Bulldog Improvements

Sarah Lawrence, University of Southern Maine 03/03/2025

Description

The goal of this is to add improvements to four aspects of the Bulldog program using AI. The four improvements are as follows.
1. Getting rid of "104": For this improvement, the code base should define a constant set to 104. All other classes should reference the constant and not a hardcoded value.
2. Consolidate randomness: For this improvement, isolate the random number generators from the rest of the code base whenever possible.
3. Javadoc: For this improvement, get the AI tool to document all of the code in the Javadoc style.
4. File Structure: For this improvement, prompt the AI to have multiple classes in multiple files.
This paper will cover the success of AI in achieving these improvements and the problems that occurred along the way. If some of the improvements have been already made it will be explained why the AI tool might already include that. The model that will be used for this is ChatGPT 3.5.

Get rid of "104"

Success:

ChatGPT was very quick to make this improvement. In total, it only took 3 minutes. The majority of the time was bug testing. I believe it was quick due to the prompts and cause I hand-placed replaced the variables. The prompt use is as follows.
Prompt:

> **Human:** The score necessary to win the game was chosen as 104. This value is embedded "deep" in the code, often in multiple locations. This is a poor style because it makes it difficult to change the winning score of the game. For this improvement, the code base should contain a constant named something like WINNING_SCORE which is (initially) set to 104. All other classes should reference the constant and not a specific number. For example, here is some code with the issue (Code: WimpPLayer class).
> **ChatGPT:** Add the constant in the Player class.

```
public abstract class Player {
   // Constant for the winning score
   public static final int WINNING_SCORE = 104;

   private String name;
   private int score;

   public Player(String name) {
      this.name = name;
      this.score = 0;
   }

   public String getName() {
      return name;
   }

   public int getScore() {
      return score;
   }

   public void setScore(int score) {
      this.score = score;
   }

   // Abstract method to be implemented by subclasses
   public abstract int play();
}
```

. Update the WimpPlayer class to use the constant

```
public class WimpPlayer extends Player {

   public WimpPlayer() {
      this("Wimp");
   }

   public WimpPlayer(String name) {
      super(name);
   }

   @Override
   public int play() {
      Dice dice = new Dice(6);
      int roll = dice.roll();
      int totalScore = getScore() + roll;
```

```
            System.out.print(" Player " + getName() + " rolled " +
                roll);

            // Use the constant WINNING_SCORE instead of the hard-coded
                104
            if (totalScore >= WINNING_SCORE) {
                return totalScore;
            } else if (roll == 6) {
                System.out.println(" and scored 0 for the turn.");
                return getScore();
            } else {
                System.out.println(" and chose not to continue, scoring
                    " + totalScore + " for the turn.");
                return totalScore;
            }
        }
    }
```

.

This logic was handapplied to the other classes to save time. The code worked for every test with no bugs.

Experiment:

Since there were no problems, I wanted to test if asking ChatGPT to make the changes for me would be more effective.

Hypothesis: I theorize it will take longer to improve.

Test: This theory is tested with lots of the same prompts to see if ChatGPT can do the same task faster.

Prompt: have WINNING_SCORE in this (Code: FifteenPlayer class)

Result: In the end, it just kept telling me, in different wording, to replace the constant with the variable.

```
if (total_score >= WINNING_SCORE) {
    return total_score;
}
```

The process of promoting, doublechecking, coping, and then pasting took more time than just changing one variable by hand. The process took 5 minutes including bug testing. It worked without any complications but did take longer than the prior method.

