Extra

Sarah Lawrence, University of Southern Maine 05/2/2025

Description

The goal of this assignment is to provide a creative outlet for expanding the Bulldog program. We must propose two significant new changes to the Bulldog program and implement them. The work that is proposed must be fairly specific and must be related to both the Bulldog game and the ideas of object-oriented design.

1 Proposed Creative Improvements

I propose the following two improvements:

1. Implementing the Command Design Pattern (CDP): CDP compacts a request as an object. Making it possible to pass different requests to a queue and can support undo."

2. Implementing the Factory Design Pattern: This pattern is used to create objects without specifying the exact class of object that will be created.

For implementing the two ideas I will be using my knowledge and the assistance of the free ChatGPT 3.5 model.

2 Implementation

2.1 Command Design Pattern

It took me and the AI over 2 hours and 12 minutes. This was because I was having trouble grasping how I wanted this to benefit the user. I knew for this design I wanted to implement a player history log and an undo. I wanted the log to show everyone's last 5 round scores. This is so players can strategize a plan agents their appointments with the knowledge they have.

So I started by changing the Player class. I added a scoreHistory ArrayList. Anytime a player ended their turn the total would be added to the scoreHistory list. Then I had the AI create a new class called CommandScoreHistory and a new interface called Command. The command has only been executed, and undo for now. CommandScoreHistory implements Command. CommandScoreHistory uses the execute to create a display for the log showing everyone's last 5 round scores. The AI produced this code.

```
 import javax.swing.*;
import java.util.List;

public class CommandScoreHistory implements Command {
   private List<Player> allPlayers; // List of all players

   public CommandScoreHistory(List<Player> allPlayers) {
      this.allPlayers = allPlayers;
   }

   @Override
   public void execute() {
      // Create a new window to display everyone's score history
      JFrame historyFrame = new JFrame("All Players Score History");
      historyFrame.setSize(400, 300);
      historyFrame.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);

      // Create a new JTextArea to display all the player score histories
      JTextArea historyArea = new JTextArea();
      historyArea.setEditable(false); // Make the area read-only
      historyArea.setText("Player Score Histories:\n");

      // Loop through all players and display their score history (latest
          5 scores)
      for (Player player : allPlayers) {
         List<Integer> playerScoreHistory = player.getScoreHistory();
         historyArea.append("\n" + player.getName() + ":\n");
```

I then asked it to create the action listener for the viewHistoryButton button. So players could interact.

```
  historyTextArea = new JTextArea();
      historyTextArea.setEditable(false); // Make the area read-only

      // Add the action listener to the View History Button
      viewHistoryButton.addActionListener(e -> {
         CommandScoreHistory viewAllPlayersHistory = new
             CommandScoreHistory(
            playerList.getPlayers() // Pass the list of players
         );
         viewAllPlayersHistory.execute(); // Execute the command to show
             player history
      });
```

In the end, I kept asking the AI to tweak some small things like showing only the most recent five scores of each player or how certain things were working. Like how I didn't know this lone of code made this window read-only.

```
historyTextArea.setEditable(false);
```

What took me the longest was the Undo. That took roughly an hour. I didn't know what I wanted for an undo so I just messed around with this design pattern so I didn't make it a huge befit to the player. I evenly settled on the history popping up in a new box so if the player wanted to close it from the game screen they could. I originally wanted it to undo the last player's turn for debugging purposes but I had a hard time grasping why the AI was making certain changes. I eventually didn't want to break my code further and decided to abandon that effort. When I improve my skills and have more time I hope to come back to the idea. This is the short code that I settled for that.

```
 @Override
   public void undo() {
      if (historyFrame != null) {
         historyFrame.dispose(); // Close the window
      }
   }
```

2.2 Factory Design Pattern

In total it to 11 minutes and 43 seconds. Me and the AI put the factory method into three new parts.
The PlayerFactory class is a factory that creates different types of Player objects like Human-Player, RandomPlayer, SevenPlayer, and UniquePlayer. It does this based on a given type. This was done by the AI.

```
 import javax.swing.*;

public class PlayerFactory {
   public static Player createPlayer(
      String type,
      String name,
      JButton rollButton,
      JLabel rollLabel,
      JLabel totalLabel,
      JLabel turnTotalLabel,
      GameGUI gui,
      Dice dice
   ) {
      switch (type) {
         case "Human":
            return new HumanPlayer(name, rollButton, rollLabel,
               totalLabel, turnTotalLabel, gui);
         case "Random":
            return new RandomPlayer(name, rollButton, rollLabel,
               totalLabel, turnTotalLabel, gui, dice);
```

