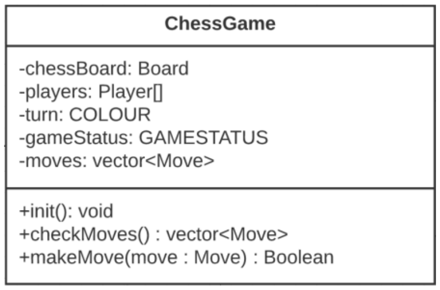
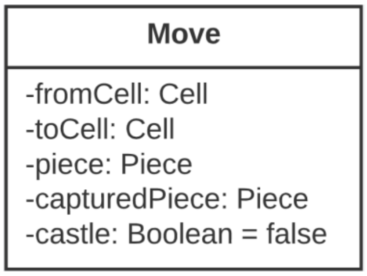
Chess Plan - Leo Feng, Richard Yuan, Rick Lyu

Introduction

Our group consists of three members, and we will be creating a Chess game. We have laid out the general structure of our program in UML in an object-oriented approach. We plan to minimize coupling so that we may each work on distinct parts of our program. Additionally, we will use version control software to make changes to the program simultaneously. In this plan we will describe the design of each class, estimated completion dates, potential extra features, and answers to the questions from the specifications.

Design





The ChessGame Class will control the flow of the program.

The ChessGame Class contains the fields:

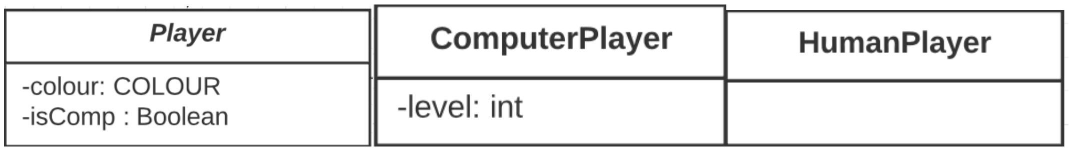
- chessBoard is of type Board and will represent the cells of the chess board. It will also act as the concrete subject of our Observer Pattern
- players will be an array of the Player type. This is to distinguish computer and human players as well as the player's colour.
- turn and gameStatus will be enumerations to represent the state of the game.
- moves will be a vector of the Move type to record made moves (initially empty).

The ChessGame Class contains the functions:

- init() which will be called from the main function each time a new game is played. This function sets up and initializes the board.
- checkMoves() will return a vector of all valid moves (packaged in Move class).
- makeMove() will check if a move is valid and add it to the moves vector if it is. Returns true if the move was made successfully, false otherwise. Will check and update gameStatus after the move is made.

The Move class contains the information of a made move stored in the fields:

- fromCell and toCell represent the cells the piece will be moving from and to.
- piece will be the piece on the fromCell (initially).
- capturedPiece will be the piece on the toCell (initially) if it exists.
- castle will be a boolean that shows if it is a castling move.



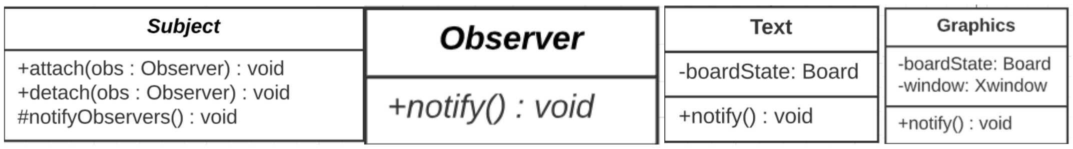
The abstract Player Class has two subclasses: the ComputerPlayer and HumanPlayer.

The Player Class contains two fields:

- colour is an enumeration, which numerically represents black and white.
- isComp is of type Boolean. True, if the player is a computer player; false otherwise.

The subclass ComputerPlayer Class has only one field:

- level is of type int that represents the four different difficulties of the computer player. Level 1 is the lowest and Level 4 is the highest.



This is the Observer design pattern. The Observer Class is an abstract class with two concrete subclasses, the Text and Graphics Classes. The Subject is also an abstract class with a concrete subclass, the Board Class which will update the state of the Board to display. The Subject Class has an aggregate relationship to the Observer Class.

The Observer Class has no field but an abstract method:

- notify() is an abstract method that will be implemented by its concrete subclasses.

The Subject Class has no field but three methods:

- attach() is used to attach the Text and Graphics to the Board.
- detach() is used to detach the Text and Graphics from the Board.
- notifyObservers() is protected and called by the Text and Graphics.

The Text Class is a concrete observer (subclass of observer). It has a field:

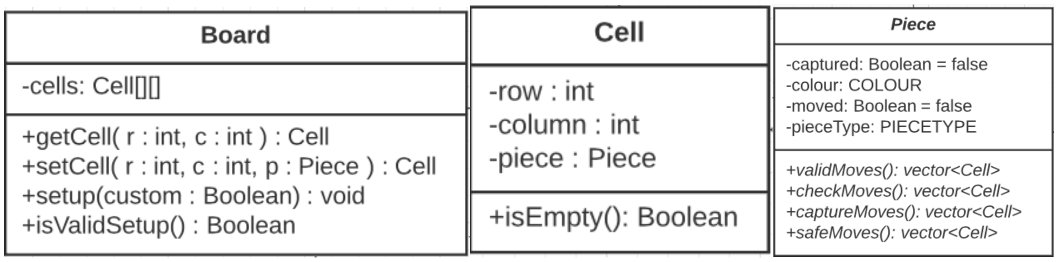
- boardState has a type Board stating the current state of the board.

