## SOLID: Liskov Substitution Principle - Erin

Tower Abstract Class:

```
public abstract class Tower {
   protected String classification;
   protected int price;
   protected double xVal;
   protected double yVal;
   protected double dps;
   protected static int playerLevel;
   protected ImageView imageView = new ImageView();
   protected static double proximity = 130;
   protected double gainedHealth = -1;
   protected int gainedMoney = -1;
   protected Upgrade upgrade;

   //getter & setters omitted//
   //non-abstract methods omitted//

   public abstract ImageView draw();

   public abstract Node createAttackObject(Enemy e);

   public abstract boolean attackEnemy(Enemy e);

   public abstract void upgradeAttack();
```

Tower1: (Tower2 & Tower3 extend Tower and implement the same abstract methods listed above)

```
public class Tower1 extends Tower {
   private Image sprite = new Image("/Images/blueTower.png");
   private Line l;
   public Tower1() {
       classification = "Blue";
       price = 30 * playerLevel;
       dps = (4 - (0.4 * (playerLevel - 1)));
       upgrade = new Upgrade();
   }
   public ImageView draw() {
       imageView = new ImageView();
       imageView.setImage(sprite);
       imageView.setFitHeight(75);
       imageView.setFitWidth(75);
       imageView.setX(xVal);
       imageView.setY(yVal);
       return imageView;
```

```
    }
    public Line createAttackObject(Enemy e) {
        l = new Line(xVal + (75.0 / 2), yVal + (75.0 / 2),
                e.getXVal() + (35.0 / 2), e.getYVal() + (35.0 / 2));
        l.setStroke(Color.RED);
        l.setStrokeWidth(5);
        return l;
    }
    public boolean attackEnemy(Enemy e) {
        if (e.getImageView().intersects(l.getBoundsInLocal())) {
            e.setHealth(e.getHealth() - dps);
            return true;
        }
        return false;
    }
    public void upgradeAttack() {
        proximity += 3;
        dps += .3;
        upgrade.upgrade();
    }
}
```

Implementation of LSP principle :

```
public static void allTowerAttack() {
    Tower currTower;
    Enemy closestEnemy;
    for (int i = 0; i < currentTowers.size(); i++) {
        currTower = currentTowers.get(i);
        if (currentEnemies.size() > 0) {
            closestEnemy = currTower.closestEnemy(currentEnemies);
            if (currTower.enemyInProximity(closestEnemy)) {
                Node n = currTower.createAttackObject(closestEnemy);
                attackAnimation(n, closestEnemy, currTower);
                if (!closestEnemy.isEnemyHealthy()) {
                    closestEnemy.getImageView().setVisible(false);
                    root.getChildren().remove(closestEnemy.getImageView());
                    currentEnemies.remove(closestEnemy);
                    Player.setEnemiesKilled();
                }
            }
        }
    }
}
```

