CS246 A5 Final Document

Members of the Group:

Manyuvraj Singh Sandhu, Jackson Howe, Blake Naumann

Introduction

The chosen project is chess. Chess is a board game which is popular worldwide and played almost everywhere. Every chess piece has a unique list of available moves and each piece behaves differently. Moreover, features like castling, capturing, en passant, and pawn promotion make the game more interesting and more challenging and fun to think about the game's implementation and code design. In addition to this, the differences in the implementation of a human player and different CPU levels (specially, CPU4) was an interesting and challenging discussion. Our team decided to implement the game of chess and the design, overview and key features of it's implementation are described as following:

Overview

Our project has a `main` interface and implementation files which are responsible for printing the scores of players, and indicating when a player wins or if the game results in a stalemate. `main` is also responsible for the setup mode of the game. It includes the headers for all the other interface files created in this object as it calls different functions from different files.

We created game interface and implementation files which include the `game` class which is responsible for getting the state of the game, contains getter and setter for the chess board, handles implementations including pawn promotion, castling, en passant, and validation of the board before leaving setup mode. Most importantly, it is responsible for the game actually being played and getting the result of the game if either the game is incomplete, results in a tie or if there is actually a clear winner.

There are board interface and implementation files, which includes a `board` class which is responsible for updating the chess board. It includes methods which count pieces and find pieces on the current chess board. The `board` class also determines whether a player is in check or not. It also has methods to set itself to a default starting position and clear its pieces.

We used polymorphism at a number of places while coding this chess game. For example, we created the `piece` superclass which has six subclasses for each individual type of piece: `bishop`, `king`, `knight`, `pawn`, `queen` and `rook`. Each piece has getters and setters for its colour and position. These subclasses have a method `updateValidMoves()` that generates all valid moves the piece can make and stores them in the piece. Each piece can then implement its own logic when determining if a move is valid for itself through a method `validate()`.

Now, to implement different players who play the game, we implemented the `player` superclass. It has `humanPlayer`, `cpu1`, `cpu2`, `cpu3` and `cpu4` as it's subclasses. `humanPlayer` is responsible for getting manual commands and determining the player's move. As described in chess.pdf, `cpu1` does random valid moves, `cpu2` prefers capturing moves and checks over other moves, `cpu3` prefers avoiding capture, capturing moves and checks. For the implementation of `cpu4`, we added an optimization to `cpu3` so that it would avoid causing stalemates and ending the game in a tie. We also created a `playerFactory` which created players as per the command.

In the view subsystem we had 4 classes, an abstract `View` class, which had a `TextView` and `GraphicsView` class that inherited from it, and the `Xwindow` class from Assignment 4 Question 4 which was owned by `GraphicsView`. When a `View` subclass object is initialized, it prints out the current board when the constructor runs. The public method `printBoard()` calls a private virtual method `printOutput()` that takes in a starting position and ending position as a parameter. The `TextView` subclass will reprint the whole board, as it is not very resource-intensive to do so, but the graphics view will only call the `fillRectangle()` and `drawString()` methods from `Xwindow` for the two tiles specified by the start and end parameters, as to minimize the changes needed to redraw the board.

Updated UML

Please see the UML.pdf file for the updated UML.

Design

At the highest level, our goal was to separate the program into 4 subsystems – game, view, board, and player – that attempted to optimize the trade-off between cohesion and coupling, thus creating a lowly coupled program in which each subsystem was highly cohesive. The board subsystem includes the classes that implement the board, as well as the classes that implement each type of piece. The cohesivity in this module comes from the fact that each class offers functionality that would physically affect a game – whether it be moving a piece, or determining what moves a piece has that are valid. The game subsystem deals with all meta controls of the game – for example, changing a player's turn, and determining if the game has a winner. This subsystem only includes one class, so it has maximal cohesion. The view subsystem deals with displaying the game to the user, and each class in the module deals with only displaying the board to the user. We aimed to follow the Model-View-Controller architecture with the board, view, and game subsystems, whereby the model is the `Board` class, the views are the text and graphical displays, and the controller is the `Game` class. The player subsystem consists of all classes associated with player creation and functionality, thus is quite cohesive.

