Design document

Anthony Sin, Tanisha Dhami, Michael Raad

Introduction

We worked on the **Chess** project, and have implemented just about all the necessary features and some extra ones too. Our game supports all of the “unique” moves that chess has to offer, such as castling, en passant, pawn promotion, and the pawn’s initial option to move two squares instead of one. In addition to this, we provide a fool-proof testing harness that allows for any game combination between a human and computer. Finally, our graphics display uses custom drawn pieces and a wooden chessboard aesthetic for the most enjoyable gameplay.

Overview

1. Game Initialization:
    - The chess program begins by taking in the variant of chess being played and creating an instance of the Game class.
    - The Game constructor is responsible for initializing the game state, including creating a unique instance of the chessboard (StandardBoard), determining the variant of chess being played, and setting the starting color.
2. Game Setup:
    - The Game instance may enter setup mode to configure the initial state of the chessboard.
    - For the initial setup, the StandardBoard class's config() method is called to set up the default configuration of pieces on the board.
3. Player Initialization:
    - Players are added to the game using the addPlayer() method of the Game class.
    - Player controllers, such as HumanPlayer and Computer1, are instantiated and associated with the game, based on the type of game being played.
4. Game Start:
    - The Game::start() method has the main loop, cycling through players and invoking their doTurn() methods.
5. Player Turn (Computers):
    - Within Computer’s doTurn(), the computer player makes decisions about the next move.
    - This involves calling methods from the StandardBoard class to evaluate the current board state, generate legal moves, and select one based on the specific logic.
    - For a player controlled by a human (HumanPlayer), the doTurn() method involves receiving input from the user to determine the desired move
6. Move Execution:
    - Once the computer player decides on a move, the move is executed on the chessboard.
    - The makeMove() method of the StandardBoard class is invoked to apply the move to the board.

7. Board Updates and Display:
    - The StandardBoard class notifies any attached displays (e.g., TextDisplay, GraphicsDisplay) about the changes on the board using the notifyDisplays() method.
    - Displays update their representations of the board by calling the outputDisplays() method of the Board class when necessary.
8. Game Continues:
    - The game loop continues through the vector of players, allowing players to take turns until a game-ending condition is met.
9. Game Ends:
    - When a game ending condition is met, the doTurn() will return an enumeration which indicates the player is out of game.
    - Then the main will update the score accordingly.
10. Program ends:
    - When cin.eof() is met, the program will finally end.
    - Main then will output the final score and the program will terminate.

Design

Class cohesion, temporal cohesion: Each class in our project is designed to have high cohesion, meaning that its methods and attributes are closely related and contribute to a specific, well-defined purpose. Further, we organize our code to have temporal cohesion, meaning that methods that are likely to be executed together are grouped closely.

MVC

- **Model (Chessboard)**: The chessboard, represented by the Board class, serves as the model in the MVC architecture. It encapsulates the internal state of the application, including the configuration of pieces, legal moves, and game rules. The Observer pattern is integrated into the model to handle communication with displays.
- **View (Displays)**: Displays such as TextDisplay and GraphicsDisplay act as the views in the MVC pattern. They are responsible for presenting the current state of the chessboard to the user. By registering as observers using the Observer pattern, views dynamically update their representations based on changes in the model through the use of the notify() function.
- **Controller (PlayerControllers):** The player controllers, including HumanPlayer and Computers, function as the controllers in the MVC architecture. They receive input from users or make decisions in the case of computer players. The controllers interact with the model to perform game-related actions, such as making moves on the chessboard. The controllers act as intermediaries between the user and the internal game state.

Decoupling and Separation of Concerns: MVC promotes the separation of concerns, allowing each component to focus on its specific role. The model encapsulates the game state, the views handle presentation, and the controllers manage user input and game logic. This separation enhances code organization, maintainability, and flexibility.

Observer pattern

- **Subject (Chessboard)**: The chessboard, implemented through the Board class, acts as the subject. It maintains a list of registered observers (displays) and notifies them of any changes in the board's state.
- **Observers (Displays):** Display classes, like TextDisplay and GraphicsDisplay, serve as observers. They register with the chessboard as observers and update their representations when notified of changes.

Flexibility and Decoupling: The Observer pattern enhances flexibility by allowing new displays to be added without modifying the chessboard class. This decoupling ensures that changes to one component do not affect the others, promoting a flexible design.

Resilience to change

1. Number of players:
Instead of relying on fixed player variables, we dynamically manage player information by storing them in a flexible vector structure. This design choice enables us to accommodate new players without the need to alter the core gameplay loop. By iterating through the vector, each player is assigned a turn, providing an adaptable solution that easily scales with the addition or removal of players.

2. Cmake:
We use cmake which simplifies the process of generating the executable. This adaptability is particularly valuable if the project structure or dependencies need to be changed, enhancing the overall flexibility of our development environment.

3. Colour
The colours of the players are stored in an enum class, allowing us to easily change it to any custom set of colours if needed. In case, we could

4. Board variant
Instead of one chess board class, we have an abstract board class. Although due to the time constraint we only implemented the standard 8x8 chess board class, the flexibility of an abstract board class allows us to easily implement more customized boards, such as 4 player chess, bigger boards with customized rules, different shapes, etc. The only constraint would be the board has to be represented by squares (two-dimensional).

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. 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.

Answer:

Changes

The basic structure of the code and the classes would be the same. We would only need to make a data structure that stores information about book moves, and make changes to our displays (or make a new display) that displays the available book moves and the related information (win rate, number of times played). We could also modify some of our computer players to utilise the data structure that stores the book moves.

