Nguyen, Vansh, Deep

CS246 Fall 2021 Project Design

Introduction

We have chosen to create a Chess game for the CS246 Fall 2021 Project.

Overview

The overall structure of the project follows the Observer pattern, where the Board is the concrete subject, and the TextDisplay, GraphicalDisplay and FancyTextDisplay (extra-credit feature) are the concrete observers. A chess game has 6 different pieces (queen, knight, rook, bishop, king, pawn) for each color (white and black), so we have a pure virtual Piece class that creates the 6 different pieces and the Empty piece, which represents an empty square. The Piece and Board have a composition class relationship, where each Board owns 64 pieces (one for each square on the 8x8 board). Moreover, each game has 2 players, either computer (4 levels) or human. Thus, we created an abstract Player class, and the Human and LevelOne to Three inherit from it and linked them to Board, using aggregation, as the players will play the game on the chess board. The GraphicalDisplay uses Xwindow to present the board, while the TextDisplay and FancyTextDisplay print the board in the terminal.

The Board class stores a 2D array of all the Pieces, so this keeps track of the state of the location of all Pieces. Thus, after any move or setup changes this array is modified and with the help of our Observer design pattern, the concrete observers are notified and updated. The 2D array is the heart of this program, and is used to fulfill most of the tasks in a chess game, for example checkmate, stalemate, and check to name a few.

Design

The first challenge we have to face is determining how we can display the board in various ways. We chose the Observer pattern because one, we want to update the board's display every time a move is called, and two, we realized that the problem we're facing is very similar to what we faced in Assignment 4. In A4, we have a text display and a graphic display showing sections of our Studio object, which is a concrete Subject. In our project, we also have a text display and a graphic display showing the entire board, which we can treat as a concrete Subject. We knew that the board is the state of the program, and so we required this board to be displayed after any change (ex. move, setup), which makes it similar to the AsciiArt picture being the state of the program. Thus, we realized that the Board class will hold the state and be our concrete subject, and the text and graphical displays the concrete observers.

For our Board class, the second challenge is finding a way to internally store the chess pieces' information. We settled on storing a 2D array of 8 x 8 Unique Piece Pointers

(std::unique_ptr<Piece>). Our Board - Piece relationship looks very similar to the Strategy design pattern, where we can choose which algorithm to run at run-time.

The next question is which array index corresponds to which square on the board. Internally, the a1 square corresponds to [0][0], h8 corresponds to [7][7]. We convert a square on the board as follows: if it's c5, then it's [c-a][5-1] on the array, d3 is [d-a][3-1], [file - 'a'][rank - 1]. Our Board interface takes in a char and an int corresponding to a square on the chessboard and the client who calls it doesn't need to know how the board is stored internally. However, we can take this abstraction further. Instead of having to remember which index corresponds to which square every time, we made a helper function called getPointerAt which takes in a char and int, auto converts the indices and returns a reference to a unique Piece pointer. Therefore, even in the implementation file, if we want to access a piece in the array, we call getPointerAt and abstracts accessing the array instead of having to remember to convert. Now, with accessing Piece pointers solved, how do we store the array of Piece pointers?

Because we're storing an array, all the types must be the same, and yet we have to store pieces of different kinds. To account for these differences, we chose to make a Piece an abstract superclass, with the different pieces (King, Queen, Rook, Bishop, Knight, Pawn) and Empty as its subclasses by inheritance. Each piece moves and behaves in a different way, for example, Queen can travel horizontally, vertically and diagonally, while Knight can travel in an L-shape. Therefore, Piece must have some pure virtual functions involving valid piece travel, and thus, we ended up with two pure virtual functions: isValidMove and valid_directional_moves.

These two functions are immensely helpful to us because when we start implementing the methods in the Board class, especially move, possibleMoveExists, and inCheck, instead of having to do if-else statements on each piece type, for the most part, we can use polymorphism to our advantage and simply call the Pieces' class virtual methods instead, with the same interface, and we have lots of examples and situations where these two functions help us a lot.

For example, in valid_move, we pass in parameters to a Piece object's isValidMove which automatically checks if the piece can start at a square and travel to the destination square, with no pieces in between and in the allowed directions. However, this assumes we're not putting our own king in check, which is the responsibility of valid_move itself, without having to know what type the piece is. If we know that move is invalid, we just return false. If true, we can move the piece to that square and call isCheck(), which checks if our own king is in check.

Another example of using polymorphism to our advantage is isPieceSafe, which is a helper function to other functions, including inCheck(). Our implementation of inCheck is as follows: all we have to do is find where our king is, and we traverse every square in the board to see if there exists a location on the board where if you move from that location to the king, that move is valid by calling the virtual function isValidMove. If that move is valid, the king is in danger – there is a piece that can "capture" it. Again, no if-else statements necessary, just the polymorphic nature of isValidMove. Because of that, it also takes into account discovered checks and double checks.

