Design Document

Claire Treacy, Michael Hana, and Simon Chase

Overview :

Our implementation of chess uses the Model-View-Controller design pattern, with the Game class as the controller, the Viewer class as the view, and the Board class as the model. The Game class is in charge of running the game, and controls the flow of logic between all the other classes. We implemented the chess board and its pieces in a Board class with a vector of unique_ptrs to pieces, which are implemented with a base Piece class, and a derived class for each type of piece. Players are implemented with a base Player class, which has derived Human and Computer classes, and the levels of computers are implemented as derived classes from Computer. Finally, the game is shown to the users through the base Viewer class, with derived Text and Graphics classes.

The program begins by creating an instance of Game in the main function, and then calling its method Game::play(), which contains a while loop to take in user input. Users can input any of the commands: “game”, “move”, “resign” or “setup”. When the user inputs “game”, a game is initialised by creating a new Board and Players. The Game class also holds on to the Viewers. When the user inputs “move”, the Game calls Board to get a list of possible moves, and then sends those to the Player whose turn it is currently, which returns a move to make. Game calls Board to make this move, and then calls the Viewers to update with a 2d character array representation of the current state of the Board, passed from the Board. The Game then checks if the end of the game has been reached with the help of the method Board::checkmate(colour), which returns whether a certain colour is in checkmate, stalemate or a different state. If the game has ended, the scores are updated accordingly, and the memory allocated for the Board and Players is destroyed. Then Game updates turn. When the user inputs “resign”, the scores are updated accordingly, and the memory allocated for the Board and Players is destroyed. When the user inputs “setup”, it calls the method Game::setup(), which begins a new while loop for input.

Game::setup() takes in any of the commands: “+”, “-”, “=”, or “done”. When the user inputs “+”, Game calls Board::place() to put a new piece of the desired type in the desired place on the board. When the user inputs “-”, Game calls Board::remove() to remove the piece at the requested position. When the user inputs “=”, Game changes the turn to the one matching the desired colour of Player. When the user inputs “done”, Game calls Board::setupReady to check if it is in an acceptable state to end setup, and if so, it returns from setup mode.

We implemented castling using methods from the King class to check if it's a valid move. In Board::checkMove, we added some checks specific to Moves that are castles. Similarly, we implemented en passant using a field in the Move class, and some checks in Board::checkMove. We implemented pawn promotion using a function we added in the Player class called Player::getPromotion(). The Human override prompts the user to input a choice of promotion, and the Computer classes choose a promotion at random. This also required some checks in Board::checkMove, to ensure the pawn promotion is valid, in which case the piece is swapped within Board using a new method, Board::promote(piece). For

rules like not allowing a player to put their own king in check, stalemates and checkmates, we implemented everything necessary within Board::checkMove.

We made a few changes from our initial plan for implementation as we worked on the actual implementation. For the most part, our implementation follows the plan we had made; we still have the same classes organised in the same way, but we added a few methods and changed the type of return value and argument types for some methods.

In Game, we changed Game::checkEnd() from a return type of an integer to a void return type. This was because we decided it would be easier to implement the ending of the game inside that method as well as the checking of whether the game had ended, rather than using it as a check from which the return value would be used in an if statement in Game::play() within the "move" option, which made things a bit messier.

We removed the method Piece::listMoves() from the Piece class since we decided that it made more sense to handle listing the moves in the Board class, making use of the method Piece::checkMove(). Within the Board class, we added a number of methods which we ended up needing to call for setup within the Game class, since having that functionality happen within the Board kept the cohesion higher.

In Viewer, we changed the argument type of Viewer::update from a 2D character array to a vector of character vectors, since we decided it would be easier to work with. This change was then also made to the overrides of Viewer::update within each of Text and Graphics.

Design:

We used the Model-View-Controller Design Pattern in our implementation for the Board, Viewer and Game classes. Our Game class acts as the Controller, while the Board acts as the Model. This helped us to decide how to split up the biggest tasks the program needed to complete in a way that was functional and logical. We further split up the program tasks between these and three other base classes: Viewer, Player, and Piece. This allowed us to keep high cohesion while minimizing the amount of coupling necessary.

The Game class interacts with the other classes on a very high level, and does not care how the methods it calls from Board, Players and Viewers are implemented. It contains everything related to the high-level running of the game, which mainly consists of determining which methods to call based on user input.

The Board class controls more specific game logic that requires knowledge of the current state of the board. It is somewhat coupled with the Piece class since their functions are highly related, but the Piece class allows for the behaviour of the different types of pieces to be separated out into individual derived classes. Each Piece deals with knowledge and behaviour specific to it, such as its colour, and what constitutes a valid move for it. The derived classes of Piece allow for overrides to determine behaviour specific to the type of piece.

The Player class controls behaviour related to the choice of moves. The Human class does this by taking in input from the user. The Computer class was a challenge to implement, since in order to make a move, the Computer class must have knowledge of the moves it is

able to make, which requires knowledge of the board. We solved this by having the Game class ask the Board class for a list of valid moves the Computer Player could make based on colour, and pass this to the Computer class to decide between. For Level2 and Level3, Game also asks the Board class for information about possible checks and captures and passes this information along as well. In this way, each of the involved classes can perform its necessary tasks without requiring knowledge of how the other classes implement their roles.

