Final Design Document

Group - Vincent Wan, Yuhong Zeng, Jonathan Kao

Overview

The overall structure of our project is as follows: the Game class is the overarching class which handles the flow of the game state, it owns two players, a graphics window, the game board, the score and a move history. The Player class shares pieces with the board and the Player class is called upon by the Game class to make moves. The player has its concrete implementations with the human and computer classes which represent a human player or one of the computer level players. The abstract Board class which is implemented as a concrete Two Player Board class and contains the information about the pieces positions. The abstract Piece class has concrete classes for each type of piece (King, Bishop, Queen, …) which contain the heavy-duty backend logic for each piece - how it can move, attack, castle, etc. These board and piece classes affect each other's control flow, access each other's fields via get and set methods and (add more). Basically, they're very highly coupled but that's what we want. This is because the board class must call a piece to move, which the specific piece then must figure out where in the board it currently is, after that it moves, but needs the board's information, so the board calls back.

Each piece is one of King, Queen, Bishop, Rook, Pawn, and Knight. These classes are pieces, so they're concrete derived classes from the superclass Piece that is abstract. Also, we incorporated polymorphism in the board class, by utilizing a matrix of Piece pointers to represent the board. That way, we can represent a blank cell with a nullptr on that cell, and a piece of (any subtype) if the cell isn't blank. This allows for the design to support the possibility of various changes to the program specification.

The Player class is an abstract superclass with the derived classes of Human and Computer (is-a). Human class is the class that the user will initialize if they are playing as a human and Computer class is the class that the user will initialize if a robot is playing. For the Human class, it is concrete and final, with fields and methods that allow for the Human class to interact with the board and pieces, move them, and do stuff. For the Computer Class, it is abstract with the derived classes One, Two, and Three representing the difficulties of the computer AI. Each of these are concrete with fields and methods that allow for them to move in ways that satisfy the Assignment guidelines for the difficulties of AI.

There is a XWindow class that utilizes the PImpl idiom in order to reduce the number of dependencies among the XWindow class, thus reducing compilation time. This is achieved by placing all of XWindow's private fields into a separate class, XWindowImpl, and using a pointer to XWindowImpl as a private field in the XWindow class. The Game class owns an XWindow for graphics, and the PImpl idiom makes it so that the Game class has no compilation dependency on XWindowImpl.

Another hierarchy of classes that we decided to make was the abstract Move class. The concrete classes of the Move class were the specific types of moves such as castling, pawn promotion, capturing, a regular move, and en passant. The purpose of this was to store the moves in a move history that could be used for updating the graphics. Only updating the graphics based on the last move in the move history would minimize the amount of redrawing required during run time.

Finally, there is a simple Score class with fields that store the current scores of the players (Player 1 and Player 2). This class includes methods that allow for the implementation to print, set, increase points for a player, and more. This is an example of good coding practice since it follows the single responsibility principle, because it only has one reason to change, which is if the score calculation changes.

Updated UML

The updated UML is below, but the changes from the original UML are outlined here. Further elaboration on these changes follow in the Design section. The first major change to the UML was the expansion to the Player class hierarchy. Originally, there was an abstract Player class with concrete classes Human and Computer, but the Computer class was made abstract with three concrete classes to represent the three levels of computer difficulties. Furthemore, the Game class, used to handle the state of the game, was introduced which changed up the relationships between certain classes. The Game class owns a board, the players, moves (in the form of a move history), the score, and graphics window. Previously the Board class owned the Pieces, but the updated UML now has the Player class owning the Pieces. Trivially, more public methods were required for the implementation as the coding progressed, so those now appear in the updated UML. In particular, the design decision was to have the Piece subclasses handle a large portion of the move logic so many new methods were introduced.

Design

One of the first challenges that our group encountered was deciding on the relationships between the classes. In particular, it was unclear which class should make a move in the program. After the command interpreter received a move command, would a player move a piece, would the board move a piece, or would the piece move itself? To promote polymorphism, the two player board class contained a 2D array of Piece pointers. High coupling was required between the pieces and board in this design since the board only had positions of Piece pointers and did not know what kind of Piece each actually pointed to. On the other hand, the piece would need to know information from the board in order to know what moves were possible. To achieve this, the piece classes also had a pointer to the board. In the end, the command interpreter in the Game class would call the move method on a player, which would then attempt a move by calling the move method on a piece. The piece move method would return a “status” which would give information on whether the move was possible; if the move was possible then the type of move would be returned back to the Game class. The Game class would then act accordingly based upon the returned information. This design worked well since it appealed to the Single Responsibility Principle. The Game class was in charge of handling the state of the game, acting as an interface between the main program, the graphics and the game state. The board was responsible for keeping track of piece positions, the pieces were in charge of the move logic, and the players were responsible for moving those pieces. This design promoted high cohesion within each class since all the elements in each class cooperated to perform a singular task.

