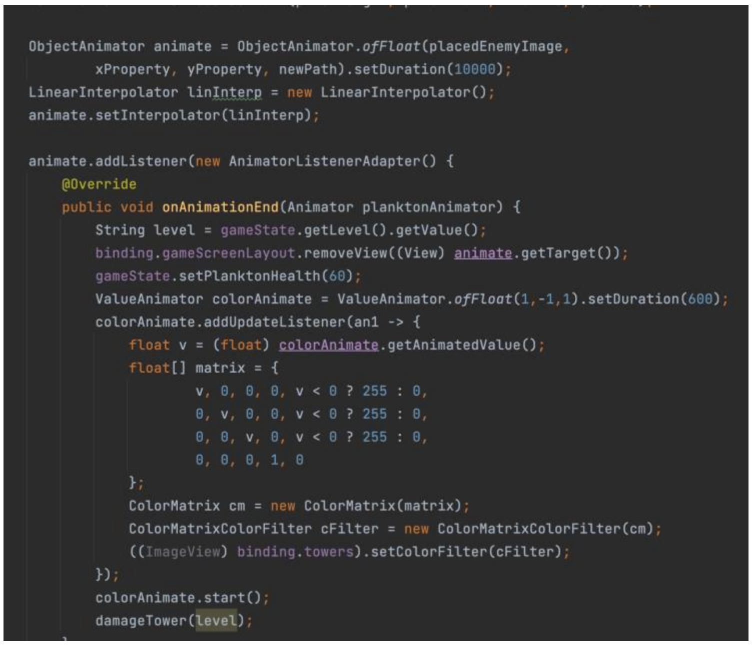
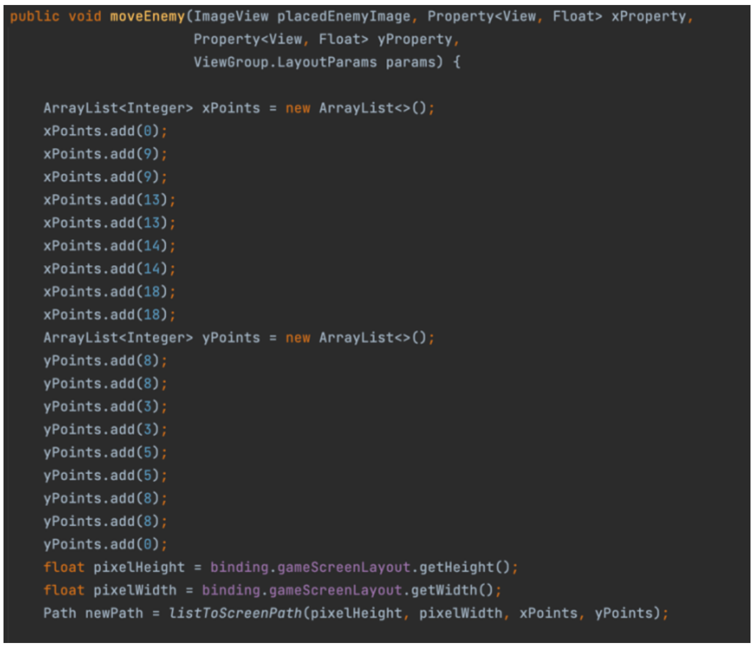
Bloaters: Long Methods

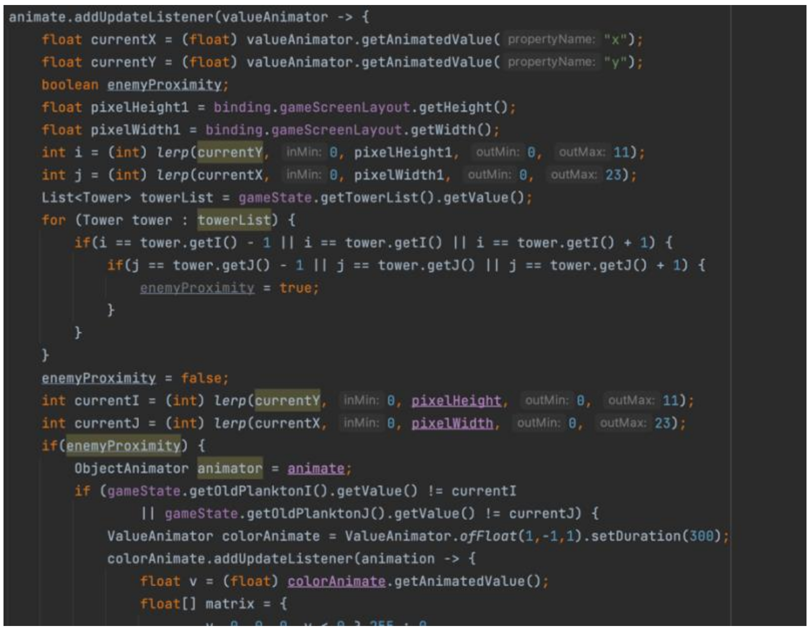
In the GameFragment class, we had a very long method by the name of 'moveEnemy' that was checking various conditions for proximities and performing all reductions and additions within it. This made the method complicated to follow and test, and hence introduced the code smell of bloaters: long methods. The pre-solved code is shown below:



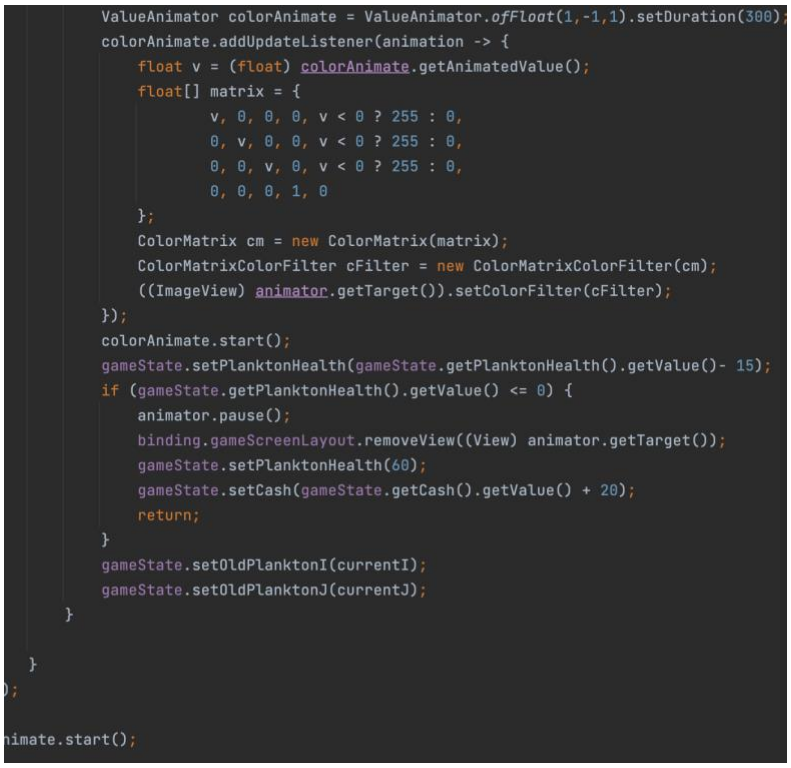
```java
ObjectAnimator animate = ObjectAnimator.ofFloat(placedEnemyImage,
        xProperty, yProperty, newPath).setDuration(10000);
LinearInterpolator linInterp = new LinearInterpolator();
animate.setInterpolator(linInterp);

animate.addListener(new AnimatorListenerAdapter() {
    @Override
    public void onAnimationEnd(Animator planktonAnimator) {
        String level = gameState.getLevel().getValue();
        binding.gameScreenLayout.removeView((View) animate.getTarget());
        gameState.setPlanktonHealth(60);
        ValueAnimator colorAnimate = ValueAnimator.ofFloat(1,-1,1).setDuration(600);
        colorAnimate.addUpdateListener(an1 -> {
            float v = (float) colorAnimate.getAnimatedValue();
            float[] matrix = {
                    v, 0, 0, 0, v < 0 ? 255 : 0,
                    0, v, 0, 0, v < 0 ? 255 : 0,
                    0, 0, v, 0, v < 0 ? 255 : 0,
                    0, 0, 0, 1, 0
            };
            ColorMatrix cm = new ColorMatrix(matrix);
            ColorMatrixColorFilter cFilter = new ColorMatrixColorFilter(cm);
