CS246 Final Project
Fall 2022
Chess
i48ahmed-k433wang-dwwei

Overview:

The program we wrote for this project implements the game of chess for users to interact with. We do so by dividing the requirements of the project specifications into several main components of the project. In this section, we will explain each of the components and their functions. More details about the implementation of these components are described in the **Design** section.

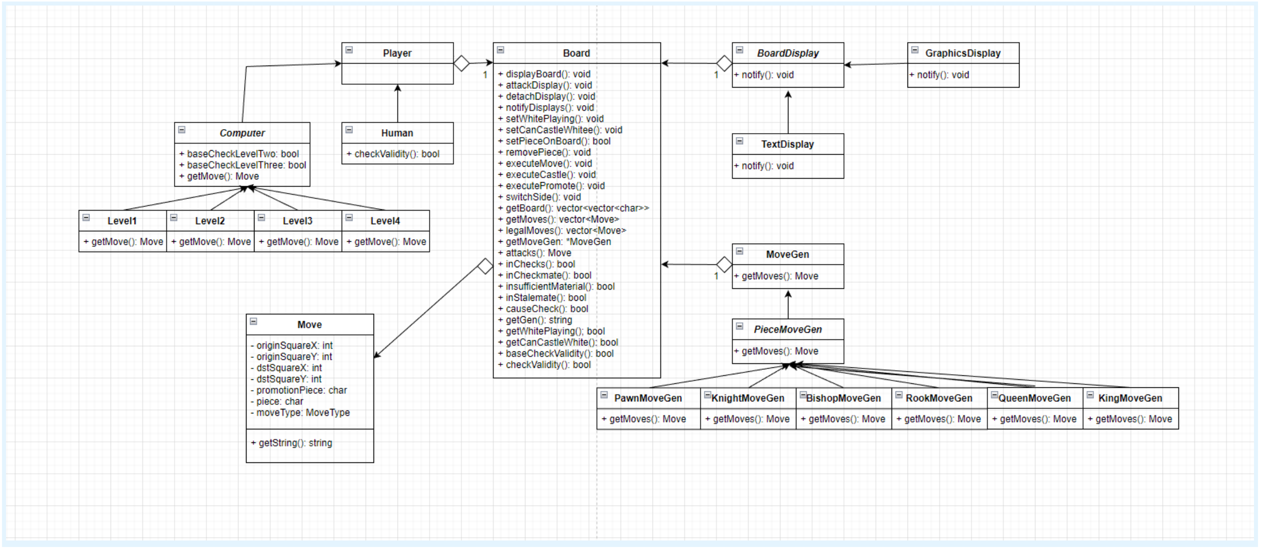
In the test harness, the score of each match is tracked every time the program is run. In `main.cc`, the test harness reads any input commands the user calls, and achieves the purpose of the command by calling various independent classes using the input information. Within this main file, there are also three wrapper functions that deal with the range of method calls each command requires. Finally, detection for the end of games and the start of new games is located in this file.

When a game is started, the program takes as input the player types for the white and black pieces. Since the program has different objectives when the "move" command is called for each, the logic encompassing each player is contained in a subclass of the virtual Player class. In the Human subclass, the program will process the move command by ensuring that the input move is a valid, legal move. On the other hand, the Computer subclass and its four inherited levels generate legal moves from scratch with preference given to the logic of their level. In order to generate such moves, the Computer classes rely on a decorator pattern of the MoveGen and PieceMoveGen classes that layer possible moves depending on the level of computer difficulty.

Another component crucial to the program is the board class. This class encompasses the logic of the chess game, and it holds onto the current chess board and its various states. This class contains various public methods that allow the `main.cc` file to easily update the board when moves are made. Within the board class, values of the game, such as whether a side can castle, which turn it is and if a player is in check are also stored here.

The final component of the project is the display. The textual and graphic displays are both notified simultaneously whenever a move is mode or a position is set up. While the textual display relies simply on the board vector and an ostream, the graphical display works through an observer pattern in its related classes.

