CS 246 Final Project: Final Design

Peiran Tao, Elaine Zhao, Martin Tang

1. Introduction and Overview

Over the past two weeks, we have fully designed and implemented a program that runs a game of chess. We started by reading the project specifications together, discussing a general structure for the implementation and planning how to approach the program.

Ultimately, we decided on splitting the chess program into a few main classes which would work together to run the game of chess.

We identified similarities to Assignment 4 Questions 1 and 2, which also had a game implemented on a grid system, so we decided to start there. From then, we used our existing knowledge on how to write and implement a basic grid with cells and displays, and decided to work from there to figure out what other classes were needed. The first one we identified was a Piece class, since each type of piece in chess has different patterns that it moves in. There would be six different subclasses of piece, one for each type of piece. The second class that we identified we had to write was one that dealt with the computer players and how they would generate their next move. We decided that we would also have a Player class, and subclasses for each player level.

We decided to split the work equally among the three sections of the program we identified. The Player class, including the different types of computer generated moves were written by Martin, the Piece class was written by Peiran and the Grid, Cell and Display classes were written by Elaine, with Peiran helping to implement movement of pieces and checking for checks in the Grid class. We decided on this distribution because it seemed like an equal amount of work per person, as well as code that could mostly be done independently. We wanted independent work to be done as much as possible to maximize efficiency and to accommodate our varying schedules, availability and other commitments. This way, we could write code independently, yet still collaborate on other aspects, such as how these classes worked together. Then, we would meet every few days to discuss progress, and if there were any major changes that needed to be made. The rest of our communication could then be simplified to updating the rest of the group about our progress, if we were running into trouble with any functions, or any other simple communication necessary.

Collaboration for this project was done through a private GitHub repository, as well as a group chat where we would send each other updates and requests for different functionalities to be added to the classes we were working on. Each new class, or any piece of code was pushed onto its own branch, and then merged after talking with the group and testing. Testing was done in the student environment by cloning the private repository into the student environments, and then compiling and running it there.

2. Design

For this project, we employed two design patterns that were taught in lectures: the Observer and Decorator patterns. Since our Grid, Cell and Display classes followed the work we did in Assignment 4 that was implemented with an Observer class, we thought it still made sense to implement it somewhat similarly. The Piece and Player classes both use the Decorator pattern.

2.1. Cell, Grid and Displays

The Cell class is a subject that is being observed by displays. Each cell has the following fields: integers row and column, pointer to a piece object, boolean white which returns true if the cell is a white cell, as well as a vector of observers. If a cell does not contain a piece, the piece pointer is set to nullptr. There are some accessor and mutator functions that assist with functionality elsewhere in the program as well, such as accessor functions for each field, as well as two mutator methods that are used to modify the piece field. The removePiece function sets the piece field to nullptr to signify that a piece has been removed from the cell, and the setPiece function sets a piece into a cell, as well as removes whatever piece might have been there before, if one existed there. These two functions then call the notify function to notify observers that the cell's contents have changed. Finally, we had attach and notify functions to serve as part of the observer pattern. The attach function would attach observers to the cell by adding a pointer to an observer to the vector of observers, and the notify function would loop through each observer in order to notify them that the cell has changed. The observer pattern functionality is very similar to that of questions 1 and 2 in Assignment 4.

The first major challenge we faced was trying to figure out how to implement the movement of pieces. Originally, we had planned to do this through the Piece class, which would modify the grid. However, we found that this meant that we had to pass a grid into the piece class, and that it would be much simpler to implement this as part of the Grid class. This is because the Grid class already had access to all the cells, as well as to the actual grid itself, so it would be much easier to directly modify it. The grid class is also responsible for other important functions with similar implementation, such as checking if a King is in check. Furthermore, it is part of the Observer pattern, where moving a piece would have to update the observers (in the case of this program, the different displays) with the new positions of pieces and whether or not they were still on the grid.

The Grid class is then where the majority of the game functionality takes place. There are many fields within the class to help describe what is going on in the grid.

- theGrid is a 2D vector of cells that serves as the grid for the game. It stores all the cells within the game. The coordinate (0, 0) corresponds to a1 on the chess grid, and (7, 7) corresponds to coordinate h8.
- td is a pointer to a textDisplay that prints a textual representation of the grid onto the console.
- gridState is a char that represents the current status of a game. gridState is set to 'w' if the white player has won the game, 'b' if the black has won, 's' if the game has resulted in a stalemate and 0 if the game is unfinished. The accessor method for this field is used in the main function to let the main function know if a game is still

ongoing or not, and how to update the scoreboard with who has won the most recent game.

