Overview

The project has 3 primary classes:

- **Chess:** This is the main class component of the project, responsible for managing the overarching interactions of the system. This is what gets instantiated in the main function.
- **Game:** This manages a single game state and encapsulates the core game logic and interactions.
- **Board:** This represents a single board state and encapsulates the logic for things like piece placement and movement.

The main relationships between these are that **Chess Owns Games**. Chess can own as many games as it wants because you can play multiple games in a session. The other main relationship here is **Game Owns a Board**. This makes sense as each game is played on one board, so each game object only owns a single board object.

Each time the *game* command is ran to play a new game, the chess object will instantiate a new game object for the game, and the game object will instantiate a new board object.

The **Chess** class also has a **owns a** relationship with 2 other classes:

- **DisplayBoard:** An abstract class with **TextDisplay** and **GraphicsDisplay** as subclasses. Responsible for displaying the board state to the user. An observer design pattern is deployed here.
- **Player:** Another abstract class with **Human** and ***Computer*** as subclasses (Computer is abstract with concrete subclasses for each level). Responsible for facilitating moves from either a computer or human player.

Chess will typically own 2 **DisplayBoard** objects, one for the console version and one for the graphical version. When the chess object gets instantiated, the DisplayBoard objects are also instantiated. Both displays are instantiated by default, but the graphics display can be omitted via a command line argument.

Chess can also own as many player objects as it since multiple games with different players can be played in a single session. Whenever a new game is created, the chess object will instantiate 2 more player objects. We store all the games and players for possible future additions.

The **Board** class also **owns *Pieces***. ***Pieces*** is an abstract class with a concrete subclass for each of the 6 types of pieces. A board object can own as many piece objects as a board can allow. When a board object gets instantiated, the board will instantiate the required number of piece objects for the desired board.

When a *move* command gets ran, the command is first parsed by the chess object. Then the chess object calls the corresponding *make_move* function for either the human or computer player. This function calls the game object to validate and execute the move with the board object.

At the same time, the chess object notifies the displays to display the board. The boards get the state from the chess object then displays the new state.

Design

Resilience to Change

The

Answers to Questions

Question: Chess programs usually come with a book of standard opening move sequences, which list accepted opening moves and responses to opponents' moves, for the first dozen or so moves of the game. Although you are not required to support this, discuss how you would implement a book of standard openings if required.
Question: How would you implement a feature that would allow a player to undo their last move? What about an unlimited number of undos?
Question: Variations on chess abound. For example, four-handed chess is a variant that is played by four players (search for it!). Outline the changes that would be necessary to make your program into a four-handed chess game.

Question 1: Openings

Question 2: Undo

Question 3: 4-Handed Chess

Extra Credit Features

Final Questions

1. What lessons did this project teach you about developing software in teams? If you worked alone, what lessons did you learn about writing large programs?

Through the process of building and developing this chess program, there were many lessons learned, and major takeaways for every member.

- **Division of Labor**: One of the most important aspects that constituted to the success of our group project was the even division of work between team member, to ensure that every member was carrying their weight in the team. We found that keeping track of how much work each group member has completed and giving additional tasks to group members who had their hands empty was effective in ensuring that no group member was too stressed out by their workload.

- **Modular Design & Documentation**: The division of labour was able to be executed smoothly as a result of the modular design of our program, as we could assign each member a part of the program that they would work on without having to wait for another member to finish their tasks first. Furthermore, the detailed documentation and comments of how certain functions and methods worked allowed other members to pick up quickly where another have left off, ensuring smooth transitions between different stages of the project.

- **Importance of Version Control**: From experiences in our previous co-op terms, we knew the importance of version control systems such as Git in team software development environments. We utilized branches to ensure that we could always maintain a working version of our software to build on top of and to ensure that each group member's work did not clash during development. Furthermore, Git allowed us to effectively conduct code review of changes, to get multiple opinions before it was merged into our main branch.

- **Test Driven Development (TDD)**: Throughout our development process, we focused much of our development efforts on the tests for our program, this allowed us to be confident in the changes we made in our code, and ensure that additional features we added on did not break the core functionalities of the chess game.

- **Continuous Integration and Continuous Development (CI/CD)**: Early on in our project, we setup continuous integration and continuous development pipeline through Github Actions to ensure that we ran our entire test suite of extensive test cases on every code push, pull request and merge. This allowed us to be confident that our changes did not break production without having to manually re-run tests.

2. What would you have done differently if you had the chance to start over?

Question 1: Lessons Learned

Question 2: Done Different