CS 246 Project Plan of Attack - *Chess*

dyli r473li[1]

1. Introduction

   This plan of attack provides an overview of class designs of our current program, as well as a timeline that showcases our current and planned milestones.

2. Breakdown and Timeline of tasks

   Since one of the group members dropped out of the course on July 16th, we re-distributed the rest of our work. Henry will be responsible for designing and developing the program, while Devin will be responsible for quality testing as well as compiling documentations.

   Timeline(as of July 18th):

   - [x] ~~[July 4] text-based display (debugging purposes)~~
   - [x] ~~[July 6] setup mode~~
   - [x] ~~[July 6] resign & draw by agreement & scoring board~~
   - [x] ~~[July 7] King & Pawn moves (incl. en passant)~~
   - [x] ~~[July 9] visible area management, Fog of War demo, check~~
   - [x] ~~[July 11] better chess board hierarchy, better move check, castling~~
   - [x] ~~[July 12] pawn promotion, checkmate, stalemate~~
   - [x] ~~[July 14] finish game demo (human vs human)~~
   - [x] ~~[July 19] error handling, design test cases~~
   - [x] ~~[July 19] Submit DD1~~
   - [x] ~~[July 21] level 1 AI (random moves)~~
   - [ ] [July 21] level 2 AI (1 move ahead)
   - [ ] [July 21] level 3 AI (2 moves ahead)
   - [ ] [July 22] graphics display (X11)
   - [ ] Optional:
   - [ ] [July 26] level 4 AI (5 moves ahead)
   - [ ] [July 26] Some easy variants (960, Horde, Atomic, Fog)
   - [ ] [July 26] Undo
   - [ ] [July 26] Submit DD2

[1] One of our teammates (*j327park) dropped out of course on July 16th.*

3. Program Design

From day 1 of designing the program, we have taken into consideration potential variations on parts of our program. To be more specific, we considered how to structure our program so that we can implement many different chess variants with relative ease (from easy-to-implement variants like Hoard, Chess 960, or Fog of War, to more demanded variants like Atomic Chess or 4-Player Chess. See https://www.chess.com/terms/chess-variants).

Here we break down some key components of a “game of chess” that we categorize into “flexible” and “fixed”, where being “flexible” means the game component is variable among different chess variants, and “fixed” means the component is fixed no matter what chess variants we are playing with. We only considered the subset of variants provided through the link above. There may be many other variants out in the wild, but for simplicity we will only include a handful ones for presentation.

- Fixed: **chess pieces** (in all variants, each player controls a subset of the six chess pieces, with one king each), piece move **rules** (in all variants, pieces can only move the way they’re allowed to; castling, promotion, en passant), **game logic** (a check is defined the same way across all variants; all players take turns to move; when king is captured, game ends, etc).
- Flexible: **chess board display** (in four-player chess, there are four sides of players; in Fog of War, certain squares are hidden under a fog), move **consequence handling** (in Atomic, capturing implies exploding 3x3 squares; in Fog of War, there is no consequences of check and checkmate), **player handling** (in 4-player chess, there are 4 players, and they can team up sometimes).

These identified fixed and variable factors will guide our overall design of our chess program. We will list a few classes and provide some high-level descriptions:

- **Board**: This is the main class that handles almost everything chess-related, from displaying the chess board through a **TextUI** (observer), to managing and (co)-owning **Piece** objects, to answering sophisticated chess-rule related questions by observing the patterns on board. **Board** should be overridden/decorated by each chess variant.

- **Piece** objects represent piece rules. They implement a **PieceIterator** object, which when applied on the **Board** will iterate through all squares that are “covered” by the current piece. They are fixed through all chess variants.
- **Player** is where we implement both human and computer player behaviours. The most important one is, of course, making a move on the **Board**. It takes care of parsing commands from player input, and translates them into actions that it later applies to the **Board**. **Player** also co-owns **PIece**s. Their behaviour may vary in 4-player mode.
- **Move** is another important abstraction from the chess games. Upon a Player object receiving a “move” command, it will parse a **Move** (either given by a user’s instruction or fabricated by computer players) which is executed onto the **Board**, and whose consequences will be applied to both **Board** and **Player**. In theory, **Move** will also handle the “undo” part of command. They should be independent of chess variants.
- **TextUI** is an observer of the **Board**, and it displays formatted command line output.
- **Controller** controls the flow of the game. It doesn’t know anything concrete about chess, except setting up games and running matches through public interfaces of **Board**.

And together, **Board**, **TextUI**, **Controller** completes the MVC architecture.

4. Questions and Answers

Question 1: 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. See for example https://www.chess.com/explorer which lists starting moves, possible responses, and historic win/draw/loss percentages. 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 plan to use a linked tree data structure to store a subset of standard openings. Each node in the tree represents an opening position, and edges represent the moves that lead to subsequent positions. The root node represents the initial board setup. Then in real-time games, we may use a pointer to traverse the tree following players’ actions, and give them suggestions of opening moves (by selecting random next opening moves, for example).

To implement such a tree structure, we may first hand-code a few openings in linked lists, and merge the lists into trees at runtime. We may use a separate class **Openings** to handle this part.

Question 2: How would you implement a feature that would allow a player to undo their last move? What about an unlimited number of undos?

To implement “undo”, we can simply add it as a public method in **Move** class, since “move” and “undo” are naturally inverse of each other. In particular, Move objects will have to remember a subset of the **Board**’s states before making a move, so that when the move is undone, the **Board** can safely go back to what it was.

To implement a series of undos, we will let the **Board** object own the **Move** objects. In particular, we will store the **Move**s as well as the captured pieces associated with this move inside a stack, and when “undo” is called, we will pop the stack. This calls the destructor of **Move**, which is overloaded to apply the “undo” method. This allows the reversal of the moves and restoration of any captured pieces. We can keep “undoing” by popping the stack until it is empty, in which case the board should be in its initial state, and no further “undo” should be allowed.

Question 3: 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. (If it’s important to your answer, state whether you’re assuming free-for-all or team rules and then answer the question. You don’t need to get too specific into the rule set changes in answering the question though; your focus should be more on what would need to be altered at the high level of the design?)

As we stated already, we incorporated a suite of chess variants into our program designs. In particular, we intend to implement chess variants mostly through overriding methods of classes that handle regular chess rules. But there should be no significant changes of class interactions and design patterns so forth. In the 4-player case, we need to override a few public methods of the **Board** that handle rule-checking. For example, we currently have a public method “**IsCastling**()” that takes two coordinates on **Board**, and tells whether it is a castling move. In regular chess, castling happens only horizontally; in 4-player chess, we also need to check vertical castling. Certain public methods also depend on who the current player is, and which team they belong to. For regular chess, this is managed with two indicators “**player**” and “**opponent**”, but in 4-player chess, we need something a bit more sophisticated.