            ((ImageView) binding.towers).setColorFilter(cFilter);
        });
        colorAnimate.start();
        damageTower(level);
```



```java
public void moveEnemy(ImageView placedEnemyImage, Property<View, Float> xProperty,
                      Property<View, Float> yProperty,
                      ViewGroup.LayoutParams params) {

    ArrayList<Integer> xPoints = new ArrayList<>();
    xPoints.add(0);
    xPoints.add(9);
    xPoints.add(9);
    xPoints.add(13);
    xPoints.add(13);
    xPoints.add(14);
    xPoints.add(14);
    xPoints.add(18);
    xPoints.add(18);
    ArrayList<Integer> yPoints = new ArrayList<>();
    yPoints.add(8);
    yPoints.add(8);
    yPoints.add(3);
    yPoints.add(3);
    yPoints.add(5);
    yPoints.add(5);
    yPoints.add(8);
    yPoints.add(8);
    yPoints.add(0);
    float pixelHeight = binding.gameScreenLayout.getHeight();
    float pixelWidth = binding.gameScreenLayout.getWidth();
    Path newPath = listToScreenPath(pixelHeight, pixelWidth, xPoints, yPoints);
```

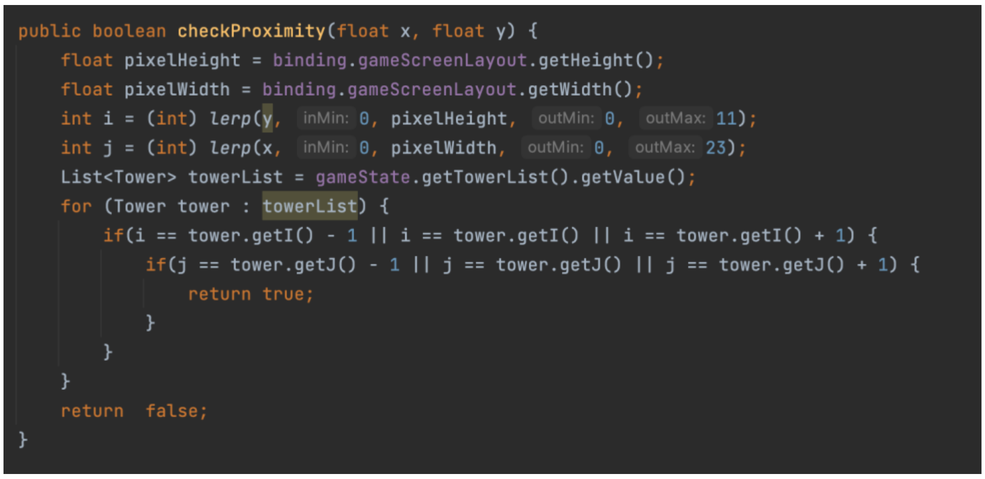


```java
animate.addUpdateListener(valueAnimator -> {
    float currentX = (float) valueAnimator.getAnimatedValue( propertyName: "x");
    float currentY = (float) valueAnimator.getAnimatedValue( propertyName: "y");
    boolean enemyProximity;
    float pixelHeight1 = binding.gameScreenLayout.getHeight();
    float pixelWidth1 = binding.gameScreenLayout.getWidth();
    int i = (int) lerp(currentY,  inMin: 0, pixelHeight1,  outMin: 0,  outMax: 11);
    int j = (int) lerp(currentX,  inMin: 0, pixelWidth1,  outMin: 0,  outMax: 23);
