Chess: Plan of Attack

Dhron Joshi(d8joshi), Haowei Guo(h48guo), and Rishabh Malhotra(r22malho)

July 15, 2016

1 Intro

The intent of this project is to apply the object oriented techniques learned in class over the term and to apply the best software engineering practices and the appropriate design patterns to making a chess game in C++. Our group decided to implement the chess game because we determined that we were familiar with the design patterns and conceptual knowledge needed to implement this game. We were also interested in the AI portion and doing research for the 4th level AI that is requried for this project.

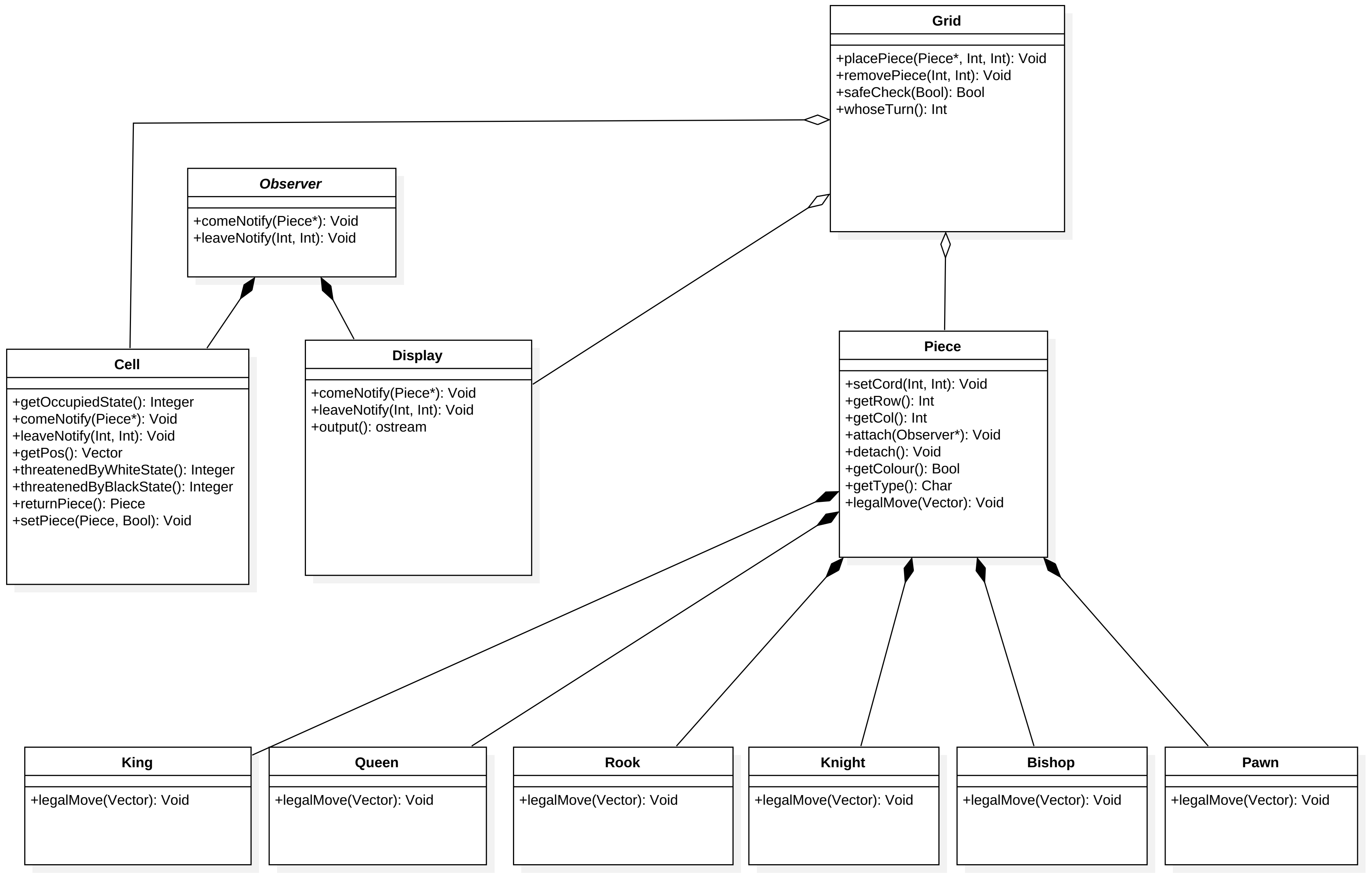
2 Class Relationships & UML

A brief descriptions of our classes:

- Observer
  - An abstract class responsible for providing a pure virtual function to controlling the notifications and communications between its derived classes.
- Display
  - A derived class of Observer class to control how the chess board appears on the screen. This gets notified of changes to the board and also changes accordingly.
- Cell
  - A derived class of the Observer class. It is responsible for notifying the Display class of changes made to the current cell in question and notifying the other significant cells during the player/computer movement of the pieces.
- Grid
  - The grid class keeps track of the game's movements and the player's turns. It has an aggregation relationship with the cell class and other methods to check the validity of the game.
- Piece
  - The piece class oversees all the pieces and stores information about any particular piece's properties; coordinates, colour, legal moves, and keeps track of it's observers.

