CS246

Chess

Final Design Document

2021 Winter

Introduction

Chess is a two-player game, which in the scope of this project is played on an 8x8 checkerboard with the bottom right square is white every time. In Chess, the player who controls the white pieces always has the first move, after which, it turns into a turn-wise game. There are six different pieces: King, Queen, Bishop, Rook, Knight, and Pawn, each with a specific set of movements. Each player can capture the other's pieces with the end goal of capturing the opponent's king which is known as checkmate.

Overview

When any game is started between human-human, human-computer, computer-computer, the **Grid** is initialized which contains the game board and reflects how the game is played out. The purpose of **Grid** is to serve as the general command dispatcher, for instance whenever a piece is moved, **Grid** relays the information or command to the necessary functions or classes so that the game can play out as intended. While **Grid** acts as the dispatcher, the command interpreter or parser would be **main.** Any commands or input made by the player would be read in by **main**, and in turn, **main** would call the respective functions inside **Grid**.

The **Grid** consists of a 2D vector of **Cell** (**std::vector<std::vector<Cell>>**), which represents the 8x8 game board. **Grid** also includes two vectors of **Piece ***, whose purpose is to keep track of the black and white pieces throughout the game, respectively. When **Grid** is initialized in default mode and not using **setup**, each **Cell** is initialized and filled in with the proper information such as if a piece is placed in it. Whenever a move is made, **Grid** would first make the check to determine whether the move is legal then call the necessary functions to make the change. This is made possible by making **Grid** also include a **Textdisplay** and **Graphicsdisplay** class whose respective purpose is to output an updated board whenever a change or move is made. Once the move is validated, calling **update** on the **Textdisplay** would apply the change to the game board display and output it.

Each **Cell** is initialized to contain a **Pos** class indicating the **Cell**'s respective position on the game board. That is, **Pos** is a class that contains the coordinates (row, column) of the respective **Cell** regarding its position on the board. Additionally, **Cell** also contains a class called **Piece** which represents which type of chess piece is in the cell

(knight, rook, pawn, etc.), if there is no piece present in the cell, the **Piece** parameter defaults to the null pointer.

A **Piece** is a superclass of classes **pawn**, **rook**, **knight**, **bishop**, **king**, and **queen**. Additionally, **Piece** has parameters **Type**, **Pos**, and **Colour**. **Type** is an enum class that take on values King, Queen, Bishop, Knight, Rook, and Pawn. While **Colour** is also an enum class that takes on values White, Black, or NoColour. To create individual **Piece**s, the factory design pattern is utilized. Which means that instead of having to explicitly call each **Piece**'s subclass constructor, the factory method takes in the default parameters **Pos** which is a class of row and column as well as the **Type** and **Colour**, and then constructs and returns the respective **Piece** with the given parameters.

Computer is a superclass of classes **computer1**, **computer2**, **computer3** for each respective difficulty level of the computer player. The **computer** class utilizes the factory design pattern, the internal method createComputer takes in a integer that indicates the level of the computer player and creates as well as returns a new instance of a computer with the corresponding difficulty level. In each subclass of **computer**, there exists an internal method getNextMove which determines the next move of the computer. For **computer1** or computer player level 1, getNextMove simply determines all legal moves that the computer can make and chooses one at random. In **computer2**, getNextMove attempts to find moves that would capture opposing pieces or result in placing the opposing king in check as well as regular legal moves. In this case, getNextMove would also prefer to choose moves that would capture pieces or place the opponent's king in check. Finally, for **computer3**, getNextMove sets out to find moves that would avoid capture, capture a piece, or places the opposing king in check as well as all random legal moves. In this implementation, **computer3** would prefer to choose moves that avoid capture, capture a piece, or places the opposing king in check over other moves.

Design

Setup

When a game of chess is started, the class **Grid** is initialized as mentioned before. The **Grid** essentially contains the entirety of the game of chess as it encompasses the physical game board (called **grid**, to distinguish between the actual class **Grid**, the board is written as lower case), along with the **Piece**s for each player. Additionally, the game board within **Grid** is set up as a two-dimensional vector of **Cell**s. The **Grid** is responsible for the implementation of the commands or inputs that are parsed from

main, as such it acts as the central dispatcher. As an example, whenever a move command is read in from **main**, the **Grid** then calls the respective internal methods which in turn calls the internal methods for both **Piece** and **Cell** to validate and implement the move onto the physical game board.

