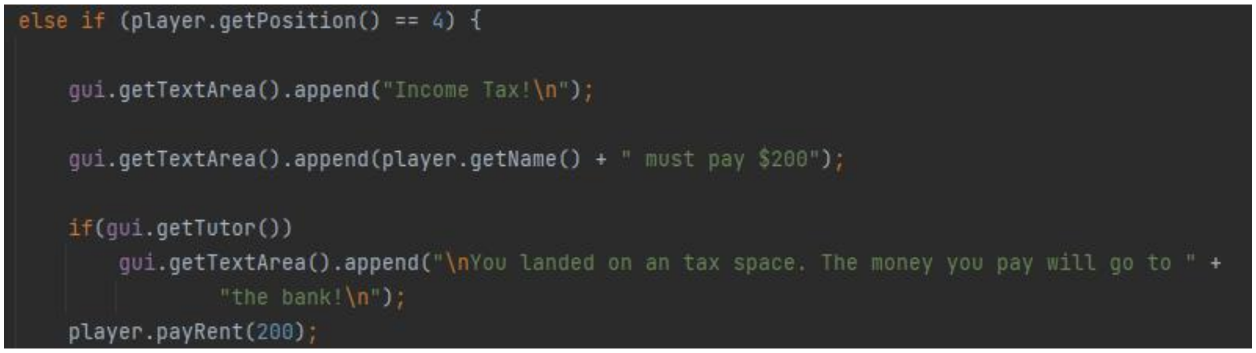
Refactoring

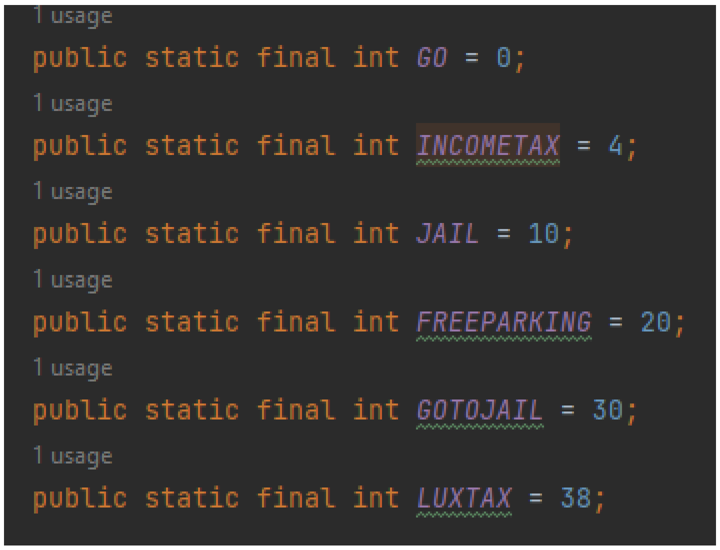
Code Smells:

1. **Magic Numbers**
   In phase 2, the OtherSpace class originally contained a large amount of seemingly unexplainable magic numbers. As the OtherSpace class is reserved for handling the miscellaneous spaces on the board such as Income Tax, Free Parking, Go To Jail, Go, the code originally contained statements such as:



```
else if (player.getPosition() == 4) {

    gui.getTextArea().append("Income Tax!\n");

    gui.getTextArea().append(player.getName() + " must pay $200");

    if(gui.getTutor())
        gui.getTextArea().append("\nYou landed on an tax space. The money you pay will go to " +
                "the bank!\n");
    player.payRent(200);
```

