Chess - A C++ Implementation

By: Roman, Alex, Anshul

Overview

Our game is composed of 7 components, the Game, Setup, Board, Players, Pieces and observers, with a few helper classes & structs that are used by the above classes.

Game

A game is just an object which has and owns 2 players and a board. The types of players are passed from input, and the board is initialized to a standard chess board, or a board from setup (explained below).

After initializing the board, 2 observers (text and graphical) are attached to the board so that the user can view the game state as it changes after each move. Game keeps track of positions that change on the board so that only new positions need to be notified to observers.

The game has a main game loop which accepts moves from a player. The game then checks if those are valid moves based on the rules of chess. Game includes the implementation for the major control flow functions of the game such as check, checkmate, stalemate, and resign. Other than check(), all 3 of these functions have the power to terminate the game, while check is a function that returns true when the given King is currently in check. This will affect how players are allowed to move.

Setup

Setup is an object very similar to a game. It has a board, but no players. It initializes the board to be empty and provides functionality that lets the user update the board by adding/removing pieces. Then, when the setup mode is done, the setup object creates a new game, passing in the board which will start a new game on that board, after which the setup object is destroyed.

Board & Square

Board is an aggregate class of Game and Square is an aggregate class of Board. Board is essentially the game stored in memory. It's hidden from the user, all they can see is what is noticed to the observer. The Board is able to initialize a chess board or to a setup board as an 8x8 2D array of squares, with both sides of the board initialized with opposite colour pieces in the correct positions. It then sets the squares to be empty or to contain a piece. Board also provides functionality for changing the actual positions of pieces in the array of squares when a move is executed in game. Squares have a piece and a color. They are the middlemen between the board

and the pieces on the board. The board always accesses pieces and their symbols through a square. Not directly to the piece. This abstraction helps hide the details of pieces.

Pos & Move

Position holds a row and column which abstracts away the coordinates of the board array, making movement functions easier. Move is a structure composed of two Positions, representing the startPos and endPos coordinates of a move.

Piece

Piece is an abstract class that defines the common data and functionality of its 6 subclasses Pawn, Knight, Bishop, Rook, Queen and King. Every piece holds a character which represents what kind of piece it is, and a player # (either p1 or p2) which says which player it belongs to. Each of the subclasses don't have additional data, only a different constructor which initializes the symbol to a specific value and a canMove() function. This function determines if a piece (based on its type) can physically make a move that is passed to it.

Player (Human || Computer)

Player is an abstract class. A player essentially just has a player #. The main purpose of Player is to represent the two players involved in each game of chess. There are two subclasses of a Player; Human and Computer. Humans and computers don't hold any additional data, but they have different functionality. Note that the computer itself is also an abstract class. A players only job is to generate moves. The player and computer both have an overridden generateMove() function. For a human player, a prompt is given to output and the generated move comes from their input. For a Computer, the generated move occurs automatically. For this reason, it can be considered "AI". Additionally, when a computer player is chosen, it must be set with a difficulty level of 1-3, and the Computer class provides several functions that build upon each other to make the computer player better moves the higher level it is.

Computer Levels

Each computer level has the functionality of all levels below it, with additional features the higher you increase the level.

Level 1: The base level simply chooses a random piece on the currPlayer's team, and plays a random valid move. The only requirement is that the move is valid, however there is no preference on which piece types are preferred to be moved first.

