Conclusion and Afterthoughts
By Ryan King

When I started this project, I assumed it would be relatively easy to do. After looking back at this assumption, this could not have been farther from the truth. I figured that I would struggle a little, but kept denying the feeling because I started my project almost one year in advance.

The main reason I chose this project is because my father introduced me to chess from age seven or eight, and since I have had years of prior experience with the game, it would be a more satisfying project that would not require much self-teaching or research.

Things I Learned

Learning More About Swing Multithreading and Event Handling

I did not choose to write Dark Blue in Java Swing arbitrarily. I already made a handful of smaller programs in Swing (blackjack, video poker) and decided that since it will still continue to be supported until 2025 or 2026, it was still a relevant platform for development.

I already learned a fair amount about `ActionListener`s in my prior two programs, as well as more about event handling from taking Organization of Programming Languages with Dr. Kumar. I figured this would be a breeze to accomplish. Again, I could not have been more wrong.

It was a tough pill to swallow when I found out about multithreading in Swing. It uses a single Event Dispatch Thread (EDT) to process events that get fired, but it also uses several worker threads to accomplish long-running tasks. In my blackjack engine, I called everything on the EDT on the click of a button. Looking back at that, I almost want to cringe. I knew how time- and resource-intensive Minimax would be, but I did not know the best way to let it run.

That was when I discovered the `SwingWorker` class. It requires the coder to override at least two methods: `doInBackground()` and `done()`. Looking through several tutorials on YouTube and other sources was slightly helpful, but it did not allow me to understand what *I* needed to get out of it; namely, allowing Minimax to run without burdening the EDT.

Accepting Help and Not Feeling Guilty

Despite the fact that I resolved to get most of this done by myself, I found myself going back to the Black Widow chess program and seeing how Amir Afghani built his engine. It was this code that showed me that my `SwingWorker` and `MouseListener` tasks were much simpler than I thought they would be. Returning a `Move` object from the `SwingWorker`'s `doInBackground()` method leads to it being received in `done()` and then processed. Many of the tutorials I found online were doing time-based executions of `SwingWorker` that would fill a progress bar or do other visual tasks and not let the user hang up the program. The multiple instantiations that come with this made the topic seem all the more confusing. Despite this setback, I ended up doing more non-trivial tasks myself such as parsing and validating a Forsyth-Edwards Notation (FEN) file and making sure castling rights, en passant tiles, and the like got parsed correctly.

The Complexity of Design

Despite that I have programmed several moderate to large-sized projects before, I did not anticipate the magnitude of this chess engine. This has required me to get more organized in order to work more effectively.

I started with the simplest classes and worked my way up to the more complex and abstract parts. I did more initial design and planning for my Piece, Tile, Board, and Move classes than I had ever done for any other project I have built. I believe this advance planning saved me a lot of headache later. Even though I changed a lot in regards to abstract methods in Piece subclasses, my class was still so well-designed that a few changes to methods meant nothing else in my source code had to change.

The Usefulness of Constant Debugging

As I worked on my "model" classes such as the piece, tile, board, move, player, and other classes that have nothing to do with the GUI, I tested them extensively on the command line and found lots of errors that made me pull my hair out. This was painful, tedious, and boring to do. However, I persisted, knowing it would save me lots of time and frustration later on.

Keep It Simple, Stupid!

The AI algorithm that I originally took from Black Widow was overly complex and ran for a long time. It also did not move very well. The second algorithm that I found on a JavaScript program was much simpler, faster, and more brutal. It barely did any

calculations compared to the Black Widow algorithm. Since it was designed to be run in real time on a web browser, I felt it was more suitable for my program. This made me realize that simpler design is better.

Trying New Design Patterns

Generally speaking, I only prefer to have one class per source file if I can help it. Sadly, this did not work out well when I assembled my `GUIBoard` and tried to make it move. The individual tiles do not know anything about any others, and the board that contains them acts as its own unit. By following Amir Afghani's design of placing the GUI tiles, board, and main GUI class together in one source file, I was able to simplify the solution I ended up using.

I also got exposure to the Factory and Builder design patterns. Though Afghani went all out and used a move factory class with a private constructor for each Move subclass, I did not want my classes to be that complicated. I just created a Factory interface for generating multiple types of objects. I had seen the Builder design pattern when I wrote my Android project in Organization of Programming Languages, but I did not know much else about it. This allows me to construct a board quickly and easily, which is useful for not needing to know about the internal workings of a class and just assigning things to it.

Future Improvements

A minor improvement I could make to the Dark Blue chess engine in the future is disambiguating moves in algebraic notation. When two or more identically typed pieces can move to the same tile on the same turn, the following field or fields may need to be specified to prevent ambiguity (in descending order of importance):

- The moving piece's file of departure if it differs from that of another piece (the piece's current column as a letter)
- The moving piece's rank of departure if it differs from that of another piece (the piece's current row as a number)
- Both of the above fields if 3 or more pieces can move to the same tile (this is highly unusual and only happens when one player makes multiple promotions)

Thus if there are three queens that can move to e1 (one located on e4, the second on h4, and the third on h1), moving the h4 queen to e1 would read "Qh4e1" instead of "Qe1". I decided not to do this because each move string is returned directly from the

`Move` object itself and I did not incorporate any provisions for scouting out identical pieces.