While prioritizing system cohesion, we also aimed for low inter-module coupling. We aimed to achieve this via only connecting different modules through association links, if need be, but ideally through functions calls that may pass objects from one subsystem to another. A main design decision we made was opting to create a data type that represented a position on the board, instead of an object that would represent a move on the board. Instead of passing around a move from module to module, we passed a tuple of positions. Our position data type, called `pos` is only 8 bytes, thus calls passing this data type are less intensive on the program's memory management facilities. One drawback of this approach was the fact that different subsystems are more highly coupled than if we were to create a move object. An example of this increased coupling is the game and board subsystems having to interact slightly more intimately to perform castling and en passant functions. While this is not ideal, using a position data type as opposed to a move object was discussed in detail in the planning stages of our program, and we believe it was the correct choice. One other decision we made to achieve lower coupling was to define virtual methods in the abstract `Piece` class that aided in castling and en passant functionality. While pieces other than the king, rook, and pawn were forced to create an empty implementation of these methods, we felt the alternative of giving the `Game` class increased knowledge about each individual piece would create much higher coupling than our current program. Another aspect of the program that leads to higher coupling is that the `View` class has an association link with the `Board` class. Attempts were made to design a program that did not have this association, but due to the specification that views were to be re-rendered with as little work possible, we thought it necessary they have knowledge of the current board.

In addition to higher level architectural considerations, our program utilizes a number of concrete design patterns. As mentioned above, the program utilizes the Model-View-Controller architecture, and we also used the observer design pattern with the game and view subsystems. The pattern is slightly augmented, as the `Game` class does not inherit from any abstract subject class, which we determined was okay, due to the fact that there is no higher level of abstraction than a game. The observers are each type of graphical display; the current implementation of the program specifies two observers – one text display, and one graphical display – but this specification can be expanded, as discussed in the resilience to change section. The concrete subject is the `Game` class, and although the traditional observer design pattern might dictate the `Board` class be the subject, we thought it best to make the `game` class the subject as it would maintain minimal cohesion between modules. Another design pattern we utilized in the player subsystem was the factory method pattern. Our `PlayerFactory` class takes in the string that the user inputs to `stdin` and creates one of the 5 types of players accordingly. As depicted in our resilience to change section, this makes it easier to deal with the addition of new player types. Finally, in some of our subsystems, specifically in the board subsystem with the `Piece` class, in the view subsystem with the `View` class we tried to follow the non virtual interface idiom for our most important methods, `isValidMove()` in `Piece` and `printBoard()` in `View`. This allows us to ensure that our design is flexible when it comes to ensuring pre and post conditions for these methods, as well as protecting against any malicious users.

Resilience to Change

Much of the consideration about the architecture and design of the program was in an attempt to make our program flexible should specifications be changed. At the highest level, the subsystem architecture is in effort to limit any major changes in functionality to at most one module, without having to totally refactor the remaining modules. For example, If a new piece type was added, we wouldn't have to change much code in modules outside of the board subsystem. Also, the `pos` data type we created has methods associated with it for processing input. If inputs were to change, our idea is that only method `convertPos()` would need to be adapted. This method takes in a string as its parameter, thus should offer much versatility when dealing with different input types, as almost all input that a user would give would either be in a string, or could be converted to a string. Furthermore, we store an internal representation of the board as a 2-dimensional array of indexes, labeled x and y, with each dimension ranging from 0-7. This allows us to accept input of any syntax, and convert it to a position easily. Furthermore, our approach allows for a variable sized board. Although the board is currently stored as a 2D array, since most of our functions loop over all rows and columns, (i.e. `updateValidMoves()` is called on all nonempty tiles on the board each time a move is made), we are able to accommodate different styles of boards with extra loop conditions. For example, in 4 player chess we could store a 2D array encompassing the board, and enforce the condition that the 4 4-tile corners are not considered part of the board.