(This code continues)

In the GameGUI class, a new method is created called createPlayerButtonListener. The createPlayerButtonListener method does not need to know the specific class of the player being created it just calls the factory method.
I created this since the AI was having trouble understanding what variables I needed.

```
 private ActionListener createPlayerButtonListener(String type) {
      return e -> {
         Player player = PlayerFactory.createPlayer(
            type,
            playerNames.get(currentPlayerIndex),
            rollDiceButton,
            rollResultLabel,
            totalLabel,
            turnTotalLabel,
            this,
            new Dice(6) // HumanPlayer can ignore this
         );
         playerList.addPlayer(player);
         nextPlayerType();
      };
   }
```

Then this final code also located in the GameGUI class uses what was created prior to creating the different player types. This was also generated by the AI.

```
      humanButton.addActionListener(createPlayerButtonListener("Human"));
      sevenButton.addActionListener(createPlayerButtonListener("Seven"));
      randomButton.addActionListener(createPlayerButtonListener("Random"));
      uniqueButton.addActionListener(createPlayerButtonListener("Unique"));
```

(This code continues)

This prior code looked like the following:

```
humanButton.addActionListener(e -> {
         playerList.addPlayer(new HumanPlayer(
            playerNames.get(currentPlayerIndex),
            rollDiceButton,
            rollResultLabel,
            totalLabel,
            turnTotalLabel,
            this
```

```
        ));
        nextPlayerType();
    });

    sevenButton.addActionListener(e -> {
        playerList.addPlayer(new SevenPlayer(
            playerNames.get(currentPlayerIndex),
            rollDiceButton,
            rollResultLabel,
            totalLabel,
            turnTotalLabel,
            this,
            new Dice(6)
        ));
        nextPlayerType();
    });
```

(This code continues) It was repetitive and had a lot of duplicated code. This new code has less duplication, but I do think having to change two spots to add a new player is a problem. While I was trying to fix this problem I ended up fixing a different duplicated issue using this AI-generated code.

```
for (String type : List.of("Human", "Seven", "Random", "Unique","Wimp")) {
            JButton button = new JButton(type);
            button.addActionListener(createPlayerButtonListener(type));
            buttonPanel.add(button);
      }
```

This helped me git rid of all these repetitive pieces of code.

```
      wimpButton = new JButton("Wimp");
      randomButton = new JButton("Random");
      humanButton = new JButton("Human");
      sevenButton = new JButton("Seven");
      uniqueButton = new JButton("Unique");
      buttonPanel.add(humanButton);
      buttonPanel.add(wimpButton);
      buttonPanel.add(randomButton);
      buttonPanel.add(sevenButton);
      buttonPanel.add(uniqueButton);
```

Trying to fix the issue of going between two spots to make a new player took 34 minutes and 45 seconds until I gave up trying to find a solution with the AI. This could be good as it separates the two, but I personally think it's not ideal. If someone comes along and wants to add a new player, they would have to make the button and then connect the button to the player in the PlayerFactory class.

3 Javadoc

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

PlayerFactory class Example:

```
 /**
* A factory class for creating different types of Player instances.
*/
public class PlayerFactory {

   /**
    * Creates and returns a specific type of Player based on the provided
       type string.
    *
    * @param type      the type of player to create ("Human", "Random",
       "Seven", or "Unique")
    * @param name      the name of the player
    * @param rollButton the button associated with rolling the dice
    * @param rollLabel the label showing the result of a roll
    * @param totalLabel the label displaying the player's total score
    * @param turnTotalLabel the label showing the current turn's total score
    * @param gui       the GameGUI instance to update game state and visuals
    * @param dice      the Dice object used by AI players to simulate rolls
    * @return          a new Player instance of the specified type
    * @throws IllegalArgumentException if the type is not recognized
    */
   public static Player createPlayer(
      String type,
      String name,
      JButton rollButton,
      JLabel rollLabel,
      JLabel totalLabel,
      JLabel turnTotalLabel,
      GameGUI gui,
      Dice dice
```

(This code continues)
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

Overall, both implementing design patterns had their strengths and weaknesses. The Command Design Pattern, due to my struggle to understand it, I had to implement a poor undo function. The Factory Design Pattern, unfortunately, resulted in having to define both the button and the function in two separate places for it to follow the factory method pattern. But they had functions that benefited the coder or the player. CDP allowed the user to view the scores of other players or how FDP helped clean up the GameGUI class. In the end, I think it was fun trying to see If I could implement these designs.