Challenges

One design challenge was the implementation of the functions check, which is a crucial method for the game of chess to work. Since the first step before determining if a king is in check is to validate if the move was legal. As such, we first wrote another internal method within each subclass of **Piece** called getValidMoves. This function would consider the **Type** of the specified **Piece** and its position on the current game board to calculate all legal places that the piece is able to move and return a vector of all valid moves. However, our original idea was to call getValidMoves inside of check to determine which legal move would result in a check. However, inside our **Grid** class, we have two instances of check (overloaded), the first check takes in a **Piece** and a **Pos** and determines if the piece was moved to the specified position would result in a check or not. In addition, this would then call the second check which simply returns if the board has a check or not. However, inside this second check, we called getValidMoves to determine all legal moves and check if any of those moves would result in a check. Yet, in getValidMoves, we also call the first check function (to determine if a piece was moved to a position would result in check), causing a circular dependency that gives off a segmentation fault. Our solution to this problem was to pass in an additional Boolean parameter into both checks as well as getValidMove indicating when it's not necessary to inspect if there is a check.

Another design challenge was the implementation of checkValidMove, which is an internal method inside each subclass of the **Piece**. This function would check if a specified move were valid or not. Originally, we set out to implement all the logic within checkValidMove, however, after some consideration, we deemed it too inefficient as given a move to a specified position on a board we would need to determine if it were valid but also if it would result in a check or not. Since the internal method of check within **Grid**, also needed to determine valid moves, we choose to write another internal method within each subclass of **Piece** called getValidMoves which given the **Type** of **Piece** and its **Pos**, would return all the valid moves. This way, both methods, checkValidMove and check could simply wrap around this function getValidMoves instead of having to be logically implemented separately.

One other design challenge was that originally, we set out to implement the observer design pattern. However, the issue was that the superclass, the observer, only takes in one type of parameter. Yet, we use different types of observers such as **Textdisplay** and **Graphicsdisplay** but this proved to be an issue since they are of different types and can not be cast to one another. As a result, we worked about this problem by instead providing an internal method called update and updateGrid inside **Textdisplay** and **Graphicsdisplay** respectively that when called, would make the necessary changes to the board. Since we decided not to use the observer design pattern, we choose to utilize the factory design pattern when creating pieces as discussed in the overview.

Changes from Due Date 1

One aspect of our implementation that we had to change from due date 1 was the design pattern. Originally, we thought to utilize the observer design pattern and have the **Grid** observe all the **Piece**'s. However, as we continued with the implementation, we realized that we did not actually need observer pattern if we just included an internal update method in both **Textdisplay** and **Graphicsdisplay** that would keep track of the current board and update itself whenever a move or change is made. In the end, we decided to implement the factory design pattern since we have numerous subclasses for **Piece** and **Computer**. This way, instead of calling the explicit constructors for each subclass, instead we call a factory method in the superclass instead indicating whichever type of piece and it will return a new instance of a **Piece** with the specified type. This also applies to **Computer** where we have subclasses for each separate level of difficulty (level 1, level 2, and level 3).

Another aspect that we changed was instead of each subclass of **Piece** (knight, pawn, etc.) having an internal method which calculates the best move for the computer players, we decided to make a separate class called **computer** and underlying subclasses for the varying levels of difficulties. Each subclass would then have an internal method that would calculate the best move for that level of difficulty.

We also decided to make a separate class called **Utils**, which would have internal methods that make our lives easier. For instance, having to determine which type of **Piece** corresponds with which character (Pawn would be represented by P or p), instead of having to manually determine the conversion every time, we can simply call the method and be done.

Resilience to Change

The definition of coupling is the degree of interdependence between different modules within a program. Obviously, a lower degree of coupling is preferred since applying changes within one module will not influence or affect other modules. Meanwhile, cohesion refers to the degree of relation between methods or functions of a module or class. As a result, higher levels of cohesion are typically sought after in development since one module should ideally only focus on one goal. Regarding our implementation of the game chess, we will discuss in further detail how we implement low coupling as well as high cohesion.