Adding on the versatility of how the board is stored is the encapsulation of piece logic. Each piece stores a set of its own valid moves, thus making it easy to update each piece's valid moves each time a move is made. If rule changes such as being able to hop over your own player's piece if you are making a capturing move were implemented, we would be able to just change the piece logic inside of `updateValidMoves()` for desired implementation. Having the pieces store their own logic makes implementing new computer logic rules for determining moves easy as well. If a computer needs to choose a move, we can always start by generating a vector of all valid moves for each piece, with the method `getValidMoves()` for that computer to perform its logic on. Also, since we follow the non virtual interface idiom with the `validate()` method in `Piece`, we can ensure that pre and post conditions are satisfied. For example, if there was a rule that the opposing player could "lock" one of your pieces for one move, we could store a Boolean field `isLocked` in `Piece`, and before piece calls `validate()` inside of its `isValidMove()` method, it could ensure that the piece is not locked, thus ensuring the preconditions of the `validate()` function swifty and simply. Using a combination of the non virtual idiom used in `validate()`, and the fact that a piece stores its own valid moves, we could provide an alternative approach to our current one if we were dealing with opening or closing move sequences. We could specify a `state` field for each variable, and ensure that the piece satisfies the preconditions for the `validate()` method before `validate()` is called, such as a pawn being able to en passant, even if said pawn was initialized in setup mode.

Another design pattern we used that offers resilience to change in our program is the `PlayerFactory` class. If other player types were required in the specification, all we would need to do is add another condition to our factory `createPlayer()` method, and we would be able to create a new piece. Since all pieces inherit from an abstract `Piece` class, there would be no need for refactoring the code that calls any method on a subclass of `Piece`. In addition to the factory method pattern, the combination of the Model-View-Controller architecture and observer design pattern associated with the board, view, and game subsystems provides a great deal of versatility. If we wanted to add more views, or different types of views, we could simply create a new `View` subclass and add this `View` object to each game's list of views. If we needed one observer for each tile of the board if we wanted to implement some really fancy graphics, we could also do this by augmenting the `printBoard()` method to not take in any positions, store each tile view in `Game`'s list of views, and our `notify()` method could call `printBoard()` on each observer.

Answer to Questions posed in chess.pdf A5

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.

In order to have opening moves, and be able to respond to following moves, we would use a tree structure that branches at each move of the opponent. For each move the opponent makes, it leads us to a corresponding move to make, down the tree. This tree would contain references to board layouts corresponding to each given point in a tree. The board layouts will be stored as 2-dimensional arrays or vectors of pointers to objects of the `Piece` class. This way, to follow the move from the book, we can copy the board configuration from the current position in the tree. We recognize that this saves a fair amount of redundant information, and it would be potentially more optimal to store a tree with position pairs representing the move to be made, however this would require a more complicated decoding of the opponent's move, rather than a simple board comparison to determine what branch of the tree to follow. It also makes it more difficult to execute a move, as it requires either passing the optimal move to the user, or passing it directly in as input, rather than copying the board state. We expect to have a function that swaps the current board with a new one, which is involved with validating moves, so this swap function would already exist, and would thus be easier for us to implement.

This implementation assumes that the book only considers “good” moves for the opponent to make, and assumes that discussing the first “dozen or so” moves means between both players, not “a dozen or so” each. If either of these assumptions are not met, the size of the decision tree would become unreasonably large, even if we stored the moves more efficiently, such as in a tuple of positions.

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

Similar to the opening handbook, we would store previous board configurations. In the case of a single undo, we would store the previous board configuration with a simple pointer or reference to the 2-d array or vector in a variable. In the case of multiple undos, we would store previous board configurations by pushing pointers onto a stack data structure, and popping them off to facilitate an undo. We are confident that our stack will be large enough to store all previous moves since there are only finite moves a player can undo.

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.

First, we would have to modify the board size and shape, and would likely store it as a large square, with some values in the corner areas to specify those as non-playable spaces. This means we would have to modify the `updateValidMoves()` method to check if a move crosses a non-playable space. We would also need to expand our `getState()` method that checks to see if a move puts an opponent in check or checkmate, as well as if it puts the player in checkmate, as there are now two other opponents to consider. Of course our setup mode and game command interpreter loop would be modified to support four players rather than two, and we would need to keep track of four players’ results. Finally, we would need to add new cases into the `updateValidMove()` method for the `Pawn` class, to facilitate pawns moving left or right as part of the two new opponents.

