Design Document for chess

by

David Knight
Tom Ebergen

Our final design and our original UML differed in many ways. After the first week we had basic functionality of the chess program running with no AI but we were fairly confident about our UML. Little did we know we would have to add many more functions in order to implement our AI and Xwindow/visual display. In addition, because of time constraints we couldn't implement the AI into its own class, which made our player.cc file very big.

The Xwindow class added a lot of functions to our UML document. We had to add draw and undraw functions for every subclass of the piece class. We added functions to the Xwindow class as well so that we could draw more detailed chess pieces. However, when it came to specific requirements for the game, we encountered many problems that required more functions. In order to implement the functionality for pawn *en passant*, castling, and detecting if your king was in check or checkmate, we had to add new member variables and functions to the board class, the players class, and some of the subclasses of piece.

Our plan of attack didn't change drastically during the course of the two weeks. We took out some functions and moved others to different classes but there were no real showstoppers. One problem we did have involved working on mutual code. Many times one partner would improve a piece of code in such a way that it would actually counteract the progress the other partner has made. Once we realized that this was a real problem the two of us decided to work completely independent parts of the code in order to avoid any type of confusion involving sharing code. The two of us decided that one person should work on the AI while the other works on basic functionality of a chess board.

David Knight, the member who worked on the basic functionality, was able to code it all fairly early on. From there he went on to code the functionality that would handle the checking of the king, the pawn *en passant*, and the castling. This would ensure a working program before any specifics were implemented. Tom, the member who worked on the AI, started with coding a simple function that would move random pieces for a computer level 1 before attempting to implement any other computer levels.

This approach allowed us to separately work on code without having to worry about updates our partners would make. Over the final three to four days we merged our two pieces of code. Although it took a while, we were able to integrate our two versions of the code very well.

Because of time restrictions, we were not able to implement a level 4 for our AI. Apart from that, our code complies with all other assignment specs along with some bonus features we implemented.

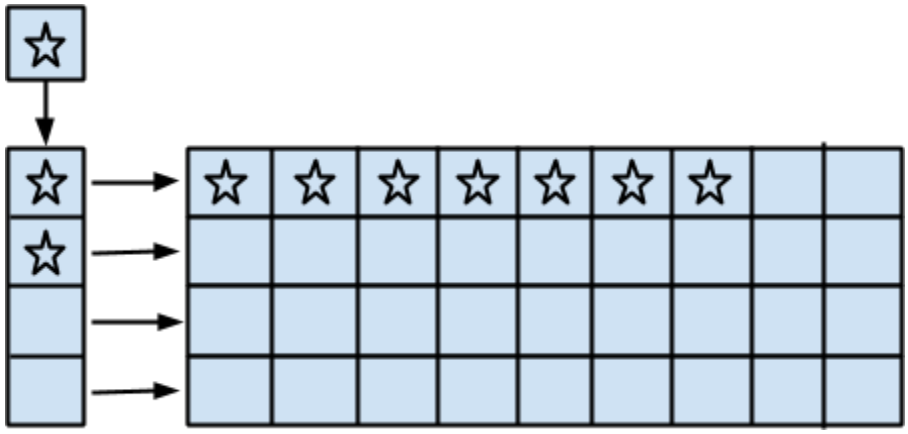
How the code works

For our chess game to work we implemented the observer pattern that was taught earlier in the year. We made a board class to act as the subject while the piece class embodied the observers. The piece class was an abstract class that was used to build all the pieces that occur on the board as well as an empty piece that was used to signify an empty board space. We used our functions istype and getletter to find out what pieces were on the board.

As noted the earlier the piece class is an abstract class that outlines the functions for all the pieces that are on the board. The piece class declares void canmove(...) as pure virtual to express the fact that it is an abstract class. However, many functions were so general to the piece class they could be defined within piece.cc. For example, getrow(), getcol() were implemented in the piece class.

Moving on from the abstract piece class we come to the individual pieces. Knight bishop and queen had no special moves so they were easily implemented. However, king, rook, and pawn all have special moves within the game that require specific variables to be declared within their classes in order to give them their special moves. For king, many functions were added to consider castling and being in check, checkmate, or stalemate. The rook had functions added as well in order to accommodate the castling move. The pawn class has many extra booleans and functions in order to do pawn *en passant* and pawn replacement. These functions handle turning on the ability to capture diagonally when a pawn has crossed it on the immediate previous move and more. In regards to all the pieces, they all had a unique canmove function that helped us determine if a piece was able to move to a position or not depending on what type of piece it was and how it could move.

