Design Document
CS246 DDL2

Introduction

This C++ Object-Oriented Programming implementation of the Chess project aims to develop a digital version of the classic board game Chess using object-oriented programming principles. This documentation provides an overview of the project, the rationale behind major design decisions, implementation details, answers to questions raised on additional features. Answers to reflection questions are also included at the end of the documentation.

Overview

The project will be organized into the following (major) components:

1. Board: handles move validation, and keeps track of the game progression.

Board is one of the most important classes in this program . Board contains a private field board, which is a vector of vectors of unique pointer to Chess. It has two functions placeChess() and removeChess(), which are responsible for changing the Chess on the board. Its core method is move() and canMove(). canMove() checks whether a move is legal by verifying whether it follows the specific chess type's move pattern, whether it is blocked by other Chess, and whether this move puts its king in check.

The first two conditions are verified by calling the Chess methods isLegalMove() and isBlock(). The last condition is verified by temporarily changing the original location's private field "empty" to true and setting the new location's field "empty" to false (pretend the original location is now empty and the new location is not empty without actually moving chess), then this function checks if opponent's Chess can move to current colour's King by again calling isLegalMove() and isBlock().

Once canMove() is verified, Board calls move() which actually moves the Chess and notify Observers. Other important methods including isCheck(), isStalemate(), and isCheckmate() are implemented based on canMove(). For instance, isStalemate returns true if a colour of all Chess do not have legal move and king is not in check, that is, canMove() returns false for all Chess pieces and for all locations. isCheckmate() works in the similar way, determining if the King has at least one canMove() return true when the King is in check.

2. Controller (main): Handles the main control flow of the game, communicating between user input and board operations, displays. It is responsible for managing some game information including setting up players matching player colour, storing player's score. Most of its job is to generate input and call corresponding class methods. For example, if the user chooses to enter setup mode, it calls Board::placeChess() or Board::removeChess() to edit the board. If the user starts "game" without entering setup mode, it calls the main function SetUpStandard which also uses Board::placeChess() and Board::removeChess() to finish a standard opening setting.

3. Displays: manages the user interface for displaying the game board, player interactions, and game status updates on both the standard output and in a graphical display. They are observers of each Chess piece, and are attached as such upon initialization.

- TextDisplay: a CLI interface displaying a series of characters which makes up a board along with its pieces. The core field of the TextDisplay class is “theDisplay”, a vector of vectors of char. It is initialized to be filled with “_” or “ ” (empty), then it updates the char when the board is updated (placeChess() and removeChess() notifies observers)
- Xwindow: named Xwindow, the graphical display defines an Xwindow object and opens up a graphical interface for users to visualize the progression of their game in a visually appealing way by calling drawBoard() and drawChessString(); board is represented by some black and white rectangles while the line numbers are illustrated by strings. Each chess piece is shown by using pixmaps. When a chess piece moves to another location, the program simply fills another rectangle on the original and new location and puts a new pixmap on the new location.

4. Players: contains all information regarding the two players of the game, namely the name, score and colour, along with a protected pointer to the board, and allows a player to make a move during an ongoing game.

- Human**:** one of Player’s subclasses, denoting a human player as opposed to an automated one. The class enables a user to manually input moves and translates them to a numbered index on the board. Prompts for a valid chess type when performing pawn promotion.
- Computer: a pure virtual subclass of player, which has various concrete Level subclasses, denoting the level of sophistication of an automated player. Its makeMove() function is pure virtual, which will be implemented in its subclasses based on moving logic, whether the automated player prefers capturing/checking, or avoids being captured. Contains two protected methods: isPromotion() checks whether the move will invoke pawn promotion, and isPawnCapture() checks whether the move is legal for pawn to capture or perform en passant.
    - LevelOne: performs random valid moves.
      Algorithm: Check if the current player is in a stalemate, if true, cannot move and return false in makeMove(), else proceed to the next step.
      Record positions of own side chess pieces and any other grid positions into separate vectors of vectors of int, respectively. Shuffle the list of player’s own side chess to get a random chess piece, loop through the available grid positions (either opponent occupied or empty grids) to store all the legal moving positions for the current piece into a 2-d vector of ints.
      If there are no legal moves for the current piece, randomly pick another one and apply the same logic. As long as any piece has a legal move, move on to

randomly pick a legal move by generating a random index in the range of the legal move list and get the coordinates of destination.
Check for promotion before move, if current moving chess is a pawn and will reach the end of the board at the destination, randomly generate a piece type between rook, knight, bishop and queen, place the chess at moving destination and remove the current chess from board. When finished, return true.
If no promotion is detected, simply move the chess and return true.

