which allow for the extension of Tower without directly modifying it, thus allow us to make changes, but avoid unforeseen errors in the Tower class itself.