- whosTurn is a char that represents the current player who should be playing. whosTurn is ‘B’ if it is black’s turn to move and ‘W’ if it is white’s turn to move. The whosTurn field replaced an earlier set of Player fields which would indicate who’s turn it was. However, we changed it to a simple char representation after we found that the only way that these Player objects were used was to access who the current player was. Thus, we simplified the program greatly by just using a char to check.
- whiteK and blackK are two King pieces that represent the two kings on the board. These are important to specifically keep track of in order to determine when the board is in check, as the two King objects can be used to determine their own positions.
- An istream in is also included as a field so that functions that require input, such as setup and other related helper functions, can take input to create a custom grid.

These fields make up the necessary functionality for the Grid class. Then, vital functions like moving a piece and checking if a king is in check are implemented through the Grid class. Note that the constructor for Grid only creates an empty board with no pieces.

Other than constructors, destructors, accessors and mutators, here are the notable functions in Grid:

- void setPiece(int r, int c, Piece &p) passes in coordinates of a piece, as well as a reference to the piece itself. Then, it calls the cell function of setPiece at the cell at the specified coordinates to set the p the cell.
- bool move(int r1, int c1, int r2, int c2) tries to move the piece at coordinates (r1, c1) to (r2, c2). There are many checks in place before a move actually occurs. First, the function checks that the coordinates are not out of bound, that a piece belonging to the current player is in the starting cell. Then it takes the vector returned by getting all moves and checks if the specified move from (r1, c1) to (r2, c2) is actually possible. If it is, then the whosTurn field is updated and changed to reflect that a successful turn has happened and it is now the other player’s turn, the pieces are actually set and removed on the board and the function returns true. Otherwise, if there is some piece in the way of it being a valid move or if the move isn’t on a correct path, the function returns false also.
- void removePiece(int r, int c) calls the removePiece function of the cell at coordinates (r, c).
- bool isValid() is used during the setup function, and returns true if the requirements of a valid grid (kings not in check, at least one king on each side, pawns not in the last rows) are met.
- void init() sets the pieces for a default board. Setting the 16 non-pawn pieces is hardcoded, which is necessary to ensure they’re placed in the correct positions.
- void clearGrid() clears the pieces from the grid. This function is used in between games to reset the board.
- void setup () takes input from the istream in and sets the pieces given by the user onto the board, as well as takes a player to be the starting player to move.
- bool validPath(int r1, int c1, int r2, int c2, int d) returns true if the move from (r1, c1) to (r2, c2) is actually possible. The integer d represents the directions of the move that are specific for the piece. The information for potential values of d is stored within the

vector of possible moves for each piece. This helper function is used in the kingInCheck and move functions.

- bool kingInCheck() returns true if whosTurn's king is in check. It does so by looping through every cell on the board to find a piece, and if the piece belongs to the opposing player, it checks its vector of valid moves to see if the king is included. If the king is included, then it means that the king is in check and the function returns true.
- The << operator is overloaded so that when outputting a grid, it outputs the text display.

The grid class is where a lot of the important implementation for chess takes place. Since there were a lot of helpful variables and functions already in the grid class, it made sense to implement somewhat similar processes, such as moving and checking for a check which both look at if a valid path exists between two cells for a piece to move to.

As mentioned above, the largest problem we ran into was not knowing how to implement the move function. Originally, this was going to be done in the Piece classes, which would have the ability to modify the grid directly. However, we thought that this would lead to less safety and higher coupling within the program, so we decided to implement it in the Grid class. First, we tried to just loop through the vector of valid moves and stop when either the specified move was found in the vector of possible moves or another piece was found that was "in the way" of this move occurring. However, this didn't end up working since the moves weren't ordered from closest to furthest, and even if they were, depending on the order that moves of the same distance were checked in, some valid moves would also not be considered valid and the piece would be unable to move (such as if there was a piece blocking a rook's movement to the left, and this was checked before a valid movement to the right, but the function we wrote would recognize the piece on the left as blocking the rook's path so the move function would return false). The way we solved this problem was by essentially re-writing the move function to look at moves of different directions separately. Then, we were able to separately consider if there were other pieces blocking the path, and thus create a move function that worked as intended.

The next problem we ran into was that we didn't know how to implement a function to see if the grid was in checkmate or stalemate. First, we started by seeing if the current player was in check. Then, using a similar approach, we decided that we would check through each piece on the board in order to see if it still had valid moves left. This way, we would know if it would be able to detect checkmate and stalemate. If the current player was in check and had no valid moves left, then this meant that the current player would be checkmated. Then, if no pieces on the board had valid moves left, it would be a stalemate. The getState function then returns a char that is used in the main loop to see the state of the board in order to update the game status after every move is made.

