

RANDOM SINGLENESS

Sarah Lawrence, University of Southern Maine

04/09/2025

Description

The purpose of this assignment is to gain experience with the Singleton design pattern and to develop a framework in which one can test methods that use randomness as part of their computations. Additionally, it provides an opportunity to assess, at least in part, the ability to implement design patterns effectively.

Build a First Singleton

The task was to use AI (ChatGPT 3.5) to rewrite that class as a singleton. Then compare the code created by that tool to the style of singleton shown in the lecture. Preparation for the task involved creating a referee class. This was to contain the program responsible for the logic that runs a game of Bulldog. Then convert it to singleton using AI. Building and comparing the AI and lecture singletons can be seen below.

AI Singleton Results:

Building the Singleton took over 2 minutes and 10 seconds with AI. This includes bug testing. Overall, it was successful on the second prompt. There were a couple of changes that the AI made to the original work. The prompts and changes are as follows.

Prompt:

Human: Rewrite that class as a Singleton (referee class code)

ChatGPT: To rewrite the Referee class as a Singleton, you'll need to ensure that only one instance of the class exists throughout the application. (code)

Some of the old referee class:

```
public class Referee {  
    private PlayerList players;  
    private int currentPlayerIndex;  
  
    public Referee(PlayerList players) {  
        this.players = players;  
        this.currentPlayerIndex = 0;  
    }  
}
```

```
public Player getCurrentPlayer() {
    return players.getPlayers().get(currentPlayerIndex);
}
```

(This code keeps going)

Some of the new referee class:

```
public class Referee {
    private static Referee instance; // Singleton instance
    private PlayerList players;
    private int currentPlayerIndex;

    // Private constructor to prevent instantiation
    private Referee(PlayerList players) {
        this.players = players;
        this.currentPlayerIndex = 0;
    }

    // Public method to provide access to the instance
    public static Referee getInstance(PlayerList players) {
        if (instance == null) {
            instance = new Referee(players);
        }
        return instance;
    }

    public Player getCurrentPlayer() {
        return players.getPlayers().get(currentPlayerIndex);
    }
}
```

(This code keeps going)

the constructor was made private and created a getInstance() method.

The only thing that didn't work was in the GameGUI the line

```
referee = new Referee(playerList);
```

It was changed to

```
referee = Referee.getInstance(playerList);
```

Then the code ran successfully.

Lecture Singleton

Building this singleton was not required. This singleton is from the Gang of Four book or lecture L03.26 slide 10, created by David Levine. What it looks like can be seen below.

Lecture singleton:

```
public class MySingleton {
    private static MySingleton instance = null;
    private int myVariable;

    private MySingleton() {
        //Private constructor to prevent iteration from outside the class
        myVariable = 0;
    }

    public static MySingleton getInstance() {
        //Lazy initialization the instance is created when the getInstance()
        //method is first called
        if(instance == null){
            instance = new MySingleton();
        }
        return instance;
    }

    public int getMyVariable() {
        return myVariable;
    }

    public void setMyVariable(int newValue) {
        myVariable = newValue;
    }
}
```

Comparison

They both succeeded in implementing the Singleton Pattern. This is because both classes restrict instantiation using a private static instance and a private constructor. They both also create the instance only when `getInstance()` is first called. They did have differences. Their purpose, parameters, and methods were a bit different. These differences can be seen better in the table below.

Change Dice class

Aspect	Referee	MySingleton
Purpose	Manages turn-based game using a PlayerList.	A generic singleton with a variable myVariable.
Constructor	Takes a PlayerList parameter.	Takes no parameters.
getInstance() Method	Requires a PlayerList argument on the first call.	Takes no arguments.
Use Case	Built for coordinating a game (turns, players, game end).	Generic example could be adapted to anything.

I prompted the AI to create the object of the Random class in the Dice class as a Singleton. When completed, it should be able to have Dice of different sizes. For example, 6 and 2 would be the ones that would be used. While ensuring they all share a Singleton Random object.

Prompt:

Human: Change things so that the object of the Random class in your Dice class is a Singleton. When you are done, you will be able to have Dice of different sizes (6 and 2 would be the ones you would use - the former in all cases, and the latter for RandomPlayer) but all of those would share a Singleton Random object. (Dice code)

ChatGPT: (new dice class code)

It took in total 1 minutes and 13 seconds to complete and bug test. The old and new code can be seen below.