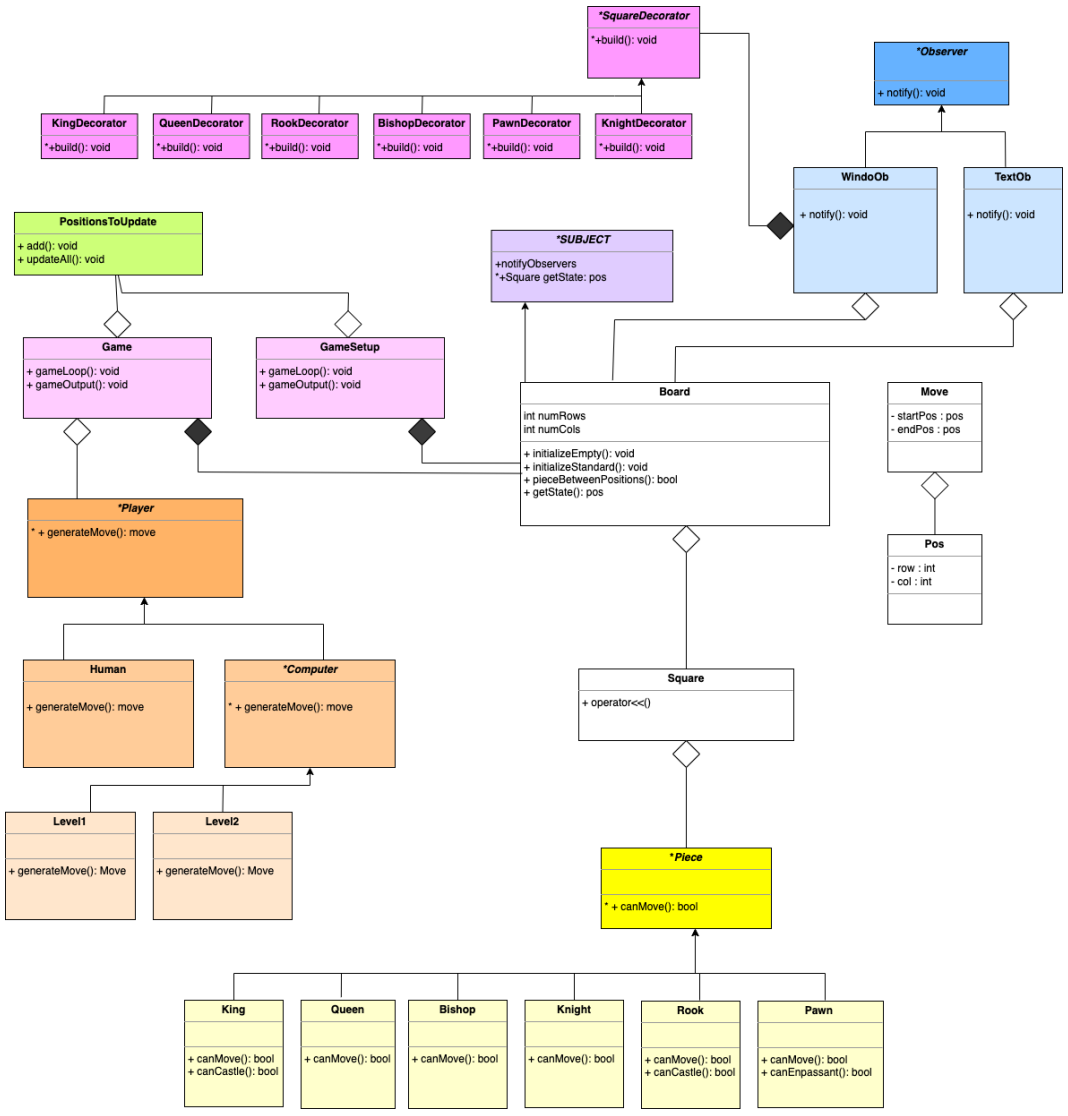
Observers

When a Game, or Setup is created, observers are attached to the board with which the game and setup concerns. Observer is an abstract class with two subclasses: text observer and window

observer. As the board is changed throughout the game, the game notifies the observers based on only the positions that were updated. For the window observer, this shows up on a graphical display. For the text observer, a text-based version of the board is outputted. Board is actually a subclass of the abstract class Subject. Subject defines the ability to attach, detach and notify observers. Also note that a window observer contains decorators. These decorators are pixelated pieces. The decorator class is an abstract class with a subclass for every type of piece. Then we call build() polymorphically on all pieces on the board, which instantiates the corresponding piece decorator.

Updated UML

The uml shown below outlines the structure described above.



Design

Observer Pattern

As the board is updated by players, how should these players be notified of the state of the game? In order to abstract away the details of output of a game from the game itself and to easily add multiple types of outputs for the same type of game, we utilized the observer pattern. The observer pattern lets you separate the data, in this case the board, from the display of the data. The board itself will always be in memory during a game/setup. To display the game visually, all the game needs to do is attach observers to the board. If the game wants them to go away, it detaches them from the board. Now, output and the game are two almost entirely separate things. To do this, the board is made a subclass of subject, meaning the board is the subject of the game. The abstract subject superclass simply defines two functions, detach and attach and notify-observers. Now the board can add observers by attaching, remove them by detaching and notify all of them at once. Note that these functions are called by game since Game owns a board; without a game there is no board. An abstract superclass called Observer simply has the virtual function notify. The subclasses of observer are text-observer and window-observer. These two concrete classes have separate implementations of notify(). For a text-observer, notify outputs a text-based board to console output. For the graphical display, it's more complicated. A window-observer has an X Window object. Upon attaching this observer, a new graphical window is opened. It displays the board in color and each of the pixelated-pieces on their respective positions. How it displays pieces is explained next. In order to minimize the amount of rendering done by the X Window object, the window observer holds an object called PositionsToUpdate(). Positions to update are visitors to the game. It is essentially just a wrapper class for a vector which stores positions that were modified by the game. The window observer has a reference to this object, and only renders the positions on the board which were recently updated on the last move. These positions are then removed by game and the process repeats.