Extra Credit Features: When we made `cpu3`, we noticed that even when it was put up against an opponent that should be much weaker, like `cpu1`,it would often end up in a stalemate, and would tie the game as a result. We implemented `cpu4,` which is similar to `cpu3`, but it avoids stalemates by checking if the opponent can make a move after a given move, and if not, it chooses another move instead. This way, although it still does not know any good strategies of searching for a checkmate, other than putting the opponent in check, it will at least not accidentally tie the game when it could easily win, as it often would tie with multiple pieces, while the opponent was left with just a king.

Final Questions

Question: What lessons did this project teach you about developing software in teams?

Manyuvraj Singh Sandhu: This project was as challenging as it was fun and interesting for me. Working with two outstanding teammates made the job much less stressful and much more interesting for me. Developing software in teams was fun especially, during the brainstorm sessions, discussions and meetings that we had to discuss about different implementations and the solutions to the problems raised along the way. This project was a mix of group and individual work. I gave my suggestions during discussions, asked for help whenever I got stuck or whenever there was a bug in my code and listened to my teammates' feedback. This project taught me some valuable lessons like teamwork, adaptability, creativity, interpersonal skills, problem solving skills, time management and most importantly, to work efficiently under pressure. I know that these valuable lessons will help me in the future both in a work environment and generally, in life.

Jackson Howe: Initially, this project seemed daunting at first, as are most large-scale technical projects. I learned that it is important to follow the development process, rather than just eyeing a result right away. The design brainstorming part of the assignment was the toughest conceptually, and definitely the part that I felt attacking as a team was most helpful. Also, I realized that it is easy to become overwhelmed by a project of this scale, but consistent communication, and intelligent division of labour between teammates allows for continuous and smooth development. There were numerous times where our group implemented pair programming, by which one person would program a portion of a method, and if they got confused, stuck, or tired, another person could step in and finish programming it. This was a really efficient and successful tactic, and one I hope to use in future large-scale group projects.

Blake Naumann: I went into this project with a pretty negative opinion on group projects. I found that when I had done them previously I found it hard to get other group members to do their part, and I ended up doing most of the work, and losing credit for my group-mates' lack of effort. This project made me realize that those feelings were a direct result of group projects in high school, and not a reflection of what university (and hopefully in the workforce) group projects are like. I found that my teammates were even more inclined than I was to get going and do well on the assignment. This project also forced me into learning a lot of things regarding the actual writing of the code, like remotely connecting to school servers with VSCode, as well as using git for version control. I also found that working in teams allowed us to come to an intelligent decision or solution quicker, as there were always 3 minds and 3 perspectives looking at a problem. There were times when I had no idea how to best solve a problem and a teammate said something that immediately clicked in my brain as being a great idea.

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

Now that our team has done a full implementation of the chess game once, if we had the chance to start over and make this game again, there are a number of things we would keep in mind before the start and there are a few improvements that can be made. For example, we would check if the function calls can be minimized where it's necessary (look for if function calls are being made in `if, else if` clauses repetitively and change that), for example, implementing a `PieceFactory` class with a `createPiece()` method, much like we had with `PlayerFactory()`. Moreover, we would look to continuously optimize our program, like checking to see if we can move any methods to a superclass, and participate in more frequent testing. It can be stressful and frustrating to debug compilation errors for such a big project that contains a lot of files. We would probably prioritize working on the graphical display a little bit more, although we did spend a fair amount of time attempting to learn about the X11 graphics library. We could've also chosen certain variable names better, and cleaned up our code right after we finished programming a small portion of the program to aid in group communication and cohesivity. Although our code uses some object-oriented programming skills, we would still check if more object-oriented programming skills and design patterns discussed in the lectures can be used somewhere in the code, to make it more efficient, such as implementing more factory methods as stated above, and perhaps following the non virtual idiom for more classes.

One aspect of the project that we didn’t get to tackle was implementing higher level cpus for more bonus points. If we had managed our time slightly better, we may have been able to make an n-level decision tree for higher level cpus.

Conclusion

Overall, our group fulfilled the project specifications to the best of our ability. This project was fun and challenging at the same time. Each team member worked on their individual responsibilities and gave their suggestions in discussions regarding design and implementation of the project. There couldn’t have been a better project for this course that puts every topic we have learned so far in this course into use. This project taught us some very valuable skills and lessons that will be helpful in the future and more importantly, in our work spaces and professional lives.