# CS 246 Chess Design Document

Yuanjing Cai (20549267), Yidan Chen (20568620)

## Overview

### class Move

In our design, the Move class acts as a “control center” of the game interacting directly with the main function, and has the following responsibilities:

- It owns a board which consists of 64 cells, and all black and white pieces. It sets up (either default or custom) the board (cells and pieces) before the game starts and clears the board when the game ends. During the game, it takes instructions from both players when they want to make a move, moves the piece to the desired location, and reflects the change on text/graphics display. It also throws an exception if the move is illegal by examining the current information stored in the cells and chess pieces.

- It records and detects the current state of the game, and interacts with the text display when a situation occurs. Specifically, it records the current scores of the two players and prints the scores when the game ends; it detects check, checkmate, stalemate, and returns the state information to main function which can be passed to the command line.

- It uses a stack to record the name and No. of the piece moved in each turn, and the piece being captured if any. When undo command is issued, it pops the record of that piece previously moved and recovers the board to the previous state.

### class Player

The Player class has a pointer to Move. During the game, it provides an interface to take in command of moves (if the Player IS-A Human), or generates a legal move (if the Player IS-A Computer). For the move to actually take effect, the Player has to pass the information of the movement to Move by calling Move::move().

### class Piece

A Piece stores the information of its own identity (what colour and what kind) and its current information in the game, including its current information and the position of the cells it can move to (nextPos). Every subclass of Piece has its own method to detect and update the cells it can legally move to in the next move, based on its type and colour. Every time it finds a such cell, except for updating its own available moves, it will also ask the cell to record its name (push back to “other”) as the one of the pieces that can move to this cell. Moreover, it also records the positions it has moved to. In the case of undo, once Move has found the piece that’s moved in last step, it can check the last position of the piece in historyPos and pop out the last position after calling undo..

**class Cell**

A Cell stores the information of its own location, the chess piece that currently resides in it, and a vector of pieces (name and their unique identifier) that can move to this cell or will be able to move to this cell once the state of the cell changes. When the state of the cell has changed (a piece enters or leaves), it notifies all pieces in that vector to re-detect the cells they can move to, and the cell will also update its own list of pieces.

**Check if a move is legal**

A move is legal if
- the start and end position exists on board
- the start position has the piece to be moved in it, and the end position is unoccupied or is occupied by a piece of the opposite colour
- the end position exists in the nextPos of the piece in the start position.
- the colour of the piece has the same colour of the player that moves it
- it does not put King in check
- if King is already in check, the move makes King avoid being captured

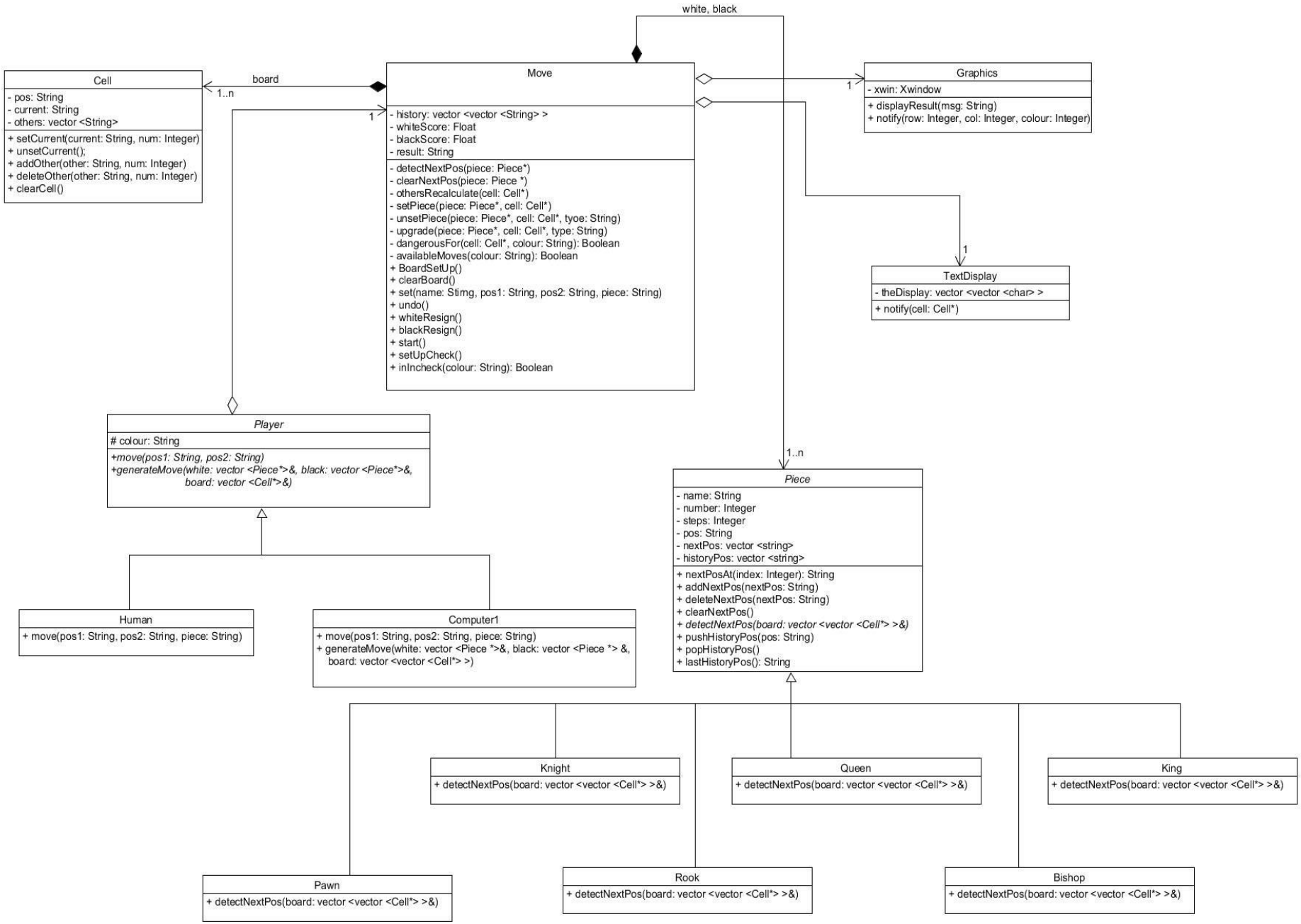
All of the conditions are checked by class Move before a move takes place.

