Final Design Document

Jonathan Cheng, Fred Yang, Sophie Yang

Introduction

For our project, we chose to design and implement a game of chess. Our game includes a command interpreter to take in text-based commands, and provides both a text-based display and a graphical display of the game board. Our implementation supports all the standard rules of chess, including the standard legal moves of each piece, as well as special rules such as en passant, castling, pawn promotion, and the 50-move rule. Our system accommodates both human and computer players, where the computer player is able to operate at one of several difficulty levels, the highest of which uses the minimax algorithm to decide the next move of a player.

Overview

Our overall project structure follows the MVC architecture. The lists below show which classes belong in which categories.

Controller:

- Game
- Player + subclasses

Model:

- Grid
- Piece + subclasses

View:

- BaseDisplay + subclasses
- Xwindow

Starting Game

In order to start an actual game, a `Game` object must be initialized and have startInterface() be called. This enters the main menu input loop, which allows the user to start a game or enter setup mode.

Setup Mode

Setup mode starts when the user enters “setup” in the main menu. This will enter an input loop that allows the user to enter different commands to directly modify the state of the board. The `Game` object achieves these modifications by making calls to public mutators defined in `Grid`. The input loop concludes after the user enters “done”.

Playing Match

A match begins when the user enters “game <player> <player>”. The `Game` object enters an infinite loop through startMatch() where it will performs the following: Display the current state of the board, alternate between calling Player::getMove() for the black and the white, modifying the state of the board using the received move, and checking for win/draw conditions to exit the infinite loop.

Getting Moves

Player::getMove() is a pure virtual method with polymorphic behaviour. In the case of any of the `CPU` classes, this entails a premade algorithm which takes in the state of the board and determines some move to return at differing levels of sophistication. In the case of the `Human` class, this entails entering a input loop which prompts the player to make a move until they enter a legal one.

Determining Legality of Moves

A `Move` is defined by a tile to move from, a tile to move to, and optionally a piece to promote to. Obviously, trying to promote when the “from” piece is not a pawn on the second last rank is not legal. Beyond this edge case, determining if a move is legal is a 2 step process. (1) “Visible” moves are determined. These are moves that simply adhere to the basic rules of chess for a specific piece. These are calculated through the pure virtual method Piece::getVisibleMoves(). (2) For each visible move, the grid is cloned (with all its state information), the move is played on the cloned grid, and it is observed whether or not the Piece’s own colour’s king is now in check. If the king is now in check, the move is illegal, otherwise it is legal.

Ending a Match

A match is ended when any of the 3 following conditions are met: (1) a `Human` player resigns. (2) There are no legal moves for the current player. (3) 50 turns have passed with no captures being made or pawns being moved. (1) is self explanatory. In the case of (2), an additional condition is checked to see if the player’s king is threatened, which would lead to a checkmate rather than a stalemate otherwise. In the case of (3), this guarantees that all matches will eventually terminate, since there is a finite amount of pawn moves and captures that can be made. Once a match end condition has been verified within the startMatch() infinite loop, the loop is broken and the match winner’s score is updated.

Updated UML

Design Decisions and Challenges

Our entire project was modeled after the MVC pattern. Model contained the logic for making moves and updating states, View contained the text and graphical displays, and Controller acted as an intermediary between the two, receiving user input and processing the data accordingly. This allowed us to separate our project into three interconnected components, and streamlined the process of creating new features and merging them together. It also promoted code reusability, scalability, and testability.

We also used the observer pattern, where the subject was the Grid class and the observers were the displays. We created an abstract BaseDisplay class, which TextDisplay and GraphicalDisplay derived from. Then, in the Grid class, we created a vector of BaseDisplays, which were notified every time the grid was updated. As the game of chess is heavily dependent on changes in state of the grid, using an event-driven architecture like the observer pattern was particularly suitable. This also promoted decoupling, as the Grid class wouldn't need to know many details about its observers. Additionally, this would also allow us to easily add other types of displays in the future, without having to modify the core logic.

We also followed many of the SOLID design principles. We adhered to the Single Responsibility Principle, by generally giving each class one single role to play in the larger game. For example, the Piece class is only responsible for representing a piece on the board and its associated behaviors (calculating legal moves, checking if the piece is threatened, and managing its position). It does not handle unrelated tasks such as managing the game state or updating the interfaces.

We also adhered to the Open Closed Principle, by structuring our code with abstract classes (that defines core features and functionality) and derived classes (which could add their own custom logic). This way, objects are closed for modification (as the core logic in the abstract classes doesn't need to be altered), but open for extension (as new derived classes representing other instances can be created and defined). This can be seen in the Piece class, which defines the core functionality needed by all pieces like isInDanger() and isThreatenedBy(), but allows subclasses that represent specific pieces to override functions like getVisibleMoves() to be easily added. This can also be seen in the BaseDisplay class, which defines core functionality, but allows subclasses that represent specific displays to override functions like notify() to be easily added.