The Graphic Class is also a concrete observer (subclass of observer). It has two fields:

- boardState has a type Board stating the current state of the board.
- window has type Xwindow. It will be used to show the game in a separate window.

Both the Text and Graphics classes have a virtual method:

- notify() is called when the state of the Board changes. It returns nothing, but it produces notification of the new state.


Board
-cells: Cell[][]
+getCell(r : int, c : int) : Cell
+setCell(r : int, c : int, p : Piece) : Cell
+setup(custom : Boolean) : void
+isValidSetup() : Boolean
Cell
-row : int
-column : int
-piece : Piece
+isEmpty(): Boolean
Piece
-captured: Boolean = false
-colour: COLOUR
-moved: Boolean = false
-pieceType: PIECETYPE
+validMoves(): vector<Cell>
+checkMoves(): vector<Cell>
+captureMoves(): vector<Cell>
+safeMoves(): vector<Cell>

The Board Class will control all the cells on the board (8x8 grid represented as a 2d array)

The Board Class contains the fields:

- cells is a 2D array that stores all the cells of the board.

The Board Class contains the methods:

- getCell() will get a single cell on the board based on the row/column.
- setCell() will set a single cell on the board with a piece based on the row/column.
- setup() will check if it is a custom board. If it is true, then it creates an empty board. If it is false, it will create a normal board with pieces.
- IsValidSetup() will check if the setup is valid according to the specifications.

The Cell Class contains information on an individual cell on the board (64 Cells/Board).

The Cell Class contains the fields:

- row is an int and represents the row location of the cell.
- column is an int and represents the column location of the cell.
- piece is of Piece type and represents the piece that is on the cell.

The Cell Class contains the method:

- isEmpty() checks if the cell has a piece or not. Returns true if the is not a piece and returns false if there is a piece.

The Piece Class is an abstract class that contains 6 subclasses for each unique piece.

It contains the fields:

- captured is of type Boolean that is true if it has been captured, false otherwise.
- colour and pieceType are enumerations that contain information of the piece.
- moved is of type Boolean that is true if the piece has been moved, false otherwise. This important for castling, first pawn move, and en passant.

It contains the abstract methods:

- validMoves() returns a vector of all the possible moves for a piece.
- checkMoves() returns a vector of all the possible check moves for a piece.
- captureMoves() returns a vector of all the possible capture moves for a piece.
- safeMoves() returns a vector of all the possible safe moves (avoids capture).

Each of these methods will be implemented within its subclass (Pawn, Knight, Bishop, Rook, Queen, King). The King subclass will have a method to check for castling moves.

<u>Schedule</u>

Task	**Date**
Main function to act as a test-harness for input	Nov 30-Dec 1
Header files for each class	Dec 2-Dec 3
Implementation of classes	Dec 4-Dec 5
Combining our implementations	Dec 6-Dec 7
Working program with all base specifications	Dec 8
Testing	Dec 9-14
Write demo.pdf, design.pdf, uml-final.pdf	Dec 12-14
Submit	Dec 15
Adding extra features	If completed before

<u>Tasks</u>

Leo: Create test harness, create header/implementation for ChessGame, Board, Cell classes, and write demo.pdf.

Richard: Create header and implementation for Player and classes for Observer Pattern and write design.pdf

Rick: Create header/implementation for Pieces and its subclasses, and write uml-final.pdf

<u>Extra Features</u>

Time permitting, we will try to include the following bonus features:

- Undo Feature
    - We already have a vector that stores all the moves so we will need a function (in our Board Class) to update our Board of cells
- Choose Pawn Promotion
- Level 4 CPU
- Show captured pieces
    - Within our Observer Pattern, we will need an extra field in our concrete subject of captured pieces.

<u>Questions</u>

Question 1 (Chess Manual):

- We could create an additional text-based command that would print out various strategies/openings.
- This can act like the man command in the Linux terminal.
- Instead of displaying everything at once, there could be a menu to select which article the user wants to read. This can all be done in the console output.

Question 2 (Undo feature):

Add a function in Board, undoMove()

- If an undo is requested, we pop the last element of the moves vector and pass it into this undoMove() function that will change the state of the board.
- Then, we redisplay the board through our Observer Pattern.
- Pops the element while the vector is not empty.

Question 3 (4-player chess):

- We would have to change board dimensions and turn order progression.
- If we write our program well, we will not have to change much of the logic concerning piece interaction (moving, capturing, castling, etc.).
- We would have to implement a system for teams and how one team would win. i.e., change checkmate logic.