## Difference between plan and actual design

In plan of attack, we planned to implement the Observer Design Pattern with both of class Cell and class Piece as subclass of abstract class Subject and Observer. Pieces and Cells are both subject and observers. Each piece has a vector of cell observers, which is a collection of cells it can move to. Each cell has a vector of piece observers, which is a collection of pieces that can move to this cell. Whenever a piece is moved, it will trigger the update of the observer list of the cells it can move to, and pieces that can move to those cells. There will be a lot of notify, attach and detach going on at the same time, and since the observer list stored in each cell and piece is a vector of observer pointers, this leads to segmentation faults that are hard to debug. So in our actual design, we still use the idea of our original plan, but now instead of using a vector of pointers in Cell and Piece, we use a vector of strings that uniquely identify each cell and piece to avoid memory issues.

Another major difference is that, in our original design, we had a Board class that owns the Cells, and Player owns all the Pieces. It was difficult for the cells and pieces to interact with each other directly. When we make a move, the Player needs to pass the pointer of the Piece to Board, and then Board to Cell, which is inefficient and breaks encapsulation. Now we have a Move class that owns the Cells (board) and Pieces, and Player simply has a pointer to Move. The Move class can then update the information or state of cells and pieces directly without passing pointers. If the player wants to make a move, it simply needs to pass which piece, the start and end position (all are strings) to Move class by calling its public method, and Move will do the actual move within itself.