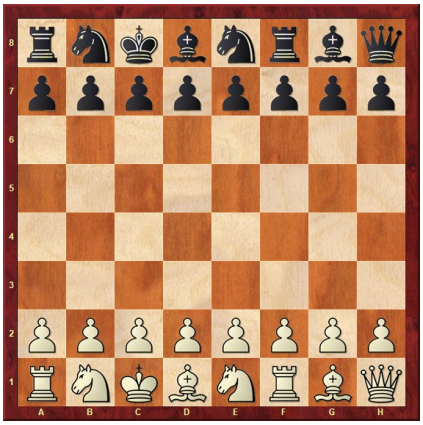
Within our implementation, there are five main classes, namely: **Cell**, **Grid**, **Piece**, **Textdisplay**, and **Graphicsdisplay**. All the classes are separated respectively to individual modules, thus changing one class is unlikely to affect another class. For example, if we wanted to implement a four-player chess mode as an extra feature, we can simply make a new grid which is a two-dimensional vector of **Cell** within the **Grid** class and overload the functions check and checkmate. There would be nothing to modify in the **Piece** or **Cell** class since the same set of rules for the pieces still apply. As a result, we can say that our implementation shows a low degree of coupling between the different modules.

In terms of cohesion, our implementation displays a high amount of cohesion. Each of the five main classes mentioned above each serve only one purpose as referenced in the names of the respective classes. For instance, **Grid** is solely responsible for managing the 8x8 chessboard, each instance of **Cell** only manages one cell of the **Grid**. Furthermore, the class **Piece** is only in charge of managing one instance at a time, **Textdisplay** only serves to print out the game board in a text format, while the class **Graphicsdisplay** is only for displaying the chessboard in terms of graphics. As such, there is no class that would be responsible for two separate purposes, demonstrating a high amount of cohesion.

Adding new features or rules

I will discuss how to add new features and rules to our program to further illustrate how our program is resilient to changes. Except for four-person chess, there is a popular

variation of chess called FisherRandom. The other rules of FisherRandom are the same as the traditional chess game except that the starting positions of the pieces are generated randomly. To add this game mode into our program, we only need to add one function and use the random move generator in our program to initialize the chessboard when the game starts. Everything else just stays the same. If we want to slightly modify the rules for each piece, we could just modify the getValidMoves() in the piece class for the specific piece. Therefore, our program can definitely handle all sorts of variations of the chess game. If we want to upgrade our computer AI, all we need to do is add another subclass under the class **computer**. We could have any number of levels we want.



Answers to Questions from Project

Question 1: ***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.***

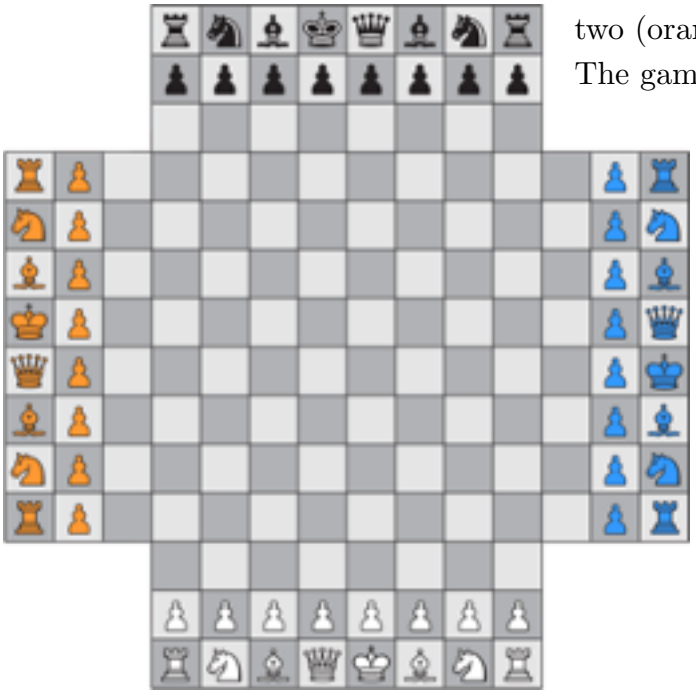
Answer: Standard opening move sequences such as the queen's gambit can have the entire procedure stored inside a vector of moves. If the player is a computer, the default behaviour can be set to complete these moves as the opening. However, since we are also interested in responding to the opponents' moves as well, we can utilize the strategy design pattern. The intention behind this design pattern is to make an interchangeable family of algorithms which ultimately allows the algorithm to act independently. The strategy design pattern is commonly used for computer player's as it allows the computer to adapt to changes as the game goes on and make decisions on the spot. This the default algorithm can be set to whichever standard opening, and as the progresses, the computer can make further decisions based off the opponents' moves and adjust accordingly.

