Final Design Document – Chess

Ajay Mistry, Kapil Bilimoria, Milan Patel

a27mistr, k2bilimo, md6patel

Introduction

The plan of attack that was created for Due Date 1 did not have any major changes over the course of development, other than the fact that the Player class was heavily simplified and did not require logic-intensive methods like move. This is because our implementation had the Board take care of whose turn it currently was, and that moves were directly called on the Board object, as opposed to a Player calling move, which was then routed through the Board all the way to the Piece. The division of work remained how it was initially planned. A few minor functions were added and rearranged between classes but there were no major changes to the overall structure of the program. A problem that we did face, however, was working on shared code, so one member's work would essentially interfere with another's work on the same class leading to some counterproductivity. Upon realizing that this was an issue that could create larger problems for us further along the project, we decided to pass along work once we deemed it safe to do so, thus eliminating the event of concurrent work limiting our progress and creating issues.

Milan was able to complete the Piece class and its subsequent subclasses fairly quickly, and this involved the functionality for moving a piece and whether a piece can move or not. Once the Piece class was complete, Milan worked on the Board itself, and all of its dependencies and setting the groundwork for it to be linked to other classes. Once the Board and Pieces were completed, we essentially had the basic functionality of Chess complete and could then focus on specific modules and enhancements, like the AI and graphics. Additionally, once the basic functionality was complete, we were able to focus on implementing specific game logic like en passant, castling, and pawn promotion. Kapil worked on the graphical interface using XWindow and optimizing the number of times the board would have to be redrawn. I created the skeleton for the command interpreter fairly quickly, and all that remained was adding specific method calls for placing and moving pieces, along with specific logic for control flow in certain cases. I also contributed to creating some methods in the Board and Piece classes. We did encounter a few issues during the development of Chess, so our initial timeline was not exactly followed. Development was mostly done one day after the initial proposed day, so this meant that debugging, testing, and fine tuning was also delayed.

Our approach allowed us to work on individual components of the program with a low number of conflicts between modules due to updates made by others. The last few days allowed us to integrate our components, fine tune components, and test our implementation.

Implementation Overview

Collectively, we decided it was best to leverage design patterns in our implementation, and the main design patterns we focused on implementing were the Model-View-Controller (MVC), Observer, and Decorator design patterns. Following MVC principles, the Board has an array of Piece pointers which acts as the Model (data/state of the program) as it had the current orientation of Pieces and overall Board state. Our GraphicalDisplay class was the View, and our Observer pattern was implemented

within MVC. The Board was the subject and the GraphicalDisplay class was the Observer that was notified once a change in Board state was made. The Controller aspect of MVC was implemented as part of our Board class. The command interpreter would issue method calls to the Board with certain parameters, and the methods within the Board class had more logic to apply before the Model was changed by routing calls to different Piece objects, or within the Board itself. The Decorator pattern was used in the implementation of graphics as a Graphical Display object was decorated with components to enhance the overall presentation of the program. The Board also HAS-A Scoreboard object which is updated when a score change is required, for example, a checkmate, or stalemate. The Scoreboard also has methods to print game-based results, like when a checkmate, stalemate, or resignation occurs.

Milan devised an interesting yet clever way of representing spaces on a chessboard. This involved converted a letter-number coordinate to an integer value, so that the Piece at a space can be accessed by indexing the array of Piece pointers (see below). Every coordinate on an 8x8 chessboard is mapped to a unique integer, so accessing an array element with an index will return the correct Piece pointer.