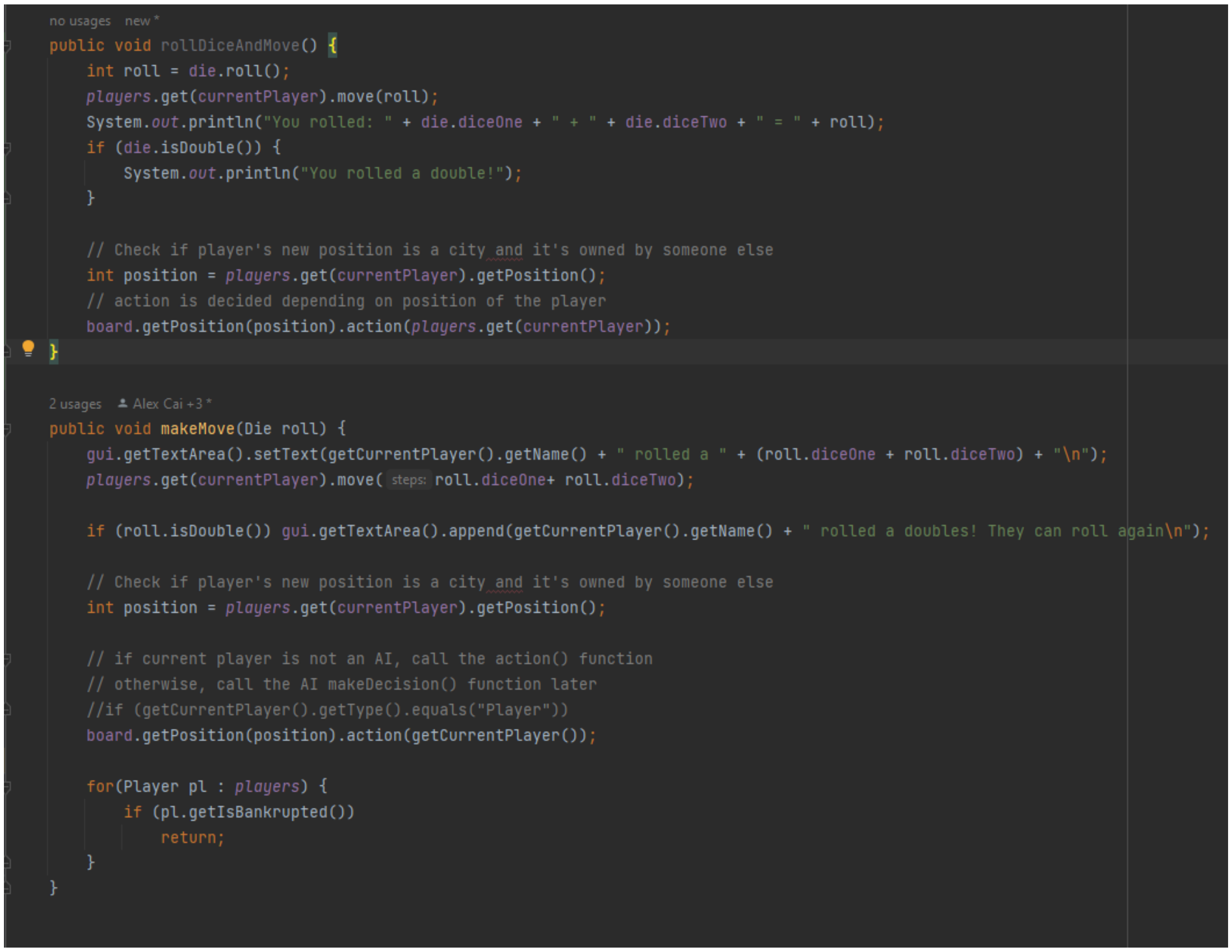
The number 4 is used for the position on the board where Income Tax lies, as it will always be in board position 4. However, this would be problematic for anyone unfamiliar with our position system or the game of Monopoly itself, and would seem like completely arbitrary numbers. We have since refactored all the magic numbers in this class by simply setting these numbers as declared constants in the beginning of the class, in order to avoid confusion and difficulty of comprehension when inspecting the code, as such:



```
1 usage
public static final int GO = 0;
1 usage
public static final int INCOMETAX = 4;
1 usage
public static final int JAIL = 10;
1 usage
public static final int FREEPARKING = 20;
1 usage
public static final int GOTOJAIL = 30;
1 usage
public static final int LUXTAX = 38;
```

2. **Duplicate Code**

After phase 2, we discovered that there was duplicate code located in the Game class. There were originally 2 different functions in the Game class to roll the dice and move:

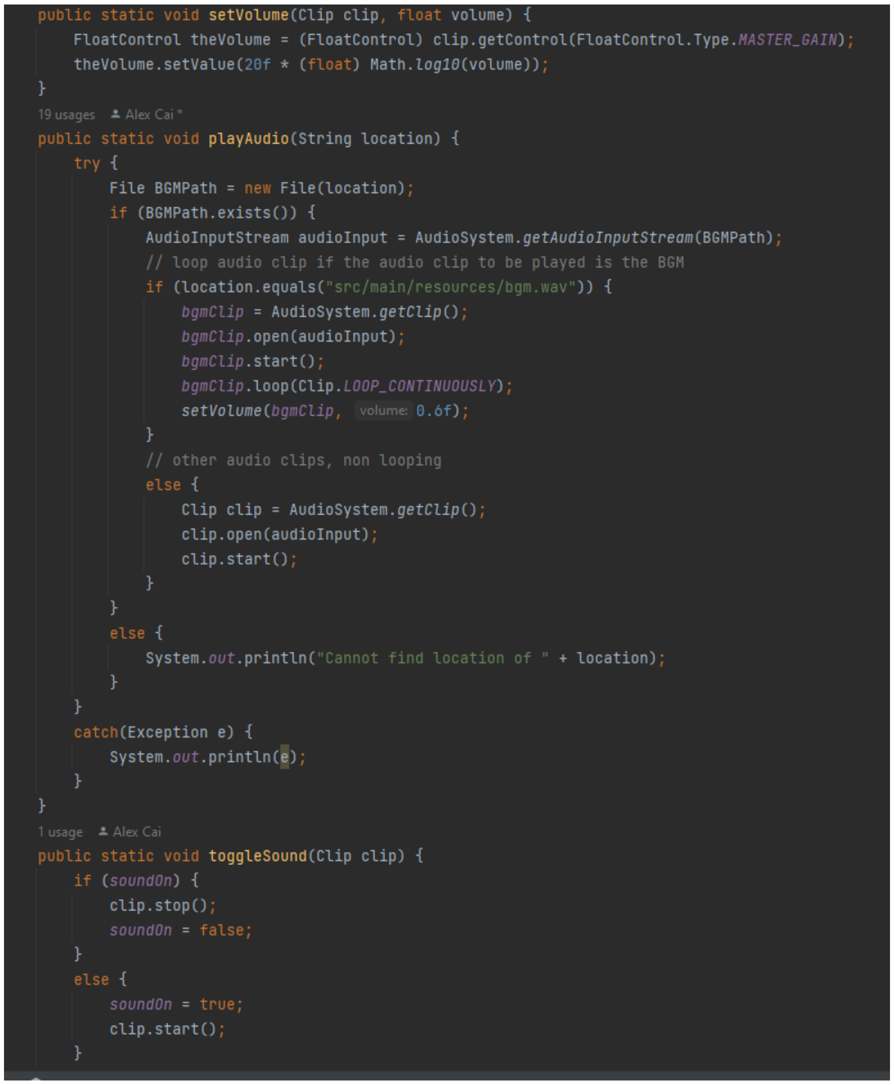


```
no usages   new *
public void rollDiceAndMove() {
    int roll = die.roll();
    players.get(currentPlayer).move(roll);
    System.out.println("You rolled: " + die.diceOne + " + " + die.diceTwo + " = " + roll);
    if (die.isDouble()) {
        System.out.println("You rolled a double!");
    }

    // Check if player's new position is a city and it's owned by someone else
    int position = players.get(currentPlayer).getPosition();
    // action is decided depending on position of the player
    board.getPosition(position).action(players.get(currentPlayer));
}

2 usages   Alex Cai +3 *
public void makeMove(Die roll) {
    gui.getTextArea().setText(getCurrentPlayer().getName() + " rolled a " + (roll.diceOne + roll.diceTwo) + "\n");
    players.get(currentPlayer).move( steps: roll.diceOne+ roll.diceTwo);

    if (roll.isDouble()) gui.getTextArea().append(getCurrentPlayer().getName() + " rolled a doubles! They can roll again\n");

    // Check if player's new position is a city and it's owned by someone else
    int position = players.get(currentPlayer).getPosition();

    // if current player is not an AI, call the action() function
    // otherwise, call the AI makeDecision() function later
    //if (getCurrentPlayer().getType().equals("Player"))
    board.getPosition(position).action(getCurrentPlayer());

    for(Player pl : players) {
        if (pl.getIsBankrupted())
            return;
    }
}
```

As you can see, these two functions are very similar and perform identical actions. The issue was that in our code, there are separate functions for an AI player's turn to move and a human player's turn to move. These functions did not call the same move() function, which resulted in inconsistencies. For a period of time, we were confused why the AI player's dice roll was not being displayed in the GUI text, and why the game never seemed to recognize when an AI player was bankrupt. We realized that it was because the AI turn function was calling the rollDiceAndMove() function, which outputted the dice roll to the console rather than our GUI text box. We decided to remove the rollDiceAndMove() function and use the makeMove() function for all instances, in order to maintain consistency. At the same time, we realized that this may have indicated a flaw in the structure of our project, having two separate functions for AI and player turns.

3. Divergent Change: Adding Functionality

We added audio into the game pretty late into development, and we began by naively adding Audio objects into every single class that needed audio, and writing essentially duplicate methods for each class, just to play and stop audio clips. Every single method that interacted with audio needed significant changes in order to create an Audio object, designate the AudioStream, etc. This proved to be very cumbersome and bloated our code significantly. To fix this, we created a utility Audio class with methods as follows:



```java
public static void setVolume(Clip clip, float volume) {
    FloatControl theVolume = (FloatControl) clip.getControl(FloatControl.Type.MASTER_GAIN);
    theVolume.setValue(20f * (float) Math.log10(volume));
}
19 usages  Alex Cai *
public static void playAudio(String location) {
    try {
        File BGMPath = new File(location);
        if (BGMPath.exists()) {
            AudioInputStream audioInput = AudioSystem.getAudioInputStream(BGMPath);
            // loop audio clip if the audio clip to be played is the BGM
            if (location.equals("src/main/resources/bgm.wav")) {
                bgmClip = AudioSystem.getClip();
                bgmClip.open(audioInput);
                bgmClip.start();
                bgmClip.loop(Clip.LOOP_CONTINUOUSLY);
                setVolume(bgmClip, volume: 0.6f);
            }
            // other audio clips, non looping
            else {
                Clip clip = AudioSystem.getClip();
                clip.open(audioInput);
                clip.start();
            }
        }
        else {
            System.out.println("Cannot find location of " + location);
        }
    }
    catch(Exception e) {
        System.out.println(e);
    }
}
1 usage  Alex Cai
public static void toggleSound(Clip clip) {
    if (soundOn) {
        clip.stop();
        soundOn = false;
    }
    else {
        soundOn = true;
        clip.start();
    }
```

This class allows us to simply call its methods globally without having to worry about creating objects, and it can be called with a single line, with the file location as the parameter. This change improved our code by reducing code bloat and enforcing the single responsibility principle: all code involving audio can be found in this one class, making it easy to examine and refactor.