Ellis Fitzgerald
COS 420: Object Oriented Design
April 29, 2025
AI Lab Book #6

LLM: DeepSeek DeepThink R1
Time Spent Overall: 1 hour
Time Spent Prompting: 20 minutes

Creating a GameStatus Class

For creating the GameStatus class I did not have an elegant approach or plan. Technically, I got to creating the GameStatus class by prompting the LLM to refactor the current code of Player classes into dividing the "play" method into "play" and "continueTurn." Despite this, I ended up getting exactly what I needed for the GameStatus object. This was not so surprising considering the object did not need to keep track of much other than the Players and the winning score. Here is the prompt I used:

This function will take the current rollsCount, the current turnScore, and a GameStatus object which should encapsulate every player and their logic on how they make their decision to roll again.

Arguably, my wording here could be seen as confusing. Am I claiming that the GameStatus object should contain all players logic on how they make their decision? Thankfully, DeepSeek seemed to understand what I was going for regardless.

Refactoring "play" method in Player

Now that the GameStatus class has been defined, I needed to refactor the play method. What this assignment aims to solve is something I noticed incredibly early on in the semester which can be seen in the first report we did:

> minimum one turn. I prevented myself from refactoring the condition in the loop to continue a `continueRolling` boolean or function, which all `Player` classes could override because we were told to keep it as simple as possible. }

Because I already had this in mind, I preferred to just do it by hand with no DeepSeek assistance.

```java
/**
 * Play a turn for this player
 * @return Int representing turn score
 */
public int play(GameStatus gameStatus) {  8 usages  new *
    int turnScore = 0;
    int rollsCount = 0;
    do {
        int roll = die.roll();
        listener.onRoll( player: this, roll);
        if(roll == 6) {
            return 0;
        }
        turnScore += roll;
        rollsCount++;
    } while(continueTurn(gameStatus, turnScore, rollsCount));
    return turnScore;
}
```

I decided to use a do-while loop with the condition being a call to the continueTurn method. I noticed that in other prompts with DeepSeek, even when not asked, it would write while(true) loops with if-break statements. This was a structure I felt to be unnecessary because the Bulldog game is perfect for a do-while considering a single roll at the start of the turn is mandatory!

Complete the “continueTurn” methods

I then defined the continueTurn in the abstract Player class by being a simple function which takes the gameStatus object, the players current “turn score,” and their number of rolls. The next thing was to simply implement the method in all the appropriate child classes. I manually typed these methods with the help of autocomplete in my IDE (IntelliJ) since they were mostly one liners that I would rather not risk any hallucinations. It is a good thing I decided to do this, as DeepSeek seemed adamant to forgetting progress we had made in previous assignments, like using Math.random instead of our Random singleton. This took me promptly 3-5 minutes or so.

```java
public boolean continueTurn(GameStatus gameStatus, int turnScore, int rollsCount) {
    return rollsCount < 3;
}
```

AIPlayer (AI Version of UniquePlayer): Rolls at maximum 3 times per turn.

```
public boolean continueTurn(GameStatus gameStatus, int turnScore, int rollsCount) {
    return turnScore < 15;
}
```

FifteenPlayer: Continues rolling until the turn score is 15 or more.

```
public boolean continueTurn(GameStatus gameStatus, int turnScore, int rollsCount) {
    return turnScore < 7;
}
```

SevenPlayer: Much like FifteenPlayer but 7 (go figure).

```
public boolean continueTurn(GameStatus gameStatus, int turnScore, int rollsCount) {
    return RandomSingleton.getInstance().nextFloat() >= 0.5;
}
```

RandomPlayer: 50% chance of continuing or bailing.

```
public boolean continueTurn(GameStatus gameStatus, int turnScore, int rollsCount) {
    return getScore() == 0 && turnScore < 5 || turnScore <= getScore();
}
```

UniquePlayer: Continues rolling until they double their score (or get 5 or more if they started with 0).

