Chess | CS246 Project DD2 Fall 2023 Report

Introduction:

For our final project, our group chose to implement the Chess game. During the process of developing the game, we ran across numerous challenges, bugs, communication issues, and moments where we were required to revisit the planning phase and come up with a new plan. This report explains this process and provide clarity on our decisions, changes, learning outcomes, as well as the main characteristics of our program.

Overview:

Model View Controller & Observer Pattern:

The *Game* class represents the *Controller* in our program. It is the part of the program that interacts with the user and delivers the results to the rest of the program.

The *Board* class represents the *Model* in our program. It works to communicate with Game in order to set up the Board at the beginning of the game. Board initializes all Piece objects in the game, using unique_ptr<Piece> (see board.h for details), which allows us to use *Smart Pointers* and provide safety to our program. Board is also responsible for adding, removing, and adjusting Pieces, while also keeping track of the status of the game.

The *TextDisplay & GraphicsDisplay* classes represent our *Views* in the program. For this program, we chose to implement the Observer method such that any change in the game is immediately reflected onto TextDisplay and GraphicsDisplay. Both classes are *Subclasses* of the *Observer* class (see observer.h for details), and are notified using the notify() method when a change to the board is made. GraphicsDisplay owns an Xwindow class (see window.h for details) which aids in creating the graphical interface for the game.

The *Square* class is owned by the Board class and has an Observer. Square represents individual squares on the Chess Board, similar to real Chess; each Square holds information about the X-Y-Coordinate it sits on and a pointer to a Piece that sits on the Square. Square is responsible for adding and removing Piece from a Square, notifying View Observers of any changes to the Square, and notifying any Pieces of changes. Square also holds information regarding whether the Piece sitting on the Square can EnPassant and/or be Promoted.

The *Piece* class is owned by the Square class and is an Observer. Piece is a *Superclass* for King, Queen, Rook, Bishop, Knight, and Pawn classes, and uses *Abstract methods* (see piece.h for details), as well as many

setters and getters to store information for each Piece object. For any relevant change to the Piece's location or status, it is notified so it can reflect the change. It also notifies TextDisplay & GraphicsDisplay for any changes made as well.

Implementing Chess Pieces:

We have designed our program in a way where every time a game is initialized, a Board is created with Squares which hold pointers to Pieces, and these Pieces can be any of King, Queen, Rook, Pawn, Bishop, or Knight. Furthermore, features including Checkmate, Check, Castling, EnPassant, and Promotion, are implemented by creating the *Move* class. We added this new class in our implementation after realizing that it was necessary to have access to all potential moves that all Pieces in the game could make, so we could determine, for example, if King can safely make a move to another Square without entering Check. This change to our plan proved to be instrumental in implementing the Computer class later in our implementation as well. Every Piece has the method, calculateMoves(), which generates and updates a vector<Move> to hold all possible moves of that Piece in its current Square, with the current environment of Pieces.

The *Move* class is owned by Piece, and essentially represents a single move that can be made by a single Piece. It uses a Piece's X and Y coordinate, and stores that along with a Destination X and Y coordinate, pointer to a Square, and a Direction (see move.h for details).

The *Direction* is a Public enumeration which represents any of the 8 compass directions, along with NONE. This enum allows the program to efficiently suggest the direction in which a move will travel.

The *Rook, Bishop, Knight,* and *Queen* Classes all only call on the calculateMoves() method, and don't have any additional complexities. The *Pawn* Class has an additional method to determine whether it can DoubleStep. The *King* Class has functionality to determine whether it is in Check, Checkmate, or neither, as well as whether it can Castle with its *Rook*. The *Rook* Class can determine its possible moves.

Implementing Human & Computer Players:

In our program, the *Human* and *Computer* Classes are inherited by the *Player* Class which is owned by *Game*. Human has no further subclasses and works to interact with data from the user's input in Game and represent the user in the program as a player of one team. Computer is a *Superclass* for 5 levels of varying complexity and works to represent one team entirely independently (no user input involved in the decision-making process).

The *StageOne* Class is-a Computer that takes the current Board of Squares and Pieces and creates a Vector<Move> (an array of Moves) with all possible moves for all pieces. Then, it randomly selected one Move from the array to use.

The *StageTwo* Class is-a computer that prioritizes capturing as many pieces as possible, at every opportunity. We use the Minimax Algorithm with Alpha-Beta pruning, but only go one level deep in our generation of possible Moves produced by the algorithm (generate one team's possible moves, and the resulting possible moves of the opponent team). We rank taking pieces very highly, and rank keeping our pieces much lower so that the "highest value move" is one where we take the highest value pieces at all costs.

