University of Waterloo
CS247 Spring 2022
Project Design Document (Chess)

Qirong He, 20814593, q48he

Tanbir Banipal, 20873605, tbanipal

1 Introduction

We decided to do the chess project since both of us are really interested in chess, and always wanted to create our own version of the game, this project provided the perfect opportunity for us to do so. In the following sections, we will be describing to overall structure of our project, how our design solved various design problems, how our design supports changes to the program, and what both of us learned from this project.

2 Overview

Our project consists a total of 16 different classes. We have various inheritance hierarchies within the whole structure. We also implemented the observer design pattern in our project. Here we will be giving a brief overview of our project's structure, with a heavy focus on the more important methods and functionality in each class.

To start off the overview, our project is initialized and started from the backbone 'Game' class. This is where we initialize the play field as well as the two players, using their corresponding classes 'Player' and 'Board'. The 'Player' class is an abstract base class, which has the pure virtual method getNextMove for getting the input/next move from its derived classes, and the pure virtual method getPlayerType for getting the derived player type. It also has methods to check if a player lost due to checkmate, or if the game is a draw due to a stalemate. The 'Human' class is a derived Player, where we implemented the input prompt for human players, and allow them to input their moves, while also checking if their move is valid. The 'Computer' is also a derived player. It calculates the next move based on current list of available moves for its side as well as the opponent, then choose which algorithm to implement depend on the difficulty.

The 'Board' class is the main play field, which contains a grid of 'Cells'. In this class, we have the init method for supporting the setup command for setting up an empty board while checking if the board is valid. We have methods that operates around the 'Cell' class such has nextMove, testMove, and badMove which moves the chess pieces around by calling methods in cell. We have getAllAvailable moves that gets all available moves a black/white player can make using Cell's methods, and getType that gets the information of a cell given the position. When the cout operator is invoked, we will simply call cout on the textDisplay to update the textDisplay, and the GUI updates according to that.

Now to be more specific with our 'Cell' class, each cell has a ChessPiece, which is an abstracted base class for the 6 different types of chess pieces we have. The 'ChessPiece' abstract base class only has 2 method, getAvalibleMove, which get the available move a chess piece could make, and getType, which gets the chess piece type of the current ChessPiece. All 6 derived classes, 'Queen, 'King', 'Pawn', 'Bishop', 'Rook', 'Knight' only has the 2 methods in the abstract base class which they override according to their own rules in the game of chess. For the observer design pattern, instead of using the usual design (i.e the Observer abstract base class, the Subject abstract base class, and their concrete classes), we decided to merge subject within 'Cell', so the 'Cell' class is also a subject at the same time, and has a vector of 'Observers' that it notify about changes that happened.

The 'TextDisplay' class is for providing a text display of our game, which has a vector of characters to represent the current state of the board. It is an observer of cell. When a change has been made, it will be notified, and set the current state of the text display accordingly. The 'GUI' class is also going to update and draw the board and chess pieces according to changes made to the TextDisplay.

To walk through the whole process of an actual game, first we construct the board on the 'game' command input. Then we construct the two players, (either computer or human based on the user input). Once we finished the game initialization, the users could input their moves if there is a human player, and they could move the computer players using the command 'move'. While a move command is received to a human player, our program will first check if that move is a valid chess move, then we check if that will cause the king to be checked (which is also invalid). If the input is indeed a valid move, we tell the board to move the piece, which triggers notify in cells and notifies the textDisplay to update, as well as changing the source and destination cell's status based on the input move. After each move, we will also run a checkmate and stalemate check (based on the list of all available moves the player and its opponent could make by calling the getAllAvalibleMoves method in board). If a player lost (or if there is a draw), we end the game, update the score, and reset everything

If one of the players is a computer and we received a move command from input, we will first get all available moves of the player and its opponent, then based on the level of computer the user inputted, run the corresponding algorithm, and make the move. Checkmate and stalemate are also checked after each move, and if game ends, we will also update the scores and clean things up accordingly.

