CS 246 Final project Chess Documentation

Intro

Chess is a board game, played on a board of 64 squares. There are two players, one is in control of white pieces, aka the white player and the other is in control of black pieces, aka the black player. There are six kinds of pieces in total: King, Queen, Bishop, Rook, Knight and Pawn. Players take turn to move and the goal is to trap the enemy king so that it cannot avoid being captured.

Overview

- Class Cell

Data Type	Field name	Description
share_ptr<Piece>	piece	piece on the cell
int	row	row number of cell on the board
int	col	column number of cell the board
Colour	colour	colour of cell on the board
Board*	board	the board that the cell stays on

  - Important Methods
    - notifyIn: notifies the list of observers when a new piece comes
    - notifyOut: notifies the list of observers when an old piece is removed.
    - setPiece: set the given Piece on the cell
    - hasObserver: returns true if there exists a Piece that can capture the piece on the cell
    - underattack: returns true if the piece on the cell is potentially to be captured.
- Class Piece

Data Type	Field name	Description
Cell*	location	the cell that the piece stays on
Colour	colour	White/black: the colour of the piece, indicating which player
int	index	1,2,3,4,5,6: indicates the type of the piece (see chart below)
bool	valid	true if the piece is placed on a cell, false otherwise

  - Subclass:
    - King
    - Queen
    - Bishop
    - Rook
    - Knight
    - Pawn

Details about Index:

PieceType	King	Queen	Bishop	Rook	Knight	Pawn
Index	1	2	3	4	5	6

  - Important Methods
    - changeLocation: modifies the location on the piece when the piece needs to be moved
    - moveIn: set the cell on the piece

- moveOut: unsets the cell on the piece

- Class Board

Data Type	Field name	Description
std::vector<std::vector<Cell>>	board	contains all cells on the board
TextDisplay*	td	textdisplay pointer
Observer<Info>*	ob	observer pointer
bool	whiteCheck	True if White is in check, false ow
bool	blackCheck	True if Black is in check, false ow
bool	whiteCheckMate	True if white king is in checkmate, false ow
bool	blackCheckMate	True if black king is in checkmate, false ow
share_ptr<Piece>	blackKing	The piece of the black king
share_ptr<Piece>	whiteKing	The piece of the white king

  - Important Methods
    - validate: check whether the setup is successful and throws error message when not.

- Class Player

Data Type	Field name	Description
Colour	colour	the colour of chess that the play controls
Float	grade	the grade of the player
Vector<shared_ptr<Piece>>	pieces	the pieces that the player has now and before

  - Subclass:
    - Human: human player
    - ComputerLevel1: computer player level 1
    - ComputerLevel2: computer player level 2
    - ComputerLevel3: computer player level 3
    - ComputerLevel4: computer player level 4

- Class Game

Data Type	Field name	Description
shared_ptr<Player>	blackPlayer	the black Player on the game
shared_ptr<Player>	whitePlayer	the white Player on the game
Board	board	the board that the game is on
Vector<Move>	previousMoves	Stores previous Moves

  - Important Methods
    - set: calls set method from Board class
    - unset: calls unset method from Board class
    - undo: undo the latest move
    - checkState: returns true if one of the players is under check
    - endGame: ends the game
    - resign: add score to the player who does not resign.
    - humanMove: calls Move method under Human class
    - computerMove: calls Move method in corresponding ComputerLevel Class

- Class Move

Data Type	Field name	Description
shared_ptr<Piece>	lastMovedPiece	stores piece that moves most recent
int	start_r	the row that the piece moves from
int	start_c	the column that the piece moves from
int	end_r	the row that the piece moves to
int	end_c	the column that the piece moves to
bool	capturing	True if the Move captures a Piece
shared_ptr<Piece>	captured	the captured piece

- Class TextDisplay

Data Type	Field name	Description
Vector<vector<char>>	textDisplay	Consists vector of vector of char