Old Dice class:

```
import java.util.Random;

public class Dice {
    private Random random;

    public Dice() {
        random = new Random();
    }

    public int roll() {
        return random.nextInt(6) + 1;
    }
}
```

New Dice class:

```
import java.util.Random;

public class Dice {
    private static final Random random = new Random();
```

```

private final int dice_sides;

public Dice(int dice_sides) {
    this.dice_sides = dice_sides;
}

public int roll() {
    return random.nextInt(dice_sides) + 1;
}
}

```

What's changed was a couple of things. One is that private `Random random;` is now private static final `Random random = new Random();`. Making it only one instance of random. Another is the addition of a new variable for the dice sides called `dice_sides`. This new variable was added to the method `dice`. Allowing the dice to change size. Here is a test that helped show that the dice can change size using the singleton random object.

DiceTest class:

```

public class DiceTest {
    public static void main(String[] args) {
        Dice d6 = new Dice(6);
        Dice d2 = new Dice(2);

        System.out.println("Rolling d6 (6-sided) 10 times:");
        for (int i = 0; i < 10; i++) {
            System.out.print(d6.roll() + " ");
        }

        System.out.println("\nRolling d2 (2-sided) 10 times:");
        for (int i = 0; i < 10; i++) {
            System.out.print(d2.roll() + " ");
        }
    }
}

```

Results from terminal:

```

Rolling d6 (6-sided) 10 times:
2 2 5 4 2 6 2 2 1 4
Rolling d2 (2-sided) 10 times:
2 2 2 1 2 2 1 2 2 2 %

```

As can be seen the 6 sided dice displays random numbers ranging from 1-6. While the two sided dice only has random numbers ranging from 1-2.

Create an abstract RandomDice superclass

My IDE didn't allow me to extract a superclass from a class. So I made the AI do it. It took me 1 minute and 47 seconds to make and bug test. My superclass kept track of the number of sides of the Dice and have an abstract roll() method. The Random object stayed with the Dice class.

Reminder of the old Dice class:

```
import java.util.Random;

public class Dice {
    private static final Random random = new Random();
    private final int dice_sides;

    public Dice(int dice_sides) {
        this.dice_sides = dice_sides;
    }

    public int roll() {
        return random.nextInt(dice_sides) + 1;
    }
}
```

New Dice class:

```
import java.util.Random;

public class Dice extends RandomDice {
    private static final Random random = new Random(); // Singleton Random
    instance

    public Dice(int sides) {
        super(sides); // Call constructor of RandomDice
    }

    @Override
    public int roll() {
        return random.nextInt(sides) + 1; // Random number between 1 and
        sides
    }
}
```

New abstract RandomDice superclass:

```
public abstract class RandomDice {
    protected final int sides;
```

```

public RandomDice(int sides) {
    this.sides = sides;
}

public abstract int roll();
}

```

JUnit Testing

Create a FakeRandom subclass of RandomDice

The FakeRandom implemented the roll method. In this class, it's not pretending to be random. I prompted the AI to perform the task here were the prompts.

Prompts:

Human: in Junit "Create a FakeRandom subclass of RandomDice. FakeRandom should implement the roll method (Dice and RandomDice code)

ChatGPT: Great! You're working with a dice-rolling abstraction where RandomDice is the base class, Dice uses real randomness....

This prompt produced two FakeRandom classes one that was simple and only tested one roll. The other was a little more complex but allowed to test multiple rolls. The two are provided below.

One roll FakeRandom class:

```

public class FakeRandom extends RandomDice {
    private final int fixedRoll;

    public FakeRandom(int sides, int fixedRoll) {
        super(sides);
        this.fixedRoll = fixedRoll;
    }

    @Override
    public int roll() {
        return fixedRoll;
    }
}

```

Multi roll FakeRandom class:

```

public class FakeRandom extends RandomDice {
    private final int[] sequence;
    private int index;
}

```

```

public FakeRandom(int sides, int... sequence) {
    super(sides);
    this.sequence = sequence;
    this.index = 0;
}

@Override
public int roll() {
    int value = sequence[index % sequence.length];
    index++;
    return value;
}
}

```

I used the second code FakeRandom class cause I wanted to test multiple rolls. Overall, This process took seconds.

