CS246 FINAL PROJECT DOCUMENTATION

Group Member: t9sun y62fan h27yin

Changes in Design and UML

We used Model-View-Controller(MVC) pattern to design the whole program. For Model, we have an additional class named Piece; for View, we used provided files to display graphics; For Controller, we just have the main function to control the flow.

We changed the methods in the classes in the UML graph, because we found that methods in the initial UML graph are not enough to design the program well, we added some methods in class Piece and Model.(see uml.pdf) We tried our best to finish the whole program, but we found difficulty to design the AI level-4 and show pictures as pieces in graphical view, so we use letters to show pieces on graphical board.

Design

Model:

When we designing Model part, we found that the most complex part is designing possible moves for all types of pieces. We don't want to do repeated jobs, thus we only use one class Piece to represent different type of pieces(King, Queen, Bishop, Rook, Knight, Pawn). Class Piece has a character field shows the piece's type and a string field shows the piece's position. We will show the methods in class Piece below:

Piece methods:

+ Piece(char type, string pos) — Constructor.

+ get_prom_string(): string — return the string containing what this piece can promote into

+ row_to_num(char): int — return an integer represents row number of this piece (converts character ;a' - 'h' to integer 0-7)

+ col_to_num(char): int — return an integer represents column number of this piece (converts character '1' - '8' to integer 0-7).

+ valid_pos(string new pos): bool — return true if the position is valid on the board, false else.(not be used in the main function)

+ set_pos(string new_pos) — set the string new_pos as the current piece's position on the board.

+ get_pos(): string — return a string to represents the position of that piece.

+ get_type(): char — return a character to represents the type of that piece.('K', 'Q', 'B', 'R', 'K', 'P')

+ set_type(char) — take in a character ('K', 'Q', 'B', 'R', 'K', 'P') and set that character as this piece's type.

+ white_black(): bool — return true if the piece's type is white, otherwise false.

+ possible_move_pos(vector<vector<char>> board): vector<string> — take in a vector of vector of characters which represents all the pieces on the board and return a vector of strings shows the possible move positions of the piece.

Class Model takes a vector of Pieces as a field.

Model methods:

+ get_current_player(): bool — return true to represent white-player, false for black-player.

+ set_current_player(bool new_current_player) — take in a boolean(true for white, false for black) and set it as the current turn' colour.

+ newgame(int a, int b) — take in two integers to start a new game(0:human, 1-4: computer1-4).

+ move(string start = "", string end = "", string prom = ""): bool — take in 0(computer move), 2, or 3 parameters to move the piece. (we did not program the moving for en passant and castling, but the whole program still can run well without these two moving)

+ valid_board(): bool — return true if the board is valid(2 rooks, 2 knights, 2 bishop, 1 queen, 1 king and 8 pawns for each side).

+ setup_add(char c, string place) — setup the new board with the character c and string place as a class Piece and add it on the new board.

+ setup_remove(string place) — remove the Piece with the position place.

+ check(bool wb): bool — take in a boolean wb(true for white, false for black) and return true if wb is checked, otherwise false.

+ get_board(): vector<vector<char>> — return a vector of vector of characters to show the represents board(pass this to View's method to display the board).

+ reset_board() — reset the board as the default board

+ resign() — actually do nothing but in case of using it.

+ check_win(bool side): bool — take in a boolean wb(true for white, false for black) and return true if side wins, otherwise false.

+ ai_move(int lvl) — take in an integer (1-4) to use different AI level(1-4) to play the game.

— different AI levels

- level 1: random moving(possible move).
- level 2: try to capture the other side's pieces and check the other side.
- level 3: try to avoid capturing and check the other side.

View

Class View uses window.h and window.cc for graphical display, View takes a vector of vector of characters as a field and a window as field for graphical display.

View methods:

+ View() — constructor

+ update_board(vector<vector<char>> board) — takes in a vector of vector of characters and display the text for the updated board.

+ invalid_command() — display lots of error messages saying that the command is not valid for the game.

+ check(bool b) — display the message for checked.

+ checkwin(bool state) — display the message for winning.

+ current_turn(bool b) — display whose turn to move.

+ score(int a, int b) — display the score message.(when press ctrl-D)

Controller

In the main function, we "communicate" with player about which function is going to be called for that kind of command. Controller is the connection between model and view. We still have a tiny bug that if one side wins, there will be another chance to move but nothing changed then show the winning message.

Cohesion and Coupling

In our program, we think our code are low coupling and kind of high cohesion. Our modules communicate just via functions calls with basic parameters or results, shared global data, affect each other's control flow. One thing we think we did not do very well is that we design the Piece class as one class for six different type of Pieces which may affect cohesion.

Questions for Due Date 2

Q1: 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?

A1: After whole process of developing software, we've learnt a lot. We know that doing team work ,we should communicate a lot for programming well, and separate the work fairly and everyone try their best to do the job. During the process, we've encountered many troubles for compiling and running, but we discussed and tried our best to find the error and fix it. We think the debugging process helped us for understanding well about C++ material and programming better in the future. For writing large programs, we should brainstorm which pattern we should use for the program and separate jobs to different people, when encountering troubles teammate should discuss well.

Q2: What would you have done differently if you had the chance to start over?

A2: If we still choose to do the chess one, we might want to separate the Piece class into six different classes for different type of pieces for programming in high cohesion. And we will start earlier to study more about graphical display to make the display looks better(using pictures for pieces). But if we can choose again, we may choose cc3k project, because cc3k does not have that difficult graphical display.

Questions in Due Date 1

(There is nothing changed)

Q1: 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.

A: When the computer need to make a move at the beginning of the game,it will have many choices. First of all,the system will evaluate the benefits of all the possible moves even in a few moves later and it will exclude all the moves that have negative benefits,then if a move can have benefits in the range of this engine's logic ,then make that move .Otherwise, compare the current board with all snapshots of saved classic games. Rank those snapshots by similarity, and check if the next move of those snapshots is valid in the current board. If not, the topmost snapshot is threw away and the next one is compared. If the list reaches its end, a random move is made(though this likely will never happen in a non-stalement game).

Q2: How would you implement a feature that would allow a player to undo his/her last move? What about an unlimited number of undos?

A: We can use a stack for undoing the last move. Using stack we can do unlimited number of undos, because we can pop the topmost "move" for each undo. Push each move command into the stack, when calling undo pop the topmost one, before their turn ends, they can call unlimited times of undo until the stack is nullptr. After the undos, we should pass the stack of moves to following methods.

Q3: 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: The view part doesn't change so much, because we pass a vector of vector of chars to the getboard() method and we just ignore the four corners of the board. For the controller, we should change the number of loops in our main function, because initially we only have two players(black and white), but now we have four. The way for calling the model should not change a lot, but for the model itself, we do change. First, AI

should be rewritten, because the data structure changes and now we have to hold for four players. Second, the size of the board changes, so we need to change the method for checking if each command is valid. Third, the way to win the game changes, because now the play has to check other three players for winning.