Similarly, in possibleMoveExists, we can call Pieces' virtual function valid_direction_moves to give us all squares that a piece can travel to without having to know exactly what piece it is. This significantly increased efficiency, since instead of checking if the piece can move to each square on the board (64 iterations of valid_move method), we now only need to call them for a few squares where the piece is only allowed to move. For example, if we want to check if there exists a move for a pawn, we first looped through all 64 squares even if the pawn cannot move, like from e2 to h8. However, our valid_direcition_moves, now returns a vector of all valid moves by a pawn, which are simply moving one up, two up, or one in each diagonal, and we loop through this vector of moves to determine if any move is valid on the board. This means we only have a maximum of only 4 iterations for a pawn from 64. This was implemented for each piece, making our possibleMoveExists method efficient.

The third challenge, then, is determining how we deal with human and computer players. In our program, when we type the word move, a human player is expected to type in input from the command line, whereas a computer does not. To save us some grief on if-else statements, and to write one single line of code, we decided to use inheritance. We have a Player abstract class, to which Human, LevelOne, LevelTwo, and LevelThree all inherit from. All Players share the same interface and a single method: make_move. Human's make_move reads from input, while the computer classes' make_move doesn't. In either case, they both return a Move object to be passed into the Board class's move method. In main, we simply write: players[?]->make_move() without any if-else statements. We have to deal with all these challenges while having to aim for high cohesion and low coupling as much as we can.

We designed our program so it follows the OOP principles - high cohesion and low coupling. Our code demonstrated high cohesion in the Board class. For example, the inCheck method is used in many functions such as valid_move, move, endSetupMode, and isGameOver. valid_move is used in functions such as move and possibleMoveExists. possibleMoveExists, in turn, helps create many types of moves, including a set of all moves in allPossibleMoves. Nguyen's personal favourite is getPieceAt and getPointerAt, which is a function that abstracts accessing the array and pretty much every single function uses it. This last example is also a good example of low coupling.

One example of low coupling in our code is the relationship between Piece classes and Board. The Piece classes need to access the Board to know about the pieces of the board. However, we abstracted accessing Pieces in the array with the function getPieceAt which takes pairs of chars and ints instead of two ints. This is because we don't want anyone to know the internal arrangement of the Pieces in the Board. To access the Board, all the client (or whoever is seeing Board) needs to know is how a chessboard looks and how chessboard indexes work and pass in the appropriate char and int. However, it is not the only example of low coupling.

Another example of low coupling in our code is the relationship between the Board and the Player classes. The player classes create Move objects and pass that Move class to the overloaded move function in Board. Board doesn't need to know how a Move object is created. It just takes it in and handles it accordingly. The player classes communicate with Board via

Move objects without knowing anything about each other. The less the classes know about the implementation of each other, the better.

Changes to the UML

First, the Board class has many, many more methods in the final version than what we originally planned in the beginning. There are a lot of methods that stayed on the final version, such as getPieceAt, inCheck, inCheckmate and inStalemate. However, as the project went on, we created many more useful methods to suit our needs. For example, we added a lot of methods related to setting up the board and methods related to making a list of various types of moves for the computers to use. We also added two methods defaultSetup() and emptyBoard() because we do not want to manually clear the board or set up the board in the default position. Not only do we add new functions and remove unnecessary ones to Board, we also need to add more methods to the Piece classes to help Board.

Second, early on, we determined that we need the Pieces to be able to identify itself by getType() and be able to determine if that piece can travel from a given square to another, polymorphically by isValidMove(). As the project went on, we also added two more pure virtual functions: make_copy and valid_direction_moves. Because in valid_move, setPieceAt, and copy operators, the Pieces need to be copied, but unique_ptrs can't copy themselves, so make_copy polymorphically makes a new copy of itself, with the same type and colour. valid_direction-_moves is created from the need to increase our optimization of finding a player's all possible moves on a given chessboard configuration and to avoid writing if-statements over and over.

And lastly, there are a few minor changes in the Player classes. Instead of having no attributes, it takes a single attribute: a Board object. Instead of make_move taking in a Board as a parameter, it takes in no parameter. We did this because instead of having to pass in a Board reference or pointer every time, it has the choice to use the Board contained within. For the Human object, it does not need it. We also removed the Computer superclass which the Levels One-Four inherit because we no longer see the point of it.

Despite all these changes, there are things in the UML that stayed the same. Almost all of the relationships between the classes are the same. We kept the Observer design pattern between the Board and the Observers. We kept a slightly modified Strategy design pattern between the Board and the Pieces. The Board still has 64 pieces and the Piece classes have a dependency on Board. The Board class and the Player classes still communicate via the Move class as a dependency (although Piece classes also have a dependency with Move via valid_directional_moves)

Although the relationship between classes stayed mostly the same, the classes themselves have changed quite a bit to suit our ever-changing needs.