Another major design decision stemmed from the challenge of selectively updating the graphics. It would be straightforward to redraw the entire graphics display after each move, however this would contradict the requirements in the specifications. We decided to address this by implementing a move history. The move history was implemented as a vector of Move object pointers where the Move class was the abstract superclass for the different types of moves (pawn promotion, capture, castling, etc. …). The addition of the move history addressed the challenge since after each move, the program would be able to check the move history and only update the graphics based on the most recent move. The move history also opened up the possibility of implementing a move history printer for when the game ended or even an undo-move system if needed. In chess there is a draw by repetition condition for if the same board state is reached three times in a row; although we did not have time to implement this, this could also stem from the move history by checking the sequences of moves. In order to implement the move history, the design of how pieces moved had to evolve. The general idea was that once a move was approved as a legal move, then the information about the move would have to be passed back and stored in the history. Initially, the move information was passed from the command interpreter, the Game class, to the Player classes which would pass the information to the pieces and board. After the move logic was handled, the move information was passed back to the Game class as a zero if the move was illegal, one if the move was a normal move, or a two if the move was a capture. In order to accommodate for the move history as well as the special moves such as en passant, the move information that was returned to the Game class had to be changed to a pair with an integer and string. The integer indicates the type of the move and the string contains extra information such as the type of piece that a pawn is promoting into. This extra information would allow for selective graphic updates.

The final major design decision lies in the conformity to the Resource Acquisition is Initialization (RAII) principle. Initially, we coded the program without any smart pointers (as we had not learned the concept in lectures yet). To transition, unique pointers were used for the fields that the Game class owned. This included the window, board, score, players and moves. The next transition would involve the board and player classes with the pieces. Originally, the first design had the player class as the owner, however it was later decided that sharing the ownership with the board class would be easier to implement for two main reasons. The first reason was that the sharing piece ownership made it easier to move the pieces that belonged to each player. The flow of the game was that the command interpreter had the player make a move, so the player having ownership of the piece pointers would make it simpler to make the move. The player class also has a vector of shared pointers for the "invalid pieces" which would be the pieces that were captured. If we wanted to implement the undo-move feature, then this would prove useful. The second main reason came from the flow of the program and the set-up mode. During the set-up mode, the players are not yet initialized so the pieces are solely owned by the board. This would be problematic if the program were to end during set-up mode or between games since the memory would leak from the pieces. To remedy this, we used shared pointers for the pieces and shared the pointers between the board and player class. This would allow for no memory leaks during any point of the game and it made sense with the UML relationships as well.

Resilience to Change

One design choice that supports the resilience to change was using abstract base classes in our implementations of the pieces, boards, moves, and players. In particular, while it was not strictly necessary, we chose to implement the board class hierarchy as an abstract Board class with a concrete twoPlayerBoard class. This was done with the specification question in mind. If we ever wanted to play chess on a different kind of board, such as the four player chess board, we would be able to inherit from the abstract Board class. This would support the possibility of different boards. Furthermore, the choice to use an abstract Piece class as the superclass opens up the possibility to add different kinds of pieces if we ever wanted to. The same logic applies to the abstract class for players and moves. The design choice allows different types of moves to be added to the rules and easier expansion if more computer levels are made.

Another instance of resilience to change lies in the use of the PImpl Idiom for the XWindow class. The XWindow uses a pointer to its implementation, XWindowImpl. The header file for the XWindow class does not have compilation dependency on XWindowImpl.h. This implies that change can occur in the implementation in XWindowImpl and the change will not cause clients using the XWindow class like the Game class to have to recompile.

Answers to Project Specification Questions

1. To implement a book of standard openings, we would store each opening as a vector of Move pointers. We had defined an abstract class Move that was the superclass for the concrete moves of each type. The main purpose of the Move classes was to implement a move history, also stored as a vector of Move pointers in the Game class. After each move, we would be able to compare the move history with the vectors for the different standard openings. This would allow us to match the openings that were still available and even give suggestions, by looking at the next move in the openings. Storing a large number of standard openings as vectors of Move pointers may eventually take a lot of space, but this design is quite simplistic and straightforward to implement.