Updated UML:



Design:

In order to deal with the various design challenges this project presented, we utilized three design patterns to deal with the challenging components of the project. In order to update the visual state of the board, an observer pattern containing the graphical and textual displays of the board notifies its subscribers (the displays) every time a move is made or a game is set up. Furthermore, two decorator patterns are utilized in the Computer and its respective Level classes and MoveGen and its respective Piece Classes. In both of these decorator patterns, layers of moves are added to the move generating logic behind the program. In the first case, layers of moves are added based on the criteria they satisfy in each level. In the second case, different moves are layered onto the possible moves to be generated piece type by piece type. With these different design patterns, and independent classes that share information through method calls, we were able to overcome the difficulties of the project.

Resilience to Change:

Our program's resilience to change stems from the various independent classes it is built from. Because of this, changes needed to be made to the program are minimized and the changes can be localized to the classes the change affects.

When our chess program is run, a match is created where the score of the white pieces and the score of the black pieces are kept track of. The test harness then supports the list of valid commands and calls the methods of attached headers to accomplish each command. Because of this, any minor changes to the input of the user or the way the game is scored and printed can all be addressed in `main.cc`. The loop that accepts input can easily be updated to include new commands or remove old commands, and if the format of move or game is changed, we would only need to change the way the input is parsed, while the call to the relevant class method and the relevant class would not need to be changed at all. For example if move inputs were now given in Figurine Algebraic Notation, we would only need to parse the input string differently in our `main.cc` loop that deals with "move" to retrieve which piece, origin square and destination square we are dealing with .In this case, our call to the board methods, and the board class which updates the state of the board after the move would remain exactly the same.

Another class that is able to be used independently from the rest of the project is our display. The text and graphic display for our board both have their own separate classes, where the code to produce the two displays is. This means that if the textual display for the board needs to be changed so that the white pieces have lowercase letters and the black pieces have uppercase letters, we can make those changes to the `textDisplay.h` and `textDisplay.cc`, and everywhere this class is called to display the board will not need to be changed. Similarly if the color specification for the graphics display needs to be refactored, we can limit the changes to `graphicsDisplay.h` and `graphicsDisplay.cc`.

The other two main components of the program are also standalone. First, the classes that control which type of player we are dealing with are compartmentalized to the player, human and computer classes. Within the computer class, any changes to the logic of each computer level can be changed in its respective level class, without affecting the others, while logic shared between all computer classes is stored in the virtual Computer base class that only needs to be changed once. For example, the general move generating logic for computers is housed in its own class so changes to it (such as modifying the rules to categorize certain legal moves as illegal now) can be changed there. On the other hand, if the player is a human, the program has different priorities, such as checking whether the move is valid, so only under the human class are its responsibilities run. With this style of implementation, changes needed to be implemented can be done so in one place, and reused code only needs to be changed once.

The final area where major changes to the specification of the project are altered is the board class. This class contains all the logic for the current board state, including checking whether a move is legal, holding onto the current board and performing basic checks to ensure a move passed by a human is legal. Since this class is called frequently, and it has an array of data representing the board, we designed it to minimize coupling. We did so by keeping its fields private, and instead only allowing the customer to access its fields through public methods. This allows for the class to be protected and ensures that function calls only pass basic results thereby minimizing coupling.

Answers to Questions:

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.

Having implemented our program, it would be fairly straightforward to introduce a map containing the names of each opening as the key of each entry, and a vector of Move's representing the corresponding moves associated with each opening in the value field. Whenever a game starting from a certain opening is required, we could simply extract the moves from that opening and input them into the board from the default position. Since the map contains all moves of that opening it would also allow the player to start from any position within the opening (not just the end necessarily). Furthermore, if a user would just like to refresh themselves of the moves and not start a game with them, the raw Move structures can be viewed instead of having to be played one at a time.

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

To undo a single move, in our Board class (containing the information of the current board and game), we could keep track of what the last move captured (type of piece or nothing). Then when the user passes a special "-" flag to the move command (e.g. move e4 e2 -), we would process this like a regular move except we do not check if the undo move is legal, if the previous move captured a piece, we place it back, and finally, the turn is passed back to the player that just undid his last move.

