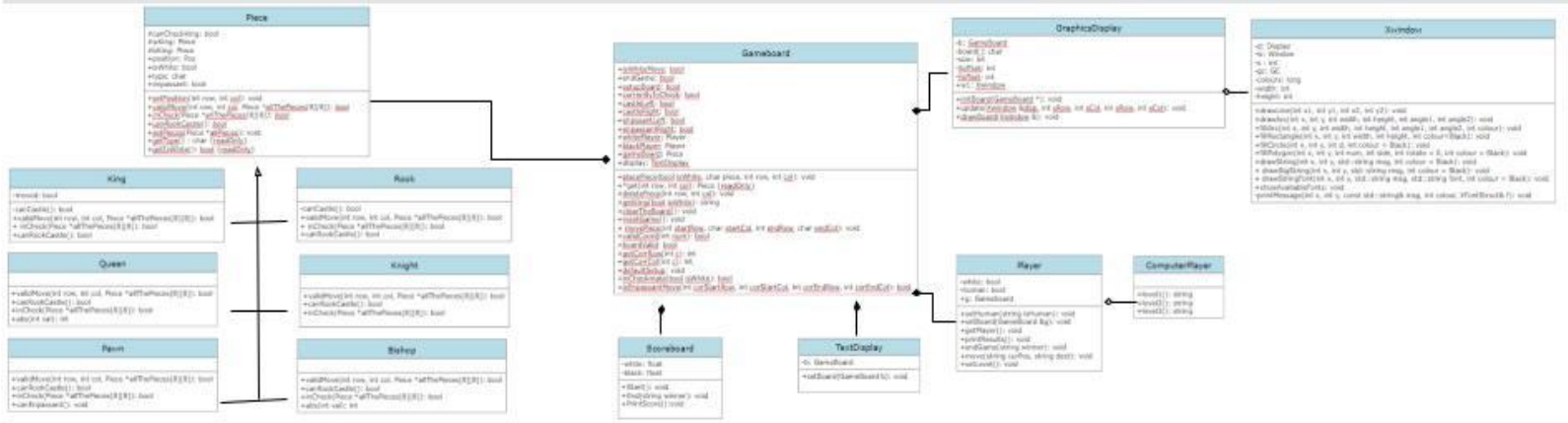
Lina Patrick, Julianne Jorda, David Kim
December 7, 2021

Chess Design Document

Introduction

For the final project of CS246, we created a standard chess game that includes features like graphical display, user input, etc. It also includes the functions of a standard chess game such as en passant, castling, pawn promotion, and of course capturing, check and checkmate. We were each in charge of different parts of the project; Lina working on the piece class and its subclasses as well as the gameboard, Julianne working on the human player and computer player classes, and David working on the scoreboard and display classes. We set clear deadlines for all of our parts and met up in person to put the project together as well as work on the documents side of the project while discussing amongst one another. For the version control, we used GitHub as it is a very popular tool used by many software developers for efficient and effective version control and source code management.

Overview



The structure of our code is by far the most complex one we have created this semester. Due to the scope and difficulty of the project, we decided to break each of the major functions/features of chess into separate classes. This decision made our code have low-cohesion and high-coupling, allowing for us to not only have more flexibility when applying changes but to also have better object oriented design. As well, this allowed each person to work on their parts with ease, and made putting the separate classes/code together into one cohesive project much easier.

The main design pattern used in our project is the Factory Method design pattern for the chess board pieces. Our Pieces class provides the interface for the creation of each type of piece, whether it is the rook, bishop, pawn, king, queen or knight. Our Piece subclasses have a

generalization association with the Piece class, they have an “Is-A” relationship with the Piece class.

An important feature of our design is that the TextDisplay, GraphicalDisplay, ScoreBoard, Pieces, and Player classes are connected to the GameBoard class with a composite relationship. GameBoard is responsible for the creation and destruction of the objects of these classes, making it a “Owns-A” relationship between GameBoard and the other, listed, classes.

Design

We encountered many different design issues when planning and working on the final. Some of the design issues were caused by programming-specific complexities, like creating AI for the computer player, and some of the issues were caused by the complexity of chess as a game, with all of its rules and different piece movements.

