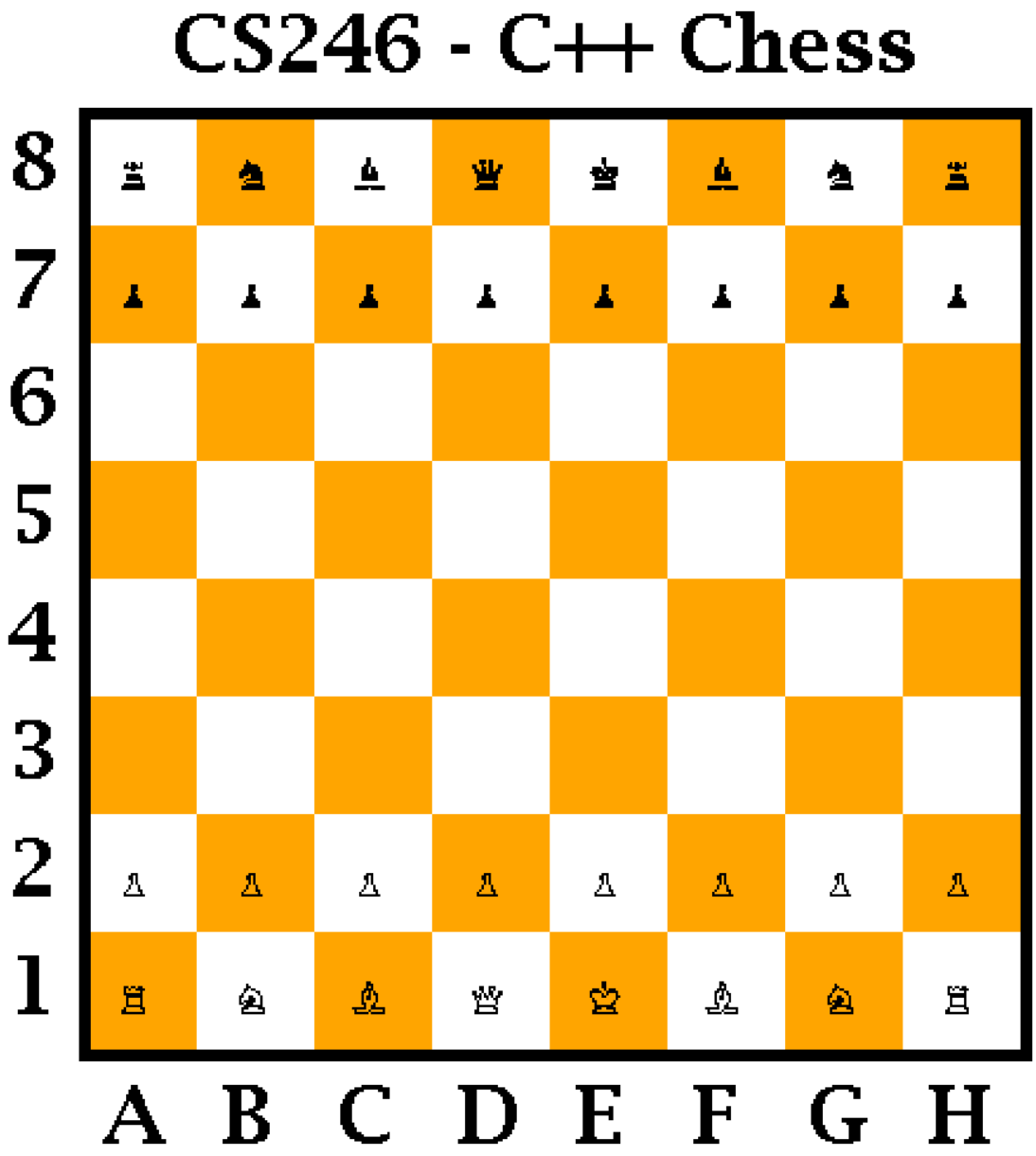


CS246 - Final Project

06.12.2022

Abdullah Shahid, Ariq Ishfar, and Soham Nagi

Introduction

This document outlines the processes behind and the insights gained from implementing a game of chess in C++ as a group project. We will go over a high-level explanation of our program, describe challenges, the adaptability of our implementation, and share our retrospection on this project as a whole.

Going forward into the project, we had the following goals in mind:

1. To produce a working implementation of the game of chess in C++ with accurate logic and checks, such that a full game of chess can be played with human and/or computer players.
2. Provide features as graphical windows or move highlighting to enhance the user experience.
3. Optimise our implementation to be computationally efficient, not reliant on dynamic memory, and adhere to software development standards.
4. Learn and have fun!

