Ellis Fitzgerald
COS 420: Object Oriented Design
March 31, 2025
AI Lab Book #4

LLM: DeepSeek DeepThink R1
Time Spent Overall: 1 hour
Time Spent Prompting: 15 minutes

Addressing the issues with my prompting:

1. Divide & Conquer:

   For this assignment I took the Professors feedback given to myself and the rest of the participants. I decided to "divide and conquer" by splitting each task into smaller pieces until it approached a "base case." Something I would define as the moment where myself doing the task would take the same amount of time (or less) than if an LLM were to do it. Including doing research.

2. Tell & Show:

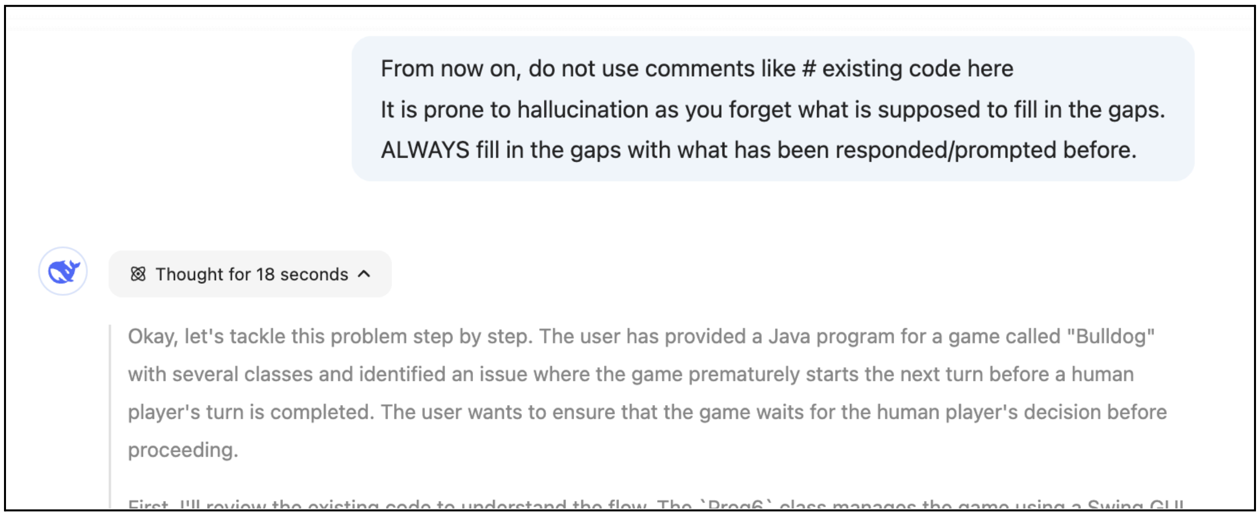
   Not only did I divide the tasks up into separate parts, I asked the LLM for its thought process given a problem so I could correct it before it generated any code. This requires an ancient debugging technique known as reading. I can read the (rather expressive) thought process that DeepSeek spits out and make sure it is at the bare minimum logical. An example can be seen in Figure 1 which covers my next change.

3. No More Hiding Code Behind Comments:

   I prompted the LLM not to put comments in the form of "insert previous code here." The reason for this is quite simple: it made broad assumptions about what code laid before. It hallucinated code that I never provided and it never generated. When I trusted these types of comments I often found myself pasting the code in individual pieces waiting for it to come together. It would never because most of its implementation relied on methods that did not exist. Figure 1 showcases me asking for this to be done which was somewhat misinterpreted as a request to revise the previous generation. Albeit, I had no issues with these comments for the rest of the assignment.

