CS 246 Final Project Design - Chess

Markus Rieg (mrieg)
William Kennedy (wkennedy)

Introduction

Our final project is chess. Over the past week, we have created a fully functional chess program for humans and computers using best practices in object-oriented programming. We have collaborated to implement full chess logic, two user interfaces, and a gameloop with scoring and input. Our effective planning and communication has allowed us to implement a number of core and extended features using well-known design patterns.

Overview

The structure of our project follows that of object-oriented design, with special attention paid to encapsulation, inheritance, abstraction, and polymorphism. The primary driver of the program is the gameloop in the game class, which allows for entry into different modes such as setup, or just starting a game. Here, the user will choose opponents, which could be human players or a number of bots, all of which are subclasses of the player class. The largest class of the project is the board, which is where the action takes place in chess, and thus all the checks for validity. The board also contains all the pieces, which are abstracted to a piece class while still allowing for individual piece movement specification. The board is observed by an abstract board observer class, which updates its subclasses, namely the saved board history game data as well as the two user interfaces. We also have a number of structs and enumerated classes in each class to more easily share cohesive information.

Design

We make use of a number of design patterns to anticipate various ways the project could change, such that new features can be added with minimal modification or recompiliation. In addition to the overview description in the above section, such resilience to change is listed in the section following.

A guiding fundamental we followed is the single responsibility principle. We have over twenty class files all at some level of abstraction, each of which contains at least a class and also sometimes related structures (e.g. position, move information), enumerated classes, or related standalone methods when they are highly cohesive. In order to keep coupling low, none of our classes are friends, although they at time may pass common structures to each other. Given our modules do not share global information, and that they pass data to each other to perform single tasks, it is reasonable to describe the coupling as low and the cohesion as high.

Our largest planning problem arose surrounding the relation of the classes of the program itself. At the core, we make use of the model-view-controller pattern given the variety of roles the different classes for a game program must play. In our case, the game class acts as the controller, the board (and related classes) as the model, and the UIs as the view. The game class runs the central game loop as one might expect, which is itself modular in the respect it calls different methods depending on the mode (e.g. setup, game). The board contains the state of the data, using an array of various pieces that have been abstracted. As expected, it sends its information to the view without knowing about its details. The view consists of an abstract UI class and its two subclasses, the text UI and graphical UI. These represent a display of the board. The computer player subclasses can also be considered mediators between the controller and model.

Another design pattern that features prominently in our project is the observer pattern, which compliments the model-view-controller pattern. We use the board as a publisher for a variety of notifications that are sent to an abstract observer, fulfilling the publisher-subscriber model. This also allowed us to have multiple different observers which, while vastly different, could interpret the same information. For instance, the move history (game data) and graphical user interface are very different classes, though they are both subscribers of the board as subclasses of the abstract observer. One of the challenges we encountered was managing the notifications for non-move events. We worked around this issue by creating different notifications, which react or do nothing in response. For instance, the move history is not updated during board setup, while the user interfaces are. Ultimately, this design pattern allowed us to cleanly integrate the move history without worrying about too much coupling.

A significant portion of our design surrounds inheritance and its results such as polymorphism. This was a core part of ensuring encapsulation. For instance, the different pieces all determine how they are able to move, while still being of type piece for ease of use on the board. Similarly, we have two user interfaces with vastly different outputs, but these can use the same fields and methods. This is also the case for the computer (bot) players.

Another design pattern we made use of was iterators, though in a less traditional sense. Parts of the standard iterator pattern were used to increment enumerated classes, such a colour, for ease of use. Similar methods were implemented for changing turns.

Lastly, we also made moves towards using standard STL modules. We acted on a preference for using vectors and smart pointers over arrays and raw pointers, though these were still used in some places. This was motivated by desire for moden C++ standards, in addition to the concision and ease of use provided by modules like algorithm and vector that let us forgo tedious management.

Outside of design patterns, there are also a few classes and structures that play important roles in the program. For instance, the game data class contains a vector of move info structures for the history, while the board has an invisible board info structure. Other programming features we used include input for the moves and plenty of conditional checks for the board.

Resilience to Change

Our design also supports the possibility of various changes to the program specification. Our use of inheritance allows a lot of flexibility when it comes to implementing new features that are likely to be related to chess and its variants.

Firstly, our Piece abstraction allows for the easy creation of new pieces for any or all players. Since each piece keeps track of how it moves and captures, no modifications would need to be done to other files, except if a new special move is created similar to en passant or castling.

Secondly, the core gameloop is not dependent on the alternation of specific player turns or colours, but rather iterating through the default given ones. Changes could be easily made to accommodate a third player or rearranged colours, though this might reflect strangely without changes to the board size or game logic.

