Rastin Rassoli Cellina Patricia

CS 246 - Project
"Design"

Introduction

The general structure of our program follows the Model-View-Controller (MVC) template. All classes are divided into these three modules. They are either showing messages or receiving inputs from the user (View), interpreting user's inputs (Controller), or handling the logic (Model). View classes communicate with the user. After each interaction with the user, related functions in the controller classes are called. Controller classes interpret those inputs and make them understandable for the model classes and call the required functions in these classes. Models handle the logic and notify the view classes. This cycle led to an appropriate implementation and high cohesion.

Design

Besides MVC, we used strategy and observer design pattern in different parts of our project. We start investigating the design of our project with the ChessView class. As the name suggests it is a class in the View module. Technically, it could be implemented in main, however to improve resilience to change we separated this class from main, and main simply creates an instance of this class and calls the function run(). In this function ChessView instantiates all necessary classes to run the program. Moreover, this function includes a main while loop which never ends. This loop waits for the user's command and when it receives the input, a related function in the ChessController is called and the given input is passed to it.

ChessController calls the required functions in the Board class. However, this class cannot accept raw inputs. For instance, *move* function in the Board class accepts Positions not a simple raw string. ChessController is responsible for these interpretations. After converting raw inputs to accepted inputs the related functions in the board class are called.

```
if (command == "move")
{
        try
        {
                string move_commands;
                getline(cin, move_commands);
                controller.move(move_commands);
        }
        catch (GameError ge)
        {
                cout << "game error: " << ge.get_message() << endl;
        }
}
```

```
void ChessController::move(std::string commands) {
        if (board->get_game_mode() != "game") {
                throw GameError{"current game mode is " + board->get_game_mode()};
        }
        if (commands == "") {
                board->computer_move();
        }
        else {
                string move_commands [3];
                stringstream ss(commands);
                int size = 0;
                while (ss >> move_commands[size]) {
                        size++;
                }
                if (size == 3) {
                        Position from = make_position(move_commands[0]);
                        Position to = make_position(move_commands[1]);
                        char promoted = move_commands[2][0];
                        board->move_promotion(from, to, promoted);
                }
                else {
                        Position from = make_position(move_commands[0]);
                        Position to = make_position(move_commands[1]);
                        board->move(from, to, false);
                }
        }
}
```

Board class uses strategy design pattern for handling pieces and computer players. This class stores the game state by keeping an array of Pieces. Each piece provides an important function called *is_valid_move*. This function determines whether a move is valid or not. By using inheritance, polymorphism, and the strategy design pattern, relevant move logics depending on the selected pieces are implemented.

Furthermore, the strategy design pattern is used in the implementation of computer player classes. Board keeps an array of players. If the player is a computer player the correct strategy for the player is selected between the available options (level1, 2, 3, and 4). Level1, 2, 3, and 4 inherit from the ComputerPlayer abstract class which has a pure virtual function called *play*. This function is used for the strategy design pattern. Board gives the current state of the game to this function and based on its logic it returns a legal move.

We chose strategy design pattern in the two mentioned cases since it helped us to separate the logic of piece and computer moves from the board. This increased cohesion. Piece classes determine whether a move is valid or not based on their own logic. The same is true for computer players. With this design, different levels could be easily implemented.

In order to reduce the repetition of code, Position class is created. This structure stores two integers x and y. Also, it provides a very helpful function called distance_squared, which calculates the distance between two positions. Another supportive class is the Move class. It incorporates all the main information in each move such as the coordinates of the move, the piece, whether a piece was removed or not, or whether the move was a castle, promotion, or En Passant. This class makes the implementation of undo possible.