2. This question about undos and infinite undos would again depend on our move history located in the Game class. To undo a move, we would access the most recent move from the move history vector. Next, we would extract the information from the move, such as the positions changed in the move, and the type of move. This would allow us to reverse the move by updating the fields for each piece and the board as necessary. The same idea applies for an unlimited number of undos. While the move history is non-empty, the program would be able to undo moves as mentioned above as the client requests.

3. With this question in mind, we first implemented the game board as an abstract class. Then we implemented the concrete class called twoPlayerBoard which was the standard chess board. To modify our program into a four player chess game, we would need to implement another subclass off of the abstract Board class to make a fourPlayerBoard class. This would include methods for inserting, removing, and moving pieces as well as a different array of pointers to represent the board. The shape of a four player game board is a bit odd, but it could be implemented as a 14x14 board with the 3x3 corners being null pointers since they are not an actual part of the board. Some modifications might also need to be made to some of the computer levels since the rules are a bit different. The piece classes which handle their own move logic would need to be adapted to the four player board as well. Lastly, the Game class would need some modifications for displaying the graphics. The Game class would also need two additional fields for the two new players and the game flow logic would need to be adapted to the new move order.

Final Questions

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

This project taught us many lessons about developing software in teams. The first lesson was a more technical one. Some group members had not really previously worked with Git so this project provided us with a chance to collaborate through Git. This is an especially useful skill to learn since it would prove beneficial for any co-op positions/jobs we look for in the future. The next lesson was that everyone has different ideas on how they want to implement a solution and that it is important to thoroughly discuss and consider all the ideas before diving into the actual coding. Often, the best implementation comes from a combination of ideas from different group members since it is hard for any single person to consider every possible factor when it comes to the implementation.

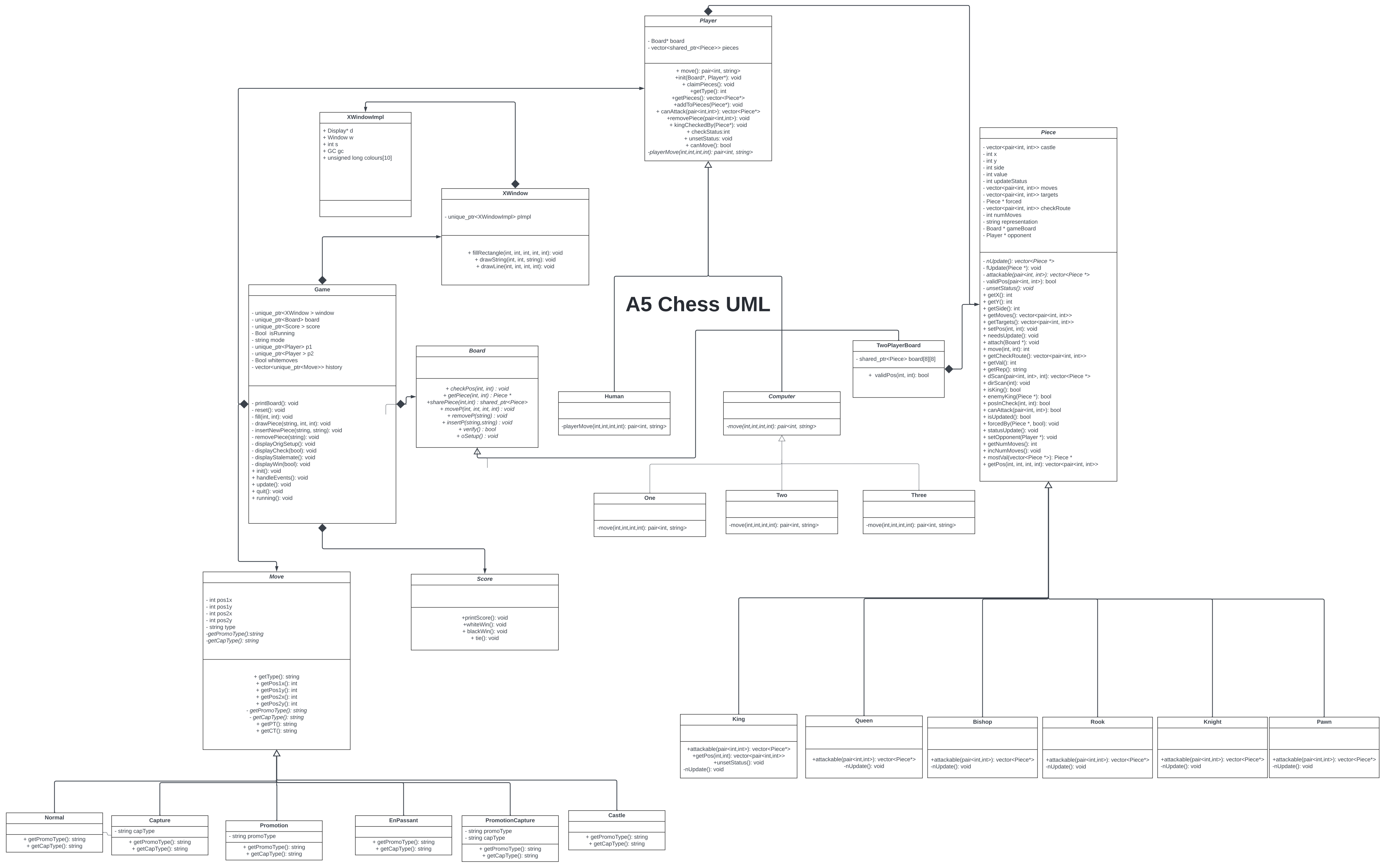
We also learned that communication between members is very key. In particular, each team member should communicate their actions frequently and with sufficient detail. Often, different members would be working on different parts and each part would eventually need to be connected together. As a result, detailed and frequent communication is required in order to properly connect all the parts. If the communication was not clear, sometimes certain parts would need to be re-coded in order to correctly mesh each part together.