Overview

The three underlying classes which form the bulk of the game's logic are:

Board: This stores the board of the current game running, and associated information as the number of moves and half-moves made and which player's turn it is. The class also has methods for detecting checks, attaching and detaching pieces, and constructions from a FEN.

Pieces: This class represents the chess pieces. There is an abstract base class consisting of fields common to all pieces as their colour, type, legal moves for the pieces stored in a vector of indexes, location, and whether or not they have moved.

The derived classes represent the 6 types of chess pieces and one representing an empty square, each overriding the method for updating the legal moves of the piece.

Player: This class represents the players and the means of providing moves. If it's a human player, the class is responsible for inputting the moves from the user. If it's a computer player, then the best move is evaluated and applied on the board. How the best move is evaluated depends on the chosen difficulty.

The overall user experience is supplemented by the following classes:

Game: This is a wrapper class containing the board, pieces, and the players. The game class is fully capable of running a single round of chess, and returns the result to a counter when the game ends.

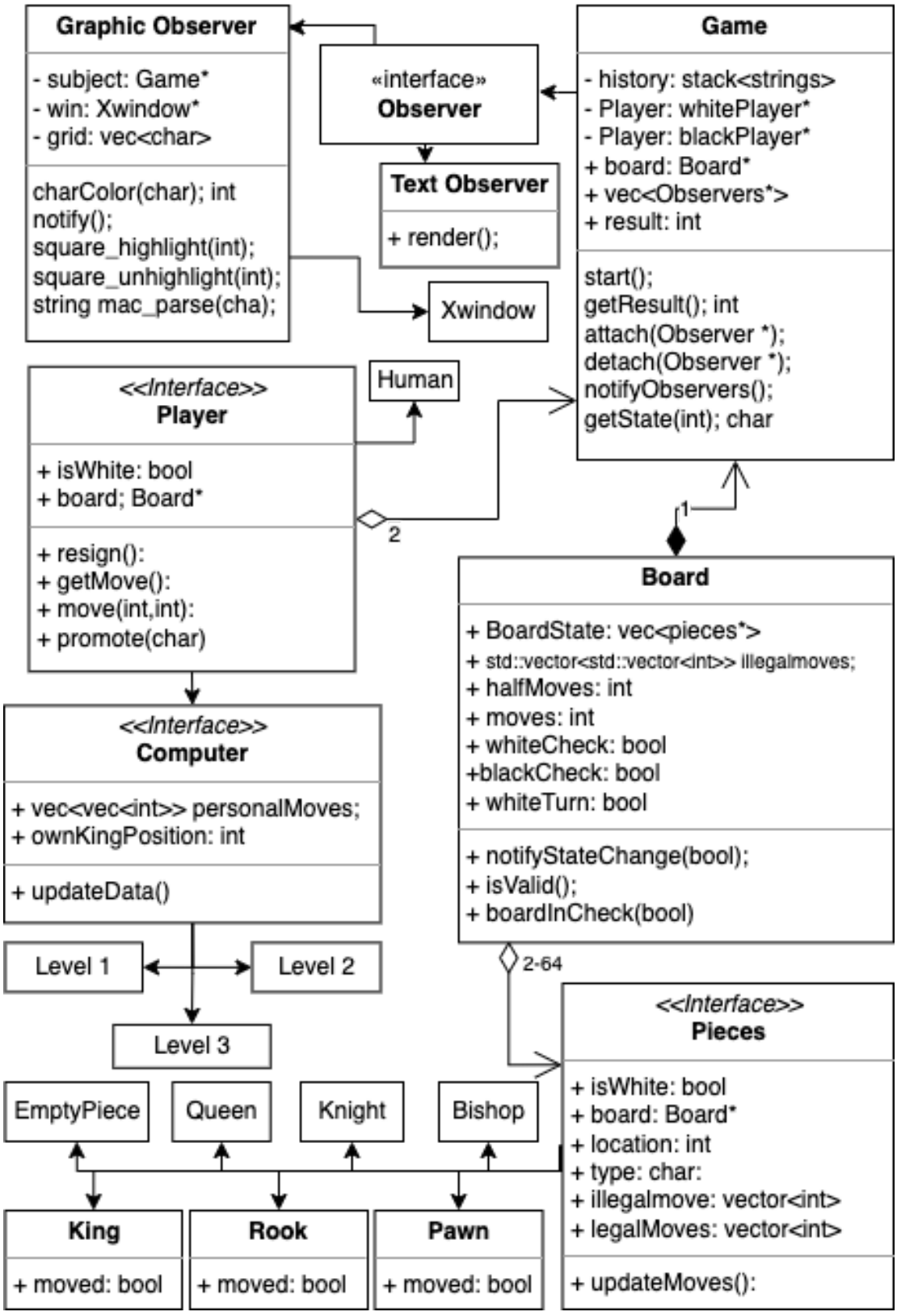
Observers: These are responsible for visually displaying the board. The text observer class displays the board on the terminal. The graphics observer class creates a graphical window that displays the board.