Decorator

The window observer also holds a vector of decorator pointers. The decorator class is an abstract superclass with a subclass called piece-decorator, where a piece is any of the seven types of pieces. Each decorator has a build() function which builds an image of the piece using the X Window display functions. When the window observer is notified, decorators are added to the window display at the corresponding pos based on the type of piece. This is another use of polymorphism which helps reduce adding decorators to a single function call, which calls the underlying decorator build() based on the piece.

Polymorphism

A Piece is a polymorphic object. It's an abstract superclass with 6 different subclasses. The virtual canMove() method in Piece is implemented in each subclass so that we can call canMove() polymorphically on the underlying pieces held by the board. It outputs a bool representing whether a certain movement path is possible. Each overridden canMove() has its own specific implementation based on the piece. So as a result, all we had to do was pass in a move to a piece on a square, without having to check what kind of piece it was.

A player is also an abstract type. We call generateMove() polymorphically based on the underlying type.

Finally, the square decorators are polymorphic objects. All we need to do is call build() based on the underlying piece held at a position to output that piece as an image.

Resilience to Change

Our program employs a fully object oriented design with all specific functionalities split into different classes and objects, while also employing design patterns to limit repeating code, making our program more maintainable and reusable.

We used helper functions, getters and setters with descriptive names. We also abstracted away details such as with a position instead of two integers. In this way, it's easy to implement our logic regardless of syntax.

In terms of how we would minimize modifications and recompilation, we tried to minimize coupling and maximize cohesion as explained below. In general, our goal was to make it so that modules are split up based on function. Modules should only interact by passing values through functions between each other. There were exceptions, but this was our goal. By doing it this way, if a new feature needs to be added, it will most more likely than not be able to interact with all of the other functions as a separate object as well. If it needs a reference to another object or to own another object, you can pass that object value as reference.

Maximizing Cohesion

Our first primary goal was to maximize cohesion. We wanted each module/class to have a specific purpose. In that way, when we are coding it's easy to figure out what object you have to reference or use when you need something done. If you need a specific task done, then you can include the file that does that task. This would also make the classes simpler and easier to read through. The less they have to do, the less data you need to store. This keeps constructors relatively simple to no more than 2 parameters.

We split up the program into several classes all working on separate overall functionalities. For example the Game only cares about the state of the game: updates board, and outputs different results like win or stalemate. By designing our program this way, any changes to the game rules could be easily implemented into our game without any modifications to the other classes. We would simply just add a function in the game class that employs the new rule change.

Another example is the Board class, all the Board class does is change the squares in an array. We created a separate class for updating changes on the board. Why should the board care how it changed? The other is a visitor that simply looks at what positions are different every move. All the square classes does it change the piece it holds. All a piece does is say if it can make a move based on its type.

As one can see, it's very easy to pick up on what each file does simply from the name of the file.

Minimizing Coupling

Our second primary goal was to minimize coupling. By reducing the dependency of the 7 main modules talked about earlier, this gave us more freedom to make changes without having to redo other parts of the project. We looked at each module/class as a client: each client needs something from someone else, and other clients need something from it. So instead of working together to produce a game, they pass things to each other (in the form of a reference or a function parameter). Each module shouldn't care how it gets what it needs, only that it gets the data, can do its function and produce some type of output which can be passed on to another module.

Firstly, looking at the game class, it's job is mainly to accept moves, change the board, and notify the observers. To accept moves, the class already has 2 players. It can call generateMove() on whatever player's turn it is currently in. In one line of code, we get a move to make on that board. The game doesn't care how it got there, only that it has a move to make on the board.

To make a move on the board, Game calls board->move() with the move that was passed in (once it checks that it's valid). The Board does not care how the move to be made got there. Only that it has a way to change the board now. Again, using one function call, passing in a move, the board get's what it needs from game().

Finally, all the Game has to do is update positionsToNotify() and then call notifyObservers() to give the observers what it needs to output the game.