Going back to the board class on the UML we can see it has a triple pointer to a piece array. This was implemented because an abstract piece cannot be created, so we had to create a pointer to an array of pointers to arrays of pointers to pieces like the following picture.



Where the star at the start of the diagram is a pointer to an array of pointers to arrays of pointer to pieces. The stars in the leftmost column are pointers to arrays of pointers to pieces. The stars in the block of squares represent pointers to pieces.

Going on, the board class contained many of the functions responsible for actually moving the pieces and setting up the board. The board class has functions for castling, invoking the pawn *en passant* rule as well as for having pieces notify each other as to where they are and who they can capture. From the board class one can see the Xwindow class and Textdisplay class that the board has. These classes simply output the information held by the board class so that it can be visually understood by whoever is playing.

Going back to the board class we can now move onto the player class. The player class declares to static players, black and white, who will be the only players during the chess game. In addition, it declares many other static functions that allow all other functions to find out whos turn it is, weather one can exit setup or not etc. Many other functions are helper functions for pawn *en passant*, castling, and checking to make sure the king is not in checkmate etc. Inside the player class is where our AI is setup. When player makemove() is called our AI takes over. Player makemove() is only called if a player is a computer. From here we access the level inside the player class to choose what AI should run. Below is an explanation of how the AI has been implemented.

Level 1: Random moves

Level one implements a simple random move. We go through every space on the board and fill an array called mine with only the pieces the belong to the player who is allowed to move. From there we declare a random number that does not exceed the amount of pieces the current player has. We use that random number to select a random piece. After that we generate two more random numbers for the column number and row number. If we manage to go through every piece without finding a random spot for it to move to, we use a brute force method. The brute force method runs through every piece in array mine and checks to see if it can move to any one of the squares on the board. If it can move to one of the squared it moves to that square. The brute force method is at the end of every AI level.

Level 2: Prefers capturing moves and checks over other moves.

Level two also uses the mine array in addition to an array called theirs which is full of pieces belonging to the opponent. The AI for level two tries to see if it can capture any of the opponents pieces. While doing so, it ranks the capture on a scale from 1 to 10. This allows the AI to capture the most powerful piece available belonging to its opponent. If none of the pieces in the mine array can capture an enemy piece, the AI moves to looking if it can check the opponent.

The AI attempts to check the opponent by locating the enemy king and storing the locations around the king that are empty that will the appropriate piece to check the king if it were in said position. For example, if the AI wanted to check and see if the rook could check the king, the AI would locate the king and store all locations that share the same column and row as the king. However, these location will only be stored if they are empty or if the contain an enemy piece. Once the system detects an enemy piece it stores that location and stops. This is too avoid moving to a row or column where a rook cannot technically check the king.

The AI will do this for every piece it owns. It will check the king once it finds its first available option. If it cannot find an available option, it will head to the functionality used in level one, ending with the brute force option.

Level 3: prefers avoiding capture, capturing moves, and checks

Level three of the AI works on avoiding capture using the dangers integer within every piece. Whenever a piece moves it notifies the opposing pieces on the board and all the opposing pieces notify the originally moved piece. Because of this the pieces can detect if they are in danger or not. Using this

functionality, the AI is able to find out which pieces are in danger and reacts accordingly. Naturally, the AI will try to avoid captures involving its most valuable pieces first and go down from there. If no piece is in danger or it is not possible to avoid capture by moving the piece, the functionality from AI level 2 will be run, which will end in the brute force method once again.

Oddly enough, after running some tests it seems our level 2 is smarter than our level 3.

From the player class we can see the scoreboard class. The scoreboard class simply keeps track of the amount of times a certain player has won. The score is kept as floats to accommodate awarding half a point for stalemate.

