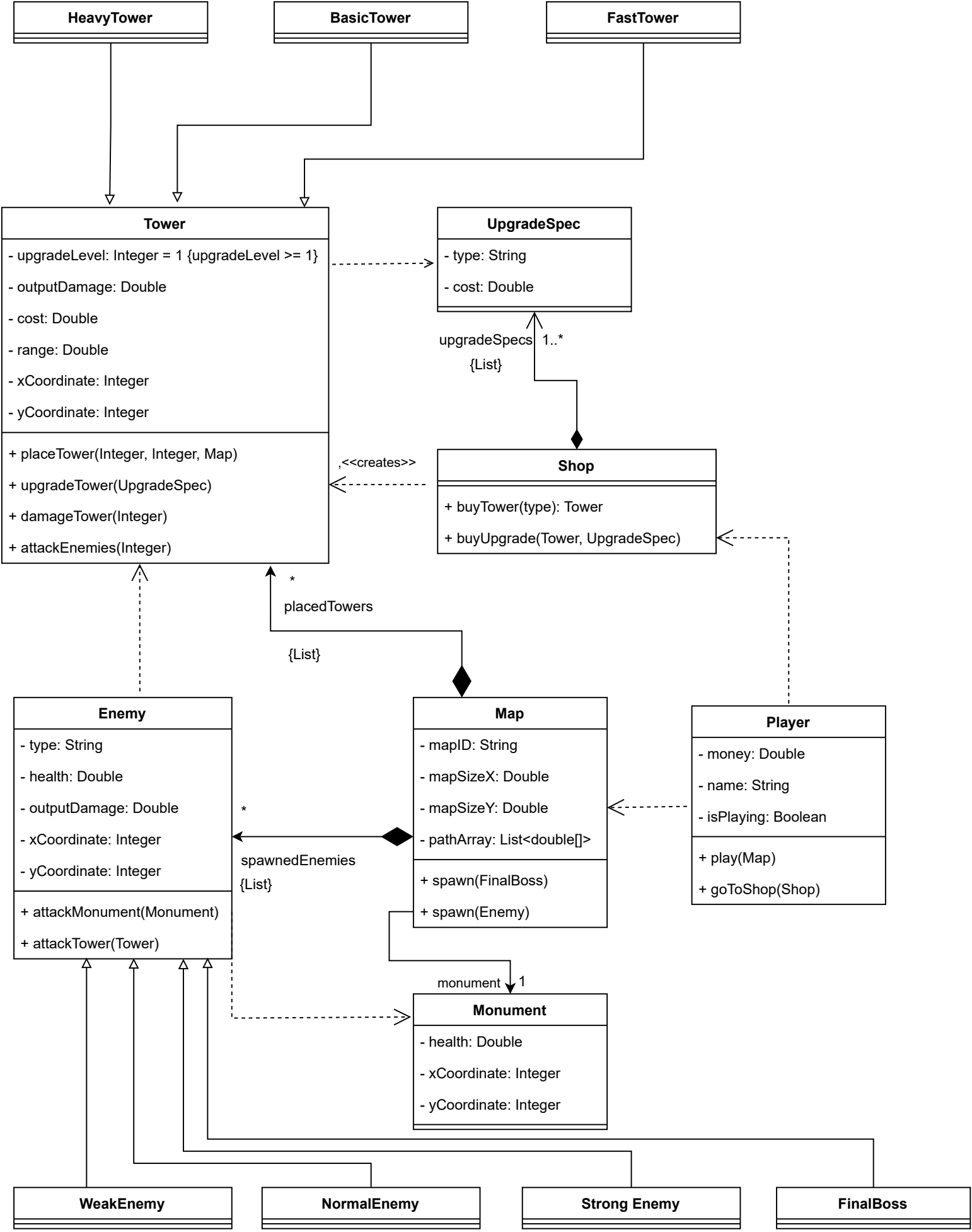
Video Submission URL: https://www.youtube.com/watch?v=UOTmNs5A3EE