3 Final UML Diagram

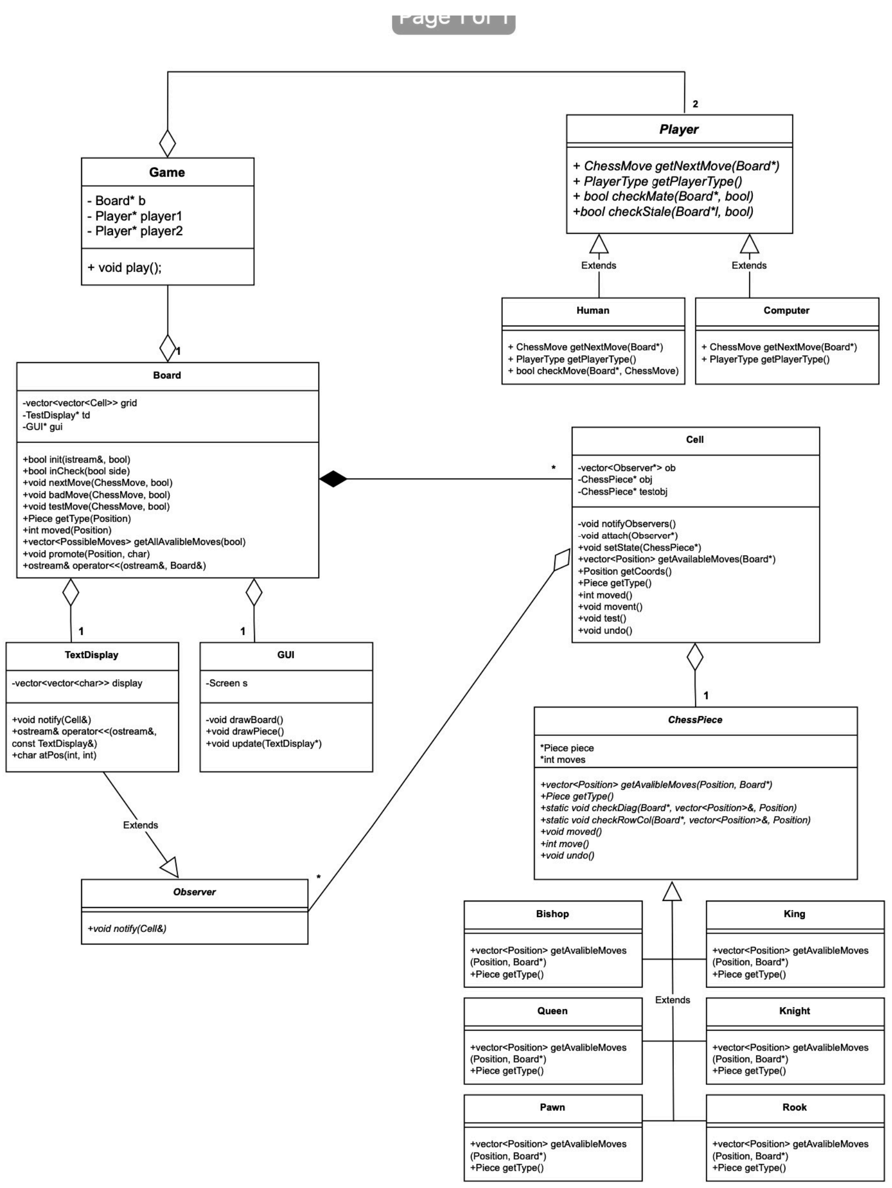
Page 1 of 1
2
Player
+ ChessMove getNextMove(Board*)
+ PlayerType getPlayerType()
+ bool checkMate(Board*, bool)
+bool checkStale(Board*l, bool)
Game
- Board* b
- Player* player1
- Player* player2
+ void play();
Extends
Extends
Human
+ ChessMove getNextMove(Board*)
+ PlayerType getPlayerType()
+ bool checkMove(Board*, ChessMove)
Computer
+ ChessMove getNextMove(Board*)
+ PlayerType getPlayerType()
1
Board
-vector<vector<Cell>> grid
-TestDisplay* td
-GUI* gui
+bool init(istream&, bool)
+bool inCheck(bool side)
+void nextMove(ChessMove, bool)
+void badMove(ChessMove, bool)
+void testMove(ChessMove, bool)
+Piece getType(Position)
+int moved(Position)
+vector<PossibleMoves> getAllAvalibleMoves(bool)
+void promote(Position, char)
+ostream& operator<<(ostream&, Board&)
*
Cell
-vector<Observer*> ob
-ChessPiece* obj
-ChessPiece* testobj
-void notifyObservers()
-void attach(Observer*)
+void setState(ChessPiece*)
+vector<Position> getAvailableMoves(Board*)
+Position getCoords()
+Piece getType()
+int moved()
+void movent()
+void test()
+void undo()
1
1
TextDisplay
-vector<vector<char>> display
+void notify(Cell&)
+ostream& operator<<(ostream&, const TextDisplay&)
+char atPos(int, int)
GUI
-Screen s
-void drawBoard()
+void drawPiece()
+void update(TextDisplay*)
1
ChessPiece
*Piece piece
*int moves
+vector<Position> getAvalibleMoves(Position, Board*)
+Piece getType()
+static void checkDiag(Board*, vector<Position>&, Position)
+static void checkRowCol(Board*, vector<Position>&, Position)
+void moved()
+int move()
+void undo()
Extends
*
Observer
+void notify(Cell&)
Bishop
+vector<Position> getAvalibleMoves (Position, Board*)
+Piece getType()
King
+vector<Position> getAvalibleMoves (Position, Board*)
+Piece getType()
Extends
Queen
+vector<Position> getAvalibleMoves (Position, Board*)
+Piece getType()
Knight
+vector<Position> getAvalibleMoves (Position, Board*)
+Piece getType()
Pawn
+vector<Position> getAvalibleMoves (Position, Board*)
+Piece getType()
Rook
+vector<Position> getAvalibleMoves (Position, Board*)
+Piece getType()

