DD2: Final Report

Ansh, Shawn, Karm

Overview

Our chess project is divided into four main components, the board, the pieces, the computer, and the observer.

Board

Chess is played on a board of 64 squares, in an 8 by 8 square arrangement. We implemented a class to represent this board as an array of arrays of pieces. We also have a lot of other fields for the sake of keeping track of things, like the colour of whose turn it is, the black and white players themselves, and the position of both kings at any given moment. There are also a few public functions to give the client access to the board class, with getters and setters to update fields, parse functions to get all the possible moves within a specific game state context, and private functions that actually take care of implementing parsers for each specific piece.

Pieces

In its default configuration, each side has two rooks, two knights, two bishops, a queen, a king, and 8 pawns. In terms of their actual implementation, all the pieces have a common template, and they differ in the moves they can make, as well as their intrinsic values (queens are worth 9 versus pawns which are worth 1). To represent the pieces, we decided to create an abstract Piece class, from which all the concrete pieces would inherit from. Each piece shares a protected name, value, colour and possibleMoves field, but they each have their own clone() implementation and own calculateAllPossibleMoves implementation (which returns all the possible moves that can be made given a current position). Pieces like pawn, rook and king have an extra inStartingPosition boolean field to facilitate moves like castling and en passant.

Computer

Since we need to have support for computer vs computer, computer vs human and human vs human, we decided to implement an abstractPlayer class, from which a human and all the computer levels would inherit. Each computer/human class shares a protected playerColour field, a pointer to the board that it is tracking, and a boolean to check whether the player is a computer or not. Human and computer classes differ in the implementation of the calculateNextMove function, where it's a no-op for human players (as it is the user is the one inputting the moves) and an actual operation for each computer, differing in the complexity of the calculations.

Observer

Even though it may have not been necessary (as we only have two observers at any given time), we decided to implement the observer design pattern. We have two concrete observers that inherit from the common abstract observer, one for rendering the text version of the board upon setup and actual gameplay, and one for rendering the graphical version using XLib, which also displays the board during setup and actual gameplay.

We finally have our driver, which imports all the classes above, and abstracts the running of the setup (should the user wants to not use the default setup), the actual game, and the cleanup (deleting of all heap-allocated memory). Our driver also wraps the game results in a struct, which has a function to pretty print the statistics of the games playing through the lifetime of the program, which it prints at the end.