4. Not Trusting What the LLM Remembers:

Finally, the last change I made was to no longer assume what the LLM remembers from previous prompts. Because LLMs are quite optimistic about what they know and how they can help: they are not good at being honest about what they do not know. It rather makes sense based on the theory behind LLMs and the data they are trained on. How can a LLM deal with data that does not exist? It either has data that quite literally addresses that something "does not exist" or it gives what is the closest guess. If I did an LLM training on the Travelling Stack Exchange and asked it about cookies, I don't think there would be data stating "it is impossible to make cookies" but instead there would be something vague or a complete absence in reference to cookies. Why? Simple, the Travelling Stack Exchange is not for baking. This would imply a result from our Travelling Stack Exchange LLM that would give something that may contain a hallucination or in Information Retrieval terms "Irrelevant" (thanks Professor Mansouri). Therefore, for the few prompts I used in this assignment I pasted entire codeblocks of the current program to make sure we were on the same page. I thought it was analogous to passing a piece of paper back and forth, or more closely related a git/source/version control with the LLM.


From now on, do not use comments like # existing code here
It is prone to hallucination as you forget what is supposed to fill in the gaps.
ALWAYS fill in the gaps with what has been responded/prompted before.

Okay, let's tackle this problem step by step. The user has provided a Java program for a game called "Bulldog" with several classes and identified an issue where the game prematurely starts the next turn before a human player's turn is completed. The user wants to ensure that the game waits for the human player's decision before proceeding.

Figure 1: Prompting DeepSeek to not abbreviate its generations with comments. DeepSeek's live "thinking" can be seen in the bottom.

Model:

I started by asking the LLM to think about the inclusion of the MVC pattern in general to our program with the entire Program 6 file pasted. I wanted to see how aggressively or passively it would implement the pattern given no guardrails or guidance.

When the LLM got back it proposed a few “candidate” sets where it could implement the MVC pattern. The one we were tasked with implementing this assignment was not one of them. The LLM was given no guardrails, the task was clearly too big. I divided it into creating just the Model and essentially summing up the task as abstracting the Player ArrayList usage into its own object. I described each of the functions that should be given, and was met with a response containing the model class. I requested the LLM to not implement the class into the Prog6 file in order for me to read and verify its work (this is something I have learned throughout this semester. Sometimes prompt engineering is actually figuring out what prompt style better fits your workflow. In this example, I chose to slow the LLM down in order to slow myself down).

This step did not require much hands on or verification as I knew it was really writing a wrapper to the ArrayList class.

I grade DeepSeeks performance a 10 out of 10. I could not really imagine doing it much differently.

View:

Just like the model section I firstly asked DeepSeek for its thought process on adding a scoreboard viewer. Unlike the first Model prompt I specified that this was going to be for the score, it would use a Dialog box (from Java Swing), and it would be at the end of every turn. In my head there were a few things to be worried about, but again I wanted to see what the LLM would mention with no guardrails so I could either introduce them or expand on them.

DeepSeek mentioned the type of Java Swing Components to use which was good to know considering I had not quite thought about the actual visual representation or structuring. It decided on using a table which had as many rows as there were players and 2 columns (one for the name and one for their score). DeepSeek also mentioned how because the data is “different” each turn it can be simply built every turn. I did not have strong enough feelings or enough knowledge to know if there is a performance downside to this, but the logic was sound enough to me so I accepted its proposal to generate the code.

DeepSeeks implementation was quite different than I expected but ultimately acceptable. It decided to inherit from the JDialog class. This somewhat confused me as it differed from previous styles for other GUI setups in our Bulldog program, but then again it also made some sense considering the component was tossed every time it was closed. Making the process of calling “new” every time quite convenient. Not to mention the modularizing of code into separate files makes the main file feel less bloated. Additionally, it also follows some of the principles we have covered as the coupling between the Viewer and the Java Swing component itself are so tight, they might as well be the same.

I grade DeepSeeks performance a 7/10. It feels like DeepSeek made a decision which was more rooted in its data than it was for the context of our current program. Despite me mentioning the style and logical difference being somewhat understandable given the unique problem, it felt like there could have been better ways to address this.