One of the issues we were having is that whenever we would move a piece with the move command, but if that move was an en passant move or a castling move, this affects 2 pieces on the chessboard instead of one. This caused a problem because within our GameBoard class we had a method called movePiece which would verify if that piece was able to move and move that piece yet it did not consider any other piece that may have been needed (a rook and/or a pawn). For example, if we were castling we would move the king 2 squares to the left or the right. This would then move the rook as well. This created an issue because we needed to update the game board as well as the graphics display. We were able to solve this issue by creating separate boolean variables within gameBoard which would turn into true/false if we needed to move another piece. Thus, within our main.cc, if those variables were true, we would then update the graphics display for those respective squares.

Another major design issue we had to solve was related to the different chess rules. Chess is a complex game, and in different circumstances pieces can behave differently, or have different abilities. For example, with en-passant we had to track that the move before was a pawn moving 2 places forward, and that it is next to the pawn you’re about to move, along with every other pawn capture condition. To track whether or not enpassant is possible we implemented a boolean variable called enpassant that became true when all the conditions for enpassant were met. Some of the other difficulties we encountered were with check, checkmate, castling and pawn promotion. With some of these issues we managed to solve them with functions, like check and checkmate, while castling required adding more complexity to the move validation functions of the Rook and King classes as well as boolean variables to help implement these moves. The fact that some chess moves affect multiple pieces, such as castling, meant that we also had difficulty deciding where to place certain functions to make the code have low coupling and high cohesion. When checking whether or not the board is in check or checkmate, we decided that it made the most sense to place the inCheck() boolean function into the King class, since it was more cohesive, yet we had to place the inCheckmate() boolean function into the BoardGame class.

This decision lowered the cohesion, however, it was the simplest way to implement checkmate given the restrictions of our code and design.

A major design issue we could not deal with in the end was the computer player. Despite starting early and working efficiently, due to our other work and exams we ran out of time to implement this design feature. Creating a computer player is a complex task, since we would have to create AI and ultimately we decided it would be a better use of our time to make sure that our chess game works correctly, and that it can handle all the other program specifications and design issues. We decided that having an otherwise well functioning program with no memory leaks was better than having a buggy game with a poorly functioning AI. However, when we were originally planning the program, we decided that the best way to implement a computer player would be to make a class that "is-a" player, meaning that it has an aggregate relationship to our Player class. We thought we would create functions corresponding to each level of difficulty, and that we would simply call the needed level of difficulty in the GameBoard class if the user decided to have a computer player. However, as aforementioned, we did not end up implementing this feature in the end.

Errors we encountered and how we solved them

Most of the errors we encountered were in the functionality of the program. When attempting to implement features of chess such as en passant, castling, and pawn promotion, there were a lot of conditions that we had to check for, as well as memory allocation of the pointers for pieces. For example, one error that we ran into was the pawn disappearing when there was an invalid input for a pawn promotion. We solved this error by creating an initial piece in the graphics update function since the issue was that we only created an end piece and we would delete the piece before it moved. Another error we encountered was running into segmentation faults when there was an invalid move. This was solved by running the gdb debugger and seeing where the issue lies; being that we were trying to delete an object that was not there. We also had memory errors throughout which we were able to solve with the aid of valgrind.

Resilience to Change

Our program and design allows for changes to the program specification fairly easily, depending on the specification that is changed. Since our program implements object oriented design with each class performing a very specific function, depending on the change in specification, only limited classes would be affected. For example, any changes to how a piece moves will only affect that specific piece class. However, a change to how the graphics are displayed, for example the colour, would only affect the GraphicsDisplay class. It is one of the main benefits of object oriented design and programming if done properly. Since each class is supposed to be highly cohesive, meaning that all the functions belong together and are doing a particular thing, and the program as a whole is supposed to have low-cohesion, meaning that the classes are not

very dependent on each other, it allows for changes to be implemented in specific classes while not affecting the rest of the code.

When we designed our program we aimed to have high cohesion and low-coupling, and after completing the project we think that we managed to do that. Since we have high-cohesion and low coupling, our code is resilient to changes in design/project specifications.

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

If we want to implement standard openings, we could create a StandardOpenings class that would store functions that would draw the board according to the chosen opening. In the main function, we would create an option for user input that would accept the name of the opening, and based on that name, it would call the needed function from the StandardOpenings class. After that however, the game would continue and our implementation would work.

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

We would need to add a vector that stores a list of the moves, and if the player wants to undo a move, we could "pop" the last move and redraw the board to whatever it would be without that move. Basically working backwards knowing whatever the last move was.