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

If we had the chance to start over, before starting to code the project, we would probably discuss our implementation more cohesively, and finalize it, so that we are all on the same page and working towards the same end result, thus avoiding a large number of conflicts, thus increasing overall efficiency and quality of results. In retrospect, we spent ample time in planning out the general structure of the project but we did not spend enough time discussing the specific implementations for the piece and the board class. In particular, it would have been better to plan out their relationships and methods more to ensure minimal coupling. We decided on letting one member start coding a few days before the others started coding. That member then made a lot of code on the piece class with his "ideal" implementation, which included a large amount of coupling between the board class, the piece class, the main method, the player class, the computer class, and the score class. Essentially, each class contained get and set methods for the other classes to call, as well as code that stops the code from the other class's control flow. Worst of all, each of the classes' fields contained pointers to other classes! However, once that member already implemented half of the methods including the logic of the pieces, like whether a king is in check or whether the move is valid, as well as the move methods, it was already too late, since the other teammates had a better implementation in mind, but were slightly forced into using this one, because we would have to start all over again otherwise. Not only that, but the implementation was quite disastrous as modifying even a small fraction of the code in a file resulted in a massive abrupt change in the code from the other files in order to make it work again. Thankfully, we were able to pick it up again by all discussing on a group call about our final implementation, and sticking to that, so we had to modify our code quite a lot, resulting in a loss of time, but we were still on track since we started early (thankfully). In short, next time, before starting to code anything related to classes that inherit off of other classes and form this web of dependencies, causing a massive amount of coupling, discuss more about the underlying implementation, and finalize it so that everyone is on the same path (so no conflicts), and that less coupling is caused. This is certainly something to look out for, especially when it comes to these big projects.

It would have also been helpful to focus on getting the “setup” utility in place earlier. This would have allowed for debugging to occur more regularly and earlier in the process. The “setup” functionality was pushed back quite a bit which led to the debugging process also stacking up. Being able to debug more regularly during the coding would have made it smoother in the end to finalize all the features.

