

Code Smells:

1. Mysterious name code smell :

```
public void h() {  
    difficulty = 3;  
    playerMoney = 10;  
    monumentHealth = 10;  
    towerPrice = 10;  
}
```

```
@FXML  
public void med() {  
    difficulty = 2;  
    playerMoney = 30;  
    monumentHealth = 15;  
    towerPrice = 7;  
}
```

```
@FXML  
public void ez() {  
    difficulty = 1;  
    playerMoney = 50;  
    monumentHealth = 20;  
    towerPrice = 5;  
}
```

Names for methods not as legible, this causes issues with reading and code formatting.

```

public void easy() {
    difficulty = 1;
    playerMoney = 50;
    monumentHealth = 20;
    towerPrice = 5;
}

/**
 * Method that sets up game with difficulty
 */
@FXML
public void medium() {
    difficulty = 2;
    playerMoney = 30;
    monumentHealth = 15;
    towerPrice = 7;
}

/**
 * Method that sets up game with difficulty
 */
@FXML
public void hard() {
    difficulty = 3;
    playerMoney = 10;
    monumentHealth = 10;
    towerPrice = 10;
}

```

This was fixed by putting names which can be easily understood by whoever reads it.

2. Excessive comments code smell

```
/**
 * Tower defense (TD) is a subgenre of strategy games where the goal is to defend a player's territories or possessions by obstructing the enemy attackers
 * enemies from reaching the exits, usually achieved by placing defensive structures on or along their path of attack. This typically means building a variety
 * different structures that serve to automatically block, impede, attack or destroy enemies. Tower defense is seen as a subgenre of real-time strategy vi
 * , due to its real-time origins,[2][3] even though many modern tower defense games include aspects of turn-based strategy. Strategic choice and position
 * defensive elements is an essential strategy of the genre.
 */

/**
 * Method to go to config from start state.
 * @param event event prompting move
 * @throws IOException exception in case of IO issues
 */
@FXML
public void goToConfig(ActionEvent event) throws IOException {
    // RQ - A way to "start the game" that moves to the initial configuration screen
    difficulty = 0;
    window = (Stage) ((Node) event.getSource()).getScene().getWindow();
    fxmlLoader = new FXMLLoader(GameApplication.class.getResource("setupScreen.fxml"));
    scene = new Scene(fxmlLoader.load(), 320, 240);
    window.setScene(scene);
}
```

Very long comment on how tower defense works, code would be readable if the comment were shorter and much more readable. We do this by providing a better summary of how the tower defense game works

```
/**
 * A type of gameplay where the player chooses and builds autonomous towers to protect a location from waves of enemies.
 */

/**
 * Method to go to config from start state.
 * @param event event prompting move
 * @throws IOException exception in case of IO issues
 */
@FXML
public void goToConfig(ActionEvent event) throws IOException {
    // RQ - A way to "start the game" that moves to the initial configuration screen
    difficulty = 0;
    window = (Stage) ((Node) event.getSource()).getScene().getWindow();
    fxmlLoader = new FXMLLoader(GameApplication.class.getResource("setupScreen.fxml"));
    scene = new Scene(fxmlLoader.load(), 320, 240);
    window.setScene(scene);
}
```

3. Cyclomatic Complexity code smell:

```
public void newEnemy() {
    if (numEnemies < MAXENEMIES) {
        int i = 0;
        while (enemies[i] != null) {
            i += 1;
        }
        switch (numEnemies % 3) {
            case (0):
                enemies[numEnemies] = new Enemy(pathLocations[0], type: "enemyGoblin", pathLocation: 0, i);
                break;
            case (1):
                enemies[numEnemies] = new Enemy(pathLocations[0], type: "enemySnake", pathLocation: 0, i);
                break;
            case (2):
                enemies[numEnemies] = new Enemy(pathLocations[0], type: "enemyDevil", pathLocation: 0, i);
                break;
            default:
                enemies[numEnemies] = new Enemy(pathLocations[0], type: "enemyGoblin", pathLocation: 0, i);
                break;
        }
        numEnemies += 1;
    }
}
```

Switch statements usually make it harder for the code to be extended. This violates the principle that the code should be open to extension but closed to modification. If we wanted to support a new case or a new scenario, it would cause potential bugs. This could be broken up into smaller functions.