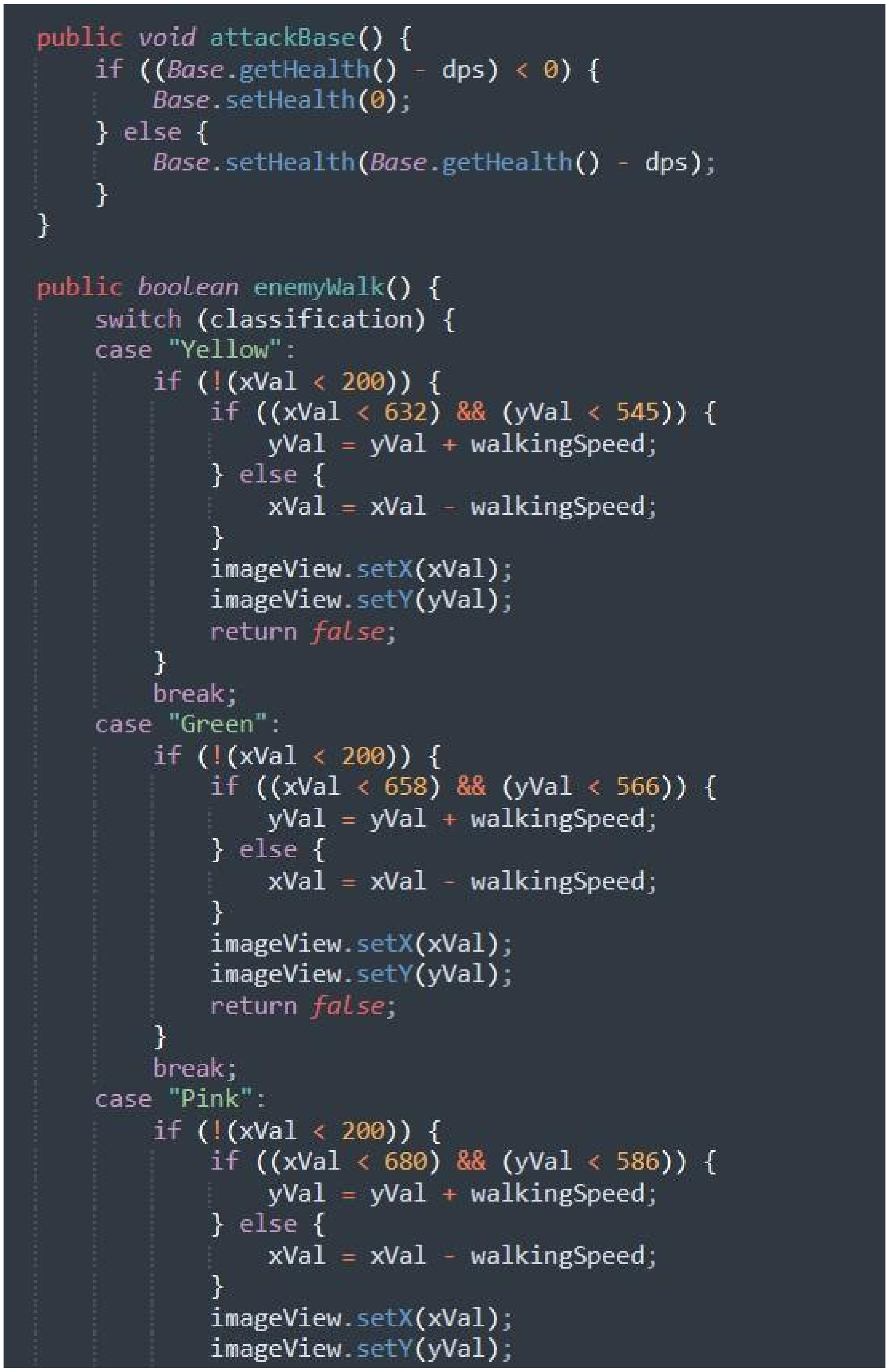
```
public static void attackAnimation(Node n, Enemy e, Tower t) {
    root.getChildren().add(n);
    new AnimationTimer() {
        @Override
        public void handle(long now) {
            t.attackEnemy(e);
            root.getChildren().remove(n);
            stop();
        }
    }.start();
}
```

```
private void pressUpgradeButton(ActionEvent event) throws Exception {
    Alert myAlert = new Alert(Alert.AlertType.INFORMATION);
    String invalid = currTower.getUpgrade().checkInvalidUpgrade();
    if (invalid != null) {
        myAlert.setHeaderText(invalid);
        myAlert.showAndWait();
    } else {
        currTower.upgradeAttack();

        Stage stage;
        stage = (Stage) upgradeBtn.getScene().getWindow();
        stage.close();

        GameConfig gameConfiguration = new GameConfig();
        gameConfiguration.start(gameConfig);
    }
}
```

Based on the LSP or Liskov Substitution Principle, derived classes must be substitutable for their base classes. In other words, the abstraction, or in this case the abstract class Tower, should be enough for the client. The highlighted code in blue follows this principle as the class calling the method is the Tower abstract class rather than the concrete Tower1, Tower2, or Tower3 classes.Therefore, the derived classes or concrete classes Tower1, Tower2, and Tower3 can be replaced or substituted by the base class, Tower.