- Class Graphics Display

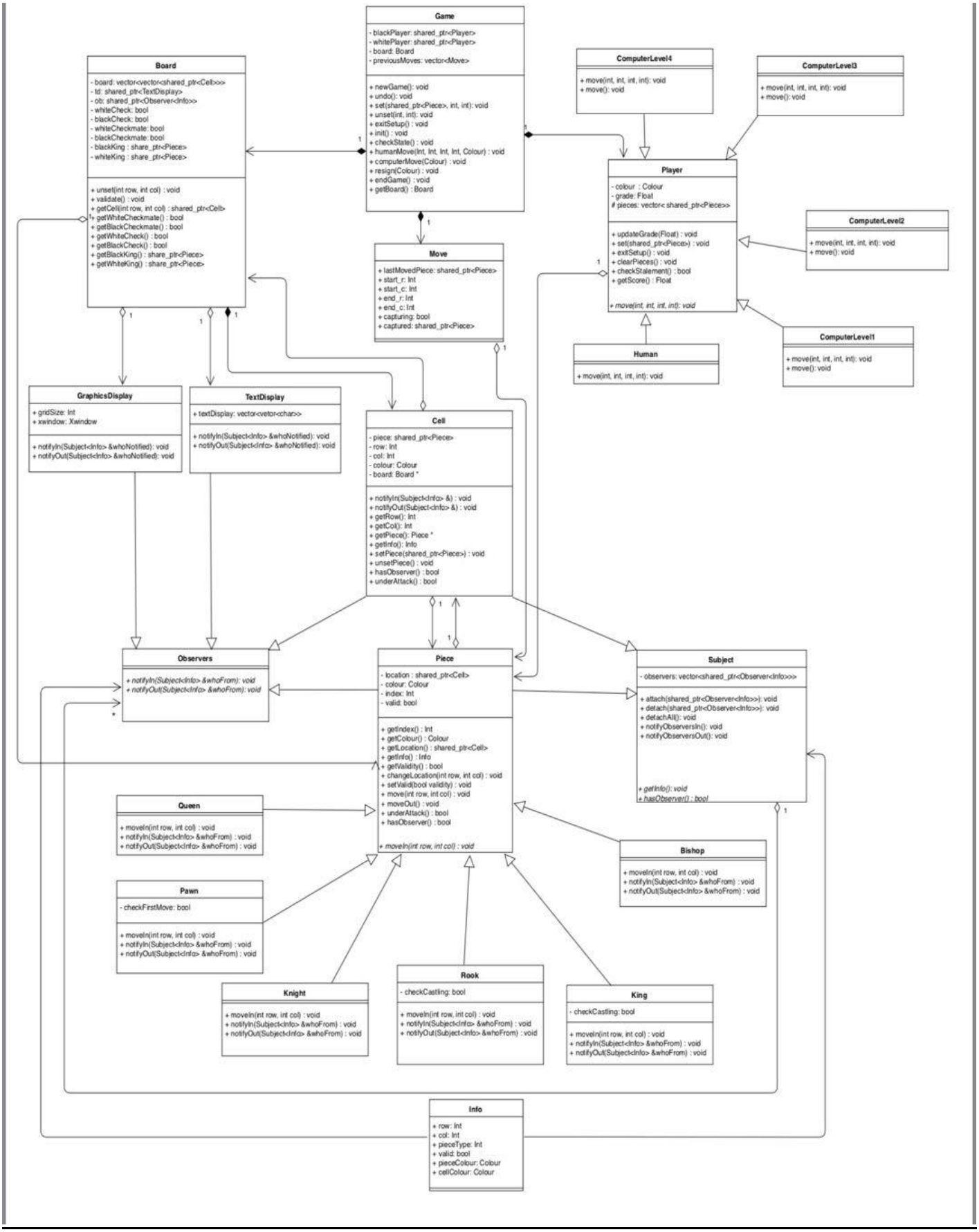
Data Type	Field name	Description
int	gridSize	the size of the each grid
Xwindow	xwindow	a window

- Struct Info

Data Type	Field name	Description
int	row	Gives the row num
int	col	Gives the col num
int	pieceType	1,2,3,4,5,6 (refer to chart in Piece)
bool	valid	True if the piece is set on the cell, false otherwise
pieceColour	Colour	The colour of th piece, indicates the colour of the player
cellColour	Colour	The colour of the cell on the board, by default

We have four observers: Cell, Piece, TextDisplay, GraphicsDisplay.

