CS 246 Final Project - Chess

Group Members:

Jaehak Kim (j54kim)
Jaden Cho (j9cho)
Juho Choi (j257choi)

I. Introduction

Among the three game options, we chose the chess game to build using object-oriented programming paradigm. During the last two weeks, we designed our chess game, planned how to implement it with UML, and carried out implementation.

This report will provide a detailed analysis of our chess game based on an Object-Oriented Programming perspective. We will explore the design that we used, discuss the points where we successfully implemented OOP principles, and consider those where we did not entirely follow these principles.

II. Overview

The program is designed based on MVC design pattern with observer pattern between the model and view components. For achieving heigh cohesion and low coupling, every class takes single responsibility and interact with each other based on the required functionalities. Player class is responsible for user input and communicate with the controller. Pieces work as sub-model due to responsible for unique movements of each pieces, so they interact with model component. For the data presentation, Model communicates with View, and Controller messages to View.

Abstraction is used to adopt variations of data presentation and of input data source. TextDisplay and GraphicDisplay implement View, and AIPlayer subclasses implement Player to produce input data for the CPU player.

III. Design

Single responsibility principle (SRP)

We designed each class has single responsivity to achieve low coupling. MVC pattern helps to simplify the structure of handling user input, commutating the input and presentation of the data. Thus, each responsibility is isolated to each component, which accomplished the SRP. Classes implementing Player class only focus on producing user input for the Game class, and Subclasses of Piece are only responsible for unique movement of chess pieces.

<u>The use of abstraction and the use of Inheritance</u>

The abstraction is another technique used to promote the low coupling. By using the Piece type in the GameBoard class, it can achieve all different movement of all piece by Piece interface. The gameboard does not need to know their concrete object type. The same mechanism is applied to View component when we switch the TextDisplay to GraphicDisplay class object.

The inheritance gives the flexibility of easy extension of new feature that shares common behaviours. For example, we used it to design unique chess pieces and AI players behaving with different strategies.

<u>MVC Pattern</u>

Our game adopted Model-View-Controller (MVC) design pattern as the base structure of the system. GameBoard class works as the model which computes and update the execution states. View class takes a role of the view component which presents the execution states of the program to the user. Finally, Game class is the controller responsible to interacting with users and handling user event.

<u>Observer Pattern</u>

In addition, we applied Observer pattern to simplify the communication between the MVC components. Our Model and View are in Subject and Subscriber relationship so that the View is directly updated the execution states by the model.

IV. Resilience to Change

In our program's design, we've prioritized 'Separate of concern' to enhance its resilience to change. This means that changes to functionality require changes only in the relevant module. We've also adopted abstraction, so code is centralized in one class for better handling and adaptability. Let's explore the points where we made our program resilient.

<u>**Piece: More Types and Changes in Rules**</u>

If there are changes and new types of piece are added, we can simply create a new subclass for the new type by inheriting the class Piece and overriding virtual methods 'possibleMoves()' and 'getType()' to implement its own algorithms for moving and add its identification in character. Then GameBoard can put the new piece to its computation and View can present it to a user. Moreover, if changes in rules are required to specific piece, we can easily adapt those changes by modifying the algorithms of the moving in a concrete implementation of the piece's class, which has no impact on the program.

<u>**Player: More types**</u>

If the changes in player are required such as adding a new type of Player, we can add the new type of player by simply inheriting the class Player and overriding a virtual method 'getResponse()' to implement a required rule for this type of player.

<u>**View: New way to display**</u>

If a new way of presenting the game is needed, we can easily reflect this need by creating a new subclass for the new way by simply inheriting the class View and overriding virtual methods 'display()' and 'notify()' to implement the new display.

<u>**GameLog: Flexibility of changing implementation detail**</u>

The class GameLog is for tracking the history of the game and undo – redo functionality. It achieved it by functionalities from Stack class object encapsulated the implementation detail. By isolating stack algorithm into the Stack class, we obtain flexibility to modify the implementation detail of the stack algorithm with zero impact to the GameLog class.