By using *ComputerPlayer* classes, *Pieces*, *Moves*, and *Positions*, the state of game changes. After this change an observer design pattern is used to notify the observers. ChessView attaches two observers to the board class: TextObserver and GraphicsObserver. Note that we separated the responsibility of receiving input and showing the output into two different classes, ChessView and Observers. Each time after board is updated, it notifies its observers which are the text and graphics observer. Board also passes a string called result to its observers. We used this string to decrease coupling. Moreover, this message reduced duplicate code. Then, the observers are notified and updated accordingly. This was a high overview of the main cycle of our project.

```
void Board::move(Position from, Position to, bool is_temp_move)
{
	if (computers[turn] != nullptr) {
		throw GameError{"computer turn"};
	}
	if (board[from.get_y()][from.get_x()] == nullptr)
	{
		throw GameError{"no piece detected"};
	}
	if (board[from.get_y()][from.get_x()]->get_color() != turn)
	{
		throw GameError{"invalid color"};
	}
	if (from.get_x() == to.get_x() && from.get_y() == to.get_y())
	{
		throw GameError{"no move detected"};
	}
	if (board[from.get_y()][from.get_x()]->get_piece_type() == 'k')
	{
		handle_king_move(from, to);
	}
	else if (board[from.get_y()][from.get_x()]->get_piece_type() == 'p')
	{
		handle_pawn_move(from, to);
	}
	else
	{
		if (board[from.get_y()][from.get_x()]->is_valid_move(this, from, to))
		{
			moves.emplace_back(board[from.get_y()][from.get_x()], board[to.get_y()][to.get_x()], from, to, false, false, false);
			board[to.get_y()][to.get_x()] = board[from.get_y()][from.get_x()];
			board[from.get_y()][from.get_x()]->add_move_counts();
			board[from.get_y()][from.get_x()] = nullptr;
		}
		else
		{
			throw GameError{"invalid move"};
		}
	}
	if (!is_temp_move) {
		handle_next_turn();
	}
}
```

```
class Position {
	private:
	int x, y;

	public:
	Position(int x, int y);
	int get_x();
	int get_y();
	int distance_squared(Position p);
};
```

```
class Move {
	private:
	std::shared_ptr<Piece> piece;
	std::shared_ptr<Piece> removed_piece;
	Position from;
	Position to;
	bool en_passant;
	bool castle;
	bool promotion;

	public:
	Move(std::shared_ptr<Piece> piece, std::shared_ptr<Piece> removed_piece,
		Position from, Position to, bool en_passant, bool castle, bool promotion);
	std::shared_ptr<Piece> get_piece();
	std::shared_ptr<Piece> get_removed_piece();
	Position get_from();
	Position get_to();
	bool is_en_passant();
	bool is_castle();
	bool is_promotion();
	void set_promotion(bool new_promotion);
};
```

This design helped us to easily manage challenging aspects of this project. By using Pieces, we were able to make valid legal moves without worrying about coupling and cohesion concerns. Computer Players made implementing different strategies very simple and the observer design pattern led to effective communication between the Models and Views.

Resilience to Change

We could divide different changes and their solutions:

- Mode changes

  By using the MVC design pattern, we have separated main functionalities of our program. This helps implementing changes related to the modes and user interfaces (input syntax). By only changing or adding new observers we can create new and different interfaces for the game without changing any other classes which is due to low coupling and high cohesion.

  On the other hand, we can easily add new menus to our project by adding new controllers without changing models. Separation of communication with the user, interpretation of inputs, and logic makes future updates and changes to the modes and user interfaces of the game convenient.

- Rule changes

  Although the project is a chess game and changing rules may not be sensible imagine new pieces or moves for some pieces be added to this game. The existence of *Piece* classes makes these changes possible without correcting other classes. By adding new subclasses of the *Piece* class, we can add new pieces with different logics and functionalities.

  On other hand if some rules such as check or checkmate, which are related to the current state of the game, change, we should only update the related functions or add new function only to the board object. If the structure of board needs to be changed, such as a 14x14 board instead of 8x8, again the only class which needs update is the board class unless a rule regarding the logic of the moves for one of the pieces change as well.

  Another update may be implementing more difficult levels for the computer player. This could be done by adding new sub classes to the *ComputerPlayer*

object. By using strategy design pattern for this functionality no change will be required for the board object.

- Fundamental Changes

  We have tried our best to use designs which are very flexible and lead to high cohesion. This even makes fundamental changes possible. For instance, as it is described in the answer to the main project questions, only a few changes are necessary for the implementation of four player chess.

  In general, any change in the syntax, the board structure and rules, and logics of moves only requires an update in the *View*, *Board*, and *Piece* classes respectively. Thus, a fundamental change which includes changes in more than one of the mentioned features requires update in more than one of the classes but still due to the structure of our program the method they communicate with each other does not change which considerably simplifies updating.

Improvements

- Display

  One significant challenge that might greatly affect the quality of the program is the graphic observer and display. Currently, we are limited to only fillRectangle and drawString for the functions in X11. There is a limitation in showing the pieces and the board with only text and colors. Moreover, the colors of the board are limited, and we could not change the font colours, making it more challenging to provide a convenient display. This is an important aspect as chess is a visual game where the pieces and the board itself is represented visually; having unclear visual aspects might hinder the game and understanding of the clients. For example, here we represent the black and white pieces with capital and small letters. People would need to spend extra effort to analyze the board, nevertheless we cannot enlarge the font size.

