Introduction

We would like to note that during the use of this program, the user can only play one game at a time. If the user would like to change the players (e.g. change from a player to a bot), then the user can input `game human computer1`, but the score of the previous game will not be continued, nor will it be displayed. Only the score of the last game run will be displayed when the program terminates.

Overview

The user interacts with the program via `main.cc`, which reads their text commands from standard input. `main.cc` then interacts with a `Game` object. Main.cc also properly handles for invalid inputs and misspelled commands. For the majority of the program's functionality, interacting directly with the `Game` suffices. main.cc also contains Observer objects, so the game can be displayed in standard output and via a graphical window.

The `Game` has fields: two `Players`, two double fields to hold each `Player`'s score, and a `Board` which represents the chess board. The Game has several functions which allow for users/ the program to end each round of play: this would update each player's score accordingly, and resets the Board.

The `Board` primarily has a two-dimensional array of `Piece` pointers, like the grid of a physical chess board. The `Board` is used to manipulate `Pieces` and their positions. We refer to the positions on the Board as tiles.

`Piece` is an abstract class, used to generalize concrete chess pieces. The concrete subclasses are: `King, Queen, Bishop, Rook, Knight, Pawn, Empty`. `Empty` is used to represent a tile which has no (physical) piece on it, the reason will be explained in the design section. `Pieces` do not inherently know their positions, nor their surroundings, so each `Piece` has a `getValidMoves()` function which takes a position and a reference to a `Board` as parameters. A `Piece` also has its color and type as char fields for convenience.

The `Board`'s second responsibility is to move pieces. The `Game` also has a function to move pieces for convenience of interaction with `main.cc`.

We refer to a `Piece` *p*'s threatened tiles as tiles which *p* can move onto, or tiles that have a teammate which *p* protects.

A `Piece` *p* protects another piece *q* if *p* and *q* have the same color and *p* can move to *q*'s position in the event that *q* moves or is captured.

Design

1. Observer design pattern:

We use the Observer design pattern to display the game through various mediums. The program uses an observer which prints the board to standard output, and another observer which draws the board in a graphical window.

`Game` is a subclass of `Subject`, which contains a list of `Observers` it should notify. In general, both `Observers` call `subject->getState(int i, int j)` to retrieve a character that represents a `Piece`, and will call it for all positions in the `Board`.

2. Strategy design pattern:

The `Computer` class employs the Strategy design pattern. This allows users to pick difficulty levels for the bots they play against. A `Computer` has an integer field which represents its level, and a function `autoMove` that takes a `Board` as a parameter, to determine what move it should make.

The `autoMove` function depends on the abstract `Difficulty` class. Depending on `Computer`'s level, it will create a local/ temporary `Difficulty` object to help determine what move to make. The concrete `Difficulty` subclasses are: `L1` and `L2`.

The strategy design pattern makes it easy to write `autoMove` for many difficulty levels, since each level can have a drastically different way to determine what move to make. For example, level 1 `Computers` will use the level 1 `Difficulty`, `L1`, to generate a random legal move by calling `L1::computerMove`. As such, this method finds all the pieces which belong to the `Computer Player`, then randomly selects one of these, then generates all of its possible moves, and also randomly selects one of these. In contrast, `L2` looks for the highest-valued `Piece` it can capture, or if it can engage a check (in similar fashion, it looks through all `Pieces` that belong to itself and their valid moves). These differing methods have the same basis, requiring to search the entire `Board`, so unifying `L1` and `L2` under the abstract class `Difficulty` allows us to call the `computerMove` functions, without worrying about how they work. All the user needs to understand is that the higher-level bot is most likely more difficult to play against.

3. Visitor design pattern

We used the Visitor design pattern to determine the state of the game after each move. We primarily use them to see if a player is in check, or if there is a stalemate. Depending on the type of visitor, they will visit every `Piece` on the `Board` and run some logic that modifies a local field. After visiting every `Piece`, the local field is accessed. Since the game can only have one state at a time, we use a helper function `Game::updateGameState()` to run both Visitors, then determine what state the game is in.