A UML diagram is also attached with this document to showcase the relationships between each of these classes and their methods.

Model::Main


Grid
+placePiece(Piece*, Int, Int): Void
+removePiece(Int, Int): Void
+safeCheck(Bool): Bool
+whoseTurn(): Int
Observer
+comeNotify(Piece*): Void
+leaveNotify(Int, Int): Void
Cell
+getOccupiedState(): Integer
+comeNotify(Piece*): Void
+leaveNotify(Int, Int): Void
+getPos(): Vector
+threatenedByWhiteState(): Integer
+threatenedByBlackState(): Integer
+returnPiece(): Piece
+setPiece(Piece, Bool): Void
Display
+comeNotify(Piece*): Void
+leaveNotify(Int, Int): Void
+output(): ostream
Piece
+setCord(Int, Int): Void
+getRow(): Int
+getCol(): Int
+attach(Observer*): Void
+detach(): Void
+getColour(): Bool
+getType(): Char
+legalMove(Vector): Void
King
+legalMove(Vector): Void
Queen
+legalMove(Vector): Void
Rook
+legalMove(Vector): Void
Knight
+legalMove(Vector): Void
Bishop
+legalMove(Vector): Void
Pawn
+legalMove(Vector): Void

3 Project Breakdown & Work Distribution

The work has been divided up so that each member has classes that they are responsible for. This includes documentation, implementation, and testing of the classes that they are assigned. This way, each member is an expert on the class that they implement and can ensure its functionality. In theory, if every class is implemented correctly, the program should work seamlessly together.

We all contributed to making the UML, and the design and functionality of the classes and subclasses of this program. Haowei is responsible for the implementation of each of the piece subclasses and the Piece class. Rishabh is responsible for implementing the Grid and Display classes, while Dhron is responsible for the Cell class. We intend to work by implementing all of the core classes first such as Cell, Grid, Display, Piece, and all subclasses of Piece, and develop a main function in which we can interact with our program; this will enable us to do testing of the core functionality while we figure out ways to implement some of the more difficult functionality like the 'computer' player and unique chess moves like castling and en passant.

After the core functionality is completed, we will focus on programming the AI 'computer', which has four levels of sophistication. We will continuously test the program during its development to make cure that each class is doing what its supposed to do, and more thoroughly once we finish the programming portion. The first three levels just seem like variations of checking specific situations of the pieces while the game is being played which seems doable, however we anticipate the 4th level of sophistication will take some time as it is more advanced to implement.

4 Estimated Goals & Completion Dates

Friday July 15th - Finish core functionality, and basic testing.
Saturday July 16th - Start working on and finish special move cases (rook, pawn).
Sunday July 17th - Finish display functionality, with windows graphics and start thinking about implementing the AI.
Monday July 18th - Work on levels 1,2,3 of the AI and finish.
Tuesday July 19th - Try looking for ways to implement a more sophisticated AI and begin development.
Wednesday July 20th - Work on and finish 4th level AI, and begin final testing phase.
Thursday July 21st - Complete final testing phase and prepare for presentation session.
Friday July 22nd - Provided the program is working, check all documentation, and run through program one last time.

We want to finish the game without the AI by the weekend because we anticipate that implementing the AI will be the most challenging and the most error-prone part of our program. We want to leave at least a week to work on the AI alone so we have plenty of time to make mistakes and fix them before the due date.

5 Answers to Provided Questions

Question: 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. Although you are not required to support this, discuss how you would implement a book of standard openings if required.

This could be implemented using a map where instead of keys, we have an opponent's move, and for values, we have our response; and feed those values into our interpreter. Since the opening moves are standardized, we only have to check a finite number of possible outcomes and do the appropriate response move. Essentially find the corresponding opponents move in our map, and let our player use the value as the response move, and continue doing this until however many moves into the game.

Question: How would you implement a feature that would allow a player to undo his/her last move? What about an unlimited number of undos?

Use a Stack data structure to keep track of the coordinates that a piece moves to. Every time a piece is moved, it's movement is pushed onto the stack, and every time a player wants to undo, the movement is popped off the stack and the piece moves to its previous position. This would allow for unlimited player undoes. If we were to implement a single undo feature, we could only keep track of the previous move and revert the piece to that location only.

Question: Variations on chess abound. For example, four-handed chess is a variant that is played by four players (search for it!). Outline the changes that would be necessary to make your program into a four-handed chess game.

We would first have to design a way to accommodate two more different colours to represent the four other players and make a sequence to cycle through all the four players' turns. The rules for four player chess are the same as regular chess so there is no need to alter any of the checks or functionality that we already have. As for forming teams, if it is not a free-for-all, we would treat two colours as the same colour so they can work together to defeat the other two colours without having the option to attack their teammate. We would also add in a rule that when a player of a specific colour is in checkmate, we consider that player 'dead' and skip their turn for the duration of the game. We would also change the win condition so that if both players of the same team have been check-mated, then there is a decisive win.