The *StageThree* Class is a Computer that *avoids* getting its pieces taken, while strategically attacking and checking the other player. It will generate 2 levels of minimax with alpha-beta pruning once again, but this time the weights of pieces will be identical on both sides. For example, if a board has our king alive that's 1000 points, if it has their king alive then that's -1000 points. We assign similar rankings to all pieces so that the computer can make accurate calculations.

The *StageFour* Class is a Computer that adds another level to the algorithm so that it can perform more computations, and we can assign values and multipliers to certain positions of pieces on the board. For example, if the Bishop can move around fluidly since it's in the middle of the board, then that multiplies the value of the Bishop by 1.2, which would change the results of the calculations. This allows us to stay in generally favourable positions.

The *StageFive* Class is a Computer that uses 6 iterations of the Minimax algorithm and Alpha-Beta Pruning, combined with values and multipliers, to predict the following 6 moves of each team, and using that to see which move is the most efficient to take the opponents highest valued Piece while saving their own highest-valued piece. This stage was created as an enhancement to the assignment and is explained in much further detail in the Enhancements section of the document below.

Minimax Algorithm & Alpha-Beta Pruning:

This algorithm works if the opposing Player will pick moves that are most optimal for them. So, with an n-deep tree of possible moves alongside initial rankings of each move computed by pieces lost v. pieces won (Net Winnings), we can work our way up the tree, alternating between values that that our opponent would want and values that we would want — again, this is under the assumption that the other Player will always want what's worst for us. Once we reach the top layer, we choose the most favourable ranking for us: we make that move.

Alpha-Beta pruning builds on the idea of the minimax algorithm, except now we store a value for either a min or a max depending on whether we are taking the min or max for this level of the tree. Once we see that something will be at most under another max we already have, or at least above an existing min we already have, we can stop calculating values that branch and move on to the next since both players will either never choose that branch or never get the opportunity to choose that branch.

Alpha-Beta pruning drastically speeds up the Minimax algorithm and leads to much greater depth and efficiency in computation.

Design:

Model View Controller

As mentioned in the beginning of our Project Overview, we used MVC to effectively bridge the gap between user input, passing data to classes, and communicating results back to the user. By using this design template, we were able to organize our code structure in an intuitive way and allow for flexibility in the process.

Game acts as our Controller, interacting with the user to determine whether to start a game, enter setup mode, and see whether each player is a Computer or Human.

Board acts as our Model, communicating with different parts of the program, including the Observers, Pieces, Players, as well as with Game, to keep track of data and communicate data with the View.

TextDisplay & GraphicsDisplay act as our View, controlling output with the user.

Observer Pattern:

In our implementation, we used the Observer Pattern to allow for clear communication between objects in an efficient and safe way. This allowed for our View to constantly receive updated information to output and allow significant parts of the Model to have contact with each other.

TextDisplay, *GraphicsDisplay*, and *Piece* are all *Observers*, and *Square* has an Observer. TextDisplay and GraphicsDisplay being Observers allow Piece to communicate (via Square) directly to the displays whenever there is a change (a Piece has moved, died, or been created). Whenever a new Piece is initialized, it can be added to the Observers easily, which allows for more flexibility and reusability within the code which results in higher safety and efficiency.

Template Method:

One major contributor to the clarity in our code is the use of the Template Method. This was used the most in the Piece class, and its subclasses, in the calculateMoves() method. We defined a pure virtual calculateMoves() method in piece.h, and then in King, Queen, Rook, Bishop, Knight, and Pawn, the implementation for calculateMoves() differed based on the way each Piece can move, and the different positions it can take. Had we implemented a different function for each Piece, then we wouldn't be able to effectively generalize all these subclasses as a Piece, and our code would have been longer, more inefficient, and prone to more bugs.

Template Method was also used when implementing the doMove() method in the Computer class. doMove() is defined as a pure virtual method in Computer, and each of *StageOne, StageTwo, StageThree, StageFour, & StageFive* have a different implementation for doMove() depending on the level of complexity associated with the level. Again, using Template Method allowed us to generalize Computer as a player so that we could create a simpler, more efficient, and safer implementation.

RAII – Using Smart Pointers & Using Vector Class:

One major issue we were worried about was having memory leaks and managing memory safely, especially in such a large program with many objects being created. We were able to avoid any such problems by using Smart Pointers. We created all objects using either unique_ptr or make_shared and as a result, all memory was managed by their respective classes.

We also used the vector class to create many different arrays. Using this class allowed us to safely use arrays, and adjust their size and contents, without interacting directly with any heap memory.

Resilience to Change:

Our implementation is highly flexible because of our use of variables rather than values, the Template Method, Observer Pattern, and MVC.

This flexibility can be seen in the implementation of our Board class (board.cc line 5). We have defined a const int BOARDSIZE which can be changed to reflect the size of the board desired. After changing the size of the board, the board can be initialized in setup mode to have pieces in specific spots as per the new size of the board.