Updated UML

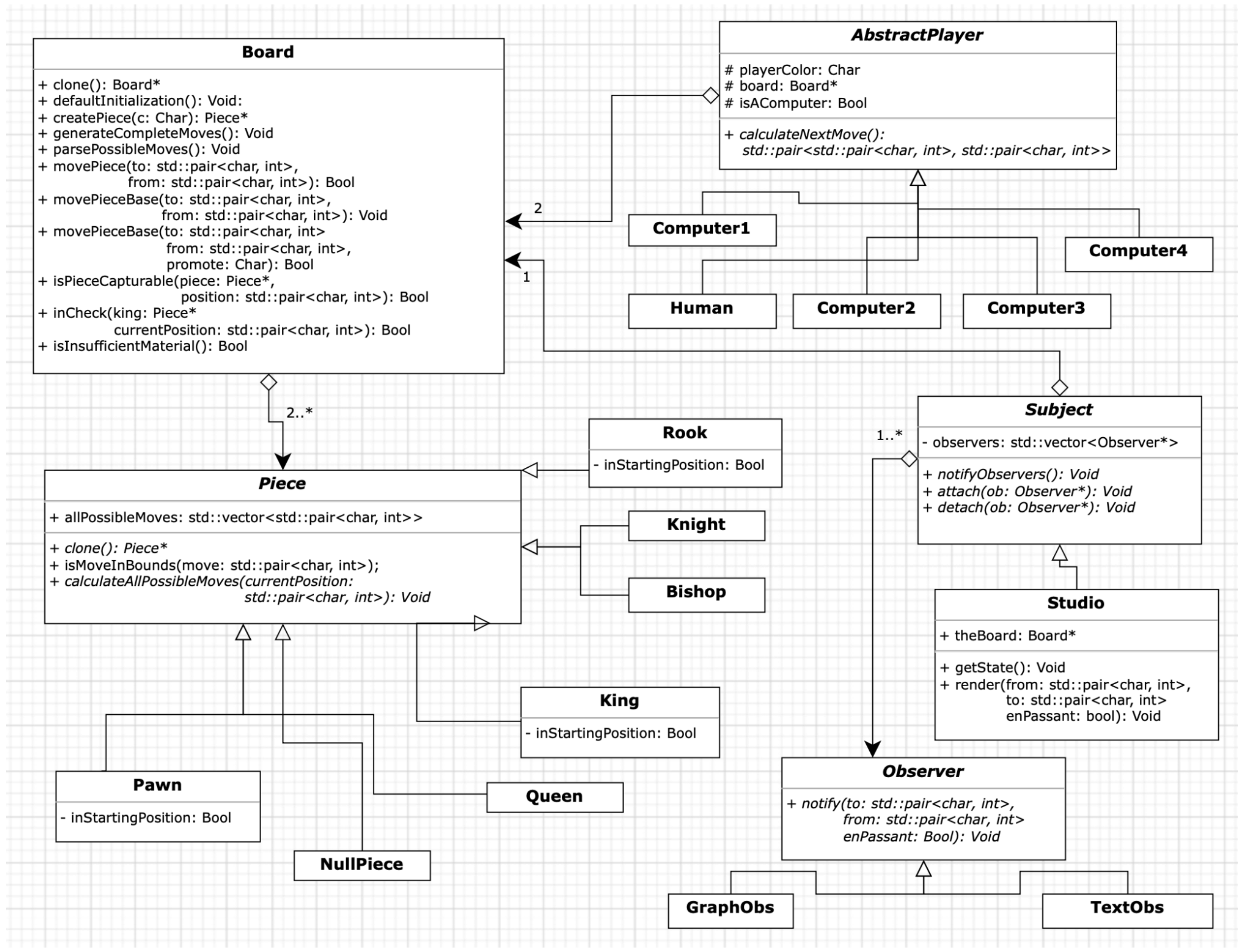
Board
+ clone(): Board*
+ defaultInitialization(): Void:
+ createPiece(c: Char): Piece*
+ generateCompleteMoves(): Void
+ parsePossibleMoves(): Void
+ movePiece(to: std::pair<char, int>, from: std::pair<char, int>): Bool
+ movePieceBase(to: std::pair<char, int>, from: std::pair<char, int>): Void
+ movePieceBase(to: std::pair<char, int> from: std::pair<char, int>, promote: Char): Bool
+ isPieceCapturable(piece: Piece*, position: std::pair<char, int>): Bool
+ inCheck(king: Piece* currentPosition: std::pair<char, int>): Bool
+ isInsufficientMaterial(): Bool
AbstractPlayer
# playerColor: Char
# board: Board*
# isAComputer: Bool
+ calculateNextMove(): std::pair<std::pair<char, int>, std::pair<char, int>>
2
1
Computer1
Computer4
Human
Computer2
Computer3
2..*
1..*
Rook
- inStartingPosition: Bool
Piece
+ allPossibleMoves: std::vector<std::pair<char, int>>
+ clone(): Piece*
+ isMoveInBounds(move: std::pair<char, int>);
+ calculateAllPossibleMoves(currentPosition: std::pair<char, int>): Void
Knight
Bishop
Subject
- observers: std::vector<Observer*>
+ notifyObservers(): Void
+ attach(ob: Observer*): Void
+ detach(ob: Observer*): Void
Studio
+ theBoard: Board*
+ getState(): Void
+ render(from: std::pair<char, int>, to: std::pair<char, int> enPassant: bool): Void
King
- inStartingPosition: Bool
Pawn
- inStartingPosition: Bool
Queen
NullPiece
Observer
+ notify(to: std::pair<char, int>, from: std::pair<char, int> enPassant: Bool): Void
GraphObs
TextObs

Figure 1: *Updated UML (pdf submitted on Marmoset)*

Here is the final UML for the current implementation of our project. The bigger version can be seen, as it has been submitted on Marmoset. When compared to the UML we had before we started coding the project, we were surprised ourselves to see how similar both UMLs were. Apart from a few helper functions and fields here and there (usually to accommodate for different pieces and different kinds of moves that we decided were best to be put in other functions), the overall public interface to our classes was pretty similar, and we think we did a good job planning out the different aspects of our classes in the beginning, as well as sticking to that plan.