    List<Tower> towerList = gameState.getTowerList().getValue();
    for (Tower tower : towerList) {
        if(i == tower.getI() - 1 || i == tower.getI() || i == tower.getI() + 1) {
            if(j == tower.getJ() - 1 || j == tower.getJ() || j == tower.getJ() + 1) {
                enemyProximity = true;
            }
        }
    }
    enemyProximity = false;
    int currentI = (int) lerp(currentY,  inMin: 0, pixelHeight,  outMin: 0,  outMax: 11);
    int currentJ = (int) lerp(currentX,  inMin: 0, pixelWidth,  outMin: 0,  outMax: 23);
    if(enemyProximity) {
        ObjectAnimator animator = animate;
        if (gameState.getOldPlanktonI().getValue() != currentI
                || gameState.getOldPlanktonJ().getValue() != currentJ) {
            ValueAnimator colorAnimate = ValueAnimator.ofFloat(1,-1,1).setDuration(300);
            colorAnimate.addUpdateListener(animation -> {
                float v = (float) colorAnimate.getAnimatedValue();
                float[] matrix = {
```

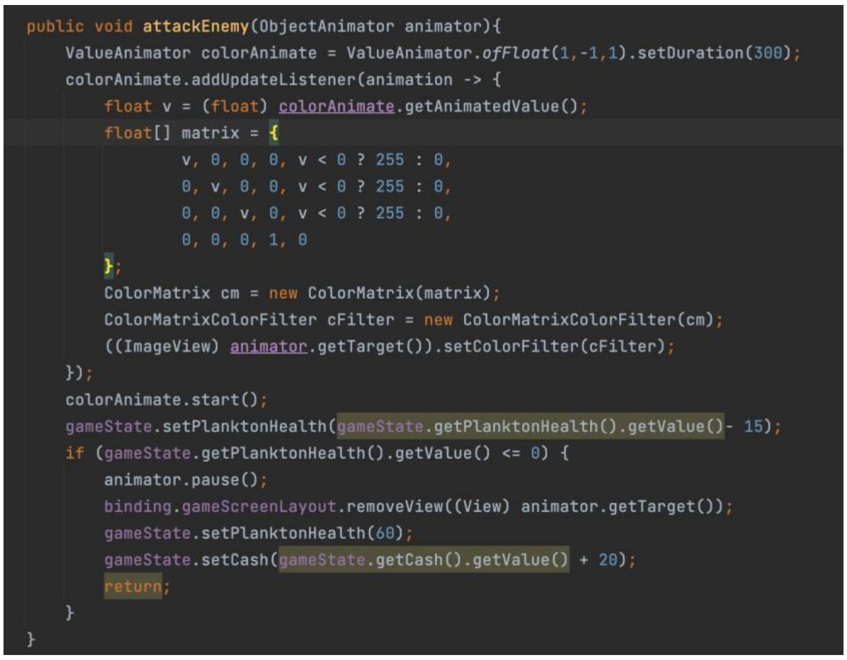


```java
            ValueAnimator colorAnimate = ValueAnimator.ofFloat(1,-1,1).setDuration(300);
            colorAnimate.addUpdateListener(animation -> {
                float v = (float) colorAnimate.getAnimatedValue();
                float[] matrix = {
                        v, 0, 0, 0, v < 0 ? 255 : 0,
                        0, v, 0, 0, v < 0 ? 255 : 0,
                        0, 0, v, 0, v < 0 ? 255 : 0,
                        0, 0, 0, 1, 0
                };
                ColorMatrix cm = new ColorMatrix(matrix);
                ColorMatrixColorFilter cFilter = new ColorMatrixColorFilter(cm);