Additionally, we followed the Dependency Inversion Principle. In general, we structured our code so that higher level modules didn't depend on lower level modules, and instead on abstractions. For example, in the Grid class, we see that it contains a two-dimensional vector of shared pointers to the abstracted Piece class, instead of specific piece types. The Grid class also contains a vector of shared pointers to BaseDisplay. Again, it relies on the abstraction of BaseDisplay, rather than any specific display implementation. This way, the higher level Grid

class is not directly tied to lower level concrete classes, promoting flexibility, maintainability, and enabling the easy substitution of different Piece types and BaseDisplay implementations.

Resistance to Change

The essence of an adaptable software design lies in its capacity to evolve, be it for integrating new features, revising rules, or incorporating additions without significantly disturbing the existing system. Our chess program has been designed with such accommodations in mind. Below we will discuss some foreseeable additions to our program and how our current infrastructure supports this extension. In later iterations of our chess game, we may introduce non-traditional pieces and unconventional rules to the standard game of chess. Our Piece class is abstract with virtual core functions. This abstraction enables easy addition or modification of chess pieces. It's as simple as deriving a new class from the abstract base and defining its movement rules. Any modifications of original chess without fundamental rule changes can also be added simply by modifying the movement rules of a pre-existing class. Advanced rules such as check conditions can also be set independently without much hassle. A major point in our design is its emphasis on high cohesion and low coupling. With high cohesion, each module is built to serve a specific purpose, making it easier to understand, manage, and modify. On the other hand, the low coupling between modules hides away unnecessary information and ensures that changes in one part do not create a ripple effect in others, enhancing the overall system's stability. This design ensures that the system's other parts remain unaffected even if rules or pieces change.

Another change we may add is expanding on our current selection of computer players with stronger and more robust decision-making algorithms. Like pieces, this addition is done relatively simply, as the logic for each computer player is defined separately and exists as its own entity. The only primary modification needed would be to create a new instance of Player and define its unique getMove() method which handles the decisions made from turn to turn.

One of the major points of potential improvements lies with our current display functionality as it is arguably our weakest aspect in comparison with modern chess interfaces. Thankfully, our infrastructure is designed with that in mind as well. The chess program's display module is decoupled from the core game mechanics and player modules. This segregation allows for an array of display interfaces to coexist without affecting the underlying game logic. A key to this design is the Observer pattern, which enables efficient communication between the game state and the various display interfaces. The strength of this design lies in its flexibility. Adding a new display interface is as straightforward as creating a new Observer and attaching it to the game state Subject. This means that we can add any number of display types - from text-based to 3D visualizations - without interfering with the core gameplay mechanics.

Project Specification Questions

Q1. Chess programs usually come with a book 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 can use a prefix tree to efficiently store and retrieve accepted opening move sequences. Each node in the prefix tree would use a possible state of the board as a key, as well as the name(s) of the standard opening(s) associated with that state of the board. Each edge going out of a node in the prefix tree would represent a possible move that can be made from the state stored at that node. In chess, standard opening moves rarely ever change, so in order to construct this prefix tree, we could simply use some sort of grammar/encoding to store the structure of this tree and we would read it into memory when the program starts. When an actual game begins, the program will start at the root of the prefix tree, it will then display all its child edges as the list of possible standard moves. Each time a player makes a move, it will traverse the prefix tree to find the corresponding new state, and redisplay the associated standard moves in that state. Our answer has not changed at all since last time.

Q2. How would you implement a feature that would allow a player to undo their last move? What about an unlimited number of undos?

To allow a player to undo their last move, we can approach it in terms of "saving" a state of the game. Before every move, we save the entire game's state into a variable - where each piece is, whose turn it is, which pawns just moved two spaces in the previous turn, and which rooks/kings have ever moved. This way, if a player would like to undo their last move, we could simply overwrite the current state with the previous state variable. If we wanted to implement an unlimited number of undos, rather than having a single variable storing the previous state of the game, we could instead use a stack to store all previous states. To undo a move, we simply pop the stack and reload the game into the popped state. This stack would grow with every move that is made in the game, and would allow unlimited undos, given that we have enough memory to store the whole stack. If we wanted to optimize the space utilization of this undo feature, we could store the changes between states rather than the entire state, and parse these changes to reconstruct the past state, rather than storing the entirety of the past state. Up to this point our answer has been the exact same as last time. However, after implementing the game, we realized that there is a common rule where if any state of the board is repeated 3 times at any point in the game, the game will be drawn. This rule exists to end infinite loops of repeated moves quickly, before they reach the 50 turn no-pawn-move-no-capture limit. If this rule is implemented, it would require a some sort of storage of the history of the states of the board. Therefore, there would not need to be lots of additional implementation added to allow infinite undos in the game, since the undo functionality can simply piggyback off of the already existing board state history from the repetition stalemate rule.