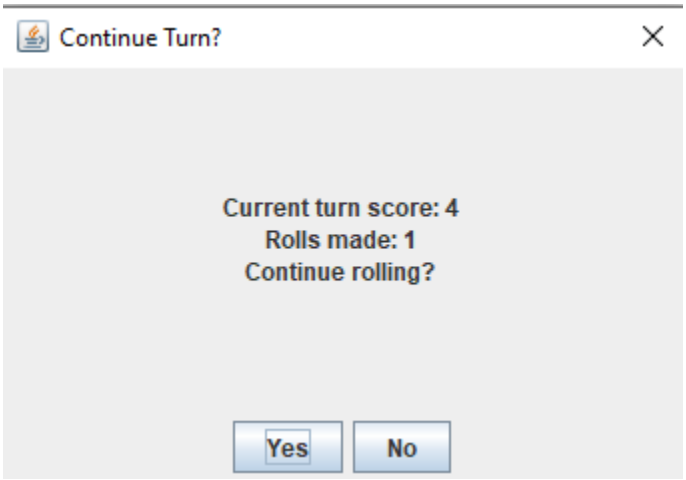
```
public boolean continueTurn(GameStatus gameStatus, int turnScore, int rollsCount) {
    return false;
}
```

WimpPlayer: Never continues to roll.

When it came to implementing the method for HumanPlayer is when I knew I needed help. I already had issues with the HumanPlayer attempting to access the view of our game with the “yes” and “no” buttons, so I asked DeepSeek to write a Java Swing program that created a Dialog box that popped up on screen which had the pressed button as the returned result. This allowed the HumanPlayer to act independently from the Referee and View classes. In order to get this result I attempted something different than previous assignments: I uploaded my Java files that were relevant to the instructions. Previously, I was copying and pasting chunks of code. This time around I thought perhaps the file “objects” may get formatted a certain way by

DeepSeek that makes it easier for the LLM to digest and understand. The files I assumed that would be necessary to refactor this decision dialog can be seen in the following image:

Given the context of previous prompts regarding the continueTurn function, I quite literally asked the LLM to generate the continueTurn function with Java Swing components that contained a yes and no button and returned the boolean of those two values. Despite me not requesting (though I technically should have) DeepSeek improvised and added text that gave the player their current score and how many rolls they have made.



The code for the entire HumanPlayer method can be seen in the following image on the next page:

```java
public boolean continueTurn(GameStatus gameStatus, int turnScore, int rollsCount) {  1 usage  new *
    // Create a custom dialog
    JDialog dialog = new JDialog();
    dialog.setTitle("Continue Turn?");
    dialog.setModal(true);
    dialog.setLayout(new BorderLayout());

    // Create message
    JLabel message = new JLabel(
            text: "<html><center>" +
                    "Current turn score: " + turnScore + "<br>" +
                    "Rolls made: " + rollsCount + "<br>" +
                    "Continue rolling?" +
                    "</center></html>",
            SwingConstants.CENTER
    );
    message.setBorder(BorderFactory.createEmptyBorder( top: 10,  left: 10,  bottom: 10,  right: 10));

    // Create buttons
    JPanel buttonPanel = new JPanel();
    JButton yesButton = new JButton( text: "Yes");
    JButton noButton = new JButton( text: "No");

    // Add action listeners
    yesButton.addActionListener( ActionEvent e -> {
        decision = true;
        decisionMade = true;
        dialog.dispose();
    });

    noButton.addActionListener( ActionEvent e -> {
        decision = false;
        decisionMade = true;
        dialog.dispose();
    });

    // Add components to dialog
    buttonPanel.add(yesButton);
    buttonPanel.add(noButton);
    dialog.add(message, BorderLayout.CENTER);
    dialog.add(buttonPanel, BorderLayout.SOUTH);

    // Configure and show dialog
    dialog.pack();
    dialog.setLocationRelativeTo(null);
    dialog.setDefaultCloseOperation(JDialog.DO_NOTHING_ON_CLOSE);
    dialog.setVisible(true);

    // Wait for user decision
    while (!decisionMade) {
        try {
            Thread.sleep( millis: 100);
        } catch (InterruptedException ignored) {
        }
    }
    return decision;
}
```

HumanPlayer: Rolls again if the “Yes” button is selected.