BULLDOG STRATEGY

Sarah Lawrence, University of Southern Maine

04/16/2025

Description

The goal of this assignment is to implement the Strategy Design Pattern architecture. The Strategy Design Pattern is defined as "a family of algorithms, encapsulate each one as an object, and make them interchangeable." [Gamma, p315] To implement the pattern Into the Bulldog I used the tool given as inspiration and broke the goal down into four steps.

Tools given:
I was provided example code the code is as follows:

```
int score = 0;
  do {
     int diceValue = d.roll();
     if (diceValue == 6) {
     return 0;
     }
     score += diceValue;
  while (continue(gameStatus));
  return score;
```

Step 1: Create Class to Encapsulate Game Status

The goal was to create a small class to encapsulate the game's status. I interpreted this as a boolean used used to check if a player has won or not.

AI Results:

Building this took over 11 minutes and 12 seconds with AI. This includes bug testing. Overall, it wasn't successful. The AI was not understanding the structure of the bulldog code. It was trying to be too much like the example code. Here is what happened with the prompts.
Prompt:

> **Human:** Create a (small?) class to encapsulate the status of the game. An object of this class was shown in the pseudo-code of the play() method and had the name gameStatus. This should be a relatively simple class. (example code) and (isGameOver class)

> **ChatGPT:** Nice — you're really close here! Your GameStatus class is quite reasonable for tracking whether the game is over and determining the winner, but from the way the pseudo-code is ...
>
> AI GameStatus Class:
>
> ```
> public class GameStatus {
>    private int score;
>    private boolean gameOver;
>
>    public GameStatus() {
>       this.score = 0;
>       this.gameOver = false;
>    }
>
>    public int getScore() {
>       return score;
>    }
>
>    public void addToScore(int value) {
>       this.score += value;
>    }
>
>    public boolean isGameOver() {
>       return gameOver;
>    }
>
>    public void setGameOver(boolean gameOver) {
>       this.gameOver = gameOver;
>    }
> }
> ```

When I saw this, I removed getScore, addToScore, and setGameOver since they are in the Player class. The only method I liked was the boolean isGameOver(). I asked the AI if it could implement the edited code it ended up giving me different versions of the following:

AI GameStatus Class:

```
public class GameStatus {
   private int score = 0;

   public int getScore() {
      return score;
   }
```

```
    public void addToScore(int value) {
        this.score += value;
    }

    public boolean isGameOver() {
        return false; // or some rule here
    }
}
```

I didn't want to watch the AI loop anymore so I implemented my method. The method used isGameOver. I had something similar located in the referee class that I combined. There was also another method in the referee class called getWinner() but it was discarded since it was only used once.

isGameOver Method:

```
    public boolean isGameOver() {
        for (Player p : players.getPlayers()) {
            if (p.getScore() >= winning_score) {
                return true;
            }
        }
        return false;
    }
```

This was more of the structure I was going for and merged better with my preexisting code. This structure has a Boolean that shows whether the game is still being played like the sudo code. This overall worked but I wanted to give the AI another chance so I gave it my code to improve and it produced this:

```
 public class GameStatus {
    private final PlayerList players;
    private final int winningScore;

    public GameStatus(PlayerList players, int winningScore) {
        this.players = players;
        this.winningScore = winningScore;
    }

    public GameStatus(PlayerList players) {
        this(players, 100);
    }

    public boolean isGameOver() {
        return players.getPlayers().stream()
                   .anyMatch(p -> p.getScore() >= winningScore);
    }
```

```
}
```

The first two seem to be constructors but the last one was different. I had never encountered a method like the isGameOver() method the AI created. I had to do some research into what it was doing and turns out it's taking the list of players, making it into a stream, and if a player has a score higher equal to the winning score it returns true else false. I didn't know what a stream was so I learned that they are like loops, but used for getting out of loops early and for parallel processing. I thought it was interesting so I kept the code the AI generated. But this doesn't implement the pattern yet.