                ((ImageView) animator.getTarget()).setColorFilter(cFilter);
            });
            colorAnimate.start();
            gameState.setPlanktonHealth(gameState.getPlanktonHealth().getValue()- 15);
            if (gameState.getPlanktonHealth().getValue() <= 0) {
                animator.pause();
                binding.gameScreenLayout.removeView((View) animator.getTarget());
                gameState.setPlanktonHealth(60);
                gameState.setCash(gameState.getCash().getValue() + 20);
                return;
            }
            gameState.setOldPlanktonI(currentI);
            gameState.setOldPlanktonJ(currentJ);
        }

    }
);

imate.start();
```

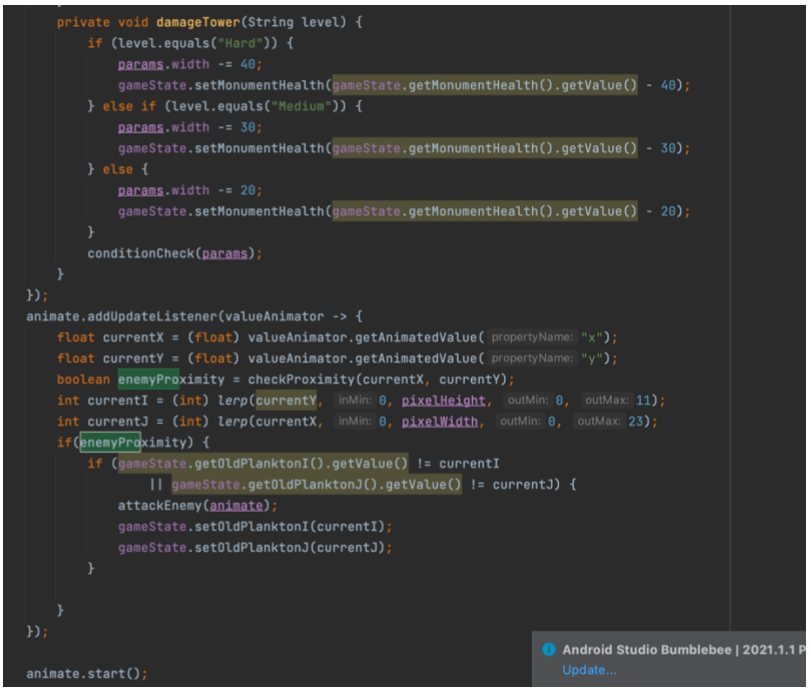
This problem was solved by introducing methods by the names of checkProximity and atatckEnemy that did the tasks mentioned earlier in separate methods respectively, returning their values to the main code. This reduced the size of the method in which all these operations were being performed substantially The new methods are shown below:



```java
public boolean checkProximity(float x, float y) {
    float pixelHeight = binding.gameScreenLayout.getHeight();
    float pixelWidth = binding.gameScreenLayout.getWidth();
    int i = (int) lerp(y,  inMin: 0, pixelHeight,  outMin: 0,  outMax: 11);
    int j = (int) lerp(x,  inMin: 0, pixelWidth,  outMin: 0,  outMax: 23);
    List<Tower> towerList = gameState.getTowerList().getValue();
    for (Tower tower : towerList) {
        if(i == tower.getI() - 1 || i == tower.getI() || i == tower.getI() + 1) {