Figure 1: Final UML diagram

4 Design

In order to solve the problem of updating the textDisplay, GUI, and status of cells, we decided to include the observer design pattern we learned in the lectures. We think the observer pattern really solves such problems where you need to update the status of something after a specific change occurred to the subject. It also made our code a lot cleaner by introducing the abstract Observer class and making the textDisplay a concrete observer to avoid having a bunch of methods in textDisplay.

Before finishing the implementation of checking whether a move is valid or not, we wanted to come up with a design that works well with other functionalities, such as checkmate, undo moves. and the computer algorithm. We finally came up with this design of operating everything around vectors of Possible Moves. We defined a PossibleMoves type to be of a struct of a position (the starting point), and a vector of positions (the destination vector that the chess piece at the starting point could reach). By implementing such a design, to check whether a move inputted by the user is valid or not, we only have to get all available moves for that side (if the user is black we get the list of all available moves black player could currently make), and it would be as simple as checking if the user's input move exists in this vector of possible moves. We also created the function inCheck which gets the list of all available moves the opponent has and check if the my King exists in one of the destination positions. If it's in the list , we know that the player is currently being checked. Having such functions make implementing checkmate straightforward as we only need to testMove all pieces into their destinations and call inCheck after each movements. If all of the movements leads to the king being checked, we know there is a checkmate, and a player has lost (of course, we will pass in the colour of the current player as a parameter). This design also helps us with implementing the different levels of algorithms for the computer, since it's going to be really easy to check which move leads to a check or capture, and making the level 1 computer will be as easy as choosing a random element from the list of all available move that computer could currently make.

We also implemented the inheritance hierarchies of Player (Human and Computer), and ChessPiece (Pawn, Rook, King, Queen, Bishop, Knight), as we think this is what makes the most sense in the real world. Since player and chess pieces are actually abstract concepts, you cannot construct them. Instead you could have a rook on the board, or you could have a human player. We defined the pure virtual methods and overrides in each of the abstract base class and derived classes to help with maintaining such structure.

5 Resilience to Change

Since we have our own system of types, for example, we would define the type Piece to be of a pair of PieceType and a boolean (which represents the player's colour), and PieceType is a enum class of the 6 pieces. This works really well if we are making change to how we choose to define a player's side. For example, if instead of 2 players, we have 3 players, then we couldn't use boolean as a representation of player's colour, we need to change the data type of that, and all we need to do is change that type within type.h, instead of going through all the functions that takes the argument and changing them one by one.

Our inheritance hierarchy of ChessPiece combined with the types we created makes it really easy to add or delete a piece type. If we want to delete a Piece type, all we need to do is delete its corresponding derived class under ChessPiece, as well as deleting it from the enum class in types.h, and we are done. If we want to add a piece, we just need to add its corresponding derived class under ChessPiece, finish the getAvailable move method for that piece's class and we don't have to change anything else, since for each chess piece, we only need the available position they could go to in order to integrate these moves into getAllAvailable moves in Board and proceed on with the rest of the operations.

The observer pattern makes it easy to change the specification of the boards. For example, if we want to change the width or the height of the board, we will just have to change the size of the grid in the 'Board' class, and also update the constructor of Board and TextDisplay to specify how we construct the default board and we are done. We don't have to change any of the methods, since all the cells are going to update the changes to textDisplay based on the movements happened on the board.

6 Answers to Questions

1. To implement the book of standard features we would store a vector of vectors that holds the piece information, previous square it was on, and the square it should go to. We would also store if a piece was killed. As long as the players continue to play the game according the most common moves for each standard opening for the first 12 moves, we would display the next recommended move in said opening. If a player deviates from the most common move for each standard opening, the players are told they strayed from the standard opening and the help information no longer prints the next recommended move. The help information is output to std::cout since that is where the players will be inputting information.