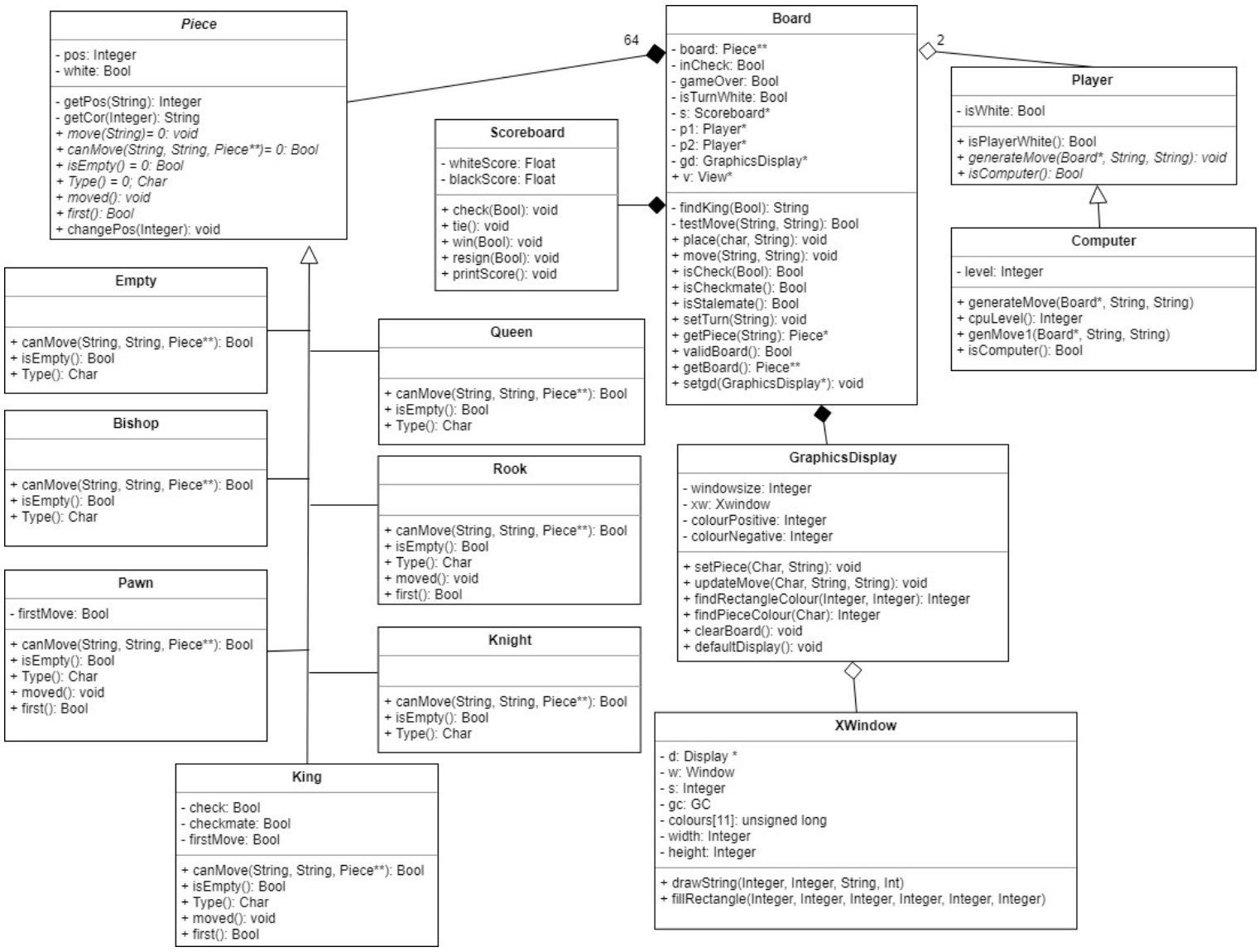
0	1	2	3	4	5	6	7
8	9	10	11	12	13	14	15
16	17	18	19	20	21	22	23
24	25	26	27	28	29	30	31
32	33	34	35	36	37	38	39
40	41	42	43	44	45	46	47
48	49	50	51	52	53	54	55
56	57	58	59	60	61	62	63
a8	b8	c8	d8	e8	f8	g8	h8
a7	b7	c7	d7	e7	f7	g7	h7
a6	b6	c6	d6	e6	f6	g6	h6
a5	b5	c5	d5	e5	f5	g5	h5
a4	b4	c4	d4	e4	f4	g4	h4
a3	b3	c3	d3	e3	f3	g3	h3
a2	b2	c2	d2	e2	f2	g2	h2
a1	b1	c1	d1	e1	f1	g1	h1

In terms of making moves and placing pieces, we decided it would be best for the Command Interpreter to route the call through the Player, then the Player calls the Board's move method, which then calls the Piece's move method. This would allow for a directional flow for a command to run through the system without direct connections between modules. This changed, however, since the Board kept track of whose turn it currently was, and simply switched the turn at the end of each turn, so calls were made directly to the Board object. The Player class is simply used to act as an object that the Board has (Board HAS-A Player(s)) and a container for the Computer class.

Moving on to Pieces, the Piece class is simply an abstract class that acts as a container for the various types of pieces in chess, namely King, Queen, Knight, Rook, Bishop, and Pawn. The Piece class

has pure virtual methods for isEmpty, Type, and canMove which determines if a move is legal and possible given the type of Piece and coordinates. Of all the pieces, Pawn, King, and Queen had the most specific logic involved due to castling, en passant, and their respective range of legal moves. Determining legal moves for pieces that can move any number of spaces in any direction was complex due to the number of edge cases involved, like if the piece was initially situated in a corner, or if it was positioned along an edge.

Updated UML



Design

One of the tougher decisions we had to make in the planning stage, which we acknowledged as a design challenge, was deciding how the game was controlled. Our choices were between the Command Interpreter interacting with the Player class, the Board class, or the Piece class. Choosing any one of these options had the potential of eliminating one of the classes entirely as they may have

become obsolete in the grand scheme of the project. Ultimately, we decided on the Player interacting with the Command Interpreter as this was the option that did not require any classes to be removed, allowing for a structured designed and clear directional flow of information between modules. However, as previously mentioned in this report, our implementation actually had the Command Interpreter interacting directly with the Board, and the Player class acted more as a container for the Computer with a level of abstraction. We found that this simplified user interactions and naturally seemed more intuitive with the Board acting as a liaison between players and the state of the program. Especially since the Board was seen as a centralized class to the program acting like a hub, we felt that the main function should be interacting directly with the Board, with the Board handling much of the specific game logic.