Bonus Features

We added a lot of bonus features to our program.We have the following additional features;

1. In setup mode the “default” command will place all the default pieces on the board.
2. In setup mode the “pawns” command will place a lot of pawns on the board which is useful for testing en passant and pawn replacement.
3. In setup mode the “easywhitecheckmate” and “easyblackcheckmate” commands will set up the board to be easy to get the opposing player in checkmate.
4. In setup mode the “instantwhitestalemate” and “instantblackstalemate” command will make the game start off in stalemate for the corresponding player.
5. In setup mode the “piececolour” arg1 arg2 (10 lowercase colors available) will redraw all the pieces on the board with arg1 as the colour for white player pieces and arg2 for black player pieces.
6. In setup mode the “tilecolour” arg1 arg2 (10 lowercase colors available) will redraw all the tiles on the board with arg1 as the colour for half the tiles and arg2 for the other half.
7. We added the feature of allowing a human player to try moving a piece again if an invalid move was made since it is very easy to make invalid moves.
8. In game mode we added the “moves” command which takes two arguments. This can only be used when playing computer vs computer.The first argument is how many turns to move and the second is the speed of the moves (1 is standard slow speed and 20 is pretty fast).
9. In addition, we actually draw pieces that represent true chess pieces using X11 commands.

Here are specific changes to the UML document.

In Piece.h

Fields that were removed

bool defended

bool white
int piecelevel1
int piecelevel2

Added fields
int wcolour
int bcolour
int tile1colour
int tile2colour
int rightshift
int downshift
xwin Xwindow
+ undraw()
+ drawletters()
+ setdisplay()
+ setpiececolours(string white, string black): void
+ settilecolours(string tile1, string tile2): void
+ redraw(): void
+ getcoords(): string
+ indanger(): int
+ incdanger(): void
+ dimdanger(): void
+ resetdanger(): void
+ getrow(): int
+ getcol(): int

In rook.h

Fields that were added
+ hasmoved(): bool
+ setmoved(): void
+ draw(): void

In pawn.h

Fields that were added
- bool enpassen
- int empassen
+ pawnexchange(): void
+ unsetempassen(): void
+ getempassen(): int
+ isempassen(): void
+ draw():void

In King.h

Fields that were removed

- bool check
- bool aftermoveincheck
- bool castlecheck
- bool checkmate
+ castle(string newcoords):void

Fields that were added

- hasmoved(): bool
- setmoved(): void
- setincheck(): void
- unsetincheck(): void
- draw():void