Consolidate Randomness

Success:

ChatGPT was partly successful at consolidating randomness. It took only 25 minutes. The new Dice class only took 2 minutes due to the prompt being descriptive. The prompts is below.
Prompt:

> **Human:** Random number generators are tricky things and it is best to isolate random number generators from the rest of the codebase whenever possible. Create a new class Dice that contains only one random number generator; the keyword 'static' is your friend, here. The class should have a constructor that takes N, the number of sides of the Dice object. The class should contain a public method roll() which produces a random number from 1 to N. Thus, the new Dice(6) would create a standard six-sided die while the new Dice(2) would effectively create a coin.
> **ChatGPT:** Provided code and example on how to use it. The code is as follows.

```
import java.util.Random;

public class Dice {
   // Static Random object to ensure only one random number
      generator
   private static final Random random = new Random();

   private int sides; // Number of sides of the dice

   // Constructor to initialize the dice with N sides
   public Dice(int sides) {
      this.sides = sides;
   }

   // Method to roll the dice and get a random number from 1 to N
   public int roll() {
      return random.nextInt(sides) + 1; // Generates a number
         between 1 and N (inclusive)
   }
}
```

Problems:

The problem was that the majority of the time was spent on hand replacing random generators, bug testing, and hand-fixing the different classes.
I handreplaced the random generators because it was such a small line of code, and I didn't want ChatGPT to make unnecessary changes. The code used to replace the random generators was

produced by ChatGPT.
The line of code that replaced the random generator was:

```
Dice dice = new Dice(6);
int roll = dice.roll();
```

With hand placing came bug testing. I tested after each class to make sure it was working properly with the new AI code. The two classes that had the most problems were RandomPlayer, and UniquePlayer class. RandomPlayer was tricky because I forgot that there were two random generators in my old code, so I found the other and replaced it. Finally, Unique was harder due to all the replacements. There are three random generators in this class. One that flips a coin. The second is if you flip heads, you roll high (4-6) and do not continue. The third is if you flip tails, you roll low (1-3) and continue. Getting the random generator between high (4-6) took some problem-solving. But a solution was found, and it was provided below.

```
Dice highRoll = new Dice(3);
int roll = highRoll.roll() + 3; // Rolls a value between 1 and 6
   if (roll >= 4) {
```

Originally, I tried to have ChatGPT figure it out but it wasn't going in the direction I wanted. ChatGPT kept making code that would if it was heads have a number generator pic a number between 1 and 6 and if it was lower than 4 then it would flip the coin again. I was hoping that if it was heads no matter what it would pick a random number between 4 and 6 and not re-flip the coin.
Example:

> **Human:** How to change the random number generators to the format "Dice dice = new Dice(6); int roll = dice.roll();" on (code: all of UniquePlayer)
> **ChatGPT:** (provides code: with "if (roll >= 4) ")
> **Human:** what happens if it doesn't roll a 4 or higher
> **ChatGPT:** If the roll does not result in a 4 or higher, based on the current logic, the code inside the if (roll >= 4) block will not execute.
> ⋮
> **Narrator:** Then it was a continuing back and forth until I handfixed the issue

Javadoc

Success:

ChatGPT was successful in replicating the Javadoc style. It only took in total of 7 minutes to get all the code in the required style. Here is an example of the successful Javadoc code created by ChatGPT.

```
/**
```

```
 * The WimpPlayer class represents a player in the Bulldog game
 * who always ends their turn after the first roll.
 *
 * <p>This class extends the abstract Player class and overrides the
 * {@code play()} method to implement the specific behavior of a "Wimp"
    player.</p>
 *
 * @author Sarah Lawrence
 * @version Spring 2025
 */
public class WimpPlayer extends Player {

   /**
    * Default constructor that creates a WimpPlayer with the name "Wimp".
    */
   public WimpPlayer() {
      this("Wimp");
   }
```

Problems:

There was only one small setback. This is because sometimes if a task was repeated often ChatGPT failed to do it. Instead, it would do something else like the task. Here is an example where ChatGPT deviated from what was asked.

Human: Don't change the code but put this in Javadoc style (giant code block)
ChatGPT: I've formatted your comment in Javadoc style at the top of your code while keeping everything else unchanged...
Human: I want all of it formatted in the Javadoc commenting style but don't change any code that's not a comment
ChatGPT: I've reformatted all comments in Javadoc style without altering any code.
Human: same with this one (Code: Human player)
ChatGPT: (Produces code in Javadoc style)
Human: same with this one (Code: Fifteen player)
ChatGPT: (Produces code in Javadoc style)
Human: same with this one (Code: Random player)
ChatGPT: (Produces code in Javadoc style)
⋮
Human: same with this one (Code: Wimp player)
ChatGPT: Here's the cleaned-up version of your WimpPlayer class with improved formatting and clarity:
Human: Is it in Javadoc style?
ChatGPT: No, the current comments are not in Javadoc style. (Provides an updated

version in Javadoc style)