The Text Display and Graphics Display are very similar to the code already completed for assignment 4. A blank TextDisplay object is constructed as a 2d vector of chars, originally alternating between ' ' and '_' to represent black/white cells as within the project specifications. Since the two kinds of displays are attached to each cell as an observer when a grid is constructed, they are notified when either setPiece or removePiece are called on a cell. Then, the char at the index specified by the changed cell's coordinates is changed to reflect the new contents of the cell, whether this be removing or adding a piece to a cell. The

Graphics Display creates a new XWindow, where the darker and lighter cells are filled in. Then, pieces are represented using a letter in the colour of the side they belong to.

2.2. Pieces

The Piece Decorator pattern implements the different kinds of moving for each different type of piece. Each Piece subclass has a field for the piece's current coordinates in the grid, a char that represents the type of piece it is, a char representing which player a piece belongs to and a 2d vector of valid moves.

The list of valid moves is stored as a 2d vector for a few reasons. The main one is so we can store pairs of coordinates as well as a direction associated with how this coordinate differs from the starting coordinate.

The main function for the Piece decorator pattern is the virtual void function find_moves(). This function is overloaded as per the specifications of how each piece moves. Each time this function is called, the valid moves vector is cleared, and then all the valid moves are recalculated. This is because the find_move() function will be called at points after a piece is moved or put in check.

The implementation of the find_moves() function takes advantage of the rules of moving pieces. For example, both queen and rook can move horizontally and vertically, and both queen and bishop can move diagonally. To save time, we implemented helper functions inside the base class Piece that add horizontal/vertical/diagonal moves to the valid_moves vectors, so that queen, rook and bishop can call these functions within their find_moves(). For Pawn, Knight and King, the logic for their find_moves() we just listed out all of the possible paths to move.

For pawn, a promotion function is also added that changes its type to that of another piece. Pawn, King and Rook also have an extra boolean field called isFirstMove, which facilitates movements like how pawns can move two moves on their first move, as well as castling.

2.3. Player

The Player class generates moves based on the specifications of the level. Then, in the main function, these moves are passed into the grid to perform moves. We decided to do it this way to prevent the Player class from directly modifying the grid, so it only had access to it but cannot directly make changes. This ensures more safety for the program. Within each level, the grid is still passed in to generate the next move.

LevelOne finds the first piece with a legal move and plays the first move from the vector generated by find moves. LevelTwo finds the first move that checks or captures, by traversing the lists of legal moves until such a move is found. If no such move exists, LevelTwo makes the same move that LevelOne would. LevelThree finds the first move to avoid capture, in the same way as the first two pieces. If no such move exists, LevelThree makes the same move as LevelTwo.

3. Resilience to Change

The design has the ability to support quite a bit of changes. First, the grid size of the grid can easily be changed by modifying the const values of the grid size in the interface to be larger or smaller. After this, the init function for a grid would also probably have to be modified since the non-pawn pieces are hardcoded in, but this would be necessary anyways to add the new configuration of a grid with a different grid size. Alternatively, if the grid size was changed, the setup function still works. Furthermore, the movement and check functions should work the same way as well, which means that the bulk of the functionality stays the same when this is changed.

Other possible changes include the addition of more special moves. Depending on the nature of these moves, the implementation of these moves can be done through either the piece class or the grid class. If, for example, new rules were added such that a bishop could perform horizontal moves, the addition of these moves to the possible moves a bishop could make would be modified in the find_moves function. This could be done quite easily, as functions exist in order to get all the moves in a certain direction (ie horizontal, vertical or diagonal), so this can be done quite easily.

Another change is adding a new type of piece. Similar to the part above, the only part of the program that would have to change is the addition of a new Piece subclass. The general move and check functions would stay the same though, since the Piece subclass would just generate a new list of valid moves. The same thing applies to the Player class, as the next move is calculated based on the vector of valid moves generated in the Piece subclasses.

Ultimately, this shows that our program is well-designed and able to accommodate many changes.

4. Answers to Questions

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. See for example https://www.chess. com/explorer which lists starting moves, possible responses, and historic win/draw/loss percentages. Although you are not required to support this, discuss how you would implement a book of standard openings if required.

Answer: To implement a book of openings, a tree data structure would be implemented. Each node would represent a move. At the beginning of the game, the computer player would perform a sequence of predetermined moves until the opening move sequence requires the player to respond to the opposing player. Then, whenever the computer moves, it checks if the tree is still being traversed. If so, it plays one of the children of the current node. Each child of a node in the tree would be all the different possible moves that the opposing player could make. This is how the computer can follow some predetermined algorithm for the first few moves and then respond to opposing moves. After the opening sequence is finished being performed, the rules for the player level are then followed by the computer player. Within our program, this would be implemented as part of any of the player subclasses, as this would be implemented as part of the functions that generate the next column and row.Furthermore, a field would have to be added, such as a vector or a stack, to keep track of previous moves. If this change was being implemented along with question 2, it could use the same record of past moves. If not, for more simplicity, a record of just the previous move would also work, and this can simply be stored as two integers, and referenced when traversing the tree to find the book’s next move.