Question 2: ***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: We can use the memento design pattern to implement this feature. The memento design pattern is a design pattern that lets us save and restore the previous state of an object. In this case, the memento pattern can save the previous state of the chess board and restore it. The memento pattern has three classes: the originator, the caretaker, and the memento. The originator is responsible for saving the state of the board to the memento object. The caretaker is responsible for requesting the save and restoration from the originator. Therefore, using the memento design pattern, we can undo the moves of both players at any time.

Question 3: ***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: Firstly, we would need to the standard 8x8 chessboard to accommodate a four-player game. This means that we would need to add three rows on each of the four sides of the board. We would then need to change the command interpreter so that it is able to parse input commands from four-players during the respective turns. Furthermore, we would need to add two additional players with the colour orange and blue in the grid object. An example of such a board is depicted by the following image.



The most popular form of the four-handed chess is two (orange and blue) against two (black and white). The game ends when both kings on the same side are checkmated. The general rule for every piece is the same. However, the pieces on the same sides cannot eliminate each other. Therefore, we only need to modify the functions or methods that determine whether a player's king is in check. Note that both kings on the same side need to be in check to win the game. We would also need to consider that each player needs to observe the pieces of both opponents, instead of just one.

Extra Credit Features

Random Scrambling of Computer Moves

As specified in the project guidelines, computer players should operate at several different difficulty levels. Starting from level 1 where the computer simply chooses to do random legal moves. Then in level 2, the computer would prefer moves that result in capturing the opponent's pieces as well as putting the opponent's king in check over other moves. In level 3, the computer would attempt to avoid capture, make capturing moves, and moves that result in a check. Finally, level 4 and above would involve even more sophisticated strategies. In our implementation of chess, we chose to randomize the computer player's choices in moves as well as increase the likelihood of the computer choosing a move that satisfies two criteria at once. This is particularly evident in levels 2 and 3, where if a valid move satisfies two criteria's such as capturing an opponent's piece as well as putting the opponent's king in check, the computer would be more likely to choose such a move as opposed to other valid moves.

Final Questions

Question 1: ***What lessons did this project teach you about developing software in team? If you worked alone, what lessons did you learn about writing large programs?***

Answer: There were numerous lessons that this project taught us.

1. It is important to communicate between group members, whether the discussion is about splitting responsibilities or deciding on deadlines. Furthermore, when developing software in a team, the foundation or groundwork implementation should be agreed upon as a group. This should be common practice even when implementing special or extra features as it might have an effect on other parts of the software. Communication can also come in the form of properly documenting code, which allows for the rest of the team to easily understand the purpose of functions and implementation details. This allows for any member of the team to continue from any point if responsibilities needed to be traded.
2. Setting deadlines for individual portions of the software is also important. Laying out expectations gives the team a sense of what needs to be completed at a particular time and the order of completion. This allows for enhanced planning and splitting up tasks and responsibilities. Furthermore, splitting up the project into smaller

individual deadlines decreases the likelihood of having to scramble last minute to deliver a finished product.

3. When a new method or function is written, it is also important to ensure that the code compiles before committing and pushing to the repository. It would not prove to be efficient if other team members must spend time debugging the issue instead of working on their own responsibilities. As well, this procedure would result in at least a semi-working product at every stage.

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

Answer: One thing we would have done differently is at the start of the project, instead of just diving straight in and figuring implementation strategies and techniques out on the fly, we would instead spend time to at least layout a skeleton plan and expectations. This way there would be fewer surprises along the way and less likely to have to apply major changes to the code, for instance, if we happened to skip over a feature by accident. Taking the time to read through the project specifications and guidelines in detail at the beginning would have been very helpful in determining the flow of the project as well as the responsibilities and tasks. By applying this thought, it would for sure have resulted in less time wasted and more efficient work.

Another thing we would have done differently is to plan out how to test the code. Chess is a complicated game with countless possible renditions and unique features or moves, meaning there are also numerous base and edge cases to cover such as castling or en passant. Instead of testing random moves and play games on the go, it would have been helpful to first design a test suite and test cases. This would have also been beneficial in figuring out the proper implementation and inner layout of the code.

Conclusion

Developing the game of chess proved to a fruitful challenge. This was the largest project that any of us had ever worked on and completed. Throughout the process and implementation, we were able to learn a lot more about the utilities and capabilities that the c++ language offers as well as the experience of developing software as a team. Despite the numerous difficulties we faced, we came away with priceless knowledge and experience as well as a final product.