Chess Plan of Attack

Jonathan Cheng, Fred Yang, Sophie Yang

Project Breakdown

For our project, we have decided to go with the chess option. We are planning on using an MVC architecture to implement our chess game in order to keep coupling low and cohesion high, as well as to maintain SOLID principles.

Model

The Model in our architecture will be composed of the `Grid` class as well as the `Piece` abstract base class and its subclasses. There will be one main `Grid` instance used in our implementation and it will own several `Piece` instances. Together these objects will store data about the current state of the board such as the position of every piece, whether a player is in check, what a particular piece's legal moves are etc.

View

The View in our project is made up of the `Display` abstract base class and its subclasses. Instances of these `Display` objects will observe the `Grid` instance using the subject/observer design pattern. Any time a player makes a move, the `Grid` instance will notify all of the `Display`s observing it. Each subclass of `Display` will be used to output the state of the board to a specific output channel. Specifically, we will have one subclass for command line output, and one subclass for graphical output.

Controller

The Controller in our project is made up of the `Game` class as well as the `Player` abstract base class and its subclasses. The `Game` class will be responsible for handling all the game logic related to our Chess implementation. To be specific, it will handle things such as starting a game through the CLI, entering setup mode, checking win conditions during a game, keeping track of scores, etc. The `Player` subclasses will implement different ways of determining the next move to be made for a specific player. For a `Human` subclass, this entails prompting the user through the command line for a new move. For a `Computer` subclass, this entails some decision algorithm used to generate a next move.

Initial Plan and Priorities

First and foremost, in order to test anything out within our chess game, we need to be able to see the current state of the grid. So our first priority will be to implement the `Grid` and `Display`

classes. At the start, we will only implement the `TextDisplay` subclass, since the graphical display is not a critical functionality.

Next, in order to be able to interact with our program at runtime, we will implement the main CLI of the `Game` class, as well as the CLI associated with `Human` players.

Next, in order to facilitate the testing of our game, we will implement setup mode so that we can test the logic of pieces in different positions without our Chess game needing enough of itself to be implemented to actually reach that state naturally.

Next, now that the foundations for the rest of the functionality are set, we can begin working on the logic for calculating legal moves for a piece as well as the procedure for moving/capturing pieces. We will start with only the most basic rules (simple moves and captures, ignoring checks, etc.), and we will incrementally add functionality for the more complex rules (checks, checkmate/stalemate, castle/en passant, etc.).

After having fleshed out the entirety of the game logic, we can now proceed with implementing the algorithms for different levels of the `Computer` players.

Lastly, we can implement the graphical interface for viewing the state of a chess game.

Delegation of Responsibilities

Sophie:

- Implement the Grid class, ensuring it properly tracks the location and state of all pieces.
- Design and implement the Piece abstract base class and its subclasses (like Pawn, Rook, Bishop, etc.).
- Write tests for the Grid and Piece classes to ensure they behave as expected.
- Once the Grid and Piece classes are stable, start working on the logic for calculating legal moves for a piece and the procedure for moving/capturing pieces, starting with basic rules.

Jonathan:

- Implement the Display abstract base class and the TextDisplay subclass for command-line output.
- Design and implement the observer pattern for the Grid and Display classes.
- Once Display is functioning correctly with the Grid, begin implementing the Game class with the main CLI.
- Start working on the win condition checks and scoring systems in the Game class.
- Research potential decision making algorithms for chess

Fred:

- Design and implement the Player abstract base class and the Human subclass, including the CLI prompting for the next move.
- Develop the setup mode, allowing for manual testing of piece positions and interactions.
- After the Human class is stable, start implementing the Computer subclass with a basic decision algorithm.

Joint:

- Continually refine and improve the Computer player's decision algorithm based on game testing.
- Create the code skeleton and design the high-level breakdown of the application
- Create the Final Report
- Testing

Estimated Timeline

Deadline 1 (July 24):

- Completes the initial development of the Grid and Piece classes.
- Completes the Game class with CLI
- Jon completes the Display abstract class and TextDisplay subclass.
- Fred completes the Player abstract base class and the Human subclass.
- Setup mode

Deadline 2 (July 26):

- Completes testing for the Grid and Piece classes.
- Implements the CLI for Human players.
- Basic move/capture rules for Piece classes.
- Win condition checks and scoring systems in the Game class.

Deadline 4 (July 28): Advanced Rules Implementation and Computer Player Development

- Implementing advanced rules (castling, etc.).
- Refining the game logic.
- Computer player class with a basic decision algorithm.
- Begins testing game with different scenarios.

Deadline 5 (July 31):

- Advanced computer players.
- Additional features.
- Final testing.
- Final Report.

Questions

Q1. Chess programs usually come with a book 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.

To implement a book of standard opening move sequences, we can use a prefix tree to efficiently store and retrieve accepted opening move sequences. Each node in the prefix tree would use a possible state of the board as a key, as well as the name(s) of the standard opening(s) associated with that state of the board. Each edge going out of a node in the prefix tree would represent a possible move that can be made from the state stored at that node. In chess, standard opening moves rarely ever change, so in order to construct this prefix tree, we could simply use some sort of grammar/encoding to store the structure of this tree and we would read it into memory when the program starts. When an actual game begins, the program will start at the root of the prefix tree, it will then display all its child edges as the list of possible standard moves. Each time a player makes a move, it will traverse the prefix tree to find the corresponding new state, and redisplay the associated standard moves in that state.

Q2. How would you implement a feature that would allow a player to undo their last move? What about an unlimited number of undos?

To allow a player to undo their last move, we can approach it in terms of "saving" a state of the game. Before every move, we save the entire game's state into a variable - where each piece is, whose turn it is, which pawns just moved two spaces in the previous turn, and which rooks/kings have ever moved. This way, if a player would like to undo their last move, we could simply overwrite the current state with the previous state variable. If we wanted to implement an unlimited number of undos, rather than having a single variable storing the previous state of the game, we could instead use a stack to store all previous states. To undo a move, we simply pop the stack and reload the game into the popped state. This stack would grow with every move that is made in the game, and would allow unlimited undos, given that we have enough memory to store the whole stack. If we wanted to optimize the space utilization of this undo feature, we could store the changes between states rather than the entire state, and parse these changes to reconstruct the past state, rather than storing the entirety of the past state.

Q3. 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 four-handed chess game

Some of the major changes would include:

1. Board size: The grid class would need to be modified to accommodate a larger board size
2. Piece setup: The default starting positions of the pieces would be different.
3. Setup mode: Setup mode for normal chess validates that there are exactly two kings of different color. With four player chess, this would need to be four kings of different color.
4. Player class: The Game class will need to be modified to support 4 players instead of 2. The turn-based logic must also be modified to rotate around 4 players. It would be best to use an enumerator rather than booleans to store whose turn it is.
5. Win conditions: Some rules and conditions for checks, checkmate must be changed. For example, checkmating one player does not end the game for all players. What happens after one player loses must also be taken into account.
6. Pawns: With 4 players, some pawns can move left and right instead of up and down.
7. Scoring system: needs to support 4 players.
8. Display classes need to render a new layout.