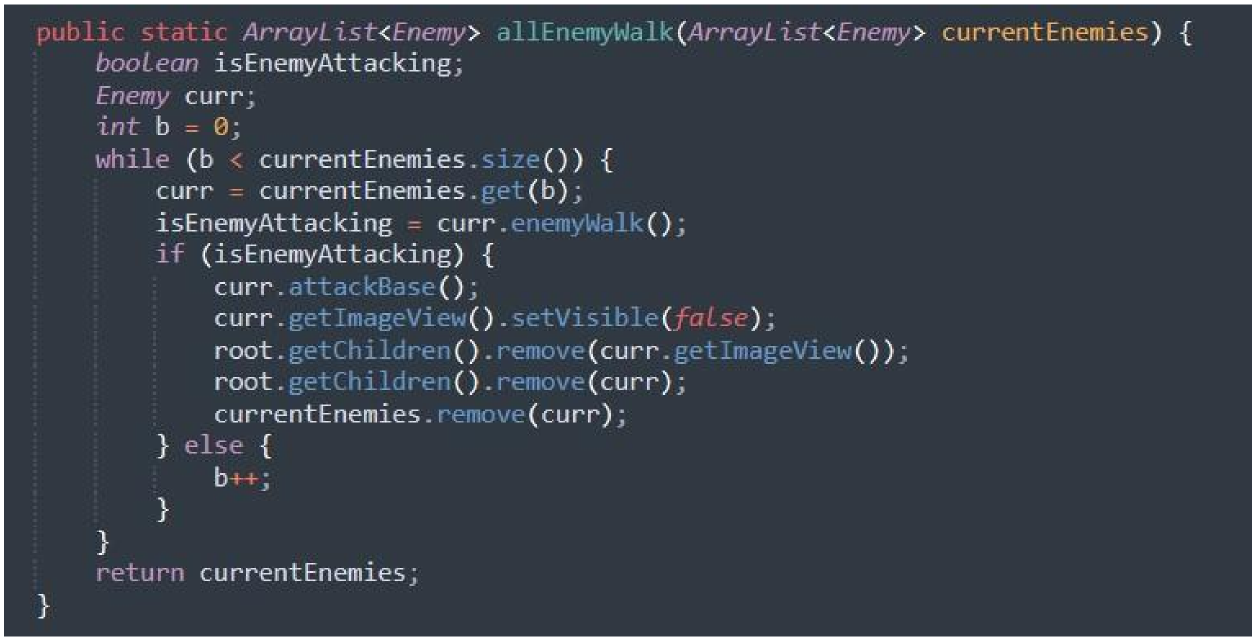
**<u>GRASP: Information Expert - Vishal</u>**

Enemy abstract class:



```
public void attackBase() {
    if ((Base.getHealth() - dps) < 0) {
        Base.setHealth(0);
    } else {
        Base.setHealth(Base.getHealth() - dps);
    }
}

public boolean enemyWalk() {
    switch (classification) {
    case "Yellow":
        if (!(xVal < 200)) {
            if ((xVal < 632) && (yVal < 545)) {
                yVal = yVal + walkingSpeed;
            } else {
                xVal = xVal - walkingSpeed;
            }
            imageView.setX(xVal);
            imageView.setY(yVal);
            return false;
        }
        break;
    case "Green":
        if (!(xVal < 200)) {
            if ((xVal < 658) && (yVal < 566)) {
                yVal = yVal + walkingSpeed;
            } else {
                xVal = xVal - walkingSpeed;
            }
            imageView.setX(xVal);
            imageView.setY(yVal);
            return false;
        }
        break;
    case "Pink":
        if (!(xVal < 200)) {
            if ((xVal < 680) && (yVal < 586)) {
                yVal = yVal + walkingSpeed;
            } else {
                xVal = xVal - walkingSpeed;
            }
            imageView.setX(xVal);
            imageView.setY(yVal);
```

Enemy walking and attacking in GameStart:

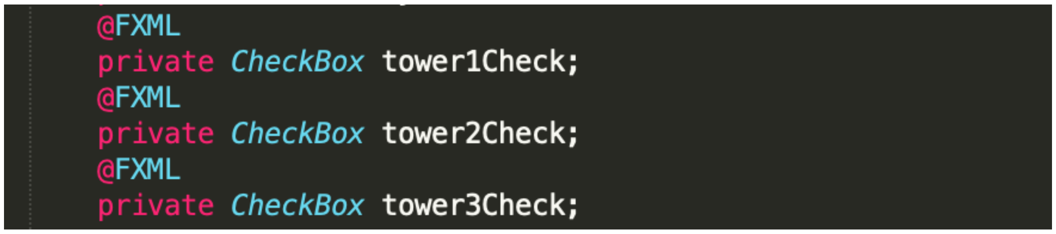


```java
public static ArrayList<Enemy> allEnemyWalk(ArrayList<Enemy> currentEnemies) {
    boolean isEnemyAttacking;
    Enemy curr;
    int b = 0;
    while (b < currentEnemies.size()) {
        curr = currentEnemies.get(b);
        isEnemyAttacking = curr.enemyWalk();
        if (isEnemyAttacking) {
            curr.attackBase();
            curr.getImageView().setVisible(false);
            root.getChildren().remove(curr.getImageView());
            root.getChildren().remove(curr);
            currentEnemies.remove(curr);
        } else {
            b++;
        }
    }
    return currentEnemies;
}
```