The Viewer class controls the output of board information to the user. We loosely based our design for the interaction between this class and the Board class on the Observer pattern, except we have all interactions happen through the Game class. The Game class requests the game state from the Board class and passes it to the Viewer class when it calls Viewer::update(board). This allows these interactions to take place without the Board or Viewer classes needing to hold onto pointers to each other, and makes it so that the Game class holds onto all other classes except for the Pieces which are owned by the Board. The Text Viewer simply prints out the board in text form to whatever ostream it is constructed with in Game. The Graphics class owns an Xwindow, and keeps track of the current state of the board in a vector of character vectors. This allows it to compare a new board to the state it is currently showing and to only make the changes necessary to achieve the new board state, which avoids the lag which can come from making too many adjustments to the Xwindow board at once.

We avoid issues resulting from misspelt input by having an else statement which catches any invalid input and outputs a message to the user letting them know it isn't valid. In the case of player input when beginning a new game, an invalid player input undoes anything that has been done so far to begin the game, and resets Game::game_running to false. In cases where argument inputs aren't valid (e.g. if "+" is input in setup mode, but the following commands aren't valid), we ensured that it will be caught before any changes are made by checking the validity before doing anything else.

Resilience to Change:

We've implemented our program with the goal of making it resilient to change, with as few changes as possible needed to make some changes to the way it runs, and minimizing the number of classes to which changes need to be made for any given change.

In general, we've set up our implementation such that for most changes that affect the overall running of the game, the necessary changes can be made within the Game class, and larger changes like adding or removing types of pieces, players or viewers can be made by creating new derived classes, and then adding or removing a few lines of code from the class that controls those objects to actually use them.

To begin with, to make changes to the size of the board, all that's needed is to change the fields Game::rows, Game::cols and Game::start to implement the desired board size. Thus, the only class that needs changes is the Game class, which makes it a lot less likely that making this change will cause any drastic errors in the way the program runs. More importantly, the Board class works the same regardless of how many rows and columns it's initialized with, as this shouldn't affect the game mechanics.

Changes to the number of players can also be made mostly within the Game class. This can be done by changing the value of the field Game::numplayers. This would likely also require a change to the size and initial setup of the board, which can be done as described above. However, the turn checker and colour system are set up to allow for more players. Changing the number of players would be a little bit more complicated since it would require a few decisions about the rules of the game, such as what happens when one player checkmates another; which players get points from this, and does the game end immediately or only once there is exactly one player remaining? Depending on the decisions made, there might be some modifications necessary to the Game::checkEnd() method. The only difficulty in implementing new players would be that it would be necessary to decide on new colours for them, which could require some slight changes to the Viewers to allow for these new outputs, especially for the text viewer since it would need a new system since capitals and lower-cases is a binary choice. While this change requires a little bit more work and is a bit less isolated within one class, that is due to the large impact it would have on the game, and these changes can be split into smaller groups which each take place within a specific class, allowing for this change to be implemented in sections, and for debugging to happen throughout.

Changes to the rules can be implemented in either the Game class or the Board class, depending on which part of the game they affect. Broader changes that affect scoring (for example, changing the scoring when a stalemate occurs) or the circumstances under which the game ends can be made solely in the Game class by making changes to the Game::checkEnd() method. Changes that are somewhat more specific, such as changes to what counts as a checkmate, would have to be implemented in one of the methods of the Board class, such as Board::checkmate(king_colour). Changes that are specific to a piece, such as changes to how pieces can move, would have to be implemented in the class specific to that piece, within its override of the method Piece::validMove(board, start, end). In each of these cases, the rule changes can be made by only changing one class, which allows for the changes to be implemented quickly and easily, and with little chance of breaking the program; any debugging required after changes are made should be isolated within the one class the changes are made in.

Adding or removing pieces can be done fairly easily. Adding a piece only requires the addition of a new derived class of Piece, with overrides of the base methods similar to how the other types of pieces are implemented. Then, a small change would be required to the starting board in Game::start so that the pieces end up actually being used in the game. Removing a piece can be done simply by removing it from the starting board in Game::start. These minimal changes allow for this process to be quick and easy, and it should be fairly easy to do without causing major issues. In general, changes to the starting layout, such as adding or removing existing types of pieces can be very easily made by changing the starting board in the field Game::start.

Adding or removing types of viewers is very similar. Adding one requires creating a new derived class of Viewer and implementing overridden methods, and adding or removing one can be done by adding or removing the lines of code creating them in the Game class, as well as the lines adding them to the vector of Viewers that Game holds on to.

Similarly, creating a type of Player just requires adding a new derived class of Player and implementing overrides of the methods. Both creating and destroying types of Players require changes to the if-else if-else statement in the while loop in Game::play() to allow users to input only the appropriate types of players.