Summary of Program Execution

Throughout the course of a single game, the interactions between these classes can be summarised as:

1. The board to be played is constructed as a **Board** class. The **Board** class contains a vector of **Pieces** that represent the chessboard and its pieces. Optionally, the user may set up a custom board with variations in the pieces present and their positions, as well as who starts first.
2. The white and black players are constructed as the **Player** class, and as a human or as one of the 3 difficulty settings depending on user input.
3. A **Game** class is constructed, taking in the board, the white player, and the black player, with text and graphics **Observer'**s classes attached to it. This class will be responsible for carrying out this round of chess.
4. The **Game** class calls on the current player to make a move.
5. The current player as the **Player** class produces the move to be played. If it is a human player, the move is provided from user input. If it is a computer player, the move is evaluated through analysis of the board.
6. This move is then applied to the board from the current **Player** class by moving the piece at the starting coordinate, represented as a **Pisces** class, to the correct position in the board. Pieces are attached and detached as necessary if a capture or pawn promotion occurs.
7. **Observer**s attached to the **Game** are notified of the updated board, and they redisplay the board accordingly.
8. The **Game** class notifies all **Pieces** present to update their legal moves according to the updated board.
9. The **Board** class analyses the current board for checks. If checkmate or stalemate is achieved, we move to step 11. Otherwise, we move to step 10.
10. The other **Player** is called on to make a move. Steps 5-9 are repeated from the other **Player**'s side.
11. The **Game** returns a number indicating the result of the game, which is used to increment our win counters. If a new game is called, we repeat from Step 1.

CHESS - CS246 - UML Diagram



Design

Implementation of Logic

The internal rules and logic is a crucial part of the program in order to produce a game experience that conforms to the rules of chess. The process of checking the validity of most moves can be described into two phases.

Getting Legal Moves:

At any given moment of the game, pieces store a vector of legal moves that the piece can undertake. This is accomplished through using the **Observer Pattern** to notify a function that updates each piece's legal moves every time a move is made on the board. The legal moves are generated by parsing through squares included in that piece's hypothetical moveset and checking certain conditions under them. If they meet the conditions, these moves are added.

For example, a rook would check pieces horizontal and vertical in all four directions, stopping if a same colour piece obstructs its path, and adding moves resulting in it landing on an empty square or one containing an enemy.

On the other hand, a pawn only checks the squares immediately ahead (two if it hasn't moved and is in the correct row of the board, so custom setups don't result in pawns stepping twice when in the middle of the board), and immediately diagonally ahead if they have pieces of the opposite colour. The colour of the piece also determines which direction it moves, upwards if white, or downwards if black.

For detection of pins, we use a function that will be described in the later phase for checks.

Checks:

The overall process of producing a list of legal moves for each piece was to create a list of pseudolegal moves for each piece, iterate through each of those moves and see if they result in the king of the same colour being checked by playing the move on the board and calling a function that evaluates the status of the board, then undoing the move and storing the data about the move. After all pseudolegal moves have been parsed, the stored data is used to filter out the illegal moves from the pseudolegal moves, leaving us with a ready-made list of legal moves for each piece at any point in the game.

The process of creating the pseudolegal moves was relatively simple, as explained in the previous section. A boolean was used to indicate whether the function that generates the moves should generate pseudolegal moves or legal moves (this will be explained ahead). If chosen to generate legal moves, a function is called on each pseudolegal move that plays the

move out on the board, updates the legal moves for each piece on the board, checks the board status to see if a check occurred, then undoes the move and resets each piece's legal moves. To avoid an infinite loop (checking to see if a move is legal by moving it and updating the board to generate the next set of legal moves to look for a check, but those moves would need to go through the same process of testing, etc), the boolean mentioned earlier was created and only looks for pseudolegal moves when testing a move on the board. If a pseudolegal move is determined to be illegal, it's stored separately (in the board), since every time a pseudolegal move is tested for any piece, the legal moves for every other piece get reset. After this, the pseudolegal moves for each are filtered and the illegal ones are removed, resulting in a list of fully legal moves.