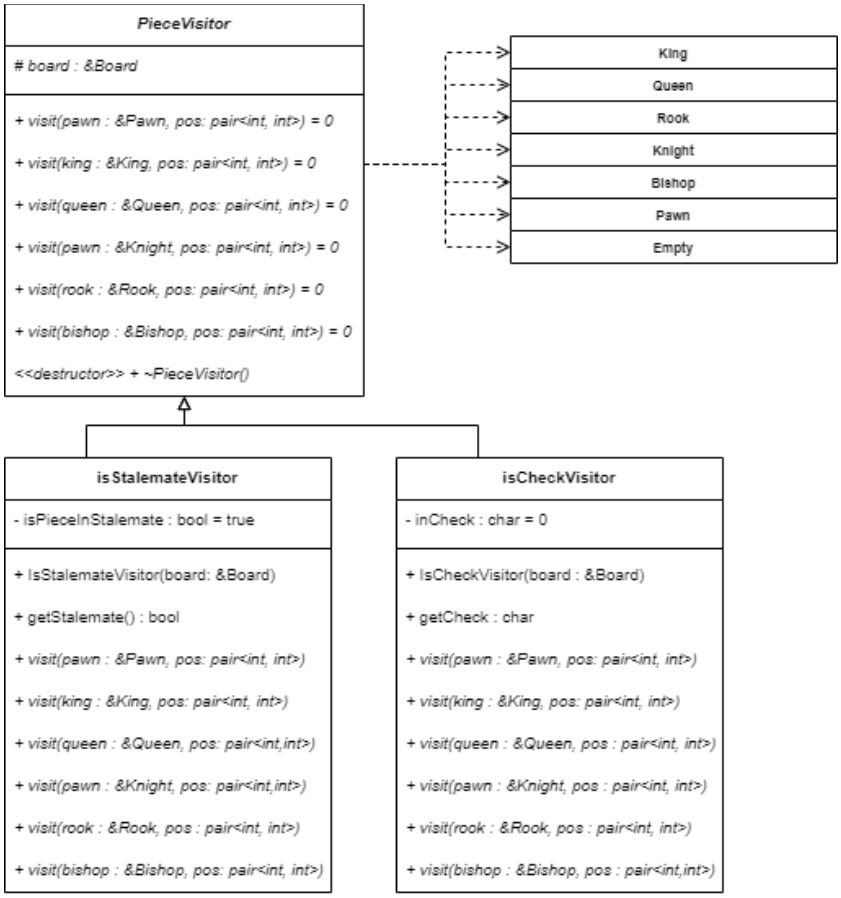

Figure 1: Visitor Design Pattern UML Diagram

The check Visitor is `IsCheckVisitor`. Upon visiting a `Piece`, `isCheckVisitor::computerMove()` looks for all the valid moves the `Piece` can make and searches the `Board` at that position for the opposing King. If so, the local character field `inCheck` is assigned a character which represents the King in check. `Game::isInCheck()` depends on the `IsCheckVisitor` (doesn't have scope outside of `isInCheck()`) and loops through all `Pieces` on the board. `isInCheck()` runs after a Player makes their move. During the loop, if the Player who just moved is in check, then the function immediately returns the character which represents which player is in check. Logically speaking, it is an illegal move, and we combine this returned character with whoever's turn it is to indicate that the Player made an illegal move, and the move is undone.

`IsStalemateVisitor` works in a similar fashion. Upon visiting a `Piece`, it returns true if the `Piece` has no valid moves to make. The visiting is run by loop inside `Game::isStalemate()`, and if all visits to Pieces of the current player's turn find that no moves can be made, and they are not in check, then there is a stalemate.

We decided not to construct a Visitor for determining checkmate. If there is a check, then we look for the King in check, construct an array of all the positions the opponent threatens, and compare whether all the King's valid moves are members of the threatened tiles. We believe that

this is sufficient logic that is better than attempting to modify the traditional Visitor design to accommodate.

After running `Game::updateGameState()`, we can use `Game::getGameState()` to extract the information we calculated.

Resilience to Change

1. Observer design pattern

By using the Observer design pattern, we are able to easily add more and different types of Observers in the future. All Observers rely on the same information that subject->getState(int i, int j) returns.

For example, we could add an Observer that has Virtual Reality (VR) or Augmented Reality(AR) support, without modifying any of the existing observers or the logical components of the program.

Or if we are not satisfied with the performance of the Xwindow graphics class, we could create another Observer that displays the game graphically.

2. Strategy design pattern

The Strategy design pattern allows us to add more difficult bots to the program. We can simply create new subclasses of Difficulty that use different algorithms to determine more "ideal" moves to make. Then, we could just modify main.cc to accept higher computer levels from the user input.