  Moreover, for setup and undo we have to render the whole board again after each command which is frustrating for the user, especially when the internet connection is not strong. We should save the last undo and setup command in order to update the board with only the last change instead of rendering the whole board again.

- Board

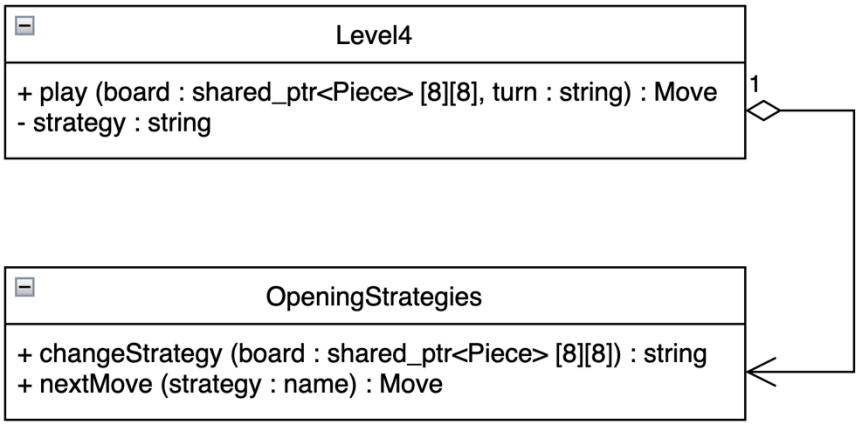
  Although we did our best effort to divide the project into as many classes as we can, still the board class is large. It includes most of the main functions and many classes communicate with it which means high coupling. Thus, for the future improvement this class should be divided into several smaller classes.

Project Questions

Our answers to the main project questions are almost the same as the answers in DD1.

- Chess programs usually come with a book of standard opening move sequences. How would you implement a book of standard openings if required?

  In our current implementation, we have an object called Move. This object includes two Positions, piece name, removed piece (in case of an attack), color, and some other information regarding exception moves such castle and promotion. Position is a simple class which contains x and y (two Integers). By using Move object, opening strategies are simply an array of Moves. With the help of another class, which we call OpeningStrategies, we can implement this requirement. Moreover, we should add a field called strategy to our Level4 ComputerPlayer (assume we are adding the book of openings to this computer player). OpeningStrategies has two main functions changeStrategy and nextMove. changeStrategy inputs the current state of the board. Different opening strategies should be either stored in a file or in a map from name of the opening to the list of Moves. By comparing the current state with different strategies, it outputs the name of a strategy. The strategy feature of Level4 ComputerPlayer should be set to this name. After each move either the move matches the expected move of the strategy or it does not. If it matches, we continue with the current strategy by calling the nextMove function of OpeningStrategies and passing the name of the current strategy. If it does not match, we should change the current strategy by again calling changeStrategy and setting the new strategy. Note that in the case none of the strategies match the move we should follow the older strategy and we will not change it.



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

  We have implemented this functionality. This is our answer from DD1:

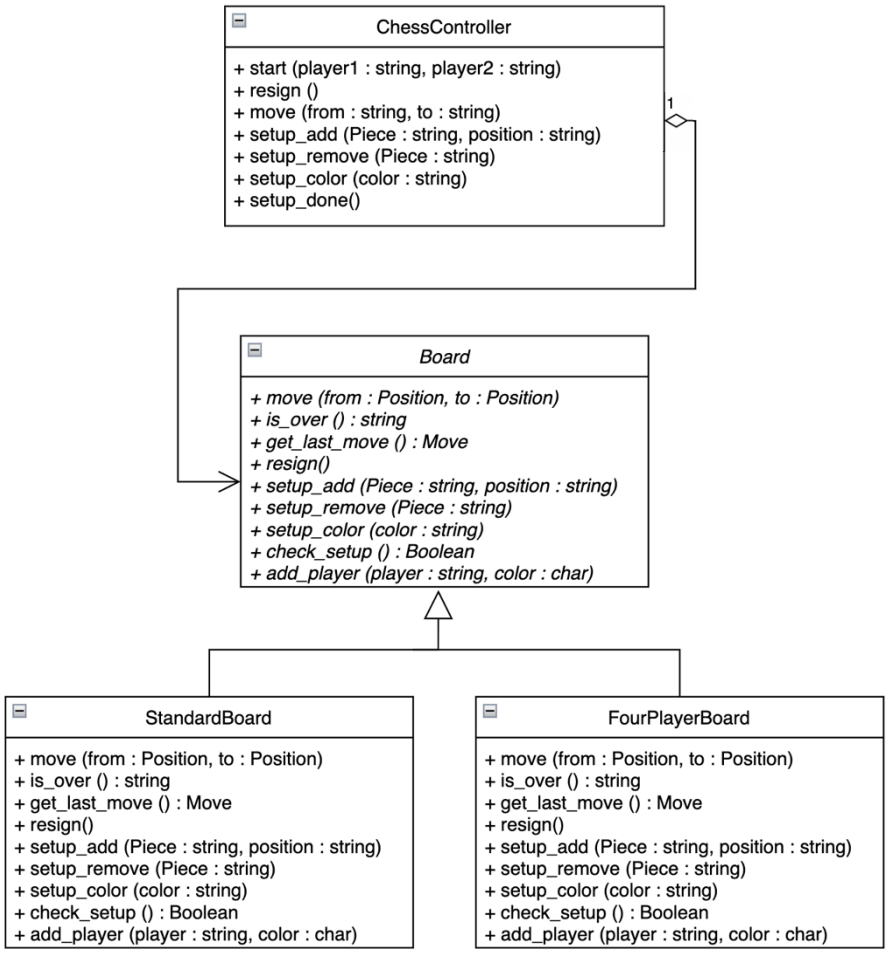
  This is one of the bonuses we have chosen to implement; therefore, with our current UML we are able to add this functionality by adding just a few more functions. As described in the previous question, we have an object called Move. This object stores, from Position, to Position, the removed Piece, and some minor information regarding special moves. Note that all of these elements are necessary for the implementation of undo. Now, by adding a simple feature called moves (Vector<Move>) to the Board, which for any call of move in our game adds the Move to itself, we can implement an unlimited number of undos. When the undo function of Board is called the last element in the moves vector should be popped and the board updates itself with the opposite direction (to to from) and in case of an attack it should bring back the removed Piece.

  Note: When the game is over due to the last move, the player does not have a chance for an undo in this implementation.

- Variations on chess abound. For example, four-handed chess is a variant that is played by four players. Outline the changes that would be necessary to make your program into a four-handed chess game.

  Generally, when we have different game modes with very similar characteristics, the strategy design pattern is very useful. The board in the chess game has very clear characteristics and in four player chess or other variations certain rules always hold. We follow the four-player chess as an example to demonstrate how

we could implement different variations. The first small change occurs in the Piece objects. Now, the Move object has two more colors and it is not just black and white anymore, so Piece objects should accept these two colors as well. Then, we create a new abstract class called Board and our current Board becomes a subclass of Board, and we change its name to StandardBoard. We create another subclass for Board, called FourPlayerBoard. All of the public functions of our previous board become a pure virtual function in our current abstract class Board. Both the StandardBoard and the FourPlayerBoard should implement these functions based on their own rules. Moreover, we put a protected field called players, which is a map from the color of the player to the type of the player (human vs computer), in the abstract class. We still need a minor change in the ChessControler as well. The start function now should also accept the type of the game. Based on the players and the type of the game, the appropriate board should be created and the players should be added using the add_player function. With this strategy design pattern, we can implement other variations as well unless there is a fundamental change in one of the main rules. The following UML illustrates the changes (only the main functions are shown):



Extra Credit Features

As we described in the demo.pdf we have implemented one and unlimited undos, smart pointers, level4, usernames, scoreboards, and graphics notification. We chose them because we found them interesting and challenging, especially undo. Undo was challenging since chess has a many different moves and special moves. This makes the design of Move class a bit difficult. The specific information it needs to store to make reverting back to a previous state always possible was a bit tricky.

Final Questions

- What lessons did this project teach you about developing software in teams?
  Although we had previous programming experiences in developing software in teams, this was a very exciting and challenging project. Drawing conclusions on design decisions and having pre-defined clean code rules were just a few of the challenges we faced. We realized using different team management and collaborative tools significantly increases the speed of developing of the project.
- What would you have done differently if you had the chance to start over?
  We started the project a bit late; therefore, we think our team should have started the project a bit earlier in order to test it thoroughly. Moreover, we did not define a clean code rule before the start of our project and it is not very consistent. We should have chosen them before the start of the project.