Step 2: Refactor the play method in the Player class

The goal was to move common duplicated code from the Player classes into the abstract class Player. This refactoring also involved creating a new method that provided each Player's decision about whether or not to continue rolling in this round.

I asked the AI to create the code while providing my original player class and each RandomPlayer's rollbutton. The RandomPlayer's rollbutton was included because it was the code that repeats in all the different player classes just slightly different. The AI with this information produced this:

```
// Template Method: shared play logic
   public int play() {
      setTurnTotal(0);
      while (continueRolling()) {
         int roll = rollDie();
         if (roll == 1) {
            System.out.println(getName() + " rolled a 1! Turn over, no
               points earned.");
            setTurnTotal(0);
            return 0; // Ends turn, no points
         } else {
            addTurnTotal(roll);
            System.out.println(getName() + " rolled a " + roll + ", turn
               total is " + getTurnTotal());
         }
      }
      endTurn();
      System.out.println(getName() + " ends turn with total score: " +
         getScore());
      return turnTotal;
   }

   // Abstract method: subclass defines strategy
   protected abstract boolean continueRolling();
   // Simulates rolling a die (1-6)
   protected int rollDie() {
      return (int) (Math.random() * 6) + 1;
   }
}
```

The following AI code has a repeat rollDie method, print line statements, and the while loop is in the wrong spot. Because of the structure of my code rollDie is instead replaced with the already existing dice.roll(). I also wasn't sure where to place the UI information. This is because I wanted UI to update each turn. So I originally kept the loop and asked it to implement the UI with it. The following AI code has a UI:

```
// Decision hook: subclasses control this logic.
protected abstract boolean shouldContinueRolling();

public int playTurn(Dice dice, RollUI ui) {
   setTurnTotal(0); // reset at the start

   while (shouldContinueRolling()) {
      int roll = dice.roll();
      ui.updateRollResult(roll);

      if (roll == 6) {
         setTurnTotal(0);
         ui.updateTurnTotal(getTurnTotal());
         ui.updateScore(getScore());
         ui.disableRollButton();
         return roll;
      } else {
         addTurnTotal(roll);
         ui.updateTurnTotal(getTurnTotal());
      }
   }

   endTurn(); // transfer points to score
   ui.updateScore(getScore());
   ui.disableRollButton();
   return getTurnTotal(); // return the result
}
```

The UI doesn't do anything, disableRollButton and while loop I ended up moving some where else, and endTurn should not exist in this. I will say the AI did do good at implementing the method boolean continueRolling(). This helped me fix the play() method. Here is the final result of the play() and shouldContinueRolling() methods.

```
public boolean play(int roll) {
      if (roll == 6) {
         // Roll a 6 ends the turn with 0 points
         setTurnTotal(0);
         return false;
      } else {
         // Add the roll to the turn total
         addTurnTotal(roll);
         if (!shouldContinueRolling()) {
            return true;
         }
      }
      // End the turn and update the score
      return false;
   }

   // Decision hook: subclasses control this logic.
   protected abstract boolean shouldContinueRolling();
```

The method is checking for a few things. First, it sees if the roll is a 6. If yes it will set the turn score to 0 and return false. The reason for the false is because there is a loop in each class, the loop needs to know if the player continues. The player doesn't continue if scores a 0 so it returns 0 to the loop. If the roll is not 6 it adds to the turn total and checks if it should continue, if yes it returns true. shouldContinueRolling() is the decision that each class makes. In the end, this helped eliminate duplicate code, and with shouldContinueRolling() it's prepared to implement the strategy.

Step 3: Create the Continue Methods in each Class

The goal was to implement distinct continue methods in each class. With the new play in Player I had to change two things in each class the rollButton and the continue method (shouldContinueRolling).