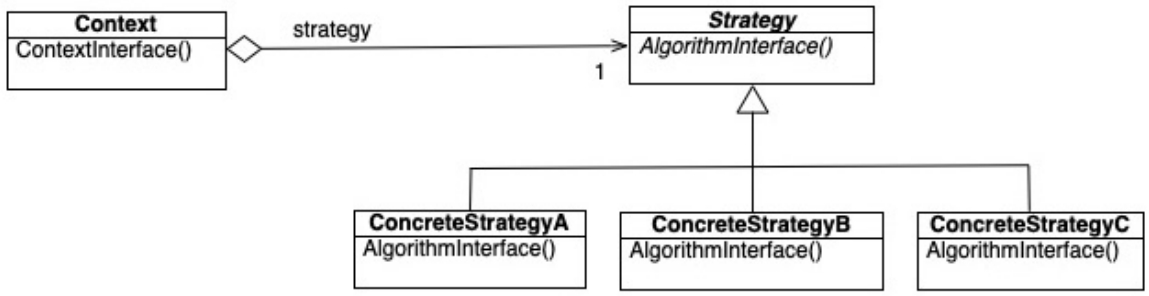

Figure 2: Strategy Design Pattern UML Diagram

In the traditional Strategy model, if we wanted to change the algorithm to use, we would also have to change the Context object.

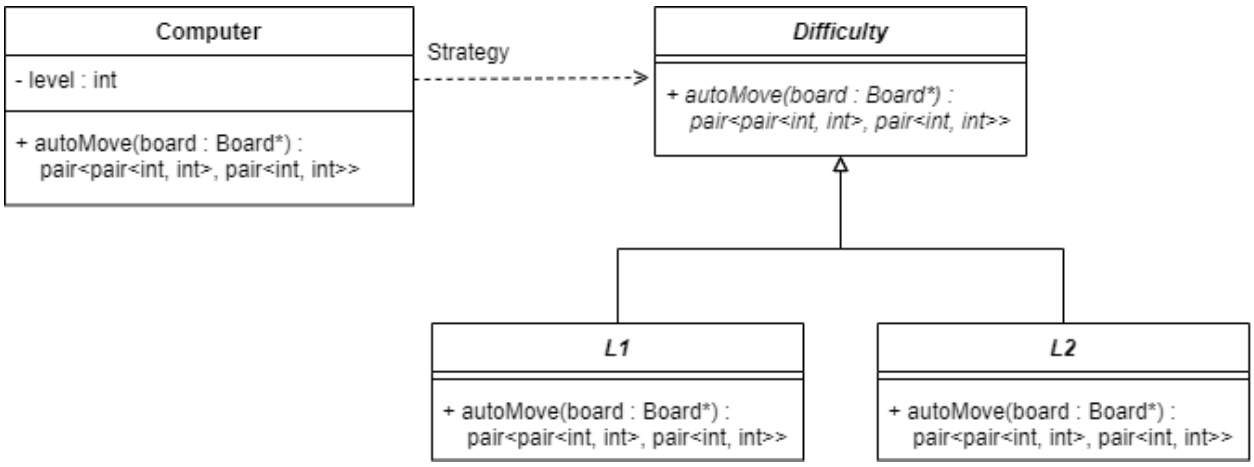

Figure 3: Modified Strategy Design Pattern UML Diagram

We modified the Strategy design pattern to have failsafe behaviour. In the event that a high-level bot cannot find a move to make, our program will generate a move from a lower level bot. We have done this by making the "Strategy" class a dependency. The Computer creates a temporary Difficulty object in autoMove() each time it wants to calculate a single move. The logic is run inside a loop, and if the current Difficulty is unable to produce a move, then we replace it with a Difficulty one level lower, and try again. Eventually, it could reach L1, which generates a random move. After a move is found, and if the level has changed, the computer will revert back to its original level.

Answers to Questions

a) 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.

- The "book" of standard openings can be an array. Each element is a queue of moves.
- When a player makes a move, all openings whose current move is not the same are removed from the "book".
- All openings whose current move do match will have their first elements popped.
- On each turn, an Observer will list out all the openings and their remaining steps.
- For example, if we have the following opening moves (O1, O2):
    - O1: W d2 d4, B d7 d6, W e2 e4,...
    - O2: W d2 d3, B d7 d5, W e2 e4,...
  - If Player White does "W d2 d4", then we will remove O2 and pop O1 to get:
    - O1: B d7 d6, W e2 e4,...
  - This would hint to Player Black to do "d7 d6", prompting Player White to follow with "e2 e4", so on and so forth.