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 make our program into a four-handed chess game, we would first need to obviously change the gameboard appropriately since in a four-handed chess board, the pieces are set up outside the square array and start there. Second, we would need to change the turns and pay more attention to it as it has to go through a rotation now instead of going back and forth between two players. Third, we would also need to pay much greater attention to the moves of the pieces themselves because what counts as a valid move or not will be different. Lastly, the win conditions need to also be adapted since there may be difficult cases such as there being multiple checks at once.

Lina Patrick, Julianne Jorda, David Kim
December 7, 2021

Extra Credit Features

One extra feature that we implemented was in the graphics display where we changed the colours of the game board as well as the pieces. We made the color of the game board green and red to represent Christmas and make it a Christmas theme as the holidays are nearing. As well, we made the text representing the pieces black and white instead of just black with upper and lower case letters. This allowed us to make gameplay easier, since it is immediately obvious which piece belongs to what player, instead of needing to remember what case belongs to what player. This created some unforeseen errors, as we tried to implement this after we finished the program with all black pieces and a regular beige and brown game board. An issue we had, was that when trying to change the colours of the font, we had to create more logic statements so that the code recognized whether it was a black or white character. We fixed these errors by doing basic debugging. We went over the code again and added in cout statements at specific parts of the code as most of the issues were due to the conditions in the if structures and this allowed us to see if the program reached a certain point; passed the condition.

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 taught us that developing software in teams is completely different from developing it on your own, because since we are working on one project together and each part of the program works in conjunction with another, there are certain parts of the code that need to be completed in order for the other person to be able to finish their part of the program. This meant that we could not work at our own pace or procrastinate at all, as we have deadlines to meet that affected other group members. Additionally, we learned the importance of version control and source code management, as we were all working in different editors/IDE's or environments. GitHub was really helpful for this since we could pull and push the changes made by one another with a single command line or the click of a button, but we still ran into some issues here and there with the branches and sometimes we would not be able to pull the changes. Thankfully this wasn't a big issue as you could go back to the previous commit in GitHub, but we can only imagine how difficult it would've been without efficient version control. On the other end of the spectrum though, we learned that developing software in a group was much more efficient and manageable than if you were to work alone. Chess is a big game and it was only possible in the limited amount of time we were given thanks to the fact that we worked alone. We could each take on different parts and put them together which lessened the work required significantly than if we were to work alone. While this also raises some difficulties in terms of understanding each other's code and debugging together since we all have different programming styles, all of us agree that we would still much rather work on it together than alone. We also learned that even though we were each in charge of our own parts, when it came to debugging the code it was

much more efficient, and important to meet as a group to do it. Being in the same room allowed for us to quickly help fix errors in the parts of the code that each person was responsible for.

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

If we had the chance to start over, I think we would manage our time a little better although we did a decent job at it and finished on time, because we were not able to implement the computer player class as it required too much time. In relation to this, we would distribute the tasks that each of us were assigned a little better as well since some classes were more time consuming than others and it was difficult to help and take over someone else's portion when they'd already started and done a portion of it. Another thing we would do differently is plan a little better as well, since most of our time was spent debugging and we feel that if we planned more efficiently and soft coded all the logic clearly and took our time with it instead of just going straight to hard coding all the implementations, we may have saved a lot more time and energy that way. An example of this is not using RAII and smart pointers, because a lot of errors and issues we ran into were caused by not using these things that we learned. If we could start over, we would definitely take some more time to look into things like RAII and smart pointers so that we could be well versed and confident in them and use them to be more efficient and save us resources such as time and energy. Finally, if we could start over we would spend more time to create a test suite in which we could test our code efficiently, because we did not take the time to do this and tested all of our code manually through coming up with tests on the spot. While this worked and in the end we were able to test our program pretty well, it would have been much more efficient to have done so through a test suite than copying and pasting the commands manually every time. This was especially clear when we finished the graphics display and tested in that environment, since we had to run our program more times than we would have needed to if we made a test suite, and each time the program ran we needed to wait for our board to set up which took some time.

Conclusion

In conclusion, we think we were pretty successful in the project as we were able to finish on time and have all of it working correctly, even though we were missing some features such as the computer player. Thanks to the project we were working on being chess, we were able to fully understand every part of the project fully as well as each and every implementation of the code. On top of this, we were able to have fun working on the project as we knew what we were working towards and when we did complete it and actually play a game with each other on our own game that we created, it was very rewarding. While it was very stressful at times, getting together and debugging for 6+ hours, the whole project was overall a great experience to work on and all of us learned a lot that we could take home for our future programming careers, hard skills as well as soft skills.