We handle some of the more infrequent moves as:

- For castling, we check if the king or rooks have been previously moved, and then check through if the passed squares are empty and not under attack. If the conditions are met, the castling is carried out.
- For en passant, we check the presence of enemy pawns to the sides of our pawn. If there is one present, and only a single half move has passed since its placement, we carry out en passant.
- While the above two pieces are handled by the **Pieces** class, pawn promotion is handled by the **Player** class. The class checks if any pawns have been pushed to the edge of the board and calls the promote function. In the human player class, this promote function takes in a string input to create a queen, bishop, rook, or knight. In the computer class, it always creates a queen.

Another component of the logic is determining the validity of a custom board setup. The **Board** class accomplishes this by iterating through the board and ensuring that each player has only two kings, and none of them are in check.

Implementation of Progression

The **Player** class is the primary means of progressing the game with each player providing the moves they want to play through the *getmove()* method, and applying them to the board with the *move()* method.

The *move()* method simply swaps the contents of the board between the piece at the starting index, and the piece at the ending index, given that it's a valid move. Captures are accomplished by deleting the piece at the ending index, creating an empty piece there, and swapping. Castling is accomplished by simultaneously moving the rook and the king. Pawn promotion is also carried out here, as described in the implementation of logic section.

While the *move()* is universal to all Player classes, *getmove()* varies between the human and computer players. In the human class, *getmove()* reads in the move from the player through standard input. In the computer class, *getmove()* evaluates the board for possible moves and randomly picks from the possible moves. The randomness is ensured through the *srand()* and *rand()* functions from *cstdlib* library.

At Level 1, the move is chosen at random. At Level 2, enemy captures and checks are preferred over other moves. At Level 3, enemy captures and checks, alongside avoiding own captures are preferred over other moves.

The **Game** class calls on Board, Player, and Observer classes in a loop to carry out a single game, as described in the summary of program execution.

Exception Handling

Our program is designed to handle to invalid inputs from the user. User inputs are involved in the game interface, getting moves, and pawn promotion.

In the **Game** class, commands are read in a loop and if an invalid command is detected, we simply continue to the next iteration and read input again. A similar process is done when reading in moves from the user in the **Human** class.

The *move()* method in the **Player** class throws an exception if it is asked to make an invalid move (trying to move opponents pieces, illegal moves), which is propagated by *getmove()* to the Game class, causing a repeat of the turn.

Changes from DD1

The biggest deviation from our original design was the point from where the move function would act from, taking it from the Game class to the Player where the methods of getting moves were stored.

Implementation of Interface

The program takes in input using STDIN through a controller to start games and setup boards. This makes it very easy to test out features as we can simply pipe in text files full of commands to the executable and ensure that the output is as expected. The human players also take in input from STDIN and they have been configured to allow the use of Long algebraic notation for chess moves which makes it easy to create tests.

In terms of output, the program relies on the use of the observer pattern in order to make it very easy to add, disable or edit the various ways of rendering the code without affecting the logic of the game. The subject is the game class which contains the appropriate functions to handle the observers and using the time allocated to this project, we were able to code 2 different renderers: a graphics observer and also a text observer.

Both observers work in quite a simple way that was discussed in class and implemented in the previous assignments but we added some features that elevate them a slight bit to improve their look and speed.

The text observer is quite simple, when it is notified, it prints the borders and labels for the bord and then it proceeds to iterate from the top left square(A8) to the bottom(H1), it uses function getState(int) with the current index as input and prints out its value.

The graphicsObserver also works in quite a similar way to the text board. Upon its construction it creates the window alongside an empty board with all the labels. When the graphics observer is notified, it proceeds to iterate through the board and print pieces at their appropriate location.

The optimization used and the aesthetic improvements made are discussed in detail in the extra credit section.

Resilience To Change

The decoupling of the game's logic between the **Game**, **Board**, **Pieces**, and **Player** classes allows us to make targeted changes to the logic to specific classes without affecting other portions of the game to an extent.

As the Pieces class handles the pieces and their movesets, any changes to the pieces can be targeted to Pieces class files without affecting the other classes.

For instance, if the King were to be able to move two squares instead of one, this would encompass changing only the conditions in calculating the legal moves for the king.