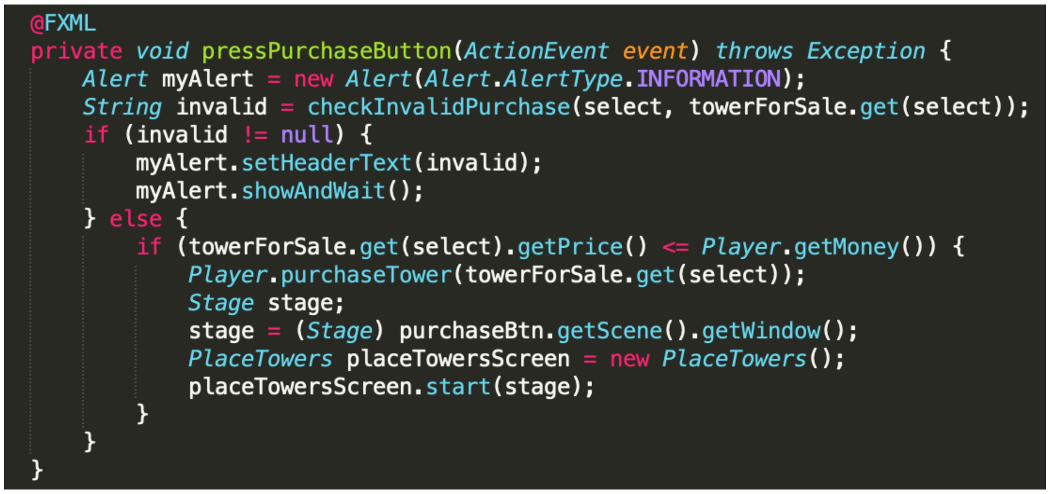
The Information Expert in GRASP is a class that controls the data and calculations for a specific Use Case. In this example, Enemy abstract class (and Enemy subclasses) control the data and calculations that allow the Enemies to walk and attack the base. When the Enemy is created, it calculates when it has to walk, and moves its sprite accordingly. When the Enemy is near the base, it calculates the damage done to the Base based on its own stats, and actually decreases the health of the Base.

**GRASP: Controller - Anika**

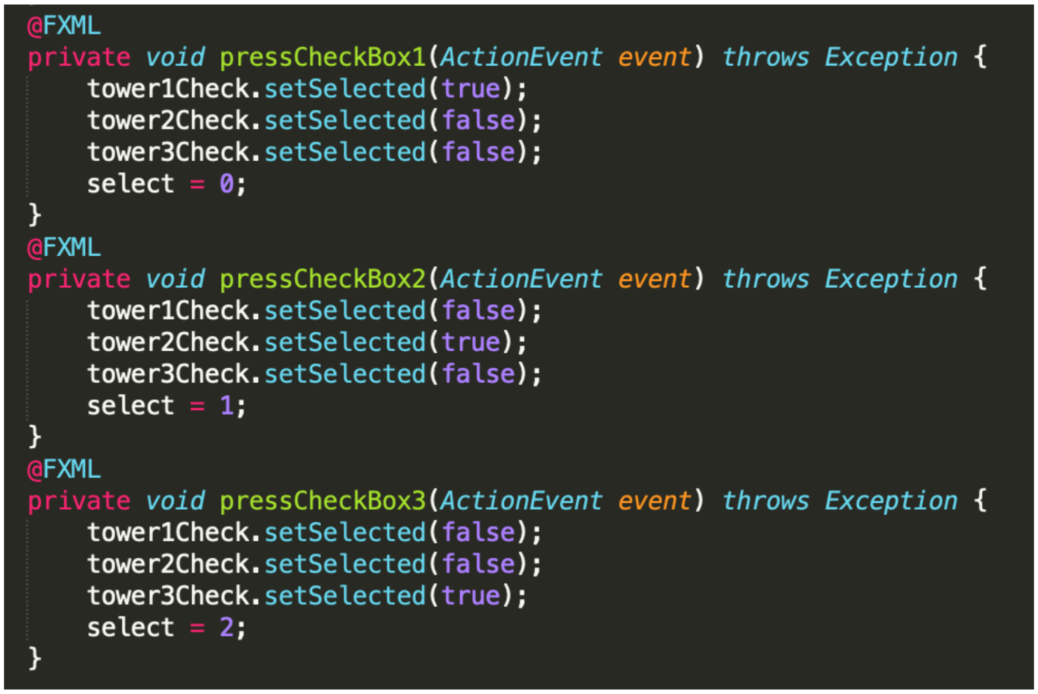
Shop Class:



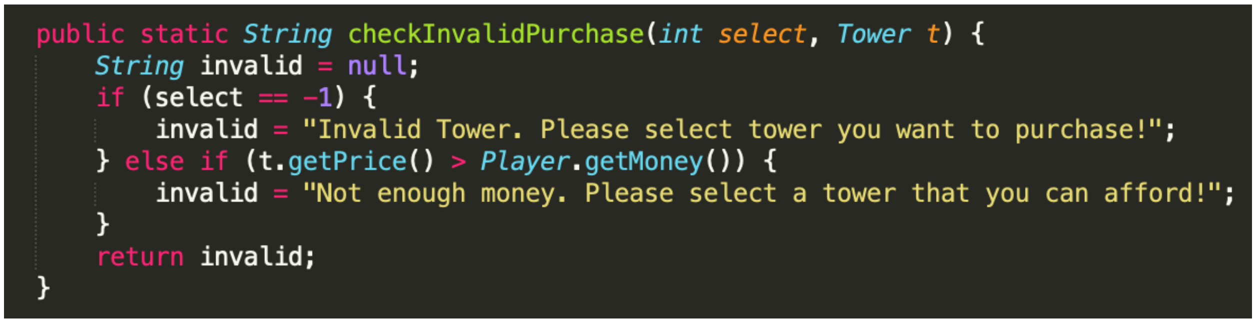
```
    @FXML
    private CheckBox tower1Check;
    @FXML
    private CheckBox tower2Check;
    @FXML
    private CheckBox tower3Check;
```



```
@FXML
private void pressPurchaseButton(ActionEvent event) throws Exception {
    Alert myAlert = new Alert(Alert.AlertType.INFORMATION);
    String invalid = checkInvalidPurchase(select, towerForSale.get(select));
    if (invalid != null) {
        myAlert.setHeaderText(invalid);
        myAlert.showAndWait();
    } else {
        if (towerForSale.get(select).getPrice() <= Player.getMoney()) {
            Player.purchaseTower(towerForSale.get(select));
            Stage stage;
            stage = (Stage) purchaseBtn.getScene().getWindow();
            PlaceTowers placeTowersScreen = new PlaceTowers();
            placeTowersScreen.start(stage);
        }
    }
}
```