Resilience to Change

We have made our project in such a way that adding additional features will not require much change to code or the overall design pattern. For example, if we were to add a new piece, we can simply create a new class which will be the subclass of Piece. Since the move method uses the isValidMove method in Piece class, the board class will not require any changes. Moreover, the computer levels will not require any changes as they use vectors generated by methods of the board class that class isValidMove method in the Piece class.

Our code can also handle changes to the main file. The whole main file can be implemented in a different manner given the user creates all the required classes in the proper order. All the fields in our classes are private in order to prevent the user from changing the values. We have provided required accessor and mutator methods such that the user can change the implementation of the main.

Our program is not however perfect, and there could be better more complex design patterns used to handle some other scenarios where we need to update our program. For example, if we want to add another complex move like en passant/castle this would require changes in multiple files. First we would need to change the isValidMove method for the piece that the move is being performed on. Then if the move can only be performed by a certain condition, for example in en passant the move can only be performed after the opposing pawn moved by 2 squares, we would need to add state variables (fields) inside our piece class and change the implementation of our overloaded move methods to handle this new move by first checking if the move is valid (isValidMove) and the condition is met.

Overall, due to low coupling in our program, changes to a single class will not require much changes in other classes. As aforementioned, if we add a new piece, no changes are required in the Board class and the Player classes. Similarly, if a new level of computer is added, no changes will be required in the board class. However, if we make changes to our Board class, the other classes will be affected and will require changes.

Answers to Questions

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. Although you are not required to support this, discuss how you would implement a book of standard openings if required.

After every turn, we create a vector of all the possible moves. To implement a book of standard openings, we can compare the vector of standard openings to the vector of possible

moves. The common moves can be stored in a new vector, using which we will decide the next move. A counter will also be implemented that will count the number of moves. After a certain number of moves(20) the computer algorithm will stop using the book of standard openings, and will use its algorithm.

Another approach that we can use is to store a tree of all possible opening moves, because a player's recommended opening moves change depending on what the opponent does. In our tree, each node, representing a player's move, has two to three children nodes which represent the other player's possible responding moves. For example, if White plays e4, that is stored in a node. Black can respond by playing e5 or c5, which are then children nodes of the White e4 node. In turn, White can respond in different ways depending on what Black plays and branch off our tree like that.

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 undo a player's last move, we can create a field that stores the player's last Move(a class in our UML), including the Piece it captures(Piece will be empty if there was no piece in the final position). This field will be updated after a new move is performed. There will be an undo method in the board class, which will get the previous move from the field and reset the board accordingly.

To undo an unlimited amount of moves, we can use a vector (stack) of all moves inside the board object. If the player wishes to undo, we pop the last move out of the stack and resets the board based on that. The undo method must be connected to the Board class.

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.

Firstly, we will have to change the size of the Piece 2D array in our Board Class to handle 4 players. So now our board will be 14x14 from 8x8. Also, the corners squares of size 3x3 are cut out in the 4-player chess board, so we will add a new Piece called "Invalid", and the Piece 2D array inside Board will have Invalid Pieces initialized at the 4 corner squares, so pieces cannot move there. We have to add new Piece colors as well in order to handle the 2 new players. Next, we have to slightly change the implementation of the Pawn class, because the direction they travel does depend on the color of the piece. For example, the right player can only move its pawns to the left, while the left player can only move it right. The top and bottom players will not need to be changed since that is our current white and black player implementation. The rest of the Piece subclasses will be fine because the directions they travel should be independent of board orientation. Then, we slightly change the implementation of the TextDisplay and GraphicalDisplay. We change the number of rows and columns iterated and we have to deal with

the corners cut out. Furthermore, we will have to modify main.cc to allow 4 players to be inputted, which is simply changing the size of the current players array used to store 2 players to 4, and loop through 4 times to store all 4 players.

Due to our design, our program can handle change effectively. Other than the changes described above, which are very minimal, we would not need to change the implementation of other complicated methods. For example, checking for invalid moves will be the same for all pieces except pawn, determining if there is checkmate, check, or stalemate, moving pieces and setup mode to name a few.

Extra Credit Features

Smart pointers

Inside the Board class, we are storing a 2D 8x8 array of unique Piece pointers (std::unique_ptr<Piece>), and it has many advantages and disadvantages.