Data structure

In order to store information about the book moves, we would store all the book moves in a tree. Here, the depth of a node would represent how many moves have been played until then (in total, counting both players). The root node would be the starting position of the board. At each step (node), the node would have one child for every book move that can be made from that state of the board. The historic win rate associated with each child can be stored with the child like a dictionary, by making a tuple. Making a move will traverse one step down the tree.

User Interface

When it is a player's turn to move, we would simply display the details of each child, which would include the suggested move itself, and its historic win rate. If the user makes a move

that we did not have data for, that would be like searching for a child node with a key that does not exist. In that case, we could print out something like “There are no games with this position found in the database.”.

Computer player

If we are implementing a computer player and we always wanted to pick the best move (based off of historic win rate as well as play count), our computer could pick the move based off of some calculation done using the win rate and number of times the moved was played (to obtain some “confidence level”). We would then traverse down the tree to the child which represents that state of the board. Now the tree is updated and our BookDisplay (if applicable) would display the updated state of the tree for the human player.

Question:

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

Answer:

Moves History

We would need to store every board state that we have reached, or every move made by each player. For this, we could use a stack. We would push each state that the board reaches onto a stack. To undo a move, we would just need to pop the topmost frame of the stack. Rather than storing the state of the board as the position of each piece, we can just store the move made in the stack with a vector<Pair<Square, Square>>. Every undo just gets the pop_back() element, and runs the reverse moveTo(destination, originally). In addition, if at any move a piece is captured or promoted, we will need to store that too so it can be undone. For this, we can push both the move (when a piece is moved) and promotion/capture (when the piece type changes/ is removed). Each item of the stack would have fields Move and PieceChange.

An example of **capture**:

Move: C1 -> D3, PieceChange: {BlackPawn, D3} -> {nullptr, 00}

An example of **promotion**:

Move: E7 -> E8, PieceChange: {WhitePawn, E7} -> {WhiteQueen, E8}

New Move structure that we have:

Move: Square from, Square to, int value, string special.

What we did in simulateMove() → locally storing the captured piece, and reverse the move back if it's invalid (puts own King in check)

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

Answer:

Our UML already accommodates possible variants of chess. In order to accommodate for different variants of chess, we must deal with: board size, number of players, and pawn direction.

Board Size:

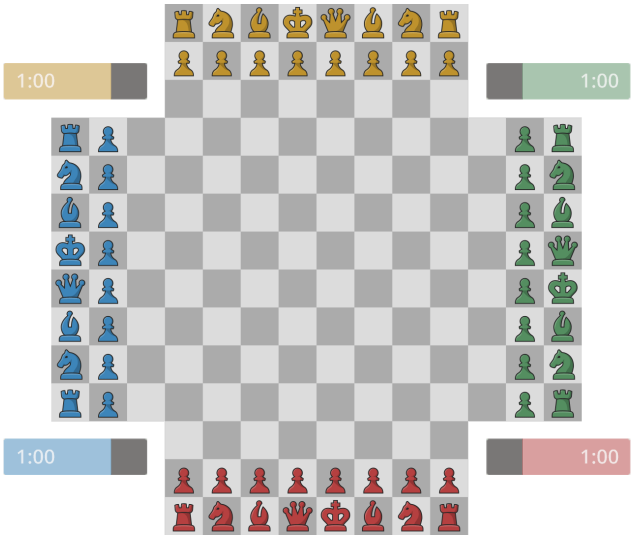
We have an abstract board size, different "concrete" boards can have a different number of rows and columns. What is considered a "legal move" is bounded by the dimensions of the board in play. Also there might be some places that specifically could not be reached by the pieces, such as 4-players chess. In the picture below, we see the corners should not be reached

Number of Players:

Our Game class has a field that holds a vector of 2 or more players. When a game is played, we traverse this vector one by one and call the doTurn() method for each player. If a player loses (is checkmated or resigns) then it is removed from the vector and will have no further turns. When only one player remains in the vector, that player has won.

Pawn Direction:

Pawns are the only piece that have different orientations depending on their colour. Each piece is assigned a colour (described by an enum which we can expand for multiple variants with different



colour pieces). For pawns, we can specify a behaviour (direction) based on its colour.

Extra credit features

- The graphics look nice - colour + piece drawing thing
- Observer pattern
- Unique pointers
- Board colors custom
- Cmake?
- Computers can resign

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?

The most important lessons were the importance of having a common writing style, documenting code well, and regularly pushing our code to the remote github repository.

We initially started off with different formats of indentation, but soon realized that we should fix a standard. Similarly, regularly pulling from the local repository before making changes and pushing them as soon as a new feature was implemented really helped us in avoiding merge conflicts.

Another important lesson was to write code designed to be read and understood by others. We added documentation for all our functions at each major step to make the code easier to read.

To add on to that, the importance of coming to a consensus on the definition of methods.. Since some of us work at a different pace, when it comes to integrating the whole program, when some of us expect this function to work in a way that others made it not to, it causes frustration and time to fix the program.

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

Given the opportunity to restart the project, we would take into account the other exams/ major assignments we had when making the Plan of attack. This would allow for a more realistic and balanced Plan of Attack.

We would incorporate a more robust testing and debugging strategy. We did test our game at every stage, but having a complete test harness would allow us to catch bugs more easily, compared to leaving the testing for the end.

Conclusion

And that's a wrap on our chess project journey! It turned out to be more than just coding – it became a lesson in teamwork, version control headaches, and the importance of writing clear code. Looking back, we realize we've learned more than we expected. We also had our fair share of fun with drawing out the pieces pixel by pixel, playing around with the colour of the board, etc. Thank you!