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

- Create a Move class
  - It has a pointer to main piece that's moved

    - Have Start and End positions
    - A pointer to a captured piece, if any (default = nullptr)
    - Have a public undo() function
        - swap piece from current position to previous
- We have a vector of moves (i.e., vector<Move>)
- When the user/game calls undo then we call Move.undo() and then pops the last element of the moves vector.

c) 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.

- Make a subclass of Invalid Piece that inherits from Piece
    - Main purpose is to mark spots on the board as invalid. When the board generates the 14x14 vector, we can mark the corners (top left/right (3x3), bottom left/right (3x3)) with those Invalid Pieces).
- isInCheck, isCheckMate, isStaleMate logic all change to account for which of the 4 players, and also which king it applies to.
- When a player is in checkmate with 3-4 players remaining, their pieces get removed to the board.
- When it's 2 players, the game logic returns back to a normal game of chess.
- Pawn promotion in 4 player chess happens when:
    - A pawn reaches the back row of the OPPOSITE team, or
    - A pawn reaches the tiles marked in red highlight:

Extra Credit Features

We developed the program with RAII in mind. This meant no raw pointers used for ownership. It was problematic at first because many of the design patterns use raw pointers, such as the Observer.

One significant issue we had with using shared_ptr for Observers was that the Observer destructors attempted to call `subject->detach(???)`
We couldn't detach a shared_ptr from the list of Observers. If we attempt to pass in make_shared<TextObserver>((*this)), then we get an infinite loop and a lot of memory usage.

As such, we did not use a custom destructor method for Observers. Instead, we created a method for the Subject called empty() which empties its array of Observers manually.

Final Questions

a) What lessons did this project teach you about developing software in teams?

It's very difficult to start these large projects. Even with an initial plan and UML diagram, we found that there were a lot of changes made. Despite our efforts to minimize coupling, the functionality of a lot of code was still dependent upon other components. This discouraged us from developing components simultaneously on our own, so we could only make progress when we were all together on the same live share platform. Eventually the core logic was completed, and we were able to incorporate individual programming time and use Git for management.

We also learned that the waterfall approach to developing software is not ideal. We prefer the flexibility of agile methodology, since a project of this size can change a lot as we fix problems. However, with three developers attempting to complete the project in a short timespan, the best we can do is create a plan in advance and stick with it as best as we can.

b) What would you have done differently if you had the chance to start over?

1. The current iteration of our program relies on `Piece`'s function `getValidMoves()` for valid move verification. However, certain pieces like `King` and `Pawn` have unique restrictions to where they can move to. For example, in normal circumstances, a `King` can move to any spot that is one unit away in any direction. However, using your `King` to check the opponent `King` is an illegal move (the two `King`s are right beside each other, and as such, you put yourself in check first). Another example would be `Pawn`s: they can move one unit forward but cannot threaten any forward tiles, and cannot move diagonally forward one unit but do threaten such tiles. This caused us many problems when developing checkmate validation, since the opponent `King` thinks it can move to a tile diagonally forward one unit from a `Pawn`. As such, we had to write a lot of additional code to identify the types of moves or potential moves, in various edge cases. Another example would be a checkmated `King` that thinks it can capture an opponent piece to escape, but all such pieces are protected by other opponent pieces.
   One solution would be to implement a `getThreatenTiles()` function for each `Piece`. In essence it would be very similar to our current implementation of `getValidMoves()`. The new versions of these functions would make a distinction between a `Piece`'s tiles it threatens, and its valid moves. This way, we can remove much of the logic in checkmate/ check verification which goes into figuring out what kind of "valid move" is being considered.
2. Many of our functions rely on searching through the entire board, but because Pieces aren't aware of their own positions, we must also pass positions as parameters to many of our functions. As such, we didn't implement an Iterator for our board, because we would have to use the position anyways, and then much of our code uses two-dimensional loops that iterate from 0 to 7.
   The advantage of the Iterator is that we can perform logic on every single element the same way for different data structures. For example, if we wanted to support four-handed chess, we would have to write complicated code multiple times to support the odd format of the board.
   If we were to do it again, we would make an Iterator that returns positions that exist on the Board, rather than return Pieces.