One major advantage of unique pointers is that we do not have to worry about explicitly managing memory using new and delete because unique pointers handle them for me. One less source of headaches gone. However, one disadvantage is that unique pointers cannot be copied. In a lot of Board's methods, we can't simply copy unique Piece pointers, so to combat this, we use a lot of strategies. For example,

1. We can use move semantics (std::move), for example, in the move method.
2. We use std::swap to swap pieces around, like in valid_move or isPieceSafe.
3. Piece class' pure virtual function make_copy, which creates an std::unique_ptr pointing to an identical version of the piece that calls it instead of if-else statements.

Also, in getPointerAt, a helper function which abstracts accessing a Piece in the 2D array, we have to return a reference to a unique Piece pointer because unique pointers cannot be copied.

Fancy Text display

Our text display was not looking very pretty and seeing the letter as pieces was pretty confusing. We knew that the terminal can print special characters and C++ has support for unicode and special characters. Therefore, we added a new display called FancyTextDisplay.

This display is similar to the text display. However, instead of alphabets we use unicode for chess pieces. For example, to print white king, we printed '265A'. This made our display look much prettier. Along with using unicode, we also added more spacing as earlier it was looking very crammed.

Pieces in graphical display

It's not that easy to design a pretty piece to display on the XWindow graphical display. However, designing a piece pixel by pixel makes rendering it on the graphical display very slow.

Therefore, we made the design of the pieces simple, made with around 5 simple rectangles. This is so it can be rendered quickly on our end, and yet still has the essential shape that we can instantly recognize and differentiate. Initially, we made the squares 50 pixels and passed in numbers to determine the size of the rectangles making up the piece. However, it isn't a good design, so I eventually passed in ratios in relation to the square side length so we can easily scale it.

Another challenge for the Graphical display is how to speed up displaying pieces. The idea is that rather than display every single square every time notifyObservers is called, it only displays the changes to the board. However, because the abstract subject's notifyObservers, as well as the abstract observer's notify take no parameters, we can't quite push the changes via parameters. Instead, the Board object has a method called getChangedBoxes which returns a vector of char-int pairs corresponding to the changes made on the Board object. When observers are notified, especially GraphicalDisplay, it calls getChangedBoxes to get the changes and loop through those changes instead, thus increasing the efficiency and speed of the Graphical display.

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?

   Nguyen:
   - Good, clear communication is essential because if we want to do something to our code, it's a good idea to justify why we're doing it the way we do and everyone in the team has a good idea of what to do.
   - Working for hours at a time is ridiculously tiring, especially when the project gets huge. Take frequent breaks. Manage your time wisely. Watch out for long coding sessions.
   - Having other people around coding with you makes you feel much less lonely, compared to coding all alone.
   - Bugs and weird, bad stuff can happen when developing software, especially in teams. Git and gdb (or a debugger) are immensely useful. Praise them.

   Vansh:
   - Working in a team increased efficiency. We divided our work and each one of us focused on the part that we knew the best.
   - It increased creativity and increased our knowledge. While deciding on the design and implementations of various features, each one of us contributed to the discussion, resulting in creating more effective solutions.

   Deep
   - Increased code quality. Since every programmer has different coding styles, and no one is perfect, we were able to learn new and better coding techniques from

each other. Also, since at times we were not working on the same feature it was important to comment code thoroughly so that we can understand each others code without having to explain it.
- Fewer mistakes and bugs. In our assignments we had to test our code individually and it is very likely that we missed certain edge cases since we coded it ourselves and tested it ourselves. However, in a group we tested each other's code and were able to fix many bugs in the process.

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

    - We completed our project in time but we did not have time to implement the bonus features. Had we started a little early, we could have implemented the additional features as well as improved on our code more.
    - Worked on our UML more. Before starting the project, we did have an idea of how to go about it. However, we did not think of how some parts might work together. For example, we knew that we had to create a virtual player class for the human and computer players and so we included that in our initial UML, but when it came to coding those classes and making them work together with Board and main.cc required a lot of thinking. This resulted in a lot of wasted time to rearrange the code to make the Player class work with the others. This could have been easily avoided if we planned it well before starting to code.

Conclusion

All in all, we were able to successfully create a working chess game. In fact, our program satisfies all the requirements in the project. On top of that we added 2 cool bonus features - fancy unicode display and Pieces inside our graphics display, and we used only smart pointers for the entire program (so no new or delete needed). This project taught us a lot about the importance of following good OOP principles (high cohesion and low coupling), and using design patterns. We already have a plan for implementing level 4 for the computer, however given our time crunch due to exams, we were not able to submit this part on time for marks. We plan on working on this and adding more features like level 4 and maybe even 4-player chess. Lastly, this project taught us the skills needed to effectively work in groups. Clear communication, and time management is crucial, and also we realized the importance of learning Git since it could save lots of time and make coding in groups much easier. In conclusion, although it seemed at first a very difficult project, we were able to enjoy the coding process and put together a working chess game.