Design

While planning out our program, we understood the need for a solid object-oriented design. The most fundamental aspect of such a design involves minimizing coupling while maximizing cohesion.

The Chess game itself has three primary classes (Board, AbstractPlayer, and Piece), which are minimally interdependent. In addition, both the AbstractPlayer and Piece classes are abstract, which means that they cannot be instantiated. Instead, we have created subclasses that inherit from these abstract classes in order to implement specific functionality. The King, Queen, Bishop, Knight, Rook, and Pawn classes all inherit from the abstract Piece class. The Piece class contains virtual methods (such as getAllPossibleMoves) which the subclasses override. This ensures that different subclasses respond differently to the same base method call (i.e. polymorphism). Similarly, the Human, Computer1, Computer2, Computer3, and Computer4 classes inherit from the AbstractPlayer class. On the other hand, the Board class is a concrete class (with no subclasses) that can be instantiated.

When the primary classes (and subclasses) are utilized in a single program, we have a Chess game with a board, pieces, and two players. In addition, these three primary classes can be modified separately with little to no effect on the others, which makes it extremely simple to change the rules of the game. For example, if we change the getAllPossibleMoves function of the Piece class and subclasses, we can modify the movement of any piece and play a variant such as Masonic chess. Similarly, if we simply change the win conditions in the Board class, we can play the Crazyhouse variant with ease.

Another object-oriented design technique we employed was encapsulation. We believe that the sensitive fields and state of an object should not be accessed directly by other classes and third-party users. Therefore, we ensured that all classified fields were “private”. If any outside members needed to view or modify field values, it would be done through getter and setter methods that were implemented within the class itself.

In addition, we implemented two design patterns that we learned during lectures. For starters, we implemented a variation of the “PImpl” (pointer to implementation) pattern. We used this idiom to determine whether or not a potential move was valid (i.e. if it would put the user's King in check). Our implementation starts by creating a deep copy of the current board (as a pointer). Then, within the copy of the board, the program makes the "potential" move. If the user's King is in check after this move, it is no longer considered a legal move. Otherwise, it is a legal move. Then, the copy of the board is deleted and the rest of the program is executed. This allowed the program to determine valid moves without ever modifying the state of the actual board. The next pattern we utilized was the Observer. We implemented both a Text Observer and a Graphical Observer so that the program would automatically display the game in the console and in XWindow whenever the state of the board changed. We could have implemented this by just individually calling the render functions for our graphical and text observers, without the need for a Studio class or abstract Subject class, but using known solutions to solve a class of problems is better than trying to implement our own, which is why we decided to use the observer class in this case.

As can be seen, our team understood the importance of good object-oriented planning and design. Not only does our design maximize cohesion while minimizing coupling, but we have also made use of essential OOP concepts such as inheritance, polymorphism, and encapsulation all while implementing design idioms such as "PImpl" and Observers.

Resistance to Change

Our project is highly resilient to change, whether that be how the fundamental concept of the game changes or just how the pieces themselves move. This is done by developing our program in such a way that each piece is independent of another distinct piece. Additionally, the board itself is an independent module that only calls on the piece at each index and handles the error checking for whether a piece can be moved.

Through this, we have achieved a program that has low coupling as each module is very loosely or not at all connected to another module. For example, in our program, each piece is completely independent of every other piece on the board and our driver will only call on the board, which calls on the pieces. This way if we were to make any changes in how the game inherently works (like rules for example) or to just implement a new feature, we will avoid any large cascading changes which could cause the whole program to break or crash. If, for example, we wish to change how the knight moves to move 5 squares up and 3 squares left or right, then all we would have to do is alter the getAllPossibleMoves function within the Knight class to recalculate all the possible moves. This way the knight can make according to the new rules given. The only other change that we need to make is to change how the new moves are parsed in the Board class. Since each piece is coded to be independent of the board the piece itself is unaware of where it is on the board so the board needs to contain all the real calculations on which move is possible given all possible moves. Everything else in the code is able to remain the same as most of the functions within the Board that handle the rest of the move checking utilize the parsePossibleMoves function for each piece. This way, minimal changes need to be made to implement new features.