- LevelTwo: prefers check and capturing moves over other legal moves
  Algorithm: Detect own side chess pieces, available grids (grids other than own side occupied grids), empty grids and opponent's king's location. Shuffle the own side chess list and available grid list to increase randomness. Loop through two lists in a nested for loop. For each own side piece and an available grid, check if the move from the current position to the grid is legal before proceed to the following steps:
    - Check if the destination sits the opponent's king, if so, capture it and checkmate.;
    - Check if the move will result in check by checking whether the chess piece after move can capture the opponent's king within one legal move and is not blocked by any other pieces;
    - If the current piece is a pawn, call isPawnCapture() to check if it is a pawn capturing move (capture diagonally or en passant)
    - Check if the move is a capturing move by detecting whether the destination has an opponent's chess;

  If no capturing or check move is applicable, shuffle the own side chess list and the empty grid list to regenerate random chess and destinations. Loop through them to get a random legal move, perform promotion if valid, this time pawn will change to queen as an advanced level strategy. If there is no promotion, simply move to the destination. Return true whenever a move is done during the process.
- LevelThree: prefers check and capturing moves, avoiding being captured.
  Contains two protected methods: inCapture() checks whether the current piece can be captured by any opponent's piece, and checkEscape() checks whether the own side king under check can move to a safe position.
  Algorithm: Scan through the board and store the positions of chess pieces (own side and opponents), available grids, empty grids into separate 2-d arrays, record the position of own side king as well as the opponent's.
  Check if my king is in check, if so, call checkEscape() and move my king to a position that will not be in capture.
  Check for check and capturing moves without potentially being captured. Loop through the own side chess list and available grids as in LevelTwo, for each legal move, check if the move will put the piece under capture by stimulating the move

and call inCapture(), if true, skip the current destination and check the next in the list; else proceed to the check procedure performed in LevelTwo.
If no check or capturing moves, loop again and check for capture-avoided moves for own side chess that will be captured.
If no own side chess pieces are in the threat of being captured, perform random legal and capture-avoided moves. Otherwise, perform random legal moves.

- LevelFour: LevelFour is built based on LevelThree, except that it establishes a new score system for different moves based on the power of this move. The score system is built based on this website: https://www.chess.com/terms/chess-piece-value. For instance, capturing opponent's queen is usually supposed to be a stronger move than capturing opponent's pawn, so capturing queen is given a higher score than capturing a pawn. And a move that makes the chess being captured will deduct a score based on the chess type. To achieve this goal, two maps are used. The first map stores different types of moves in the key and stores the corresponding point in the value. In the second map, the key contains all of the legal moves, being represented by x1, y1 (original location), x2, y2 (new location), and the corresponding score of this specific move is contained in the value.
  The algorithm works in a similar way to LevelThree, the Computer4 would start by scanning the whole board, and it is designed to prioritize the move related to escaping from being checked and checking the opponent's king. Other than that, it would randomly pick a move with the highest score from the map and move. By doing this, Computer4 can make the strongest move instead of moving randomly.

5. Chess: representation of individual chess pieces and their coordinates on the board
   - King: a subclass of the Chess superclass representing the King piece, the central piece around which the game is structured. The King can move in any direction, but only one space at a time, and can also perform castling with the Rook, given that none of the two pieces have moved.
   - Queen: a subclass of the Chess superclass representing the Queen piece, a piece that can move in any direction, and for as many squares as it desires.
   - Bishop: a subclass of the Chess superclass representing the Bishop piece. The Bishop can only move in a diagonal manner, but has no restriction on the number of spaces moved.
   - Knight: a subclass of the Chess superclass that represents the Knight piece. This piece is the only piece that cannot be blocked by other pieces. It moves in an "L" shape (any orientation).
   - Rook: a subclass of the Chess superclass denoting the Rook piece, which can only move in one axis at a time for as many squares as it wants. If neither of the Rook or the King have moved, it can castle with the king (i.e. move to the other side of the King)
   - Pawn: a subclass of the Chess superclass representing the Pawn, only moving forward (one of two squares depending on whether it has previously made a move). It captures by

moving one space diagonally (either regular or en passant capture), and can be promoted to any of the Knight, Rook, Bishop or Queen upon reaching the end of the Board.