2. To implement a feature allowing a player to undo his/her last move, we would store the original position of the piece that made its move along with its current position. We would also store what piece was stored at the current square last if that piece exists. To undo the move, we would move the chess piece back to its position before the move and restore any killed pieces. After that, we would clear the last move data. To undo every move a player made, we store this information into a vector. Upon undoing the move, we will pop the most recent move information.

3. The first change we would have to make is change the board to support the new board design. The second change is we would have to add new colors. In addition, we would have to implement two new turns. Third, we would have to update the algorithms that determine if a move is valid. Fourth, we have to change the conditions in which a game ends. A player getting a checkmate should not result in the game ending unless both partners in a team are checkmated. A checkmate can also be cleared. We would also have to allow pawns to jump over a team mates pawns. We would also have to allow pawns to go in the reverse direction. Upon reaching the back row of the teammate, we would have to reverse the direction of the pawn. The preferable implementation for 4-handed chess, however, is to not do it. In addition, since we are UWaterloo students, we are not sure if we even have 2 other friends to play the game with.

7 Extra Credit Features

We didn't implement any extra credit feature. Although we were very close to finishing up undoMoves

8 Final Questions

1. The lessons this project taught us include the importance of communication, the importance of properly designing the project from the beginning, using git, splitting up the tasks, and the importance of commenting code. Regarding the the importance

of communication, we learned that this is super important to developing software in teams as when we were experiencing broken behavior, we would have to consult with the other member with the expected results of functions they wrote and the reason for the implementation. Failure to communicate issues we were experiencing made us spend entire days trying to figure out why our code wasnt working as expected. With Qirong, when setting up the list of moves a Chess Piece could make, the board function he wanted to use was returning different Chess Pieces than the ones he wanted to use. This was because Tanbir had planned the function behaving differently. Likewise when Tanbir was trying to solve the case of Chess Pieces disappearing, he didnt notice that functions Qirong had implemented modified the board directly, and by extension, the text display and GUI. If we had consulted with each other regarding what we planned on doing with the board and player functions, we would have realized a day earlier that one person planned on the functions in board being used only in the event of making a final move, while the other planned on using the functions in board to test for certain cases. If we had known this was the case, we could have implemented a new board designed exclusively for testing for conditions while the board was the board to use when making moves. Secondly, we learned the importance of planning projects thoroughly from the beginning since we ended up having to make drastic changes to our class structures at the very end. Next, we learned that git is very useful in sharing changes we made. We also discovered early on that we kept running into merge conflicts so we needed to be upfront with each other what files we would modify and for what reason. After that, we learned how to split up tasks into core objectives so we could finish the project on time. We also had to learn how to prioritize tasks so the other team member wouldnt be stuck waiting on features to be implemented before being able to progress. And finally, we learned the importance of commenting code. When we were trying to understand what functions were doing verses their expected behavior, we realized had to consult each other what we though the functions should do. This was ok for the first two days, but eventually we ended up forgetting what we though the functions should do.

2. If we had the chance to start over with this project, we would have spent more time at the beginning planning the design of our project. The reason for this was because even though we already spent time planning our project at the beginning, we did not properly go through what the program should do for all of the functionality at the beginning. The parts that we plan out in the beginning, like the ChessPiece classes and their methods were a breeze to implement and didnt stray too much from the initial design. This was not the case for other classes like Board and GUI. For these two classes, we had to implement new objects and methods that we initially didnt plan for every time we wanted to add new behaviors to to the program. This resulted in the code for these two classes becoming horribly broken and with odd behaviours like chess pieces just disappearing from existence for no good reason. The GUI class was arguably the biggest casualty from us not planning the design of our project in great detail. While planning how to do the graphics output, we had initially planned on

the GUI being an observer of each cell in Board. In addition, we did not perform any research into SDL and when we were implementing our design for the GUI class halfway through the project, it became clear that the entire implementation was a disaster. The GUI wouldnt output any pieces that werent being moved and it would often color the entire board a single color. In the end, we had to hold discussions deciding if we should scrap the entire thing and make a new GUI class that wasnt an observer. In addition, we would have made an attempt to get a third team member. We often felt like we were performing tasks unrelated to our core objectives since we needed functionality that was not a core objective of the other member. In addition, we would have been able to spend more time testing our project and learning about broken behaviors and having the third person investigate the cause of the broken behaviors.

9 Conclusion

In conclusion, we are proud of our work for this project, even if what we wrote may not be the most efficient implementation for what we were planning on implementing. During the course of this project, we were able to reinforce our knowledge of concepts taught in class, like the inheritance hierarchy, the observer pattern, and graphics output. In addition, we were able to further improve our skills regarding team work, and feel like we were able to apply our knowledge in a practical setting. If presented with the opportunity to do this project again, we would most definitely take up the offer.