In terms of cohesion, we aimed for our codebase to have high cohesion which we achieved. Each class inherently does only one process. Each piece itself is responsible for calculating all the possible squares that it can move to. Though the board has many functions within it they all work towards the common goal of setup and playing the game by determining if the next move is possible and making the next move. For example, when movePiece (our function to move the piece) is called it will check if the move is possible such as if the player's king will be in check after the move, or if the move is even a valid move that the piece can make then it will call the base piece-moving function which will actually perform the move operation on the board. Additionally, if the move is not possible then it will return a boolean to the driver informing the user that the move is invalid. Finally, the driver itself also has high cohesion as its fundamental purpose is to read in computer/human inputs and either move the piece on the board or inform the player that they are in check/checkmate/stalemate and start the game if the player permits. This way, if someone wishes to change how the game would work then as long as the fundamental purpose of each class is maintained then new changes should be easy to implement.

Through coding our program to have high cohesion and low coupling, any new features that need to be added can be done in a way that wouldn't cause drastic effects on the rest of the codebase or the fundamental functionality of the program itself. Changing game aspects like how each piece can move will only require changes to around two or three functions which primarily are the getAllMoves function within the class for the piece you wish to change and the move parser within the board class to make sure all the moves remaining are legal moves that don't put the King in check.

Questions To Consider (DD1)

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

This is something that could be implemented within the computer, which tries to look in this database of book openings for the moves that are most popular or ones that statistically give the player the best opportunities to win. Although it could take a bit of work to develop, this seems like a graph problem, and we could try to implement a version of a graph to determine a pathway to go in terms of playing moves.

Assume we start with a fresh board with all their pieces in their starting position. Then after every move is played, the number of book moves reduces as some openings require very specific moves to be played. After another move, this number reduces even more, and then there is an even smaller subset of moves to check. We could continue to do this until we're so deep in the game that we have exhausted the opening book moves.

At first thought, a good start would be to just have a bunch of opening move sequences as arrays, and after every move is played, we can match the current sequence of moves with the opening moves, find one that matches, and play the next move that is in that specific opening sequence. Since every move reduces the subset of opening moves, we know that we will reach the point where there are no more opening move matches, and then we know we're out of opening preparation and the computer (or the human player) can continue playing normally.

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

This seems like a rather trivial task considering how we as a group have planned out the implementation of our chess game. The Board class is the class that keeps track of the board state and takes responsibility for making moves, determining whether moves are legal or not, and determining when there's a winner.

To implement the feature where the user can undo their last move, all we would need to add to the Board class is another field, which is given the last move that it allowed. Every time the user gives a move and the Board class determines that the move is legal and can be made, the Board class will make the move and at the same time, save (in that new field) its opposite move. For example, if the move was "move e2 e4", we could save the same move but flip the direction, like "e4 e2." This essentially reverses the move that was made should the user want to undo the move. We wouldn't even need to keep track of the actual piece that was there because there can only be one piece on one square at any given time.

If we wanted to keep track of an unlimited number of move updos, we would have to change the field's type to hold an arbitrary number of moves, ideally in something like a vector. Then for any number of times the user wants to undo their move, they can do so. Here, however, we would have to keep track of both players' moves, as multiple undo's would also have to undo the other players' moves. We could keep track of when we have to undo the other player's move with another boolean field for example, or we could save "pairs" of moves, where if there is only one move in the pair (x.first is populated but x.second isn't) then we know that only one player (with the white pieces) is the only move that we have to undo, otherwise, we have to undo both moves.

Either way, this would be relatively simple to implement with a couple of fields, depending on whether we want to be able to support unlimited undo's or just one.

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.*

Implementing a four-handed chess game would require several changes to both the abstract Piece class and the Board class. This is due to the fact that four-handed chess is a two-versus-two game in which the primary goal is to checkmate one of your two opponents while ensuring that neither you nor your teammate gets checkmated.