- Empty: a subclass of the Chess superclass. The Empty class is used in a vector of Chess to denote that no piece is currently occupying a particular square on the chess board.

The new UML diagram introduces several changes and additions compared to the old one, reflecting modifications in the design of the chess-related classes (in UML, changes are shown in bold). Below is a description of the changes in these classes and their interactions:

Instead of using a isFirstMove boolean value field to track the first move status in some subclasses, we have changed to counting the number of moves made (if the relevant information is needed). This is because we need to keep track of whether a pawn has made its first two-squared moves to implement an en passant capture. That is, we need to check whether an opponent's pawn is on its second move, therefore it is insufficient to only return a boolean value.

A Board pointer has been added to the Chess class to access global information from the board within a chess piece to verify legal moves, such as when a piece needs to perform castling or capture other pieces.

Chess is now stored in the Board as a smart pointer.

Parameters of new and old coordinates have been added to various functions of the Chess and Board classes and subclasses in order to gauge the validity of moving from one point to another.

Many methods that return booleans have been accounted for in main, thus were removed from the controller as initially envisioned. Printing methods are also found in the main as opposed to the text and graphical views since in those an operator overloading was deemed more suitable. However, one boolean method was added in the Computer class, for the computer to decide to which piece it would like its pawn to be promoted to.

More enum classes, especially involving the game's state, indicate a more systematic approach to managing states and to avoid encountering unspecified behaviour.

As we were not aware of the specifics of displaying Xwindows, we have in the updated UML included various draw methods–for the board, the individual pieces in the form of pixmaps, as well as the strings for displaying coordinates.

Design (specific techniques used to solve design challenges in the project)

1. **Design Pattern**

The main design pattern in this project is the Observer pattern, utilized to update the display in the graphical interface of the chess game. The Observer pattern establishes a one-to-many relationship between objects, where the subject (a particular square on the board) maintains a list of its dependents and notifies them automatically if the board (more specifically, an individual square) is updated.

In the context of the chess game, the interface components, such as the text and graphical displays, act as observers. They register themselves with each of the board's spaces–at the time it

is constructed–to receive notifications whenever the game state changes. For example, during initial setup, when a player makes a move, captures a piece, or the game ends, Chess notifies the observers about these events.

Chess pieces implemented by the Chess concrete subclass acts as the subject and maintains a list of registered observers. Whenever a relevant event occurs (i.e. a piece moves, a piece is captured, a piece type is changed by pawn promotion), the board notifies all the registered observers by invoking a notify() method. The Chess's notifyObserver() method then calls each Observer's notify() method, giving the Observer the original and new location of the moving chess piece, and a reference to the chess piece itself to convey information. Specifically, passing reference of the chess as a parameter intends to give Observer information about the piece type in situations like pawn promotion. The observers then retrieve the updated game state from the game engine and update their display accordingly.

Any changes to the game state are automatically reflected in the relevant display, ensuring that players can see the current state of the game without manually querying the update. In this way, when a new Display is added to the program, the program can simply attach it to the Observer's list and implement the new Display's notify() method without adding any extra code.

Besides the Observer pattern, the Factory Method pattern is also applied in the implementation of Player class, consisting of two subclasses – Human player and Computer player. The abstract Player class has a pure virtual method makeMove(). In concrete player subclass Human, makeMove() is overridden by retrieving input from iostream by cin, allowing real-people interaction. On the other hand, Computer is an abstract subclass which is inherited by four subclasses with distinct difficulty levels, and each subclass implements makeMove() by following different strategies corresponding to computer level. Therefore, the Player class acts as a public interface for all types of player; each player generates makeMove() input by different logic, but all of them eventually calls makeMove(), invoking Board's move() method and updating changes.

2. Cohesion and Coupling

Between the different classes implemented, high cohesion is achieved by organizing the program into distinct components such as the main input handler, game display and outputs, board and chess pieces. Each component will have a specific responsibility and will be designed to be independent and reusable–for example, the individual pieces are all designed to inherit from a larger Chess class, each with its own custom properties, while also retaining the main attributes of a chess piece. This concretisation of polymorphism provides flexibility and the ability to write code that can work with objects of different types.

