Ben Gaudreau

Professor Levine

COS 420

March 26, 2025

THE MODEL

Creating a class to represent the Model (an encapsulation of a player List) was incredibly simple. With less than a hundred lines of functional code, getting an AI tool to write the class took only a single prompt to get it mostly correct. Less than five minutes of prompting followed by fifteen to twenty minutes of human edits (mostly to standardize the formatting and extend functionality) was all that was necessary to finish the job.

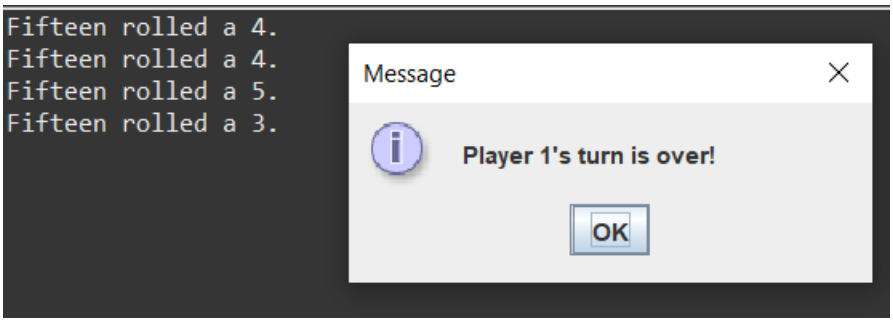
One interesting thing to note involves the AI tool's inability to remain consistent even in such a small snippet of code. Note the difference in the error handling procedures generated by ChatGPT, seen below. My best guess has to do with the return type of the two methods – in this case, a non-String return type prompted the AI tool to throw an exception instead of returning a raw String.

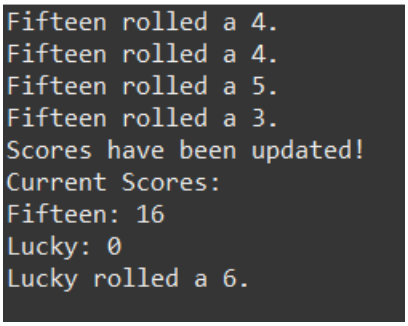
```
public String getPlayerName(int index) {
    if (index >= 0 && index < players.size()) {
        return players.get(index).getName();
    }
    return "Invalid index";
}

public int getPlayerScore(int index) {
    if (index >= 0 && index < players.size()) {
        return players.get(index).getScore();
    }
    throw new IndexOutOfBoundsException("Invalid index");
}
```

THE VIEW

Implementing the Model was simple, but getting the View to work was absolutely trivial. With a whopping 14 lines of code needed for the viewer class, only in the most unfortunate circumstances would my AI tool have failed to accomplish this task. The only change I made to this portion of the class was to refactor the print statements to utilize printf(). It seems that ChatGPT has a preference for using println() statements instead, but I tend to use formatted strings, especially when variable values are mixed in.

Adding a main method to check the View's functionality took more coding than the class itself. Since its purpose was solely for testing, I ended up removing it from the final version of the program at this iteration. I neglected to even check if any of the players reached the winning score. Sure enough, however, updating the Model caused an update to the View, which can be demonstrated in the two screenshots below. All in all, this process took about fifteen minutes to complete.





(A FifteenPlayer takes their turn, popping a dialog box when finished. Clicking "OK" causes their score to be updated in the Model, which in turn notifies the View. The scores are then printed as per the viewer's update() method.)

INTEGRATION

Now this is where things get tricky. I know for a fact that my integration of the prior two components into the Bulldog program are nowhere close to achieving a "true implementation" of the MVC pattern. Unless I am mistaken, the GUI elements that are strictly used for the scoreboard should be included in the View, and not the Controller. However, due to how my particular Bulldog program

has evolved over the course of the last several iterations, making this happen in code was not feasible in its current state. I believe that another code refactorization is in order.

One interesting thing that I noted while attempting to have my AI tool make these integrations, however, was the fact that it was unable to understand that the methods in the Model that changed data values already included calls to notifyFollowers(). In every situation that ChatGPT used one of these methods, it would immediately append a call to notifyFollowers(), resulting in doubled View updates when only one was necessary. I made these changes manually, but I imagine that following up with a prompt to remove all instances of the notify call. Now that I am writing this, I realize that it would be better to have the notifyFollowers() method be private instead of public in order to prevent this misuse; something I missed in my initial analysis of the assignment and failed to catch in the AI tool's implementation of the method.

Since this was a more involved process, it took a decent while longer than simply creating the components that were now being linked together. Adding those onto the existing program resulted in numerous bugs, some of which were fixable with follow-up prompts and others requiring manual edits. I cannot say that this iteration of the program is at a state that I am entirely happy with, but it is a "working end-product" that shall have to suffice until I dig around in the code some more.

CONCLUSION

This iteration is a prime example of why it is so important to ensure a strong foundational design before writing lots of code. The difficulty in adding design patterns on top of an existing design cannot be understated, especially when making a change at the core of the program's operation. The likelihood of rewriting a majority of the current Bulldog code in order to accommodate patterns from this assignment and future ones is high, but also necessary as projects grow and evolve. Such a process would be analogous to a lizard shedding its skin when molting.

As for the efficacy of AI tools in the mimicry of software design patterns, I believe that, like I have experienced thus far, these tools currently do a great job at creating and using a template. With the ubiquity of these patterns in online discussions, there's a lot of information for these models to train with. For mounting them on top of existing code, however, AI tools become less reliable. As these tools grow more advanced, it may very well be the case that they will become better designers than programmers.