A Hint was also given that "Most programmers would suggest that the HumanPlayer class would be a poor choice for this initial test." So I started with the RandomPlayer class. This first one to change took over 1 hour and 17 minutes. Like the "Class to Encapsulate Game Status," the AI had a hard time with the structure of the code.
The first one I did was the Randomplayer's rollButton. I was able to simplify it with the play class. It turned out looking like this:

```
public void addActionListeners() {
      rollButton.addActionListener(new ActionListener() {
         @Override
         public void actionPerformed(ActionEvent e) {
            if (!isCurrentPlayer()) return;
            int roll = 0;
            boolean result = true;

            while (result){
               roll = dice.roll();
               result = play(roll);
               updateUI();
            }
            rollButton.setEnabled(false);
         }
      });
   private void updateUI() {
      rollResultLabel.setText("Roll: --");
      turnTotalLabel.setText("Turn Total: " + getTurnTotal());
      scoreLabel.setText("Total: " + getScore());
   }

   protected boolean shouldContinueRolling() {
      Random random = new Random();
      return random.nextBoolean();
   }
```

The while loop rolls the dice and determines whether the player can continue or not, it then updates the GUI. I felt the GUI shouldn't be put in the Player class so I instead made it into a method in case I wanted to change it later (It is repetitive so I might make it in a separate class in the future). The AI kept putting the while loop inside the player class but since I wanted the GUI to constantly update for the user I moved it back to the individual classes. I had similar difficulties with asking AI to do the other classes but since I had the bulk of it done I did the rest by hand.

The most difficult for me was to wrap my head around the Human player class. I want to use the AI to help so I asked it to take the human class I have, look at a class that was complete and implements play, and implement the play method into the human class.

On the first prompt, it changed play, which I didn't want since that might affect the other classes. It then had a complete rollButton that didn't rely on the changes of the play method, so I implemented it. The final struggle was what to put for the continuation. The AI produced this.

```
protected boolean shouldContinueRolling() {
   return false;
}
```

I tested it and it wasn't working so I changed it to true instead and it worked. This allowed the human players to make decisions with the GUI. In the end, it took me 3 hours and 32 minutes to complete. Overall I think the combination of the AI and my knowledge of the code helped complete this.

I have included all the different continue code spinets below.

Code Spinets

Random Player:

```
protected boolean shouldContinueRolling() {
      Random random = new Random();
      return random.nextBoolean();
   }
```

Seven Player (Fifteen Player is the same as 7):

```
 protected boolean shouldContinueRolling() {
      turn_total = getTurnTotal();
      return turn_total <= 7;
   }
```

Unique Player:

```
    protected boolean shouldContinueRolling() {
      return roll == 3 && roll == 5;
   }
```

Human Player:

```
  protected boolean shouldContinueRolling() {
      return true;
   }
```

This was the final peace for implementing the Strategy Design Pattern architecture. This shouldContinueRolling() allows the play() code to use different strategies based on the different players. But the overall code wouldn't be complete without formatting.

Step 4: Javadoc

Putting the code into Javadoc style took over 3 minutes and 9 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.

RandomPlayer class Example:

```
  /**
    * Attaches an {@link ActionListener} to the roll button.
  * When triggered, the player will automatically roll the dice in a loop
  * until a stopping condition is met (rolling a 6 or exceeding a safe
     turn total).
  */
 public void addActionListeners() {
    rollButton.addActionListener(new ActionListener() {
       @Override
       public void actionPerformed(ActionEvent e) {
          if (!isCurrentPlayer()) return;

          int roll = 0;
          boolean result = true;

          while (result) {
             roll = dice.roll();
             result = play(roll);
             updateUI();
          }
          rollButton.setEnabled(false);
       }
    });
 }
```

(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

The Strategy Pattern allows different algorithms to be swapped easily. Starting with refactoring, I first created a GameStatus class to manage whether the game was over and eventually learned about Java Streams. Next, I refactored the Player class to eliminate duplicated logic by moving common behavior into an abstract play() method and introducing a shouldContinueRolling() to make strategy decisions. Finally, I implemented the new play() and changed the functions with shouldContinueRolling(). The AI did have trouble with these tasks but did help with improving my base code and helped me learn some new techniques. Overall the AI implemented the Strategy Design Pattern in a Java-based "Bulldog" dice game.