We use a helper method to instantiate a new enemy and then call it in newEnemy() so we don't have to check number of enemies and instantiate new enemies in the same method.

```
public void newEnemy() {
    if (numEnemies < MAXENEMIES) {
        int i = 0;
        while (enemies[i] != null) {
            i += 1;
        }
        instantiateNewEnemy(pathLocations[0], enemyGoblin: "enemyGoblin", i: 0, i);

        numEnemies += 1;
    }
}

private void instantiateNewEnemy(Button pathLocation, String enemyGoblin, int i, int i1) {
```

4. Contrived Complexity Code Smell:

```
public static void interact(Tower tower, TileObject enemyTile) {
    Enemy target = null;
    int damage = 0;
    for (Enemy enemy : enemies) {
        if (enemy.getLocation().equals(enemyTile.getLocation())) {
            target = enemy;
        }
    }
    if (target != null) {
        String eType = target.getType();
        String tType = tower.getType();
        if (tType.equals("PaperTower")) {
            playerMoney += 1;
        } else {
            if (tType.equals("IceTower")) {
                if (eType == "enemySnake") {
                    damage = 2;
                } else if (eType.equals("enemyGoblin")) {
                    damage = 1;
                } else if (eType.equals("enemyDevil")) {
                    damage = 0;
                }
            } else if (tType.equals("RockTower")) {
                if (eType == "enemySnake") {
                    damage = 0;
                } else if (eType.equals("enemyGoblin")) {
                    damage = 2;
                } else if (eType.equals("enemyDevil")) {
                    damage = 1;
                }
            } else if (tType.equals("IceTower")) {
                if (eType == "enemySnake") {
                    damage = 1;
                } else if (eType.equals("enemyGoblin")) {
                    damage = 0;
                } else if (eType.equals("enemyDevil")) {
                    damage = 2;
                }
            }
        }
    }
}
```

This is a contrived complexity code smell because it seems to be using complex designs as in excessive loop statements instead of just using simpler designs. We can fix this by using simpler designs such as

function, we could just implement this in the IceTower method, the RockTower method, the PaperTower method and the FireTower method in the GameFunctionality class.

This can be seen in the code below, the functionalities were added to respective methods so now it's more concise and readable

```
public class Tower {
    public IceTower(Button location, int x, int y) {

        super(location, "IceTower", x, y);

        if (eType == "enemySnake") {
            damage = 2;
        } else if (eType.equals("enemyGoblin")) {
            damage = 1;
        } else if (eType.equals("enemyDevil")) {
            damage = 0;
        }
    }

    public RockTower(Button location, int x, int y) {

        super(location, "RockTower", x, y);

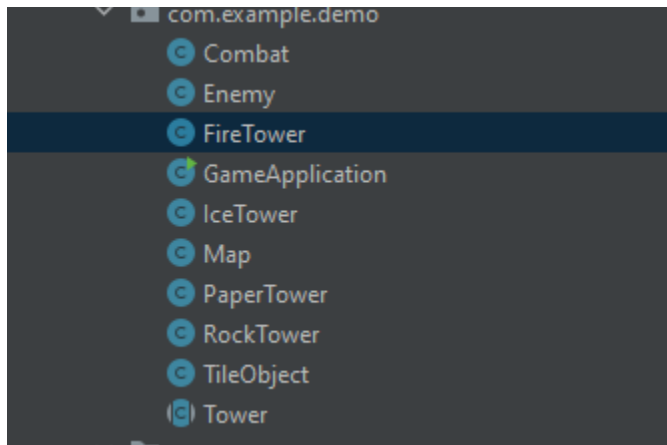
        if (eType == "enemySnake") {
            damage = 0;
        } else if (eType.equals("enemyGoblin")) {
            damage = 2;
        } else if (eType.equals("enemyDevil")) {
            damage = 1;
        }
    }

    public FireTower(Button location, int x, int y) {

        super(location, "FireTower", x, y);

        if (eType == "enemySnake") {
            damage = 1;
        } else if (eType.equals("enemyGoblin")) {
            damage = 0;
        } else if (eType.equals("enemyDevil")) {
            damage = 2;
        }
    }
}
```

5. Lazy Class/Freeloader Code Smell



We have different classes for different types of towers, this is not very optimal. We don't really use the class itself when running the game application. We could have probably used variables and declared the fxml files inside of them. Instead, the way we are doing it is just taking up more memory in the file.

```
package com.example.demo;

import javafx.scene.control.Button;

/**
 * An Ice tower.
 */
public class Tower {
    public IceTower(Button location, int x, int y) {
        super(location, "IceTower", x, y);
    }
    public RockTower(Button location, int x, int y) {
        super(location, "RockTower", x, y);
    }

    public FireTower(Button location, int x, int y) {
        super(location, "FireTower", x, y);
    }
    public PaperTower(Button location, int x, int y) {
        super(location, "PaperTower", x, y);
    }
}
```