            if(j == tower.getJ() - 1 || j == tower.getJ() || j == tower.getJ() + 1) {
                return true;
            }
        }
    }
    return  false;
}
```



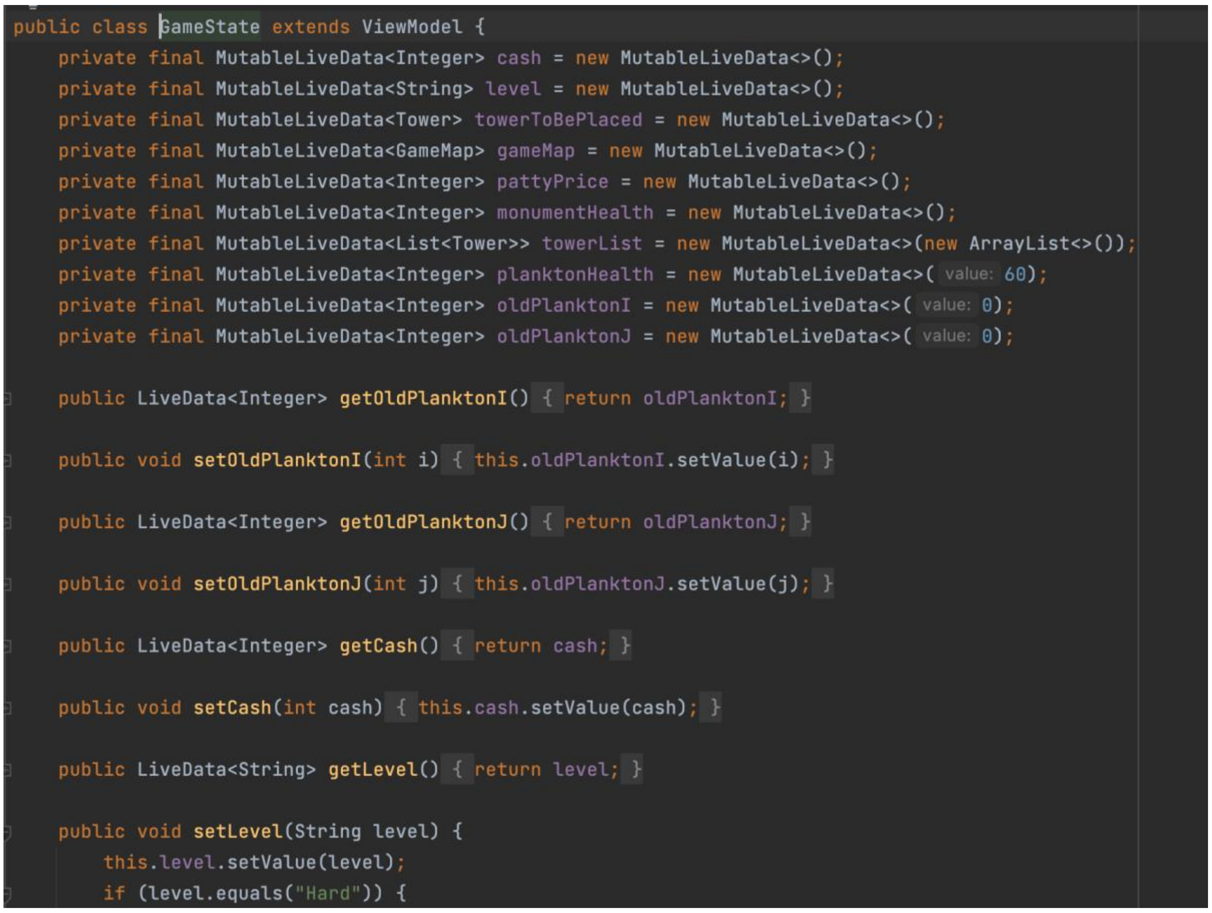
```java
public void attackEnemy(ObjectAnimator animator){
    ValueAnimator colorAnimate = ValueAnimator.ofFloat(1,-1,1).setDuration(300);
    colorAnimate.addUpdateListener(animation -> {
        float v = (float) colorAnimate.getAnimatedValue();
        float[] matrix = {
                v, 0, 0, 0, v < 0 ? 255 : 0,
                0, v, 0, 0, v < 0 ? 255 : 0,
                0, 0, v, 0, v < 0 ? 255 : 0,
                0, 0, 0, 1, 0
        };
        ColorMatrix cm = new ColorMatrix(matrix);
        ColorMatrixColorFilter cFilter = new ColorMatrixColorFilter(cm);
        ((ImageView) animator.getTarget()).setColorFilter(cFilter);
    });
    colorAnimate.start();
    gameState.setPlanktonHealth(gameState.getPlanktonHealth().getValue()- 15);
    if (gameState.getPlanktonHealth().getValue() <= 0) {
        animator.pause();
        binding.gameScreenLayout.removeView((View) animator.getTarget());
        gameState.setPlanktonHealth(60);
        gameState.setCash(gameState.getCash().getValue() + 20);
        return;
    }
}
```



```java
    private void damageTower(String level) {
        if (level.equals("Hard")) {
            params.width -= 40;
            gameState.setMonumentHealth(gameState.getMonumentHealth().getValue() - 40);
        } else if (level.equals("Medium")) {
            params.width -= 30;
            gameState.setMonumentHealth(gameState.getMonumentHealth().getValue() - 30);