Board.h

Fields that were added

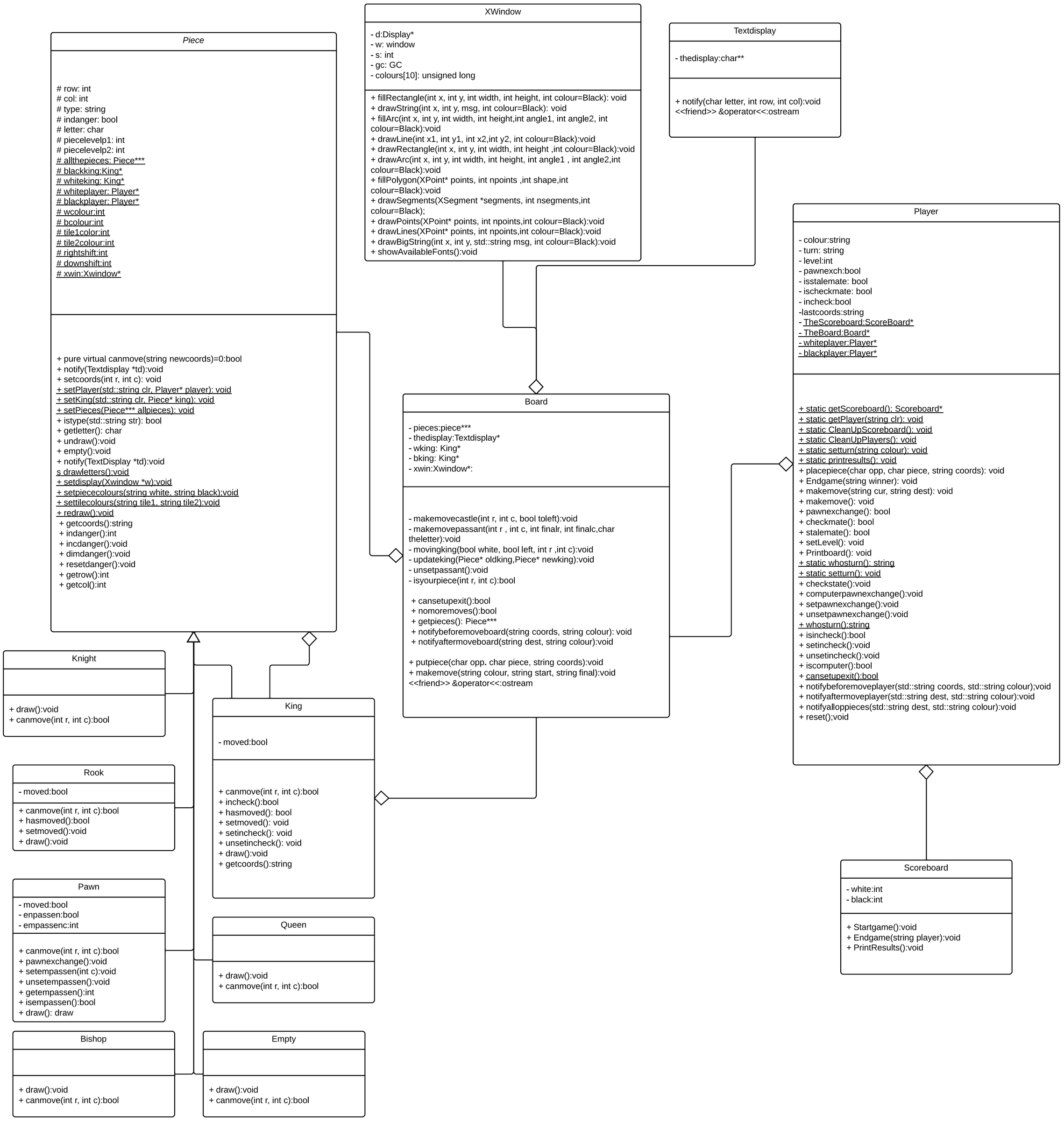
- xwin Xwindow*
- makemovecastle(int r, int c, bool toleft): void
- makemovepassant(int r, int c, int finalr, int finalc, char theletter): void
- moveingKing(bool white, bool left, int r, int c): void
- updateking(Piece* oldking, Piece* newking): void
- unsetpassant(): void
- isyourpiece(int r, int c): bool
+ cansetupexit(): bool
+ nomoremoves(): bool
+ getpieces(): Piece***
+ notifybeforemoveBoard(string coords, string colour): void
+ notifyaftermoveBoard(string dest, string colour): void

Player.h

Fields that have been added

- bool incheck
- string lastcoords
+ checkstate(): void
+ computerpawnexchange(): void
+ unsetpawnexchange(): void
+ whosturn(): string
+ isincheck(): bool
+ setincheck(): void
+ unsetincheck(): void
+ iscomputer(): bool
+ cansetupexit(): bool
+ notifybeforemoveplayer(string curren, string colour): void