Our program is widely resilient to change in the Computer class. Essentially, the accuracy of Computer's choice of play can be improved if the Minimax algorithm is iterated more times. In the *StageFive* class, we implemented 6 iterations. This can very simply be altering the number of times the Minimax algorithm is used.

Answers to Questions:

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. Discuss how you would implement a book of standard openings if required.

To implement a *book of standard opening move sequences*, we would create a tree of boards representative of possible openings. If a move is made that correlates with a certain opening, the program will continue searching through that branch of openings – the program will continue utilizing the tree of openings until there are no possible opening options available. After that, the program will continue determining the best possible moves utilizing our algorithm. Our answer from the DD1 report remains the same for this question.

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

In the DD1 report, our answer to this question was to create a deep copy of the entire Board after every turn of the game and store it in a vector<Board>. Looking back on this answer, it seems rather resource intensive and would not work with the current implementation of our game.

Instead, we would implement it similar to the undoMove(Board&, Move, Colour) method in computer.cc. The idea is to store information of the attacker's initial Square and destination Square. If the move needs to be undone, you will return the attacker back to its initial Square and return the destination Square back to how it was before the Move was made.

Variations on chess abound. For example, four-handed chess is a variant that is played by four players. Outline the changes that would be necessary to make your program into a four-handed chess game.

In the DD1 report, our answer to this question was to create 4 Player objects and adjust our initialization method such that a 14x14 size Board can be made. We also had said that no other changes would be needed.

For our program to be able to handle four-handed chess, we would need to create a new way to initialize the Board, as four-handed chess board is not the traditional square board that our program can handle. Furthermore, our Computer class including all the subclasses of the Computer class would need to adjust using the Minimax algorithm with 4 teams instead of 2. We would need to change the initialization of the Board and include 2 new Colours in our enumeration. We would need to change the implementation for all calculateMoves() functions in every Piece such that moves are compared against the moves of the 3 other teams instead of just one other team.

Extra Credit Features:

As an additional feature, we implemented the *StageFive* Class which is a Computer that uses 6 iterations of the Minimax algorithm and Alpha-Beta Pruning, combined with values and multipliers, to predict the following 6 moves of each team, and using that to see which move is the most efficient to take the opponents highest valued Piece while saving their own highest-valued piece. This means that the algorithm goes through a total of 12 moves to determine which highest valued piece can be taken, and which can be saved, and moves accordingly.

Furthermore, we also completed the entire assignment using Smart Pointers without any memory leaks. We did this using unique_ptr and make_shared.

Final Questions:

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

One of largest learning outcomes of this project was the importance of planning before execution. As a group, we did not emphasise planning very much, and our initial UML did not incorporate many of the changes we have now added. Had we spent more time planning our process, maybe the UML would have been more accurate to the final product.

Another big thing we learned was the importance of interpersonal communication. Diversity of thought is an asset which, if used correctly, helps everyone out tremendously. Telling each other when we run into problems and constantly sharing ideas keeps a level of transparency and allows each member of the group to incorporate their own ideas. Often, sharing your thought process results in a team member coming up with an even better solution!

Code clarity & documentation is vital in working with others. When a team member writes a bunch of code but leaves very little documentation, it becomes challenging for the next person to work with it. Furthermore, poor documentation makes debugging that much more difficult, and asking others for help with debugging next to impossible.

Using Git & Version Control Systems is the largest (industry applicable) technical skill learned through this project. We learned the importance of Git the hard way when one member of the group pushed changes to existing functioning code which caused the whole program to break. We panicked initially, but thanks to Git, we

were able to restore the previous version. Furthermore, it made collaborating and sharing code very simple, and allowed us to work parallel with each other without conflict.

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

If we had the opportunity to do things differently, we would spend more time planning before starting to code. Better planning would have allowed us to divide tasks in a more efficient way. The way we divided code initially was by classes rather than functionality. We soon realized this was not going to work as this method required much of the code being completed before we could begin testing even the most basic functionality of the program.

We would also have emphasized documenting code throughout the process rather than just at the end. This would have made sharing code, debugging, and working with each other's code a lot simpler. It would also have helped us avoid redundant coding (creating multiple functions with similar functionality).

Conclusion:

Overall, as a group we were able to deliver a complete implementation for Chess, with every aspect of the game working to the best of our ability. We were also able to implement a Computer with 4 required levels, and a Fifth level as a bonus feature. We completed the entire assignment without the use of delete statements, and no memory leaks. This was a very large achievement for us, as we had to restart our code multiple times, and went through many challenges to reach the finish line. There were countless moments of stress and panic, and each one taught us more.

We gained a lot from this assignment, as we learned many important aspects of software development through collaboration, Git, and the use of Object-Oriented Principals taught in class. We learned how to work on code in a group setting, solve complex challenges together, and create a real, tangible, and unique program that we will be proud of for a long time.