We also made sure to make almost all data (with only 1 or 2 exceptions) private for every class to prevent large dependencies of fields from class to class. We used getters and setters (only a few

setters were used between classes) instead. In this way, we can control how another module is setting data. For example, a square can set its own piece. A board (which holds squares) can only set the piece through a square object. A game can only get/set a piece through the board object. In this way, the modules only interact with their nearest "companion". The game should not have to access a piece directly for example.

Answers to Chess Specific 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.

We could make an additional class called intialMoves containing potential moves (start and end position). This class could contain a simple up until a certain number of moves/nodes.

Then we could call getMove() on an initalMoves object which would take in a player's current position (which the Game class can easily provide) and it would return a corresponding position from the tree that the player should move to.

The move supplied to the board would be different depending on if the player is a human or a computer. For a computer, simply use the getMove() for the first given number of moves and then switch back to getting moves from generateMove() (the overridden function in Computer).

For regular players, their getMove() could be outputted as a suggestion to the user such as "move e2 to e4."

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

Moves are supplied to the board from input as an initial position and an end position. Our game currently has a canMove() function that determines if the move can be made on the board. After which the piece in the position is moved to the second position. From here, a few things could happen:

1. The opponent's piece in position 2 is taken and removed from the board
2. The piece moves to a new square and does not take anything.
3. A pawn is promoted to a different piece
4. Castle

One option would be to keep a copy of the board and reinitialize the board to this older version. However, we could do it more efficiently. In order to undo the last move, we would need to hold

on to the previous move, and then re-supply it to the board in the opposite direction. The start and end position would flip, moving the piece back to its original position. But we also have to keep track of the other pieces on the board that were affected (ie: was a piece taken, was a castle performed?) The Game class (which holds onto the board) could have a vector which holds onto a set of actions that were made on the last move (ie: move to empty position, piece removed, pawn promotion). Then when the player selects undo, these actions are each popped from the stack and applied to the board in the reverse (from end pos to start pos).

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.

Other than changing the number of players and the board layout, the implementation would be relatively similar to what we already did.

Our current game is built with 2 players. We would have to add 2 new Player types. Then we would also have to change the board. We could keep it as a square and then make the corners an invalid move position for the player. That way, we can keep the board as a 2D array. Instead of switching between p1 and p2, the Game class would loop through turns 1-4 for each player. The legal moves and everything else would be the same. A legal move on a 4-way board is the same. So the actual logic of checking for valid moves would be the same, not including the check that they're not moving to the forbidden corners we defined. So the main changes would be to the Game class which would be relatively simple. The underlying board and piece logic would be the same.

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?

Overall, developing software in teams is a completely different ballgame than developing individually. Especially on large scale projects with a lot of classes, such as this one.

Organization
One of the key lessons this project taught us was that organization is one of the most important things that must be maintained. Organization of code, with useful documentation should be maintained in every file to allow the other group members to understand the code if they need to make future changes. We also learned we must keep all of our files and copies of the project organized, by giving each project version meaningful names, since there were a few times where we began editing the wrong version of the project, causing a loss of work.

Planning of work

Another key paradigm of work we had to maintain was good planning and distribution of work. Because there are so many moving components to this project we realized each person should try to implement their code using as little code as possible from the other members to prevent coupling and to make debugging easier at the end, since there will be limited reliance on other functions.

Making Edits

In terms of actually working on the project, one thing we learned was to never edit the same file as another member while they worked on it. We ended up creating a github repository with the project, and quickly found out that merge conflicts are very easy to get, if two people work on the same file simultaneously.

Adapting to Style

The biggest lesson we learned was to be patient and always communicate when any issue arises. One of the biggest challenges was adapting to each other's style of coding, with different syntax as well as different ways to approach problems. As a result we quickly learned, communicating with the group early is the best way to solve any confusion about a certain method of implementation of a function for example.

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

If we had the chance to start over, we would have used a different approach. We decided to each develop different aspects of the game and then merge them together. However, a couple weeks after we initially met, we lost sight of our initial plan. As a result, we each coded classes with different levels of privacy coupling and cohesion. This made it very difficult to merge them together and we did not understand each other's code. It would have been better to start working together from the beginning, developing the same main structure of the program, and then branching off to develop functions separately.

Additionally, we would have made sure to do rigorous testing from day 1 instead of coding in large batches and testing everything at once. Overall we think this was the most valuable project we've each ever done. It taught us a lot and we hope to learn from our mistakes.