Another difficult decision we had to face was where and how to handle user input and output. As we were developing, we noticed that we were handling output in a variety of classes and it was not organized. However, as one of our goals was to follow the MVC design pattern, we refactored methods dealing with input and output and consolidated them in the "View" of MVC, especially since our "View" was supposed to act as an "Observer" to the "Model". Although not all instances of input and output achieved this, we did manage to centralize much of the I/O, with only a handful of instances where a class would directly output to std::cout.

Resilience to Change

In terms of the implementations of the design patterns, I believe that our design supports the possibility of changes to the program specification to a certain degree. However, if the dimensions of the board were to change, or even the number of pieces, a considerable amount of code would need to be changed. This is because of how a Board is represented within an array. The canMove method which is implemented at each Piece would also face some issues with adapting because there are hard-coded numbers representing spaces due to our decision to map coordinates to unique integers. Some logic regarding the positioning of pieces on the Board and if a Piece can move would have to be refactored to support changes to the program specification. However, supporting more commands in the command interpreter should not be an issue as the code is extensible to additional logic. If a new type of chess piece were to be added, we do not think it will be difficult to implement as it would just be another subclass of the abstract Piece class. The only specific portion that would need to be implemented is this new piece's range of motion. Adding more than two players, regardless of whether they are humans or computers, would be difficult as this would mean that a considerable amount of the Board class would have to be changed to support this addition. For this requirement, more logic would be integrated into the Player class instead of letting the Board handle more than two players as we believe this would be a better design decision for the future. Adding more game logic to the Player class would likely require a large refactor of the Board class to split up the responsibilities between the Board and Player classes.

Answers to the Project Specification Questions

Our answers to the project specification questions did not change from Due Date 1.

Question: 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.

To implement a "book" of standard opening move sequences, we would create 2 hashmaps with one having the type of Piece being the key, and an array of strings to be valid opening moves. The second hashmap would have initial Board states as the key, and an array of strings to be responses to the opponents' moves. For the sake of being one "book", these 2 hashmaps can be wrapped up in a vector.

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

To allow for undos in this game, we would implement a Stack data structure using a vector to store the Board states. Undoing one move would be popping from the vector, and the new "top" of the Stack would be the desired Board state, so the Pieces should be reassigned to align with the new "top" of the Board state Stack. A Stack would allow for an unlimited number of undos as Boards will repeatedly need to be popped from the Stack.

Question: 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.

In order for four-handed chess to be implemented, the Board would have to be enlarged from its initial 8x8 dimensions. Additionally, two more Players would need to be created, along with two more sets of Pieces, and the Scoreboard would need to be modified to account for the addition of two more Players. During the "setup" block of the Command Interpreter, players would be allowed to play on teams or to play alone. Invalid coordinates, namely the nine corners, would have to be deemed "out of bounds" so a new Board invariant would need to be made and followed. The methods for a "check" would need to be refactored due to the increased number of Kings.

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?

From our work experience, we knew that planning was crucial prior to starting development, and this project did not alter that fact. We were aware that we needed to set realistic and achievable goals and deadlines, and that we needed to meet often to gauge our progress on the project. We also tried to follow a loose implementation of the Agile methodology by having a quick discussion, like Agile's daily scrum, every day on how the project was progressing, any problems we encountered while working, and what we hope to achieve by the end of the day. One thing we can say that we all learned was that we should have broken our tasks into smaller, more specific tasks for a more accurate picture on how the project was progressing. We also learned that effective communication is key when developing software in teams, and we are thankful for our decision to discuss our progress every day.

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

One thing we would have definitely done differently if we started over is building a test suite, or at the very least, construct test cases the moment the UML was completed for Due Date 1. We would have covered all of the game logic in the creation of the UML and could develop our testing strategy with the game logic still fresh in our minds. This would allow us to think of some unit tests to isolate certain parts of the program. Another thing we would do differently is alter our timeline for deliverables to account for tasks not going as planned. This would essentially create a safety net for tasks and we could have backup plans for tasks we anticipate may be difficult. In hindsight, something we would have done differently is make the Player class' exclusive responsibility to interact with the user through std::cin, and the Player class would then process and route additional method calls to and through the Board object. We would also plan and implement exception classes from the beginning of the project. Something else we would have done differently is be more organized with our version control system, GitHub. We feel that we should have worked in separate branches for certain features, or for one member's work at the least, and then merge the branch into master, rather than all of us committing and pushing directly on the master branch.

Conclusion

Overall, we believe this project was a success and that we worked well together to achieve our goals. This project was a great learning experience in planning the design of a large program and allowed us to perform some project management tasks. This project also allowed us to identify and experience all parts of the Software Development Life Cycle (SDLC), and this experience was definitely beneficial in creating and working on programs in the future.