6. Excessive use of literals code smell

```
private static Button[] pathLocations;

private static int[][] path = {{1, 1}, {1, 2}, {1, 3}, {1, 4}, {1, 5}, {1, 6}, {1, 7}, {2, 7}, {3, 7}, {3, 6}, {3, 5},
    {3, 4}, {3, 3}, {3, 2}, {3, 1}, {4, 1}, {5, 1}, {5, 2}, {5, 3}, {5, 4}, {5, 5}, {5, 6}, {5, 7}, {6, 7}};

/**
 * Length of the game path.
 */
private static final int PATHLENGTH = 25;

/**
 * List of all path objects (in order).
 */
private static TileObject[] path;

/**
 * 2D array of all objects on the map.
 */
private static TileObject[][] map;

/**
 * Final variable for maximum number of enemies.
 */
private static final int MAXENEMIES = 10;

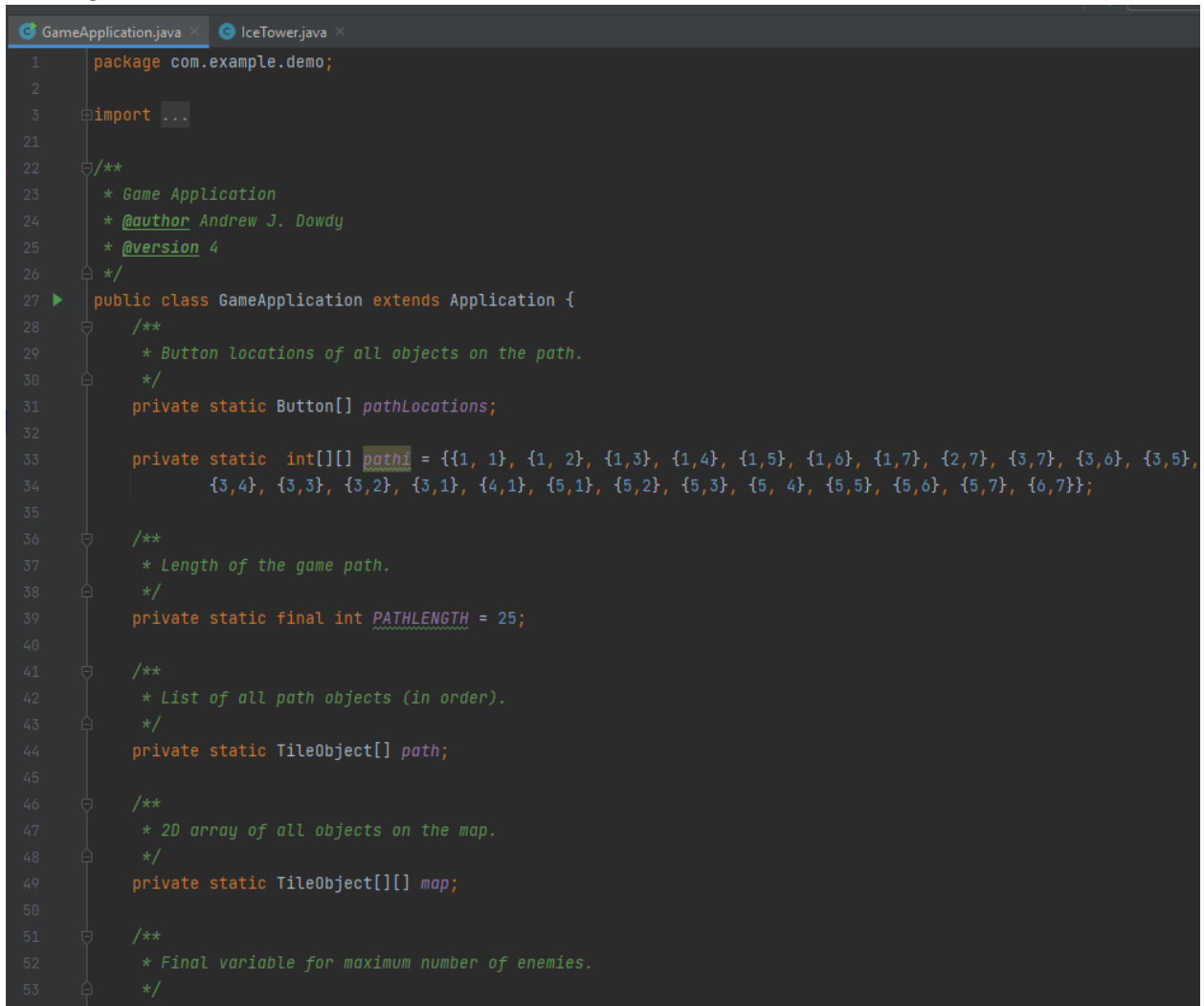
/**
 * Gets the maximum number of enemies
 * @return maximum number of enemies
 */
public static int getMaxEnemies() { return MAXENEMIES; }

/**
 * Combat animation timer.
 */
private static Combat combatController;

/**
```

We use static variables for our Game Application. This is not a good use because it prevents extension, it's not as object oriented, it prevents re-usability and it's difficult to write unit tests for it. Static variables maintain state across instances hence making it more difficult to write unit tests for it. It is very difficult to isolate changes from static variable to a single test.