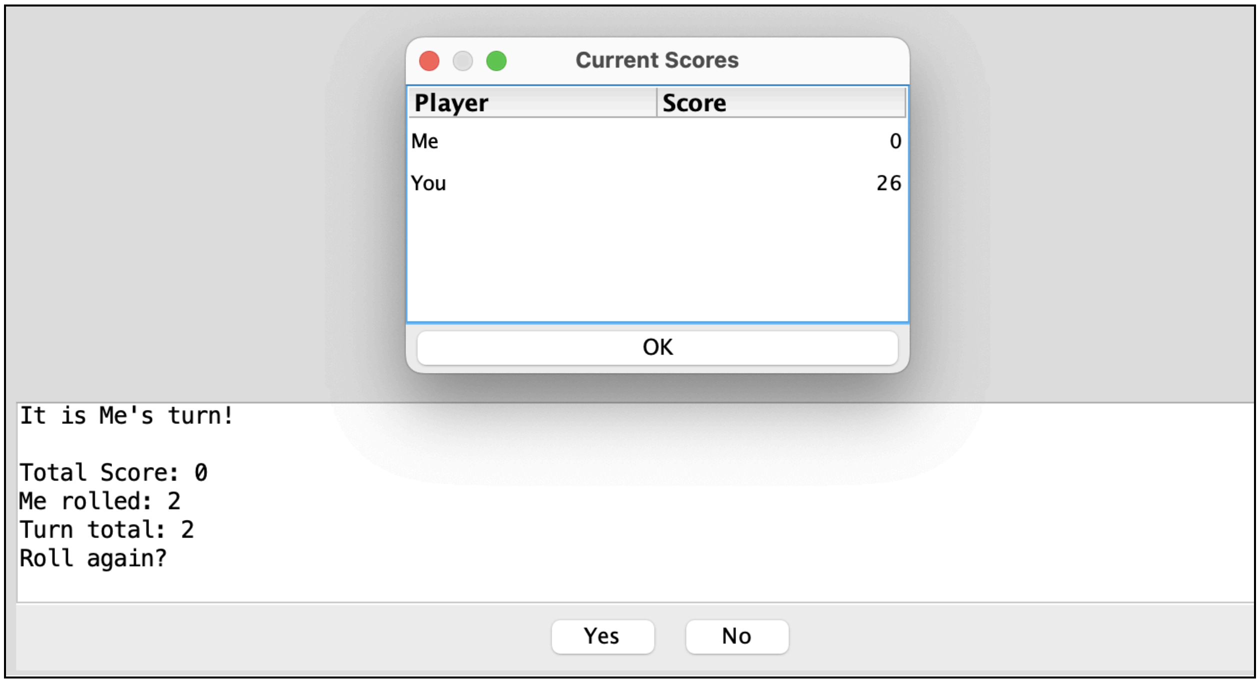
Integration into Bulldog:

For both the Model and the Viewer, after I had approved and received the generations for the new classes, I pasted the Prog6 file *again* with my prompt asking for the implementation. For example, for the Model I requested the LLM to replace the ArrayList usage and other functions with the new Model class. In order to make sure there were no hallucinations or assumptions, I created a temporary file that contained the generation from DeepSeek. This was my way of not losing what I currently had and to also make sure at the very least there were no syntax errors. I then quickly glanced at both files side by side to make sure the implementation was up to par.

The Viewer was less complex. When I first asked for the LLM's thought process, it shared the one function that would be necessary in Prog6 for it to work: "showScoreboard()" It was so simple I pasted the showScoreboard implementation, but then manually implemented the showScoreboard() where I knew it was appropriate in the gameflow.

In Figure 2 you can see the scoreboard properly working as intended. It pops up a dialog which showcases the current score. It properly updates each turn and closes when pressing Ok. As the assignment mentions, it is easy to interact or notice the difference in HumanPlayer games, as with automated players they move so fast the Dialogs start to stack as seen in Figure 3. To fix this, I would await the close of the dialog, or probably force close the dialog depending on the player. Perhaps I would prevent the scoreboard from showing if it is already on screen. I also wanted to implement sorting of the scores, but because it was not a requirement for the assignment I chose to omit it due to the fact that I am late to other assignments and DeepSeek keeps saying it's busy.