+ notifyaftermoveplayer(string dest, string colour): void
+ reset(): void


XWindow
- d:Display*
- w: window
- s: int
- gc: GC
- colours[10]: unsigned long
+ fillRectangle(int x, int y, int width, int height, int colour=Black): void
+ drawString(int x, int y, msg, int colour=Black): void
+ fillArc(int x, int y, int width, int height,int angle1, int angle2, int colour=Black):void
+ drawLine(int x1, int y1, int x2,int y2, int colour=Black):void
+ drawRectangle(int x, int y, int width, int height ,int colour=Black):void
+ drawArc(int x, int y, int width, int height, int angle1 , int angle2,int colour=Black):void
+ fillPolygon(XPoint* points, int npoints ,int shape,int colour=Black):void
+ drawSegments(XSegment *segments, int nsegments,int colour=Black);
+ drawPoints(XPoint* points, int npoints,int colour=Black):void
+ drawLines(XPoint* points, int npoints,int colour=Black):void
+ drawBigString(int x, int y, std::string msg, int colour=Black):void
+ showAvailableFonts():void
Textdisplay
- thedisplay:char**
+ notify(char letter, int row, int col):void
<<friend>> &operator<<:ostream
Piece
row: int
col: int
type: string
indanger: bool
letter: char
piecelevelp1: int
piecelevelp2: int
allthepieces: Piece***
blackking:King*
whiteking: King*
whiteplayer: Player*
blackplayer: Player*
wcolour:int
bcolour:int
tile1color:int
tile2colour:int
rightshift:int
downshift:int
xwin:Xwindow*
+ pure virtual canmove(string newcoords)=0:bool
+ notify(Textdisplay *td):void
+ setcoords(int r, int c): void
+ setPlayer(std::string clr, Player* player): void
+ setKing(std::string clr, Piece* king): void
+ setPieces(Piece*** allpieces): void
+ istype(std::string str): bool
+ getletter(): char
+ undraw():void
+ empty():void
+ notify(TextDisplay *td):void
s drawletters():void
+ setdisplay(Xwindow *w):void
+ setpiececolours(string white, string black);void
+ settilecolours(string tile1, string tile2):void
+ redraw():void
+ getcoords():string
+ indanger():int
+ incdanger():void
+ dimdanger():void
+ resetdanger():void
+ getrow():int
+ getcol():int
Board
- pieces:piece***
- thedisplay:Textdisplay*
- wking: King*
- bking: King*
- xwin:Xwindow*:
- makemovecastle(int r, int c, bool toleft):void
- makemovepassant(int r , int c, int finalr, int finalc,char theletter):void
- movingking(bool white, bool left, int r ,int c):void
- updateking(Piece* oldking,Piece* newking):void
- unsetpassant():void
- isyourpiece(int r, int c):bool
+ cansetupexit():bool
+ nomoremoves():bool
+ getpieces(): Piece***
+ notifybeforemoveboard(string coords, string colour): void
+ notifyaftermoveboard(string dest, string colour):void
+ putpiece(char opp. char piece, string coords):void
+ makemove(string colour, string start, string final):void
<<friend>> &operator<<:ostream
Player
- colour:string
- turn: string
- level:int
- pawnexch:bool
- isstalemate: bool
- ischeckmate: bool
- incheck:bool
-lastcoords:string
- TheScoreboard:ScoreBoard*
- TheBoard:Board*
- whiteplayer:Player*
- blackplayer:Player*
+ static getScoreboard(); Scoreboard*
+ static getPlayer(string clr): void
+ static CleanUpScoreboard(): void
+ static CleanUpPlayers(): void
+ static setturn(string colour): void
+ static printresults(): void
+ placepiece(char opp, char piece, string coords): void
+ Endgame(string winner): void
+ makemove(string cur, string dest): void
+ makemove(): void
+ pawnexchange(): bool
+ checkmate(): bool
+ stalemate(): bool
+ setLevel(): void
+ Printboard(): void
+ static whosturn(): string
+ static setturn(): void
+ checkstate():void
+ computerpawnexchange():void
+ setpawnexchange():void
+ unsetpawnexchange():void
+ whosturn():string
+ isincheck():bool
+ setincheck():void
+ unsetincheck():void
+ iscomputer():bool
+ cansetupexit():bool
+ notifybeforemoveplayer(std::string coords, std::string colour);void
+ notifyaftermoveplayer(std::string dest, std::string colour):void
+ notifyalloppieces(std::string dest, std::string colour):void
+ reset();void
Knight
+ draw():void
+ canmove(int r, int c):bool
King
- moved:bool
+ canmove(int r, int c):bool
+ incheck():bool
+ hasmoved(): bool
+ setmoved(): void
+ setincheck(): void
+ unsetincheck(): void
+ draw():void
+ getcoords():string
Rook
- moved:bool
+ canmove(int r, int c):bool
+ hasmoved():bool
+ setmoved():void
+ draw():void
Pawn
- moved:bool
- enpassen:bool
- empassenc:int
+ canmove(int r, int c):bool
+ pawnexchange():void
+ setempassen(int c):void
+ unsetempassen():void
+ getempassen():int
+ isempassen():bool
+ draw(): draw
Queen
+ draw():void
+ canmove(int r, int c):bool
Scoreboard
- white:int
- black:int
+ Startgame():void
+ Endgame(string player):void
+ PrintResults():void
Bishop
+ draw():void
+ canmove(int r, int c):bool
Empty
+ draw():void
+ canmove(int r, int c):bool