Q3. 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 four-handed chess game.
Some of the major changes would include:
Board size: The grid class would need to be modified to accommodate a larger board size
Piece setup: The default starting positions of the pieces would be different.
Setup mode: Setup mode for normal chess validates that there are exactly two kings of different colors. With four player chess, this would need to be four kings of different color.
Player class: The Game class will need to be modified to support 4 players instead of 2. The turn-based logic must also be modified to rotate around 4 players. It would be best to use an enumerator rather than booleans to store whose turn it is.
Win conditions: Some rules and conditions for checks, checkmate must be changed. For example, checkmating one player does not end the game for all players. What happens after one player loses must also be taken into account.
Pawns: With 4 players, some pawns can move left and right instead of up and down.
Scoring system: needs to support 4 players.
Display classes need to render a new layout.
Our answer has not changed since last time.

Extra Credit Features

Smart Pointers

All memory management in our program is handled through smart pointers (shared_ptr) and standard library containers (vector, pair, etc.). There are also no circular references between shared_ptr's, so our code is memory leak free without using any raw pointers.

Help Manual

In any given CLI context (main menu, setup mode, move prompt), the user can type the command "help" which provides a list of valid commands and their argument signatures for the current context.

Default Promotion

If the player does not specify a piece to promote to during a move that would normally require promotion, the pawn is promoted to a queen by default

RNG Seeding

Users can pass an optional integer argument to the program when calling it through the command line which will act as the seed for the sequence of random number generations during games.

Final Questions

Q1: 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?

One of the most important things we learned about developing software in teams during this project is the importance of planning out all the components and program flow in detail. Since we knew that this would be a large project, over the span of multiple weeks, with many different files, and multiple developers working on it, we wanted to make sure we planned out as much of the code as we could before actually implementing it. We started by planning out all of our classes and drawing out our UML diagram to see how classes would interact with each other. Then, we traced through an entire game of chess, seeing where different classes came in and how the flow of the program worked, revising our classes and UML diagram as we went. This ensured that every member of the team had a clear idea of the big picture, as well as the individual classes involved, and set us up for success in foreseeing any challenges and issues. As a result, we had to do minimal refactoring during our implementation process, as is evident in the comparison of our due date 2 UML and our due date 3 UML.

Another takeaway we gained is the importance of setting up a good system that would make it easy for many developers to collaborate, whether pair programming in person or working individually and remotely. After some initial experimentation, we decided that the best approach for us was to use a collaborative version control software, and to divide the program into areas and components that served very independent roles in the final product. We decided to use Git and GitHub, creating branches for model, view, and controller, and having each person generally focus on implementing features that fell under one of the three branches. This made it incredibly easy to work on independent units and features, test them, and then merge them into the main branch, with very minimal merge conflicts or code overlap.

Finally, we learned that as the product and team gets larger, communication becomes more and more essential. Looking back, many of our issues came from insufficient communication. For example, multiple team members would end up working on the same thing, resulting in duplicate code and an inefficient use of resources. There were other times during debugging when a problem had already been fixed, but due to a lack of thorough communication, not everyone in the team was notified, and we would spend extra time debugging issues that had already been fixed. In the future, we will aim to communicate more frequently, and set up communication channels that ensure all team members are updated on all changes.

Q2. What would you have done differently if you had the chance to start over?

In general, we were very happy with the way we chose to collaborate and the final product we created. However, there are a few things we would do differently.
Even from the beginning, our approach was to aim for a product with all the features fully working. As a result, we essentially waited until the end to do the majority of the testing. The biggest thing we would do differently is try to get a minimum working product as soon as possible, and test each new feature we add as we add them. Firstly, this would ensure that if something were to go wrong and we couldn't complete the entire game, we would at least have a minimum working product. Secondly, this would have made debugging easier, as once we had nearly the entire project complete, there were certain bugs that were incredibly hard to track down and fix, and it took much longer than necessary to narrow down issues.

Additionally, if we were to start over, we would integrate more collaborative coding and code reviews among team members. This project, like all other assignments in this course, is a valuable learning opportunity, and we would use it more as an opportunity to learn new techniques and good coding practices from each other.