## Updated UML


white, black
Cell
- pos: String
- current: String
- others: vector <String>
+ setCurrent(current: String, num: Integer)
+ unsetCurrent();
+ addOther(other: String, num: Integer)
+ deleteOther(other: String, num: Integer)
+ clearCell()
board
1..n
1
Move
- history: vector <vector <String> >
- whiteScore: Float
- blackScore: Float
- result: String
- detectNextPos(piece: Piece*)
- clearNextPos(piece: Piece *)
- othersRecalculate(cell: Cell*)
- setPiece(piece: Piece*, cell: Cell*)
- unsetPiece(piece: Piece*, cell: Cell*, tyoe: String)
- upgrade(piece: Piece*, cell: Cell*, type: String)
- dangerousFor(cell: Cell*, colour: String): Boolean
- availableMoves(colour: String): Boolean
+ BoardSetUp()
+ clearBoard()
+ set(name: Stirng, pos1: String, pos2: String, piece: String)
+ undo()
+ whiteResign()
+ blackResign()
+ start()
+ setUpCheck()
+ inIncheck(colour: String): Boolean
1
Graphics
- xwin: Xwindow
+ displayResult(msg: String)
+ notify(row: Integer, col: Integer, colour: Integer)
1
TextDisplay
- theDisplay: vector <vector <char> >
+ notify(cell: Cell*)
Player
# colour: String
+move(pos1: String, pos2: String)
+generateMove(white: vector <Piece*>&, black: vector <Piece*>&, board: vector <Cell*>&)
1..n
Piece
- name: String
- number: Integer
- steps: Integer
- pos: String
- nextPos: vector <string>
- historyPos: vector <string>
+ nextPosAt(index: Integer): String
+ addNextPos(nextPos: String)
+ deleteNextPos(nextPos: String)
+ clearNextPos()
+ detectNextPos(board: vector <vector <Cell*> >&)
+ pushHistoryPos(pos: String)
+ popHistoryPos()
+ lastHistoryPos(): String
Human
+ move(pos1: String, pos2: String, piece: String)
Computer1
+ move(pos1: String, pos2: String, piece: String)
+ generateMove(white: vector <Piece *>&, black: vector <Piece *> &, board: vector <vector <Cell*> >)
Knight
+ detectNextPos(board: vector <vector <Cell*> >&)
Queen
+ detectNextPos(board: vector <vector <Cell*> >&)
King
+ detectNextPos(board: vector <vector <Cell*> >&)
Pawn
+ detectNextPos(board: vector <vector <Cell*> >&)
Rook
+ detectNextPos(board: vector <vector <Cell*> >&)
Bishop
+ detectNextPos(board: vector <vector <Cell*> >&)

## Techniques used to solve design challenges

### Castling

In class Piece, the field “steps” counts the number of steps that the piece has taken so far. If the King and Rook has never moved, then castling is legal.

### En Passant

The most challenging part of implementing en passant is to check if the piece being captured has moved two squares forward. This motivates us to implement the undo feature (details in extra credit features). With undo, we are able to tell if the piece to be captured was moved in the last turn and if it has moved exactly two squares forward in that move. If yes then en passant is legal.

### Upgrade

Upgrade is triggered when a pawn goes to the last row or first row of the board. If the type of piece the pawn would be upgraded to is not provided, an exception would be thrown. Upgrade is implemented in the similar way as being captured, except that in this case the pawn is “captured” by a newly initialized piece with the same colour.