HeavyTower
BasicTower
FastTower
Tower
- upgradeLevel: Integer = 1 {upgradeLevel >= 1}
- outputDamage: Double
- cost: Double
- range: Double
- xCoordinate: Integer
- yCoordinate: Integer
+ placeTower(Integer, Integer, Map)
+ upgradeTower(UpgradeSpec)
+ damageTower(Integer)
+ attackEnemies(Integer)
UpgradeSpec
- type: String
- cost: Double
upgradeSpecs
{List}
1..*
,<<creates>>
Shop
+ buyTower(type): Tower
+ buyUpgrade(Tower, UpgradeSpec)
*
placedTowers
{List}
Enemy
- type: String
- health: Double
- outputDamage: Double
- xCoordinate: Integer
- yCoordinate: Integer
+ attackMonument(Monument)
+ attackTower(Tower)
*
spawnedEnemies
{List}
Map
- mapID: String
- mapSizeX: Double
- mapSizeY: Double
- pathArray: List<double[]>
+ spawn(FinalBoss)
+ spawn(Enemy)
Player
- money: Double
- name: String
- isPlaying: Boolean
+ play(Map)
+ goToShop(Shop)
monument
1
Monument
- health: Double
- xCoordinate: Integer
- yCoordinate: Integer
WeakEnemy
NormalEnemy
Strong Enemy
FinalBoss

Within our code, we made use of the SOLID principle of Interface Segregation Principle, as each of the enemy classes implements the Enemy interface. Within the interface are the methods utilized by all enemies, getters and setters in this case, to interact with the game, whilst methods and variables individual to specific enemies are kept within the respective class.

```
public interface Enemy {
    int getDamage();
    int getHealth();
    void setHealth(int health);
    float getxLoc();
    void setxLoc(float xLoc);
    float getyLoc();
    void setyLoc(float yLoc);
    int getPathDir();
    void setPathDir(int pathDir);
    int getMoney();
    float getHealthPercentage();
}
```

In our code, we used the GRASP Advanced principle of Indirection. It means that we have reduced coupling and introduced different components and assigned the responsibility to create different objects/towers. In our code, we have a TowerInterface that is implemented by other classes for different towers.

```java
public interface TowerInterface {
    //define other shared methods

    int getxLoc();
    int getyLoc();
    public int getDamage();
    public int getRange();
    public void setRange(int range);
    public void setDamage(int damage);
    public void setUpgrade(boolean upgrade);
    public boolean getUpgrade();
    public int getTower();

    int DAMAGE = 0;
    int RANGE = 1;

    default List<Enemy> attack(List<Enemy> enemyArray, List<Float> attackArray) {
        for (int i = 0; i < enemyArray.size(); i++) {
            if (Math.abs(enemyArray.get(i).getxLoc() - getxLoc() - 37.5) < (150 * RANGE)
                    && Math.abs(enemyArray.get(i).getyLoc() - getyLoc() - 37.5) < (150 * RANGE)
                    && enemyArray.get(i).getHealth() != 0) {
                enemyArray.get(i).setHealth(enemyArray.get(i).getHealth() - DAMAGE);
                break;
            }
        }
        return enemyArray;
    }
}
```

As you see below, there are the other classes that implement this interface.

```java
public class Tower1 implements TowerInterface {
    private int damage = 5;
    private int range = 2;
    private int xLoc;
    private int yLoc;
    private boolean isUpgraded = false;
    private int tower = 1;
```

```java
public class Tower2 implements TowerInterface {
    private int damage = 5;
    private int range = 1;
    private int xLoc;
    private int yLoc;
    private boolean isUpgraded = false;
    private int tower = 2;
```

```java
public class Tower3 implements TowerInterface {
    private int damage = 15;
    private int range = 5;
    private int cooldown = 0;
    private int xLoc;
    private int yLoc;
    private boolean isUpgraded = false;
    private int tower = 3;
```

Finally, the class tower factory takes care of creating new towers for the game to ensure there is low coupling between classes, and only TowerFactory can create new Tower objects, not the classes that define them. All adapter classes, Tower1, Tower2, Tower3, and TowerFactory implement or use TowerInterface to ensure this property.

```
public class TowerFactory {
    public static TowerInterface getTower(int type, int x, int y) {
        if (type == 1) {
            return new Tower1(x, y);
        }
        if (type == 2) {
            return new Tower2(x, y);
        }
        if (type == 3) {
            return new Tower3(x, y);
        }
        return null;
    }
```