The various levels of the computer are an example of inheritance, allowing objects to inherit properties and behaviors from the Computer superclass. It enables code reuse by defining common characteristics in a superclass and allowing subclasses to inherit and specialize those characteristics. For instance, while LevelOne determines moves in a randomized manner, higher

levels include more sophisticated logic towards a higher win rate. Inheritance promotes a hierarchical organization of classes, code reuse, and extensibility.

It is clear that low coupling is a good style that should be achieved since it makes the program easier to debug and accommodate for changes. Ideally we want each part of our program to only work for one target. Working towards this goal, TextDisplay and GraphicDisplay finish their displaying mission simply by retrieving information about Chess in parameters of notify() method.

However, there undeniably exists a codependency between the Board and Chess classes. Chess owns the function isLegalMove() that identifies whether one move follows the specific chess type's moving pattern, and it also has functions isBlocked() that determines if the current piece is blocked by other chess pieces when it tries to make a move. Together these two functions are important components of the Board method canMove(), returning boolean true if a move is legal. The Board needs to use Chess's method to identify whether a move is legal, and at the same time Chess piece needs to know whether a move is blocked by other pieces on the board. Therefore, in our design, it is to some extent unavoidable to maintain a high coupling between Chess and Board. To reduce the chance of problems occurring at runtime, appropriate accessors and getters are injected into those classes. This will allow the components to interact with each other through well-defined interfaces, reducing the dependencies between them.

3. **Encapsulation**

All the fields of classes are either private or protected in this program, prompting other classes to access and make changes through setters and getters. In class Board, users can only edit the Chess on the board by calling public methods placeChess() and removeChess(). Initializing a chess is achieved by calling placeChess(), and capturing a chess is implemented by calling removeChess(). This guarantees that invariants are respected. In this way, each type of chess is initialized by make_unique() with description, coordinates, and observers being set properly.

4. **Memory Management**

In terms of memory management, no new or delete is used in our program. All the heap memory is allocated through vectors or smart pointers. Specifically, the private field of Board "board" is constructed by a vector of vectors that contains unique pointers of Chess, so the "board" has the ownership of each "Chess". The use of unique pointers means that each Chess piece's memory is automatically reclaimed when it goes out of scope, eliminating the need for explicit memory deallocation. This helps prevent memory leaks and enhances the overall stability of our program. Additionally, the vector of vectors provides a flexible structure for representing the game board. Although it was not needed as per current specifications, if we were to resize the board (i.e. if specifications were to change overnight), we would easily be able to accommodate such changes in the game state without manual memory management concerns.

Additionally, vectors provide constant-time access to elements, hence reducing the time users spend manipulating the board during gameplay.

Before we added XWindow, there was not any memory leak in our program, and everything was freed nicely when we ran valgrind. However, after we added our XWindow, which included drawing numerous pixmaps, a run through Valgrind shows that there is a memory leak with no call stacks showing. We suspect that this is attributed to X11, especially the pixmaps we drew. Therefore, we try to free each pixmap by using the function XFreePixmap(), but it still appears that some memory leaks exist. Besides leaks caused by the external library, there should be no memory problems in our program.

Resilience to Change

The reusability of the game components enables us to add additional features without making any changes to existing ones. One such feature is adding levels to the computer class. By applying the Factory Method Pattern, one need only to add a new subclass to Computer and define a dominant logic for making moves. If a Computer needs to reach a more sophisticated level, we may determine makeMove() by consulting, say a book of standard opening move sequences.

Additionally, new Chess types can also be added to the program easily by implementing a new subclass of the Chess abstract class and overriding isLegalMove() and isBlocked() methods. Or, if an existing piece's move rule were to change, one would only have to change the affected Chess subclass's isLegalMove() and isBlocked() checks without modifying larger classes concerning the actual game.

Building a new display can be achieved by creating a new subclass of Observer and attaching it to the Observer's list. If the rules of the game change (eg, number of players increase to four), main is able to handle this change by creating more players and changing turn rules.

Answer to Questions

a. Q: 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. See for example https://www.chess.com/explorer which lists starting moves, possible responses, and historic win/draw/loss percentages. Although you are not required to support this, discuss how you would implement a book of standard openings if required.