The first red ChatGPT prompt is where the model made the error. The error was that it didn't replicate the Javadoc style like the others. The second red ChatGPT prompt is where it acknowledged it wasn't in Javadoc and fixed the mistake. One reason this might occur is the lack of clarity. Saying the specified task every time might fix this problem. Example: instead of prompting "same with this one" use "put this code in the Javadoc commenting style (provide code)". The final reason could be due to ChatGPT's random yet cohesiveness of its responses. What I mean is ChatGPT is often re-wording things. In improvement one it kept rewording the same thing even though it was the same result. This is helpful for people having a hard time understanding but not when you want to replicate the same results repeatedly. There might be other reasons but those are the only reasons I have concluded for now.

File Structure

Success:

This improvement has already been made. When it happened, it took roughly 4 minutes, most of it being bug testing. There is a hypothesis on why it did so well on this. The AI might have done this due to good prompting. The good prompting is theorized because it only took one prompt to procure this code. The prompt is all the bulldog game instructions. Including instructions on how the classes are separated and what's in each class. There were no problems with the code, and it ran successfully on the first try.

Experment:

Since there were no problems with this improvement, I created another hypothesis and put it to the test.

Hypothesis: ChatGPT being welltrained on code is not well supported theory. This is causing the model to be trained on large amounts of text data from the internet. This could mean anything and everything, not just code. Though ChatGPT sometimes displays good code when, like the Javadoc, sometimes it doesn't like in the Javaswing experiment. This claim doesn't mean it succeeded in this improvement just by being welltrained.

Test: This theory is tested with one prompt like the original but without explanations of the classes and what to put in them. Just the game instructions.

Prompt:

Game Play:
On each turn, a player repeatedly rolls a six-sided die until either a 6 is rolled or the player decides to end their turn.

- If the player rolls a 6, their turn is over; they score nothing, and control passes to the next player.
- If the player rolls any number other than a 6, the number is added to their score for that turn.
  - If the player chooses to continue, they roll the die again, applying the rules above.
  - If the player chooses to end their turn, they add their accumulated turn score to their overall score, and control passes to the next player.

The first player to score at least 104 points wins the game.

The most interesting part of the game, of course, is the strategy that each player uses to decide when to end their turn. A conservative strategy ends early, taking guaranteed small scores. An aggressive strategy favors longer turns, balancing the risk of losing all accumulated points against the reward of large scores.

Result: As expected, the result was just one java file, not many classes. The code isn't a complete class ether. This can be seen below.

```
import random

def roll_die():
    return random.randint(1, 6)

def player_turn(player_name):
    turn_score = 0
    while True:
        roll = roll_die()
        print(f"{player_name} rolled a {roll}")

        if roll == 6:
            print(f"{player_name}'s turn is over. No points scored.")
            return 0

        turn_score += roll
        print(f"{player_name}'s turn score: {turn_score}")

        choice = input("Do you want to roll again? (y/n): ").strip().lower()
        if choice != 'y':
            return turn_score

def play_game():
    scores = {"Player 1": 0, "Player 2": 0}
    winning_score = 104
    players = list(scores.keys())
```

```
    while True:
        for player in players:
            print(f"\n{player}'s turn")
            turn_points = player_turn(player)
            scores[player] += turn_points
            print(f"{player}'s total score: {scores[player]}")

            if scores[player] >= winning_score:
                print(f"{player} wins with {scores[player]} points!")
                return

if __name__ == "__main__":
    play_game()
```

This further proves that good prompting resulted in this improvement succeeding.

Conclusion

Altogether, ChatGBT did well in implementing the improvements. The one it had the most trouble with was consolidated randomness, but it still partly got it in the end. With combined efforts, it did well with the improvement of getting rid of "104". From then on it was the most successful with Javadoc. Overall, repeating prompts can cause more problems than good, and the combined efforts of the human and the AI were efficient.