V. Answers to Questions

1. ***Chess programs usually come with a book of standard opening move sequences, which lists 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*.**

   Simply speaking, the movements from a book of standard openings is a strategy that AI Player can take. Therefore, we will use the Strategy design pattern and implement a subclass of the Strategy class using Turing machine algorithm that respond to every move the opponent makes according to a book of standard openings. Lastly, a new AI Player subclass contain this strategy object to produce movements.

2. ***How would you implement a feature that would allow a player to undo their last move? What about an unlimited number of undos?***

   To implement this feature, we would implement tracking of player moves and an undo feature using a stack data structure based on a vector. The "GameLog" class will contain a private integer field named count to count the number of a player's moves to prevent reverting back to a point beyond the initial move. Since we are using a stack, we would be able to "pop" a move from the log and set the current state to the result of the pop. For an unlimited number of undos, we can modify this method so that it takes an int argument n, representing the number of moves to be undone. We can then iterate n times, removing the front of the stack each time. Therefore, such an implementation allows for undoing one or unlimited moves made during the game

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.***

   We would use the abstraction on Controller, model and view class that extend the varying program requirement. Therefore, we can design subclasses that implement business logic of the four-handed chess, such as FourHandedController, FourHanedModel, and FourHandedView, and our main program can choose which subclass concrete object to create depending on whether it plays original chess game or the four-handed one.

   Alternatively, if both games share pair amount of common behavior, we can adopt the non-virtual interface idiom so that we can benefit from reuse of the code and just implement unique behaviors into private virtual methods to have the system to adopt new requirements. For example, we define setPiecee method as private pure virtual method and implement it to place piece in different chess board.

VI. Extra Credit Features

<u>Undo and Redo</u>

While playing chess, users often want to undo a move they made, or conversely, cancel an undo. To enhance user convenience, we decided to add this feature. We created a GameLog class using the stack algorithm to store each move in undoStack or redoStack. To maintain resilience to change, we isolated the tack algorithm. The challenge we faced was reflecting an undo on the chessboard with a move from the stacks. We solved this by creating a move opposite to the last move from the undo Stack and processing it through GameBoard’s doValidMove(shared_ptr<Move>) as if it were a new move.

VII. Final Questions

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

<u>Jaden Cho</u>

This was by far the largest program I have worked on in a team. I learned that it is crucial to extensively test and debug the code I was tasked with to avoid spending excessive time on debugging after merging everyone's code. I also learned that it is important to plan out the project so that it ensures low coupling and high cohesion. This makes the program more efficient and easier to debug in the future. Finally, I learned that one of the most important aspects of developing software in teams is communication. In particular, every team member should be communicating on what they are currently working on and ask for assistance if needed.

<u>Jaehak Kim</u>

Working on this project in a team setting reinforced the importance of clear communication and the division of labor based on individual strengths. It taught me that effective version control and regular code reviews are crucial for maintaining a coherent codebase and ensuring that everyone's contributions align with the project goals.

<u>Juho Choi</u>

As working on this large project, I have learned that it is very important to set the rules at the beginning on using GIT and testing a program. Otherwise, there might be unexpected errors when merging our tasks. Additionally, it became clear that dividing tasks and delegating them to team members should be based on functionality to ensure consistent deliverables at all times throughout the project.

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

Despite of our effort, there are still area it needs to reduce the level of coupling. For example, in the GameBoard class, the code to construct piece objects explicitly uses constructors of subclasses of Piece class, and this tightly couples the GameBoard and the subclasses. We may adopt simple factory pattern to construct the objects, so the GameBoard becomes loosely coupled to the factory object, and it has no impact even when the subclasses of the Piece change. This will promote to meet the Open-closed principle of SOLID principle.

In addition, many classes are tightly couple by sharing responsibilities, so change in one place affect the other component. As the size of program grows, the impact becomes huge, and it makes the debugging not manageable. Therefore, a solution would be practicing coding to the interface that completely delegate functionality to classes providing the functionalities.

Lastly, we would apply useful design patterns that effectively solves design challenges. For example, factory and strategy patterns we described above.