Updated UML


Game
- blackPlayer: shared_ptr<Player>
- whitePlayer: shared_ptr<Player>
- board: Board
- previousMoves: vector<Move>
+ newGame(): void
+ undo(): void
+ set(shared_ptr<Piece>, int, int): void
+ unset(int, int): void
+ exitSetup() : void
+ init() : void
+ checkState() : void
+ humanMove(Int, Int, Int, Int, Colour) : void
+ computerMove(Colour) : void
+ resign(Colour) : void
+ endGame() : void
+ getBoard() : Board
Board
- board: vector<vector<shared_ptr<Cell>>>
- td: shared_ptr<TextDisplay>
- ob: shared_ptr<Observer<Info>>
- whiteCheck: bool
- blackCheck: bool
- whiteCheckmate: bool
- blackCheckmate: bool
- blackKing : share_ptr<Piece>
- whiteKing : share_ptr<Piece>
+ unset(int row, int col) : void
+ validate() : void
+ getCell(int row, int col) : shared_ptr<Cell>
+ getWhiteCheckmate() : bool
+ getBlackCheckmate() : bool
+ getWhiteCheck() : bool
+ getBlackCheck() : bool
+ getBlackKing() : share_ptr<Piece>
+ getWhiteKing() : share_ptr<Piece>
ComputerLevel4
+ move(int, int, int, int): void
+ move(): void
ComputerLevel3
+ move(int, int, int, int): void
+ move(): void
Player
- colour : Colour
- grade: Float
pieces: vector< shared_ptr<Piece>>
+ updateGrade(Float) : void
+ set(shared_ptr<Piece>) : void
+ exitSetup() : void
+ clearPieces() : void
+ checkStalement() : bool
+ getScore() : Float
+ move(int, int, int, int): void
ComputerLevel2
+ move(int, int, int, int): void
+ move(): void
Move
+ lastMovedPiece: shared_ptr<Piece>
+ start_r: Int
+ start_c: Int
+ end_r: Int
+ end_c: Int
+ capturing: bool
+ captured: shared_ptr<Piece>
Human
+ move(int, int, int, int): void
ComputerLevel1
+ move(int, int, int, int): void
+ move(): void
GraphicsDisplay
+ gridSize: Int
+ xwindow: Xwindow
+ notifyIn(Subject<Info> &whoNotified): void
+ notifyOut(Subject<Info> &whoNotified): void
TextDisplay
+ textDisplay: vector<vetor<char>>
+ notifyIn(Subject<Info> &whoNotified): void
+ notifyOut(Subject<Info> &whoNotified): void
Cell
- piece: shared_ptr<Piece>
- row: Int
- col: Int
- colour: Colour
- board: Board *
+ notifyIn(Subject<Info> &) : void
+ notifyOut(Subject<Info> &) : void
+ getRow(): Int
+ getCol(): Int
+ getPiece(): Piece *
+ getInfo(): Info
+ setPiece(shared_ptr<Piece>) : void
+ unsetPiece() : void
+ hasObserver() : bool
+ underAttack() : bool
Observers
+ notifyIn(Subject<Info> &whoFrom): void
+ notifyOut(Subject<Info> &whoFrom): void
Piece
- location : shared_ptr<Cell>
- colour: Colour
- index: Int
- valid: bool
+ getIndex() : Int
+ getColour() : Colour
+ getLocation() : shared_ptr<Cell>
+ getInfo() : Info
+ getValidity() : bool
+ changeLocation(int row, int col) : void
+ setValid(bool validity) : void
+ move(int row, int col) : void
+ moveOut() : void
+ underAttack() : bool
+ hasObserver() : bool
+ moveIn(int row, int col) : void
Subject
- observers: vector<shared_ptr<Observer<Info>>>
+ attach(shared_ptr<Observer<Info>>): void
+ detach(shared_ptr<Observer<Info>>): void
+ detachAll(): void
+ notifyObserversIn(): void
+ notifyObserversOut(): void
+ getInfo(): void
+ hasObserver() : bool
Queen
+ moveIn(int row, int col) : void
+ notifyIn(Subject<Info> &whoFrom) : void
+ notifyOut(Subject<Info> &whoFrom) : void
Bishop
+ moveIn(int row, int col) : void
+ notifyIn(Subject<Info> &whoFrom) : void
+ notifyOut(Subject<Info> &whoFrom) : void
Pawn
- checkFirstMove: bool
+ moveIn(int row, int col) : void
+ notifyIn(Subject<Info> &whoFrom) : void
+ notifyOut(Subject<Info> &whoFrom) : void
Knight
+ moveIn(int row, int col) : void
+ notifyIn(Subject<Info> &whoFrom) : void
+ notifyOut(Subject<Info> &whoFrom) : void
Rook
- checkCastling: bool
+ moveIn(int row, int col) : void
+ notifyIn(Subject<Info> &whoFrom) : void
+ notifyOut(Subject<Info> &whoFrom) : void
King
- checkCastling: bool
+ moveIn(int row, int col) : void
+ notifyIn(Subject<Info> &whoFrom) : void
+ notifyOut(Subject<Info> &whoFrom) : void
Info
+ row: Int
+ col: Int
+ pieceType: Int
+ valid: bool
+ pieceColour: Colour
+ cellColour: Colour

Design

Compared to UML on DD1, the structure did not change a lot, we make adjustments on the fields and methods under classes. Take board as an example, we realized that instead of having one variable indicating the colour of the player that is in check, setting two separated variables whiteCheck and blackCheck that return Boolean values works better. This is the same for checkmate.