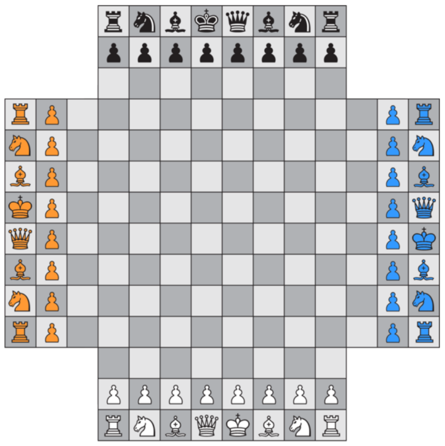

Figure 1. Taken from Wikipedia (https://en.wikipedia.org/wiki/File:Four-handed_chess.png)

In terms of the Board class, this variant requires an 8x8 square board with 3 rows of 8 cells each extending from each side. The easiest way to implement this would be to use a 14x14 square board with 36 internally "dead" squares (i.e. the squares that would have been removed from a 14x14 board to produce the board in Figure 1). These dead squares would be defined manually and any movement to those squares would be deemed illegal. In that case, the movePiece(), getPieceAtPosition(), and parsePossibleMoves() methods would all remain the same besides accounting for the additional squares. The only other thing we would have to account for is that we would have four players, each with a unique colour, and the whosTurn field would shift in a clockwise manner. The logic for determining whether or not a player is in check would remain the same, but we would add an additional check to ensure that the player "giving" check is not their own teammate (in which case the player is not in check). The logic for determining whether or not a player is in checkmate would remain the same, but we would have to additionally ensure that his teammate cannot help him avoid the check in any way. Thus, the Board class would require several changes to ensure that the four-handed variant works properly.

In terms of the Piece class, we could keep the allPossibleMoves() method of the Pawn, King, Knight, Queen, Bishop, and Rook class the same, and simply change the isMoveInBounds() method so that the boundary is extended to a 14x14 coordinate system, excluding the “dead” squares. Any move produced in these new, extended bounds is a valid member of allPossibleMoves(). All other logic remains the same.

Since we have decoupled the Board and Piece class, it is easier to identify what changes are required (most of which are within the Board class).

Questions (DD2)

Question: *What lesson did this project teach you about developing software in teams?*

Throughout this project, we learned many valuable lessons about how software is built in a team environment. The biggest thing we learned was how communication is crucial for the success of any large-scale project when developing software. In the beginning, our team originally went into the assignment with the divide and conquer mindset where each teammate would take on specific tasks and in the end, would try to merge everything together. However, early on in the development phase, we realized that this technique was not feasible as we were not communicating our thought processes thoroughly with each other. With this flaw, different features were delayed as there were dependencies needed from other functions that were not implemented as maybe one teammate didn't consider that the feature being worked on by someone else needed a specific value from the function being built. For example, when developing the En Passant functionality when the board was originally being developed there was no consideration for how a pawn capable of being captured through en passant would be stored and remembered for the following move. After discovering this problem changes had to be made to the Board class to accommodate en passant.

Another lesson that we learned was that frequent merges and testing were necessary. Having constant checks while new features are being developed and merging all the work being done together allowed us to ensure that what we have built so far was functional. Merging new features into the main branch allowed us to check if the new feature worked along with everything else that was already built and additionally being able to track where we were before the new changes were implemented allowed us to perform regression testing as each feature was introduced. This helped determine if anything new was interfering with the functionality of previously built working and tested features as this indicated to us that the new feature that was just pushed in had bugs or interfered with the previous version of the code.

Question: *What would you have done differently if you had the chance to start over?*

If we were given the chance to start over the first thing that we would have done differently would be to plan things more thoroughly. For example, figuring out what the logic for the program would roughly look like including the special aspects of chess like en passant, castling, and promotion. When first developing our program we didn't plan for how those three special moves would fit into our game until the rest of the program was completed. This caused many problems down the line as we had to work around everything that was implemented before as we didn't want to break anything that was already working. If we had coded our program from the start with these extra moves and their requirements in mind from the beginning implementing these extra moves would have been much faster and probably with fewer problems.

Conclusion

All in all, the project provided us with a great opportunity to apply everything we learned throughout the course and build a real-world program. Throughout the journey, we were able to gain a deeper understanding of the concepts and see how the different design patterns are used in real life along with object-oriented programming principles. Additionally, we were able to learn about what software development will be like in the real-world working with team members all working on the same codebase. We were also able to dive deeper into debugging and detecting problems on a large program, unlike the previous assignments where the codebase was at most a few hundred lines. In the end, though we ran into multiple problems and roadblocks along the way, each hurdle we had to overcome taught us a valuable lesson which we kept in mind as we continued on with the development of our program.