In our code, we used the advanced GRASP principle of polymorphism to effectively protect the detailed implementation of our towers under an interface for all types of towers. This allows us to effectively create a “pluggable software component” to not only render the towers without duplicate code or if statements, and also allows us to keep the rest of our code stable. In our code, the Tower1, Tower2, Tower3 classes implement the TowerInterface interface.

```java
package com.example.java;

import java.util.List;

public interface TowerInterface {
    //define other shared methods

    int getxLoc();
    int getyLoc();
    public int getDamage();
    public int getRange();
    public void setRange(int range);
    public void setDamage(int damage);
    public void setUpgrade(boolean upgrade);
    public boolean getUpgrade();
    public int getTower();

    int DAMAGE = 0;
    int RANGE = 1;

    default List<Enemy> attack(List<Enemy> enemyArray, List<Float> attackArray) {
        for (int i = 0; i < enemyArray.size(); i++) {
            if (Math.abs(enemyArray.get(i).getxLoc() - getxLoc() - 37.5) < (150 * RANGE)
                    && Math.abs(enemyArray.get(i).getyLoc() - getyLoc() - 37.5) < (150 * RANGE)
                    && enemyArray.get(i).getHealth() != 0) {
                enemyArray.get(i).setHealth(enemyArray.get(i).getHealth() - DAMAGE);
                break;
            }
        }
        return enemyArray;
    }
}
```

```java
package com.example.java;

import java.util.List;
//Regular Projectile Tower
public class Tower1 implements TowerInterface {
```

```java
package com.example.java;

import java.util.List;
//AOE Tower
public class Tower2 implements TowerInterface {
```

```java
package com.example.java;

import java.util.List;
//Sniper Tower? Cooldown?
public class Tower3 implements TowerInterface {
```

In our code we utilized the SOLID principle of single responsibility. We can see in the example below that MainActivity is only responsible for onCreate which sets and checks whether the start and quit buttons are pressed. This satisfies single responsibility since MainActivity only has one purpose which is to effectively manage a view lifecycle.

```java
package com.example.java;

import ...

public class MainActivity extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);


        Button startButton = findViewById(R.id.button);
        Button quitButton = findViewById(R.id.button2);
        startButton.setOnClickListener(l -> { //Switches to config screen
            Intent i = new Intent( packageContext: this, ConfigScreen.class);
            startActivity(i);
            finish();
        });
        quitButton.setOnClickListener(l -> { // Closes out of app
            MainActivity.this.finish();
            System.exit( status: 0);
        });
    }
}
```

In our code, we used the advanced GRASP principle of protected variations more specifically encapsulation. We used this principle to protect attributes that represent internal states that are not directly accessible. This helps us hide background details that the user does not need to know and how things in our tower defense game works. It is also helpful so that one change in a class does not change other classes directly. For example, the below code hides the unnecessary details from the user. We can use getters and setters to access and make changes to the private variables in the Enemy3 class.

```java
package com.example.java;

public class Enemy3 implements Enemy {
    private int health = 30;
    private int damage = 15;
    private float xLoc;
    private float yLoc;
    private int pathDir = 1;
    private int money = 30;
    public Enemy3() {
        this.xLoc = (float) 37.5;
        this.yLoc = (float) 337.5;
    }

    @Override
    public float getHealthPercentage() {
        return ((float) health) / ((float) 30);
    }

    @Override
    public int getHealth() {
        return health;
    }

    @Override
    public void setHealth(int health) {
        this.health = health;
    }

    @Override
    public int getDamage() {
        return damage;
    }

    @Override
    public float getxLoc() {
        return xLoc;
    }

    @Override
    public void setxLoc(float xLoc) {
        this.xLoc = xLoc;
    }

    @Override
    public float getyLoc() {
        return yLoc;
    }

    @Override
    public void setyLoc(float yLoc) {
        this.yLoc = yLoc;
    }

    public int getPathDir() {
        return pathDir;
    }

    public void setPathDir(int pathDir) {
        this.pathDir = pathDir;
    }

    @Override
    public int getMoney() {
        return money;
    }
}
```