A5 Chess UML


Player
- Board* board
- vector<shared_ptr<Piece>> pieces
+ move(): pair<int, string>
+init(Board*, Player*): void
+ claimPieces(): void
+getType(): int
+getPieces(): vector<Piece*>
+addToPieces(Piece*): void
+ canAttack(pair<int,int>): vector<Piece*>
+removePiece(pair<int,int>): void
+ kingCheckedBy(Piece*): void
+ checkStatus:int
+ unsetStatus: void
+ canMove(): bool
-playerMove(int,int,int,int): pair<int, string>
XWindowImpl
+ Display* d
+ Window w
+ int s
+ GC gc
+ unsigned long colours[10]
XWindow
- unique_ptr<XWindowImpl> pImpl
+ fillRectangle(int, int, int, int, int): void
+ drawString(int, int, string): void
+ drawLine(int, int, int, int): void
Game
- unique_ptr<XWindow > window
- unique_ptr<Board> board
- unique_ptr<Score > score
- Bool isRunning
- string mode
- unique_ptr<Player> p1
- unique_ptr<Player > p2
- Bool whitemoves
- vector<unique_ptr<Move>> history
- printBoard(): void
- reset(): void
- fill(int, int): void
- drawPiece(string, int, int): void
- insertNewPiece(string, string): void
- removePiece(string): void
- displayOrigSetup(): void
- displayCheck(bool): void
- displayStalemate(): void
- displayWin(bool): void
+ init(): void
+ handleEvents(): void
+ update(): void
+ quit(): void
+ running(): void
Board
+ checkPos(int, int) : void
+ getPiece(int, int) : Piece *
+sharePiece(int,int) : shared_ptr<Piece>
+ moveP(int, int, int, int) : void
+ removeP(string) : void
+ insertP(string,string) : void
+ verify() : bool
+ oSetup() : void
Human
-playerMove(int,int,int,int): pair<int, string>
Computer
-move(int,int,int,int): pair<int, string>
TwoPlayerBoard
- shared_ptr<Piece> board[8][8]
+ validPos(int, int): bool
One
-move(int,int,int,int): pair<int, string>
Two
-move(int,int,int,int): pair<int, string>
Three
-move(int,int,int,int): pair<int, string>
Piece
- vector<pair<int, int>> castle
- int x
- int y
- int side
- int value
- int updateStatus
- vector<pair<int, int>> moves
- vector<pair<int, int>> targets
- Piece * forced
- vector<pair<int, int>> checkRoute
- int numMoves
- string representation
- Board * gameBoard
- Player * opponent
- nUpdate(): vector<Piece *>
- fUpdate(Piece *): void
- attackable(pair<int, int>): vector<Piece *>
- validPos(pair<int, int>): bool
- unsetStatus(): void
+ getX(): int
+ getY(): int
+ getSide(): int
+ getMoves(): vector<pair<int, int>>
+ getTargets(): vector<pair<int, int>>
+ setPos(int, int): void
+ needsUpdate(): void
+ attach(Board *): void
+ move(int, int): int
+ getCheckRoute(): vector<pair<int, int>>
+ getVal(): int
+ getRep(): string
+ dScan(pair<int, int>, int): vector<Piece *>
+ dirScan(int): void
+ isKing(): bool
+ enemyKing(Piece *): bool
+ posInCheck(int, int): bool
+ canAttack(pair<int, int>): bool
+ isUpdated(): bool
+ forcedBy(Piece *, bool): void
+ statusUpdate(): void
+ setOpponent(Player *): void
+ getNumMoves(): int
+ incNumMoves(): void
+ mostVal(vector<Piece *>): Piece *
+ getPos(int, int, int, int): vector<pair<int, int>>
Move
- int pos1x
- int pos1y
- int pos2x
- int pos2y
- string type
-getPromoType():string
-getCapType(): string
+ getType(): string
+ getPos1x(): int
+ getPos1y(): int
+ getPos2x(): int
+ getPos2y(): int
- getPromoType(): string
- getCapType(): string
+ getPT(): string
+ getCT(): string
Score
+printScore(): void
+whiteWin(): void
+ blackWin(): void
+ tie(): void
Normal
+ getPromoType(): string
+ getCapType(): string
Capture
- string capType
+ getPromoType(): string
+ getCapType(): string
Promotion
- string promoType
+ getPromoType(): string
+ getCapType(): string
EnPassant
+ getPromoType(): string
+ getCapType(): string
PromotionCapture
- string promoType
- string capType
+ getPromoType(): string
+ getCapType(): string
Castle
+ getPromoType(): string
+ getCapType(): string
King
+attackable(pair<int,int>): vector<Piece*>
+getPos(int,int): vector<pair<int,int>>
+unsetStatus(): void
-nUpdate(): void
Queen
+attackable(pair<int,int>): vector<Piece*>
-nUpdate(): void
Bishop
+attackable(pair<int,int>): vector<Piece*>
-nUpdate(): void
Rook
+attackable(pair<int,int>): vector<Piece*>
-nUpdate(): void
Knight
+attackable(pair<int,int>): vector<Piece*>
-nUpdate(): void
Pawn
+attackable(pair<int,int>): vector<Piece*>
-nUpdate(): void