We implement the Game class to combine and connect all the information, instead of setting players (human or computer) on the Board. In UML DD2, we move some functionalities in Board to Game, which make the structure more explicit. Game class also acts as a connection between Board class and Player Class so that they can call method each other. At the same time, we restore all game information like score, endGame and checkState in Game class then we populate the results to the main functions. Another advantage of Game class is that whenever we want to add new player, we simply add one more field in Game, which simplifies the process.

We use the data type shared_ptr <> for our observers: Cell, Piece, TextDisplay, and GraphicsDisplay since it can be auto delated and avoid memory leak. We take advantage of the Observer Pattern to add or remove association between subject and observer at any point. Also, it supports the principle of loosing coupling between objects that interacts each other. TextDisplay and GraphicDisplay observe all the cells on the board, and whenever, the cell has changes, these two classes give updates based on cell's notifications.

We add "valid" variable in Piece class such that it updates the state whether the piece is placed on a cell. In this case, even when a piece is removed, the pieces are still restored in the "Pieces" variable in Player class.

We designed 4 levels of computer and players can choose at the beginning. For ComputerLevel1, it takes valid moves randomly since we have an observer in Cell and if it has observers, we move. For ComputerLevel2, we set some preferences on the random moves like capturing moves by checking the pieceType of the cells that I can attack. For ComputerLevel3, based on the previous two conditions, we have one more feature that prevent ourselves from being captured by the opponent. For ComputerLevel4, the random move is more cautious, and it gives preference to capture the other's pieces under the condition that your own pieces are safe. The strategy is that we check our status first and make decisions.

Also, we split the "notify" function into two parts: "notifyIn" and "notifyOut". Thus, whenever a new piece comes or the original piece is removed, we notify its list of observers respectively.

Resilience to Change

We have designed our modules and classes properly to adapt to changes as quickly as possible. We have multiple helper functions to check each requirement. Take "checkLocation" as an example. This helper function in main takes in a string and returns true if the string represents a valid coordinate and false otherwise. So, next time when we have request on the position, we may simply modify this piece of function. Other possible changes like adding a new player or increasing the number of players can be completed by adding fields in Game class, which is a key feature of our design.

Answers to Questions

1. 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.

   Answer: We will add a map structure for this implementation. Names of standard opening move like Bishop's Opening and Scotch Game are restored as keys. The values of the map structure are made up of corresponding opening move sequences. Sequences are stored in another map structure, in which keys are colors and step numbers such as Step1W, Step1B, Step2W, Step2B…. and values indicate respective moves. These opening moves are fixed and immutable.

2. 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?

   Answer: As shown in UML, we have a Move structure to store all historical moves. For one player to undo his/her last move, we call the undo method in the Game class, and we consider the undo function as a new move. For example, player's latest move is from e3 to f3, our undo conducts a new move from f3 to e3. If a piece is captured in the latest move, we have another undo Capture method to put the captured piece back to its original position. For unlimited number of undos, we use stack to store previous moves and pop off the last move every time we call the undo method. The main idea stays the same in the new UML. We have a field previousMoves in Move which restores all historical Moves.

3. 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.

   Answer: To allow multiple players, we need to implement new initial method to set up different sizes of board. Also, we can modify the Game Class to make it available to attach new players with different colors.

Extra Credit Features

- Undo

  We restored all the historical moves in the field "previousMoves" under class Game so that when it reads in the "undo" command, it calls the "undo" method under Game class. We always keep "previousMoves" updated whenever a piece is captured or a pieced is removed. Therefore, we can undo unlimitedly since all moves are restored safely. More importantly, all the removed pieces are restored in pieces in Player class, and we keep the info updated.

Final Questions

1. What lessons did this project teach you about developing software in teams? If you worked alone, what lessons did you learn about writing large programs?

   Answer: Communication and collaboration should be attached great importance in team development. Even though we share the UML, we still need to sit down together and look into details and relationships between classes and functions. We found it more efficient when we spent more time coding together since problems can be resolved in time. Meanwhile, since we had different schedules, we gave updates on the progress in our group chat. We distributed our work appropriately so that when debugging, members will be in charge of their part correspondingly. This also helps brainstorming the design. Additionally, skill sets sharing is also crucial for team working, group members introduce new techniques that they learn or discover outside the group and we discuss how we can apply the methods to our case.
2. What would you have done differently if you had the chance to start over?

   Answer: If we were able to start over again, we would have started coding a little bit when designing UML since we changed the UML a lot from version 1 to version 2. I don't mean starting code right from beginning. Some ideas turn to be not feasible when we actually code. This may indicate that we should have started earlier than we have planned. Besides that, we all enjoy the process.