After creating the code I wanted to test it. I wanted to see if AI could create a Junit test to make sure the code was working properly. The AI created the following.

```

import static org.junit.Assert.*;
import org.junit.Test;

public class FakeRandomTest {

    @Test
    public void testFixedRoll() {
        RandomDice dice = new FakeRandom(6, 4);
        assertEquals(4, dice.roll());
        assertEquals(4, dice.roll()); // Should always return 4
    }

    @Test
    public void testSequenceRolls() {
        RandomDice dice = new FakeRandom(6, 2, 5, 3);
        assertEquals(2, dice.roll());
        assertEquals(5, dice.roll());
        assertEquals(3, dice.roll());
        assertEquals(2, dice.roll()); // Loops back
    }
}

```

In the testFixedRoll test, the dice is initialized to always return the fixed value 4. The test checks that multiple calls to roll() consistently return 4. In the testSequenceRolls method, the dice are initialized with a sequence of values (2, 5, 3). The test checks that roll() returns these values in order. It loops back to the start once the sequence ends. These tests ensure correct behavior from FakeRandom, making it useful to check for errors. After performing the tests on the code

it passed.

Create a SevenPlayer class

SevenPlayer was designed just like FifteenPlayer but stops at 7 rather than 15. My FifteenPlayer I found out wasn't working properly so it was fixed and changed to SevenPlayer. This took 12 minutes with AI to fix. Changing it from 15 to 7 took me only seconds. Here is some of the code below.

SevenPlayer class code:

```
rollButton.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent e) {
        if (!isCurrentPlayer()) return;

        // Initialize Dice object
        Dice dice = new Dice(6);
        int roll = 0;
        turn_total = getTurnTotal();

        // Continue rolling until 7 is rolled
        while (turn_total <= 7) {
            roll = dice.roll();

            if (roll == 6) {
                rollResultLabel.setText("Roll: " + roll);
                turnTotalLabel.setText("Turn Total: 0");
                setTurnTotal(0);
            }
        }
    }
});
```

(This code keeps going)

Create a FakeRandom test class

I created a JUnit test case with sufficient tests to verify that your SevenPlayer class works properly. I don't have a panel with a green bar so to show that it works I will provide Visual Studio's method of showing the tests pass.

First, let's talk about the tests the AI created. I didn't prompt it to create certain tests I just asked it to make tests that would check if SevenPlayerTest is working properly. With this prompt, the AI created three tests. The first test was testStopsAtSix, this test checks if rolling a 6 resets the turn total 0. It also makes sure the total is 0 since getting a 6 shouldn't add anything to the total. This test can be seen below.

SevenPlayerTest class testStopsAtSix test:

```
@Test
```

```

public void testStopsAtSix() throws Exception {
    FakeRandom fakeDice = new FakeRandom(6, 6); // rolls a 6 immediately
    setupPlayer(fakeDice);

    SwingUtilities.invokeLater(() -> rollButton.doClick());

    assertEquals("Roll: 6", rollResultLabel.getText());
    assertEquals("Turn Total: 0", turnTotalLabel.getText());
    assertEquals("Total: 0", scoreLabel.getText());
    assertFalse(rollButton.isEnabled());
}

```

(This keeps going)

The second test is named `testRollsUntilOverSeven`. It mimics rolling a sequence of 3 and 2 repeatedly until the total exceeds 7. For example, it rolls a 3, then a 2, adds them together, and checks whether the sum is greater than 7. If not, it continues adding another 3, and so on. Once the total exceeds 7, the test verifies that the turn total is updated correctly.

```

@Test
public void testRollsUntilOverSeven() throws Exception {
    FakeRandom fakeDice = new FakeRandom(6, 3, 2); // sum = 8
    setupPlayer(fakeDice);

    SwingUtilities.invokeLater(() -> rollButton.doClick());

    assertEquals("Turn Total: 8", turnTotalLabel.getText());
    assertFalse(rollButton.isEnabled());
}