7. Large class code smell



```
1 package com.example.demo;
2
3 import ...
4
5 /**
6  * Game Application
7  * @author Andrew J. Dowdy
8  * @version 4
9  */
10 public class GameApplication extends Application {
11     /**
12      * Button locations of all objects on the path.
13      */
14     private static Button[] pathLocations;
15
16     private static int[][] path = {{1, 1}, {1, 2}, {1, 3}, {1, 4}, {1, 5}, {1, 6}, {1, 7}, {2, 7}, {3, 7}, {3, 6}, {3, 5},
17                                     {3, 4}, {3, 3}, {3, 2}, {3, 1}, {4, 1}, {5, 1}, {5, 2}, {5, 3}, {5, 4}, {5, 5}, {5, 6}, {5, 7}, {6, 7}};
18
19     /**
20      * Length of the game path.
21      */
22     private static final int PATHLENGTH = 25;
23
24     /**
25      * List of all path objects (in order).
26      */
27     private static TileObject[] path;
28
29     /**
30      * 2D array of all objects on the map.
31      */
32     private static TileObject[][] map;
33
34     /**
35      * Final variable for maximum number of enemies.
36      */
37     private static final int MAXENEMIES = 10;
```

The GameApplication.java file has about 1000 lines of code. We could probably significantly reduce our code by using some efficient methods in implementing our code such as removing methods which share the same functionality, removing lazy methods etc

We fix this code smell by putting the game functionality into a separate class. This would reduce this large class and probably make it to around 500 lines, reducing it almost by half.

This is shown in the screenshot below.

```

package com.example.demo;

import javafx.scene.control.Button;

public class GameFunctionality {
    public void processEnemies() throws IOException {
        for (Enemy enemy : enemies) {
            if (enemy != null) {
                // move along path until monument
                if (enemy.getPathLocation() < 23) {
                    // move down path
                    path[enemy.getPathLocation()].updateType("map");
                    enemy.setPathLocation(enemy.getPathLocation() + 1);
                    path[enemy.getPathLocation()].updateType(enemy.getType());
                } else {
                    // hit monument - damage
                    damageMonument(1);
                    deleteEnemy(enemy);
                }
            }
        }
    }

    @FXML
    public void buyTower(ActionEvent event) throws IOException {
        // determine what tower is being purchased
        Button source = (Button) event.getSource();
        String towerType = source.getText();
        String id = currentPurchase.getId();
        int x = id.charAt(0);
        int y = id.charAt(1);
        if (playerMoney >= towerPrice) {
            // subtract funds
            playerMoney -= towerPrice;
            // select tower
            Tower newTower;
            // make new tower using player input on tower location and type
            if (towerType.equals("PaperTower")) {
                //System.out.println("User purchasing paper tower");
                newTower = new PaperTower(currentPurchase, x, y);
            } else if (source.getText().equals("IceTower")) {
                //System.out.println("User attempting purchasing paper tower");
            }
        }
    }
}

```

8. Code Duplication code smell

```
5  /**
6   * Method that sets up game with difficulty to easy.
7   */
8   @FXML
9   public void easy() {
10       difficulty = 1;
11       playerMoney = 50;
12       monumentHealth = 20;
13       towerPrice = 5;
14   }
15
16   /**
17   * Method that sets up game with difficulty to medium.
18   */
19   @FXML
20   public void medium() {
21       difficulty = 2;
22       playerMoney = 30;
23       monumentHealth = 15;
24       towerPrice = 7;
25   }
26
27   /**
28   * Method that sets up game with difficulty to hard.
29   */
30   @FXML
31   public void hard() {
32       difficulty = 3;
33       playerMoney = 10;
34       monumentHealth = 10;
35       towerPrice = 10;
36   }
```

The 3 methods are similar to each other. We could have made one method and used constructors to set parameters for difficulty, playerMoney, monumentHealth and TowerPrice. This would probably improve code readability and code scalability.