If we were to introduce a new chess piece, this would mean creating a new Pieces class for the new pieces, and minor changes to the board set up to accommodate the new piece.

If new gameplay features were to be implemented as undo or offering a draw, this would most likely mean simply implementing them in the **Game** class without affecting the other class. Undo could be implemented as a field that keeps track of the history of the board within the class that loads a board, and draw offers consist only of reading in a yes or no.

Changes to player behaviour can be implemented directly to **Player**. We can add new methods to grant players additional actions other than moving pieces-, and in the instance of the Computer class, we can make direct changes to the AI logic or add new difficulty levels.

The most drastic and difficult to accommodate change would be to the chess board itself, since our classes operate with the assumption of an 8x8 board with two players. However, we can work in a structured manner, making alterations to the Board class, then to the Pieces class to adjust to the new board, then to the Player class, and finally to the Observers for rendering the board. Only incremental changes will have to be made for the Game class

Project Specification Questions

Question 1 - how would you implement a book of standard openings if required?

Standard openings and chess games are often encoded in Portable Game Notation (PGN). There are several opening books available online that go through all the potential lines (different variations) of a particular opening. The LiChess database in fact provides access to millions of user games as well as a collection of grandmaster games; this can be used to build a list of openings and their variations.

We can add a PGN parser that compares the current moves played with that of existing openings and lines. This would allow us to add features such as showing what openings have been played (intentionally or not), and providing computer players with what moves to play next by seeing which opening and line they are playing.

Question 2 - Undo Moves - How would you allow a player to undo their last move? What about an unlimited number of undos?

We can add a vector board to the game class that functions as a stack. When a move is made, the previous board is deep copied and pushed into the stack.

Undo, alongside unlimited undos, would simply be iterating backwards through the stack, displaying the board as we go, and loading the chosen board into the game.

Question 3 - Four Player Chess :

As previously discussed, we would start with making changes to the Board class to accommodate the non-rectangular board and the check function to work with the board. Two additional players will need to be attached to the Game class and the game loop has to be adjusted for four players.

The logic checking in the pieces generating legal moves will have to be updated to work in a four player board, with the colour member field being switched to a string of 4 possible values.

The observer classes need to be updated to be able to iterate through the non rectangular board and display the squares accordingly.

The player classes should only need an update of the colour field to accommodate four colours.

Extra Credit Features

Fen Construction

The game allows for constructing setup boards using FEN notation which is the standardised notation for representing board states. It allows for very easy testing, importing chess puzzles or games from friends. This was challenging since we had to parse through the FEN and convert it to meet our board representation

Prettier Graphics and Optimization

- Graphical Colours
    - Implemented custom colored board by reading x11 documentation to add colours
- Graphical Symbol Pieces
    - Installed chess font onto x11 in our local systems to allow for the rendering of pieces through the x11 drawstring function
- Graphics Optimizations
    - Tracked the state for all 64 squares of the board through a vector of characters and only printed things if the state had changed. This greatly speeds up rendering as unchanged squares are not reprinted.
- Text Symbols
    - We implemented unicode symbols in our text render, this was tedious as C was set up to handle them greatly as chars. This meant we wrote a function to convert the piece names to symbol strings right before printing

Move Highlighting

If a human player tries to make a move and inputs the starting square or piece, all of its legal moves are then highlighted. This was done by writing functions that draw and hide a blank rectangle around squares. The indexes from a piece's legal moves were then converted into xy coordinates that were printed onto the screen.

Final Questions

Question 1 - What lessons did this project teach you about developing software in teams?

This project was extremely insightful and helped us understand the several challenges and difficulties that come when working as a team.

- Coordination and synchronising schedules with your teammates is important for maximising productivity.
- We all properly learnt how to use the git ecosystem which was extremely important not only for this project but also future workplace settings.
- We learnt how to communicate and plan software development as a team as well as well as the genuine importance of commenting

Question 2 - What would you have done differently if you had the chance to start over?

The most important change that we would make is to follow proper git procedure to keep out code bug free at all times. Instead of always creating proper branches and doing merge requests, we simply gave every group member the ability to push onto the main branch. This was fine for the most part, except once when a bug didn't get discovered until after several commits which was a very difficult thing to debug. We would ensure that the team was always available at the same times as well. Another important thing is for everyone to have proper understanding about the entire codebase so they can work on all the possible modules instead of being limited to their own work. All in all the project was a great learning experience and we would certainly do even better if we had another try.