        } else {
            params.width -= 20;
            gameState.setMonumentHealth(gameState.getMonumentHealth().getValue() - 20);
        }
        conditionCheck(params);
    }
});
animate.addUpdateListener(valueAnimator -> {
    float currentX = (float) valueAnimator.getAnimatedValue( propertyName: "x");
    float currentY = (float) valueAnimator.getAnimatedValue( propertyName: "y");
    boolean enemyProximity = checkProximity(currentX, currentY);
    int currentI = (int) lerp(currentY,  inMin: 0, pixelHeight,  outMin: 0,  outMax: 11);
    int currentJ = (int) lerp(currentX,  inMin: 0, pixelWidth,  outMin: 0,  outMax: 23);
    if(enemyProximity) {
        if (gameState.getOldPlanktonI().getValue() != currentI
                || gameState.getOldPlanktonJ().getValue() != currentJ) {
            attackEnemy(animate);
            gameState.setOldPlanktonI(currentI);
            gameState.setOldPlanktonJ(currentJ);
        }

    }
});

animate.start();
```

Change Preventers: Shotgun surgery

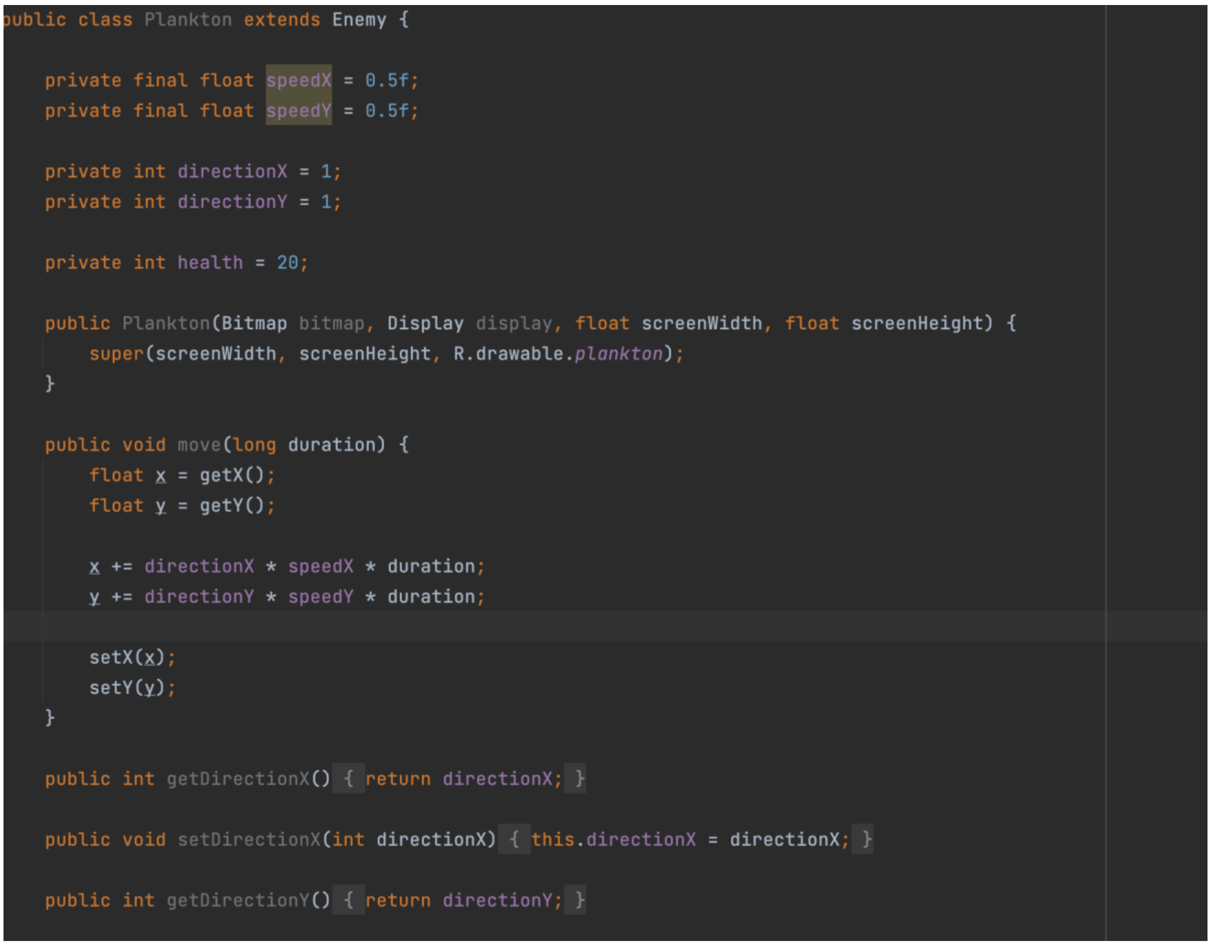
In developing most of our code earlier, we did not use any standard variables and just assigned various values to each of the different objects we had. This led to us having to change the values of the object everywhere they were declared because they were not universal. It was also party due to the fact that the same objects were declared differently in different classes. This led to very less cohesion between different classes. Also, complex Android Studio data types had to be written from scratch because of this.