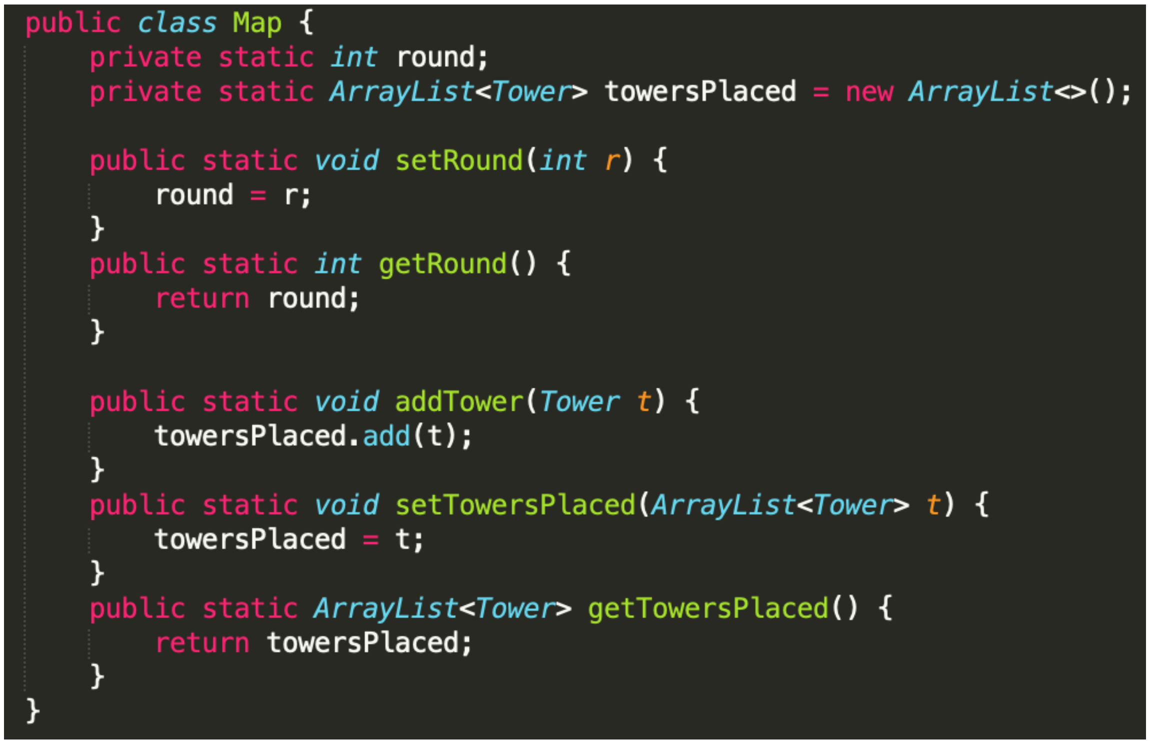
```
@FXML
private void pressCheckBox1(ActionEvent event) throws Exception {
    tower1Check.setSelected(true);
    tower2Check.setSelected(false);
    tower3Check.setSelected(false);
    select = 0;
}
@FXML
private void pressCheckBox2(ActionEvent event) throws Exception {
    tower1Check.setSelected(false);
    tower2Check.setSelected(true);
    tower3Check.setSelected(false);
    select = 1;
}
@FXML
private void pressCheckBox3(ActionEvent event) throws Exception {
    tower1Check.setSelected(false);
    tower2Check.setSelected(false);
    tower3Check.setSelected(true);
    select = 2;
}
```


```java
public static String checkInvalidPurchase(int select, Tower t) {
    String invalid = null;
    if (select == -1) {
        invalid = "Invalid Tower. Please select tower you want to purchase!";
    } else if (t.getPrice() > Player.getMoney()) {
        invalid = "Not enough money. Please select a tower that you can afford!";
    }
    return invalid;
}
```

In MVC architecture, the controller communicates with the user. The application logic of the checkboxes in the Shop to purchase towers is written directly to event handler methods of form control elements. This layer communicates directly with the user and writes to the console if the user has not checked a box to purchase any tower or does not have enough money to purchase the tower that they have checked the box for.

**GRASP: Low Coupling - Anika**

Map Class:



```java
public class Map {
    private static int round;
    private static ArrayList<Tower> towersPlaced = new ArrayList<>();

    public static void setRound(int r) {
        round = r;
    }
    public static int getRound() {
        return round;
    }

    public static void addTower(Tower t) {
        towersPlaced.add(t);
    }
    public static void setTowersPlaced(ArrayList<Tower> t) {
        towersPlaced = t;
    }
    public static ArrayList<Tower> getTowersPlaced() {
        return towersPlaced;
    }
}
```

Towers have a composition with the map, so the tower element and map element are coupled. The tower object depends heavily on the map object as the towers can't exist without the map and the map class must have access to tower objects for proper gameplay. We reduce the impact of change on these dependent elements so that coupling remains low.

**SOLID: Single-Responsibility Principle - Tuan**

Upgrade Class:
A class should only have one reason to change. In other words, a class should only have one job. For example, this class only has one job which is upgrading one of the towers that the player currently owns. This class will have its supporting functions that help to upgrade the tower which is set upgrade price and set upgrade level. The main job of this class is to upgrade which is the upgrade method and storing information after upgrade in the class itself.

```java
package main;

public class Upgrade {
   private int upgradePrice;
   private int upgradeLevel;

   public void setUpgradePrice(int u) { upgradePrice = u; }
   public int getUpgradePrice() { return upgradePrice; }
   public void setUpgradeLevel(int u) { upgradeLevel = u; }
   public int getUpgradeLevel() { return upgradeLevel; }

   public Upgrade() {
       upgradeLevel = 0;
       upgradePrice = 30;
   }

   public void upgrade() {
       Player.upgradeTower(upgradePrice);
       upgradePrice *= 2;
       upgradeLevel += 1;
   }

   public String checkInvalidUpgrade() {
       String invalid = null;
       if (Player.getMoney() <= upgradePrice) {
           invalid = "Not Enough Money. Cannot Purchase Upgrade for Tower.";
       }
       return invalid;
   }
}
```