Overall, this has been my best experience with the LLM so far, but it makes sense considering MVC does not aim to solve any particular problem, but instead is there to organize our data and program in a specific way.

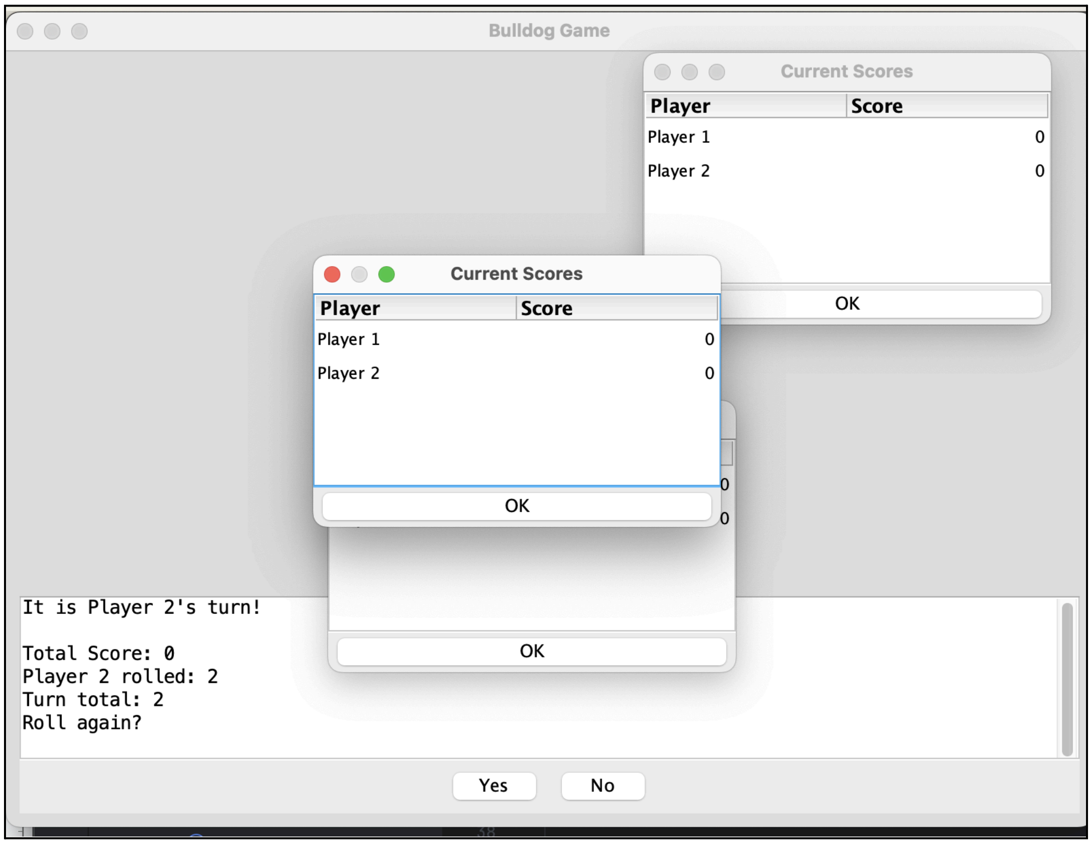

Current Scores

Player	Score
Me	0
You	26

```
It is Me's turn!

Total Score: 0
Me rolled: 2
Turn total: 2
Roll again?
```

Figure 2: The scoreboard working in Bulldog.


Bulldog Game

Current Scores

Player	Score
Player 1	0
Player 2	0

Current Scores

Player	Score
Player 1	0
Player 2	0

```
It is Player 2's turn!

Total Score: 0
Player 2 rolled: 2
Turn total: 2
Roll again?
```

Figure 3: Scoreboards stacking constantly with automated players.