We solved this problem we encountered by creating a 'GameState' class that held values for various objects across the classes and could be used to reference them as they were changed throughout the course of the program. This also led to less complexity in changing between different fragments and led to easier coding and readability. A screenshot of a part of the class 'GameState' is shown below:



```
public class GameState extends ViewModel {
    private final MutableLiveData<Integer> cash = new MutableLiveData<>();
    private final MutableLiveData<String> level = new MutableLiveData<>();
    private final MutableLiveData<Tower> towerToBePlaced = new MutableLiveData<>();
    private final MutableLiveData<GameMap> gameMap = new MutableLiveData<>();
    private final MutableLiveData<Integer> pattyPrice = new MutableLiveData<>();
    private final MutableLiveData<Integer> monumentHealth = new MutableLiveData<>();
    private final MutableLiveData<List<Tower>> towerList = new MutableLiveData<>(new ArrayList<>());
    private final MutableLiveData<Integer> planktonHealth = new MutableLiveData<>( value: 60);
    private final MutableLiveData<Integer> oldPlanktonI = new MutableLiveData<>( value: 0);
    private final MutableLiveData<Integer> oldPlanktonJ = new MutableLiveData<>( value: 0);

    public LiveData<Integer> getOldPlanktonI() { return oldPlanktonI; }

    public void setOldPlanktonI(int i) { this.oldPlanktonI.setValue(i); }

    public LiveData<Integer> getOldPlanktonJ() { return oldPlanktonJ; }

    public void setOldPlanktonJ(int j) { this.oldPlanktonJ.setValue(j); }

    public LiveData<Integer> getCash() { return cash; }

    public void setCash(int cash) { this.cash.setValue(cash); }

    public LiveData<String> getLevel() { return level; }

    public void setLevel(String level) {
        this.level.setValue(level);
        if (level.equals("Hard")) {
```

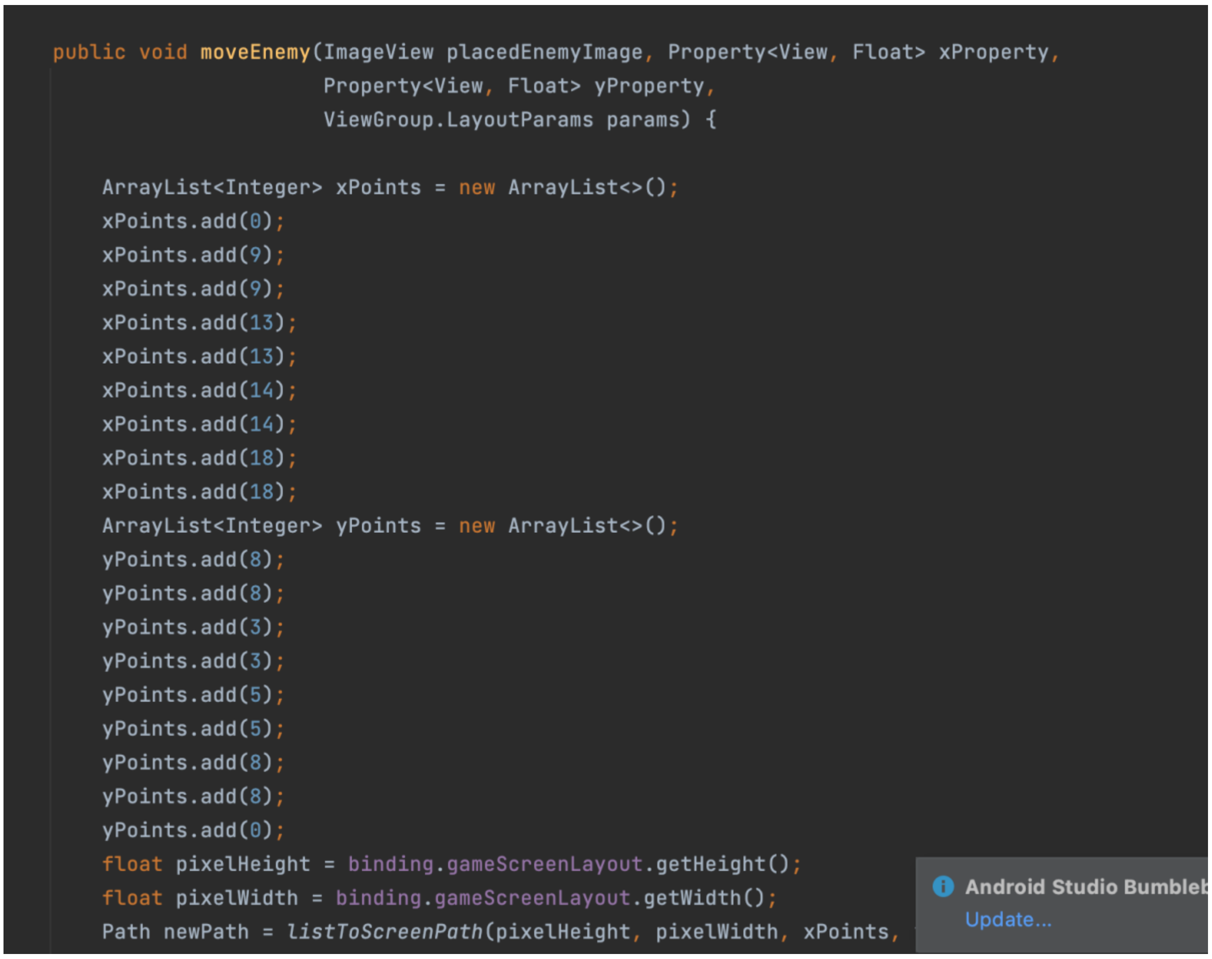
Dispensable: Lazy Class