To undo an unlimited number of undos, in every game we could track every single move from the start and queue them in a vector. If we then count the number of moves that have been played, we can subtract the number of undos from the number of total moves and revert the board state to the position of this number. The turn of the player is then determined by the parity of the turn count. After each game, the vector is destroyed and ready for a new game.

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.

To implement four-handed chess, we would first have to modify our board array to accommodate the size of the new board. Then, we would have to update our bourd logic to accommodate for the non-rectangular shape of this board in order to check whether a move is out of bounds or not. We would then modify the text and graphic display to support the dimensions of the new boards, as well as the addition of two sets of pieces (which could be represented by numbers for one player and punctuation marks for another in the textual display and various colors for the graphical display).

However, our methods for generating moves for the AI, and checking if a player is in check would still be appropriate, as would our harness for move inputs, aside from having to refactor the way we check who's turn it currently is in order to allow for four players. Within the test harness, we would also modify the code under the "game" command to be able to read inputs for four players.

We would also need to modify the methods we are using to detect checkmate, as it would no longer be the case that a single player must be checkmated in order for the game to end. Instead, the game would end when either two players are checkmated or all players except one resign. To detect this, we would need to add in an additional loop within the checkmate method to detect multiple checkmates.

Finally, since the game doesn't end when the first player resigns or is checkmated, we would need to modify the scoring logic and detection of the conclusion of a game. To do so each player would be allocated a score which increases every time they capture an enemy piece. When a player resigns, the game does not end unless it is the third player that has been eliminated. The standings at the end of the game are then a reflection of the score of each player.

Final Questions:

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?

This project was a new experience in developing software as part of a group for some of us and a unique chance to work with classmates at Waterloo on a coding project for all of us. As such, each of us took away valuable lessons from this project.

To begin with, one of our very first challenges was the scale of our program. Since the scope of the project was fairly large for this project, as it was designed for a group, and not to be completed by an individual, the depth of the program was intimidating for us. While designing the UML Diagram and the plan for Due Date 1, we had trouble knowing for certain which classes we would need and how we could integrate them together. The project's many specifications were easy to lose track of and it was difficult to decide what tasks should be addressed at which points in the program. In the end, one of the most helpful things was to break the project up into several, independent components that each could stand by itself. This strategy in our plan allowed us to make progress on different parts of the project simultaneously, minimized testing and reduced our problems to a single problem of making these components compatible when they were all finished.

Another issue we faced during the implementation of the project was communication. Since we were working on our program simultaneously, it was important to always communicate with each other in order to understand where each group member was working. This allowed us to average merge conflicts when the program was updated. We also learned about the importance of communicating our visions on what classes to create and how to implement each of them. This way, even if one of us didn't code a certain part of the project, they still understood its function, dependencies and logic, allowing us to always have a clear picture of the entire program.

Finally, we each learned an important lesson on time management. The lesson we learned here was two-fold. At the beginning of the project, we incorrectly assumed that since there were three group members that the time each of us had to invest would be the time needed to complete the project divided by three. However, we soon realized that for a group project it was not this simple. Throughout this project, we found that it was helpful to be present together in planning and coding stages so that we would have a more cohesive approach towards this project. This taught us how to manage our times effectively and arrange periods where we could discuss our project despite our conflicting schedules.

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

With the ability to start the project over with our newfound experience, we would be able to solve many of the minor frustrating problems we had to deal with (various debugging errors, inheritance issues and compiling issues). Furthermore, with a clear idea of the major drivers of a successful program, we could devise a UML and Plan that would allow for classes to be written with much more cohesion and efficiency.

However, these factors would only help us if we worked on another project with similar guidelines and specifications. The broader skills we learned from this project include better understanding the strength and weaknesses of each other and delegating tasks of our plan in a manner that would allow us to take advantage of this. The lessons we learned above would also cause change if we were to undertake another project. With a better idea of the scale of the project, improved communication skills, and a more advanced idea of the time constraints of this project, we would allocate more time for the three of us to discuss the planning stages of the program, where we designed each class and chose tasks. This would allow the actual implementation of the program to go much smoother.