Changes to the input stream can be made by changing the input stream used in Game::play() and Game::setup(), as well as changing the input stream passed to the constructor of the Human class. Changes to the output stream can be made by changing the output stream passed to the constructor of the Text class. For example, the program could identically write to a file to keep a record of the games.

In all cases, we have tried to minimize the number of changes needed to make changes to the way the game is played, and we have tried to keep changes localised within as few classes as possible so that any necessary debugging due to changes can be as straightforward as possible.

Answers to Specification Questions:

Because the basic structure of our project has stayed very similar from the planning stage, our answers to the questions in the specification of chess have stayed very similar. For a more detailed look into how four-player chess could be implemented within our implementation, we ask that you reference our description of how changes to the number of players can be implemented within the section describing how our program is resilient to change (paragraph 4). Please see our previous responses below:

1. (Assuming we want the computer to be able to use these openings, not to allow the user to view the book of openings.) To fulfill this, we would store the openings in a dictionary, mapping the vector of previously played moves to a vector of standard opening's responses. We would also keep a vector of the moves played in a game so far (as we would for the undo function discussed below). Every time the computer makes a move, we would look up the previously played moves in the dictionary. If we found a match, we would use one of the standard openings corresponding to those previous moves. If not, we would proceed as normal.

2. We would (and did) implement this feature by using a Move structure with fields such as “std::pair<int, int> start”, “std::pair<int, int> end;”, “ and “std::pair<bool ,char> capture”. We would then store a vector of Moves, keeping track of these. If a user called undo, we would return to the state described in the previous Move, returning the piece moved to its original position, and any captured pieces to the current position of the moved piece.

3. Four-handed chess does not change most of the rules of the game, but there are some major differences that must be implemented. Larger board dimensions would be required in order to accommodate 2 more sets of pieces. This also could also introduce the concept of a non-square board (board could be cross shaped). Since

there are four players, the game does not necessarily end when someone is checkmated, therefore the game must continue as long as there are two uncheckmated players. Implementing this could be a challenge since an implementation for regular chess would not check if there are still more players who are not checkmated. Another challenge could be checking whether a player is in check, which is difficult with four players. Determining what to do with pieces that remain of an already checkmated opponent poses an issue as well. Do we keep them on the board? Do we delete them all? These are only a few of the many possible challenges that can arise when implementing four-player chess.

Extra Credit Features:

We added a few extra credit features to our program. To begin with, we avoided handling our own memory by implementing everything using smart pointers (particularly unique_ptr) and vector. This took some planning and time to implement since we needed to ensure we used smart pointers that met our needs in all cases, and since we added this in after writing a lot of our code, we had to make sure that the changes we made lined up across our different classes in terms of argument and parameter types in function calls.

We also added an extra feature where, when a new move is made, it is highlighted blue in the Viewers, to allow users to more easily distinguish where new moves happened. This involved adding some checks into the Graphics::update(board) which marks squares on the board as blue if they have changed since the last update.

Another feature we added was the ability to undo moves. This was challenging since it required us to keep track of the moves that had been made and their order, and to be able to revert back to previous states of the game without losing any information. We accomplished this using the field Game::move_history, a vector of moves, and with an additional input option in Game::play() which makes the appropriate changes to revert the game to its previous state. One difficulty was making sure that we didn't miss any changes that needed to be reverted or any cases of move types that could have to be reverted. This required planning and testing to work out. This also required the addition of a new field in the Move class to keep track of which piece got captured.

Answers to 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?

This project taught us that when developing software in teams, it is good to get an early start. There are some new debugging issues that can come up due to incompatibility of parts of the code written by different people if some details of the implementation are not discussed ahead of time, and it is good to catch these as early as possible since these can require rewriting multiple sections of the code. For this reason, it is also good to have regular

meetings while code is being written, even if it has been split up into sections that can be worked on individually. This also allows discussion throughout the process of coding about how other related sections are being implemented, which can lead to more cohesive code overall. We also learned that setting early deadlines is a good way to get things done on time. We set early deadlines, and although we tried to stick to them, some things took longer than expected, so we ended up still being short on time towards the end of the project. It was a good thing we had set our deadlines as early as we did, since this allowed us much more room for error.

Another thing we learned was that it's a good idea to find out what everyone's strengths and weaknesses are towards the beginning of working together, since this allows work to be distributed in a way that makes the project more efficient, and leads to a well-made end product. We also discovered that modularization of the code not only has benefits relating to debugging and code organisation, but also allows for work to more easily be split up into sections that can be worked on individually and merged together later.

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

If we had the chance to start over, we would have more meetings earlier in the process, so that we could ask each other questions about the different parts of the implementation as we worked. This likely would have avoided some issues we ran into where we had to rewrite a few sections of the code to work together better. We would also put more time into planning out how we would debug and test certain sections of the code. Some sections were easier to debug than others, and it would have been better to go in knowing this and with a plan to approach the more difficult sections. We also think it would have made our lives easier if we'd come up with a basic outline of a style guide towards the beginning of the project so that we would have had more consistent code between the three of us. Finally, we would be more careful about planning out who's working on what files at what times to avoid merge conflicts.