The last code smell we solved for our code in this exercise is that of a Lazy Class. In developing M4, specifically the enemies section, we had thought of creating a generic Enemy class and having different types of enemies as sub-classes extending from the parent class. However, as we wrote the code, and in this exercise, we realized that the sub-enemy specific classes are not needed at all because we had eventually decided to eliminate much differences in how enemies differed from each other except for their visual appearance. There was just one useful method in the sub-enemy class which was move. Essentially, all methods and variables for the sub-enemy class were supposed to be common. This is depicted below:



```java
ublic class Plankton extends Enemy {

    private final float speedX = 0.5f;
    private final float speedY = 0.5f;

    private int directionX = 1;
    private int directionY = 1;

    private int health = 20;

    public Plankton(Bitmap bitmap, Display display, float screenWidth, float screenHeight) {
        super(screenWidth, screenHeight, R.drawable.plankton);
    }

    public void move(long duration) {
        float x = getX();
        float y = getY();

        x += directionX * speedX * duration;
        y += directionY * speedY * duration;

        setX(x);
        setY(y);
    }

    public int getDirectionX() { return directionX; }

    public void setDirectionX(int directionX) { this.directionX = directionX; }

    public int getDirectionY() { return directionY; }
```

This we decided to eliminate the Plankton class and replace the move method elsewhere in the code as shown below. It also made the coding logic easier for us.



```
public void moveEnemy(ImageView placedEnemyImage, Property<View, Float> xProperty,
                      Property<View, Float> yProperty,
                      ViewGroup.LayoutParams params) {

    ArrayList<Integer> xPoints = new ArrayList<>();
    xPoints.add(0);
    xPoints.add(9);
    xPoints.add(9);
    xPoints.add(13);
    xPoints.add(13);
    xPoints.add(14);
    xPoints.add(14);
    xPoints.add(18);
    xPoints.add(18);
    ArrayList<Integer> yPoints = new ArrayList<>();
    yPoints.add(8);
    yPoints.add(8);
    yPoints.add(3);
    yPoints.add(3);
    yPoints.add(5);
    yPoints.add(5);
    yPoints.add(8);
    yPoints.add(8);
    yPoints.add(0);
    float pixelHeight = binding.gameScreenLayout.getHeight();
    float pixelWidth = binding.gameScreenLayout.getWidth();
    Path newPath = listToScreenPath(pixelHeight, pixelWidth, xPoints,
```

Android Studio Bumbleb
Update...