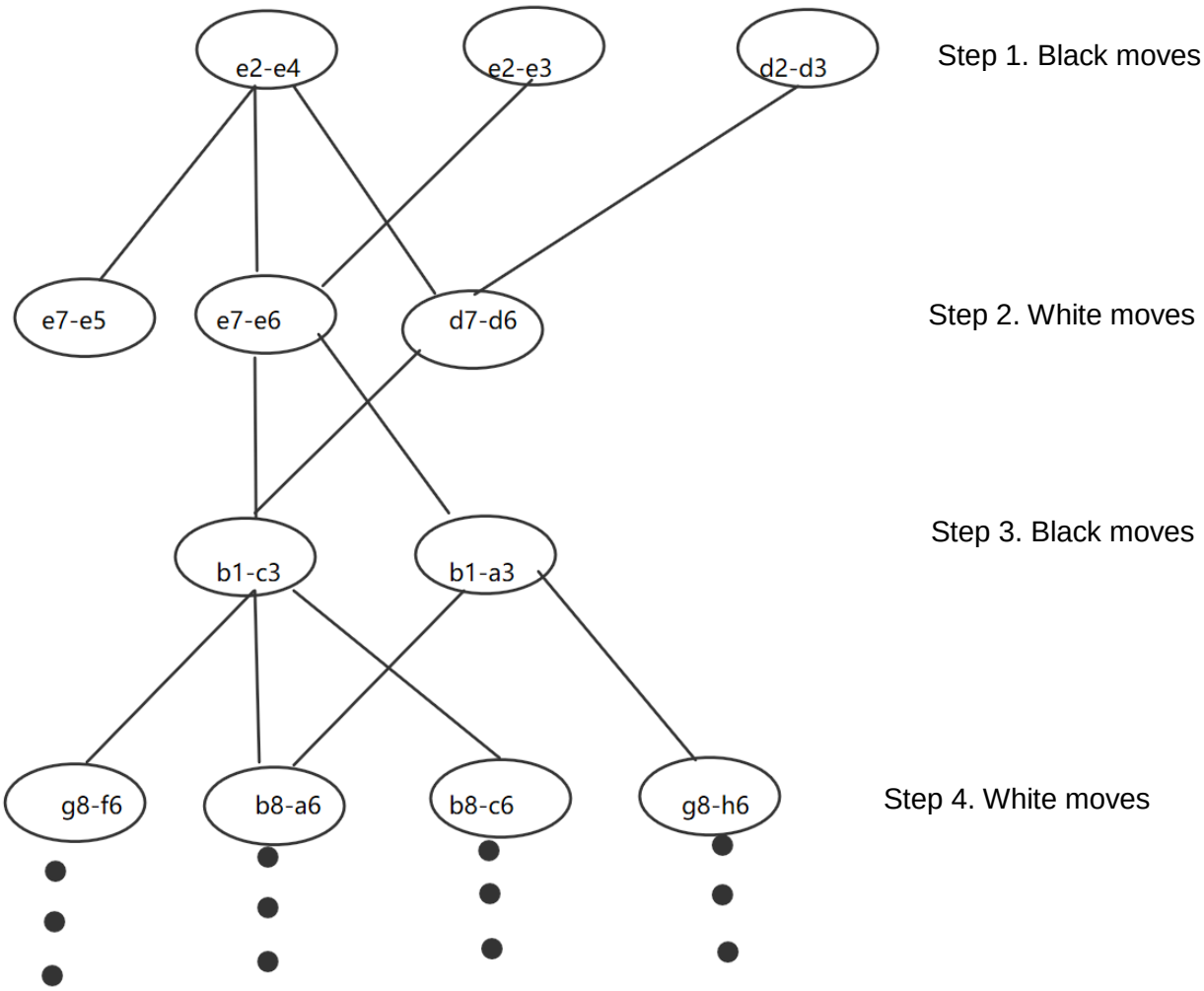
## Resilience to Change

In our design, the class Move is responsible for moving pieces only and move strategies are provided by players using visitor pattern. If for example there is a change in how different levels of computer players should play, we can simply change the way of generating legal moves (Computer::generateMove()) of the computer.

## Answers to Questions

1. Implement a book of standard openings

In the following chart, we use a network to summarize the typical first few moves from various standard openings. The network is implemented as a linked list and pre-stored as a field within the Move class.




The network has n levels, where n is the length of the longest series of standard moves available. The nodes in each level are the recommended standard steps for a player. Child nodes for a node represent the moves recommended for the other player to make in the next step, given this node is taken as the current step. If the other player chooses to not take any of the steps in the child nodes, they will exit this network, and the standard openings will no longer apply. Computer players will always choose standard moves from this network, until the network is exhausted or its opponent exits the network.

The Move class has a pointer that points to the node a player has just made. If the other player chooses to take a step from the child nodes, the pointer will be updated to pointing to the node that has been chosen. If the other player chooses to not take any of the steps in the child nodes, the pointer is set to nullptr which suggests that they have exit the standard opening.

2. Implement unlimited undos
See **Extra Credit Features.**

3. Implement four-handed chess game
If a four-handed game is requested, in main we would need to create another two Players. Within the Move class, we need to change the size of board to accommodate four players, and add another two vectors to store the pieces of those players. The way that pieces finds

the cells it can move to also needs to change, e.g. a Queen now needs to check the cells that are more than eight squares away.

## Extra Credit Features

### Undo

In general, our idea is that each Piece keeps track of the positions it has moved to, and within Move class, it keeps track of the pieces that has been moved in sequential order.

Each Player has a private field “historyPos” which is a stack implemented as a vector of strings. Each string is a position it has moved to. The Move class has a stack to record the name and No. of the piece moved in each turn, and the piece being captured if any. When undo command is issued, Move finds the piece that was moved in last step, checks the last position of the piece in its historyPos, puts it back to its previous position, and finally the last item of the two stacks are poped.

undo is implemented to deal with 3 different cases:

- Common Case:

Pop a vector from history inside class Move. Look up information of the piece in the first field of the vector, recovering the piece by setting it on its previous position, which can be found by looking up the last element in historyPos in class Piece.

- Castling:

If the piece about to be undone is a King which has moved 2 steps to the left or right, pop out another record from history to undo the rook.

- Captured or Upgrade:

If the second field of the history record popped out is not an empty string, recover that piece in the same way.

## Final Questions

The most important lesson we have learned in this project is that, when writing large programs as a team, we must be very careful at the design stage and make sure that every method we write is feasible, and stick to our design once we start to implement it. Otherwise we would encounter more and more problems in implementation stage, and would have to make changes to the original design, and constantly modify the original design will cause many potential problems.

If we had the chance to start over, we would implement the core part of the program and do thorough testing before adding any complicated features. Otherwise, the program would be too difficult to debug if any problems occur. Also, a design with low coupling is essential on the stage of debugging.