```

The final test the AI created was called `testRollsAndThenSix`. This test had one roll then a roll with a 6. This was to test if the turn total would be 0 despite rolling a 2 earlier.

```

@Test
public void testRollsAndThenSix() throws Exception {
    FakeRandom fakeDice = new FakeRandom(6, 2, 6); // should reset to 0
    at 6
    setupPlayer(fakeDice);

    SwingUtilities.invokeLater(() -> rollButton.doClick());

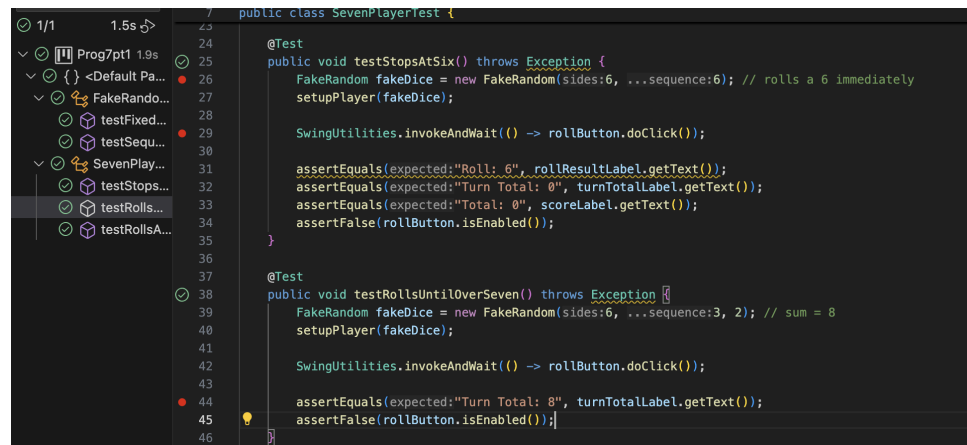
    assertEquals("Roll: 6", rollResultLabel.getText());
    assertEquals("Turn Total: 0", turnTotalLabel.getText());
    assertFalse(rollButton.isEnabled());
}

```

In the end, my code ended up passing the AI tests. Since I don't have a completion bar to show that this is true instead I'll show the check marks. A green check mark means the test passed,

while a red circle with a dot indicates a failure. In the figure below, all the tests are listed on the left side, and the code for each individual test is shown on the right.

Results:



The only change I would make is creating more tests. One such test would be making sure the total is updated correctly. I would do this; however, I moved this feature outside of the Seven-player class due to repetitive code. Since this is moved this is outside of the scope of this task. But this will be experimented on in the future.

Javadoc

Putting the code into Javadoc style took over 5 minutes and 6 seconds with AI. The headers "Written by Sarah Lawrence, with the assistance of ChatGBT 3.5." were done by hand.

The process was to do one class at a time and ask the model to format the code in Javadoc. The model had no problems completing the tasks. Here is some example code showing the successes of the model.

Main class Example:

```
/**
 * A player implementation for the Bulldog game that keeps rolling the dice
 * until their turn total exceeds 7 or they roll a 6 (which ends their turn
 * with 0 points).
 * This class uses Swing components for interaction and display.
 */
public class SevenPlayer extends Player {
    /** Button to initiate a dice roll */
    private JButton rollButton;

    /** Label to display the result of a dice roll */
    private JLabel scoreLabel;

    /** Label to display the score result of a turn */
}
```

```
private JLabel rollResultLabel;  
  
/** Label to display the cumulative total of the current turn */  
private JLabel turnTotalLabel;
```

(This keeps going)

I did this because in another experiment I tried asking the model to repeat the same step of changing the code to Javadoc. But I did it without specifying the step each time. When I did this the model had problems repeating the task. I then did another experiment where I used this method and it worked so I tried it again. I can confirm this method is far more effective than the first.

Conclusion

Altogether, ChatGBT did well in implementing all the steps. The AI didn't have trouble at all with these tasks. This could be due to the fact that it was done in parts and not all together or it just does very well with tasks like these. The only one that took the longest was Create a FakeRandom test class. This was due to user error as I struggled to set up and figure out how to use Junit in Visual Studios. Overall I would say the AI was successful at completing these tasks.