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

Answer: There should be a command that reverts the board to the position before the last move. To do this, the previous move should be stored, and a function should be made such that the previous move should be reversed based on the start and end coordinates of the previous move. In addition, if a piece was captured with the previous move, it should be returned to the end coordinate of the previous move. To implement unlimited undos, all previous moves should be stored. The moves would be stored in a stack structure, so new moves can be pushed to the top of the stack, and then moves can be popped off the stack to undo them. Since the records of moves would be stored as previous coordinates a piece has been at, the first item at the bottom of the stack would be the piece's starting position. Then, some condition within the undo function could be added to prevent the stack from being empty as long as the piece is alive. More specifically in the program we have implemented, further functionality would be added within the piece class. A stack field that stores all of the previous moves would be added in order to implement this, and after each valid move is performed, it would be pushed onto the stack. This also allows an unlimited amount of undoes to be performed, as the stack of previous moves is specific to each piece, and it stores all of them.

3. 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.

Answer: The board and coordinate system would have to be different so that the Grid class, TextDisplay and GraphicsDisplay class should be changed. These changes are made to accommodate more players, since the program would need a new coordinate system and board size. The Cell class would remain the same as the individual cell does not change even though there are more players. This is because the basic cell functionalities and fields, such as the colour of the cell and what piece it contains are still the same. Similarly, chess pieces still function the same way, so the Piece class is still the same. The Player class will be changed in a few ways. First, the different computer levels would need different rules in determining the next best move, since there are more possibilities, and the computer player would have to be able to decide whether to prioritise pieces from certain players, as well as choosing between more moves. For example, it may prioritise capturing pieces from the winning player (or the player with the most pieces remaining on the grid). The Grid class would also have to be modified to have new logic to check win conditions, as well as redefining what check and checkmate mean. Furthermore, it would have to be able to see which order players win over others in, as well as when a game is considered over.

Within our program, a lot of the movement implementation would stay the same, as well as how pieces are captured and how to recognize checks, since these are not reliant on how many players there are, but rather recognizing other pieces as not part of the current player's. The other changes would be to the main function, how turns are implemented in grid, and how computer players calculate moves as there needs to be a stronger mechanism to prioritise certain moves over others, since a larger amount of viable moves would be available.

5. Final Questions

1. Question: 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?

Answer: We learned a lot about teamwork, collaboration and writing software in teams. First, we had all met at the start of this project, so we had to adapt to working with new people very quickly since this project was completed over a pretty short period of time.

One main issue that we addressed was dealing with version control, and making sure everyone was consistently pushing the latest versions of their code, as well as communicating when these changes were made. Since we had decided to split the project up to maximize the amount of independent work for efficiency, we had to communicate when changes to certain functions that were used in other parts of the program were made. For example, if someone modified how the Piece classes would generate valid moves, they would send a message informing everyone to pull new code. This worked most of the time, yet still needed adapting to since each change, no matter how minor, should have been communicated as soon as they were pushed just so the rest of the group was updated and could modify the code they were working on accordingly. Furthermore, this would make testing as code was written much easier too.

We all gained more experience with the general structure of using a git repository to manage code, and it was a valuable learning experience to work with a structured workflow. We also learned how to add onto code that others had written. This requires strong commenting, documentation and communication among group members.

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

Answer: There are a few changes we would make if we had to start over. The first would be to plan all of our group meetings far in advance rather than meeting, then discussing when we would hold the next one there. This way, we could plan around future schedule conflicts and meet more. This didn't pose as much of an issue as it had potential to, but the possibility of this happening was still there.

Second, we would plan out our program more and prioritise working on a more robust and detailed plan. We found that our initial plan had some overlapping functions, such as moving pieces being handled in both the Piece and Grid classes. We originally did this to accommodate for whichever route we ended up choosing with the implementation, but discussing this further could make the actual code writing process faster, as we wouldn't have to write both, then discuss which one works better.

Third, we would try to finish debugging earlier to implement other extra features. Unfortunately, due to other commitments and generally being more busy around this time of the year, as midterms and final projects for other courses were going on as this project was being completed, some of us were not able to dedicate as much time as we could otherwise if we planned our time around this better. However, we still were able to accomplish quite a lot.

6. Conclusion

In conclusion, our program is well-designed and able to perform the necessary functions of a basic chess game. We ran into some problems with implementation, but were able to effectively solve them to create a complete program.

We were able to stick to the original plan we made mostly, with the three major classes implementing the design patterns we originally planned to use. The observer and decorator classes lend a lot of functionality to how the game is necessarily implemented.