Another abstraction we implemented is that of the board observer. This abstraction allows new classes to be implemented that may receive updates from the board and respond with output or other actions. This is used to keep track of our move history, but also both of the user interfaces. The user interfaces themselves are also abstracted to allow for more display methods, whatever that may be.

Furthermore, we also allow for new different kinds of computer players who may have different behaviour from the existing ones; this could be done by adding a new subclass to bot. Lastly, we have the possibility for a change in the board size.

Answers to Questions

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.*

A book of standard openings could be implemented in different ways, depending on how the game is being displayed. In a text-based UI, a list of the most common opening moves in algebraic notation can be provided by typing in the command line, say "openingbook." Alternatively, this information can be provided on the side of the ASCII chessboard, toggled on with a setting. Such a sidebar would be easily implemented in our game as we already have an existing one for move history that could be retrofitted or abstracted upon. Similarly, a panel with a list of opening moves can also be implemented in a graphical user interface, much like the text UI.

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

The ability to undo is a feature of our game. Much like most standard chess programs, we keep a complete history of every move made. While displayed using algebraic notation, we also store the initial location, destination, piece captured, and other data so that any move can be undone without recalculating the board state from the start of the history. This information is stored in a vector, which increases in length every time a move is made, and allows for as many undos as the moves played.

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. (If it's important to your answer, state whether you're assuming free-for-all or team rules and then answer the question. You don't need to get too specific into the rule set changes in answering the question though; your focus should be more on what would need to be altered at the high level of the design)*

Changing the program to allow for chess variants such as four-handed chess would require a few modifications. For instance, the size of the board is much larger and has spaces where pieces cannot move or check through (the corners of the square). The order of play would also need to be modified to account for the additional 2 players that would be added, and the colours of the new pieces. Moreover, more must be done when it comes to tracking move history and valid moves. However, some aspects of the variant are easier to manage, such as calculating king-capture rather than checkmate. Ultimately, one of the more complex parts of the program would be tracking which variant of chess is being played and what rules to apply.
These are all considerations that we have made when designing our classes and code to be abstractable. For instance, our gameplay is not hardcoded based on the alternation of two players or colours, but rather iterating through the turns of players. However, we have not implemented any of our game's features for variants with more than two players.

Extra Credit Features

We have implemented a few features that are not specified in the doc.
Firstly, we display the latest few moves played in algebraic notation, much like most modern chess programs available online. This was implemented using the publisher-subscriber model to get notification from the board class. While we had some difficulty implementing this class in relation to others, in the end it was not too challenging.
Our main extra feature is the ability to undo an infinite amount of times. This was achieved by saving information about every move made including initial/final position, a captured piece, and flags about the board. A particular challenge we encountered was putting the piece from the saved moves back on the board, as we previously assumed it would be constant. As we did not want to increase coupling, we ended up settling on clone methods for each piece subclass.
Another feature we made use of is that of smart pointers. While we still made use of some raw pointers, we worked towards using the STL containers for memory management to avoid

memory leaks. We also tried to make use of more STL imports such as vector and algorithm, in order to make the code clearer and more concise while also better reflecting industry.

Final Questions

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

This project has taught us a lot about developing software in teams. Having developed an initial planning document, a UML diagram, and effective communication strategies before the project started allowed contentions issues to be resolved early on. Moreover, talking our ideas out to each other by using a whiteboard and virtual UML diagrams helped ensure that we were on the same page, or come to a compromise if we were not. This project also reinforced the necessity for planning and communication in our development, allowing us to delegate responsibilities and work on separate files using git for version control. Lastly, this project has also taught us the usefulness of pairs programming. Having another person to read over the code or just be a rubber duck greatly increases the efficiency of both generating ideas and debugging. It has also allowed us to become better readers of code, considering our different coding styles.

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

If we had the chance to start over, there are a number of changes we may do differently. Firstly, we would spend a lot more time planning out the design. Having an initial correct plan rather than changing it as we code could increase the speed at which we work. Another thing that we could do better is documenting the code for each other as we write it, rather than at the end. There is also some room for improvement when it comes to time management. It would have been nice to have more time to implement all the extra features we were considering, including more possibilities for chess variants. Other features that would be nice to have given more time would be improved computer player moves using an evaluation algorithm and minmaxing, as well as better consideration for special moves such as castling, which are currently implemented explicitly as edge cases. Moreover, if we had the chance to start over, abstracting our classes for larger number of players, gamerules, and other expansion possibilities would be nice. Additionally, while our current classes are quite modular, it would be nice for them to communicate more effectively between each other, even if it increases coupling.

Conclusion

In a week's time, we have created a fully functional program for chess using object-oriented design, complete game logic, resilient design patterns, and two user interface displays. While occasionally experiencing challenges with communication, delegation, and time management, we implemented good design practices tot he best of our ability. Should we do this again in the future, we would like to use more abstractions that allow for chess variants, make fuller use of smart pointers, and put a larger amount of time into planning at the beginning.