   A: Add a new superclass for a Book of opening moves and associated responses array as if it were a physical book. This class could store fields (the aforementioned fields, i.e. opening moves and responses along with the respective historic win/draw/loss percentages for each response using a STL map (Have a unique key for each opening move, and store as value statistics and responses).

b. Q: How would you implement a feature that would allow a player to undo their last move? What about an unlimited number of undos?
A: We already have a lastMove vector of vectors of integers, representing the coordinates of moving from point A to point B. We could therefore implement a vector of these lastMove(s) to represent moves history. To undo, retrieve the last element (the actual last move) from the vector and update the display to reflect the previous game state, (the move would have to be erased from the vector). After each executed move, we should emplace the latest move to the back of the vector. If we were implementing the unlimited number of undos feature, we might want to implement an undo function which performs the single undo a variable number of times, until there are no more items in the vector and we have reached the starting state.

c. Q: 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.
A:
1. Create more types in the Player class to account for more players (add two players)
2. Modify the board to board size to accommodate four players, that is, add an additional 3x8 grid to each side of the existing grid by resizing the vector field in the Board class.
3. We would need to modify our transferLoc() function, which currently acts as a helper to translate user input into integer board coordinates. These additional squares would also need to be mapped to a coordinate on the freshly redimensioned board.
4. There would need to be additional symbols on top of existing capital letters for white and lowercase letters for black to denote the pieces of the other players.
5. Instead of switching turns between white and black, there would need to be another logic (e.g. clockwise/counterclockwise) for the players to take turns if they were all competing against each other.
6. There needs to be a logic for when the pawn reaches the end of the board both horizontally and vertically (Chess.com).
7. Modify the point system that supports recording points and adding points to each player following point earning rules. This point system can be implemented by adding a private field “point” to class Player and incrementing “point” after each move. Resigning should give a different amount of points, depending on how many players lost by resigning versus playing.
8. Change the ending rules
    - when a player is checkmated or resigned, its pieces become inactive, but the game continues until a winner has been pronounced
    - when three players are checkmated or resigned, the game end

9. Non-existing grids on the board that are illegal for placing chess need to be distinguished from blank grids by including enum class or private field (with a public accessor) in the Chess superclass.
10. There is a need to add two more sets of Chess subjects to the vector field of Board class.
11. If players are playing in teams,

- change the checkmate and stalemate logic to consider pieces of both enemies
- adapt the piece-taking logic, a player cannot take the piece of their teammate
- pawns only get promoted upon reaching the edge of the opposite side. whose distance increased.
- when one player from a team resigns, the game ends with the other team winning.

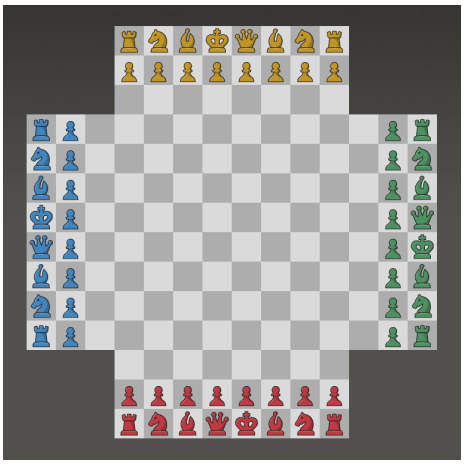

Figure 1. 4 Player chess from Chess.com

Final Questions

d. Q. 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?
A. The most significant lesson we learnt is the importance of a unified public interface. When working on a medium to large size program like this, considerable time should be spent on planning on the structure and deciding the public interface before writing actual code. An organized approach to collaboration also needs to be established from the get-go, since it is easy for work to get messy without structured deadlines and task divisions.

e. Q. What would you have done differently if you had the chance to start over?
A. Our team has done great work given the time constraints we were subjected to. We successfully implemented most features as outlined in the project specification, and everyone contributed significantly to this collective success. Reflecting on our project, there are surely areas where we could enhance our approach. For instance, the whole

debugging process could have been more streamlined for better efficiency. We also settled for a complex collaboration system without first researching on better alternatives which would facilitate coding together.

Conclusion

This Chess project completed as part of the Fall2023 edition of CS246 at the University of Waterloo aims to provide an interactive, computed version of the classic game Chess, solving implementation problems with appropriate design solutions. By following object-oriented programming principles and utilizing modern collaboration technologies, we strove to create a visually-pleasing and complete game for players of all skill levels.

References

Chess.com, The board of standard 4 Player Chess [Image]. Chess.com. URL (https://www.chess.com/terms/4-player-chess)