Design Document

Introduction

This term, numerous object oriented and C++ concepts including classes, smart pointers, and inheritance,were used to develop the card-playing game, Hydra with various features. The objective of this project is to demonstrate capability in these skills. Moreover, this project has been an excellent learning opportunity, giving practice to apply those skills.

Overview

The following classes were used to implement the game:

Play_Game:

This is perhaps the most important class. It is entirely responsible for the operation of the game. It keeps track of which player's turn it is, updates the heads, and the player cards as necessary. It is also responsible for reading in from input for the necessary moves by the player. If the game is in testing mode, it ensures that the user gets to choose the state of the board -- that is, the number of cards each player gets, and each player decides which card they wish to play. Once the game has finished, it outputs a prompt stating the winning player, and then exits.

Board:

The board class is the backbone of all the other classes, "pulling them together". It stores all the information of the game state: each player, the heads, and it is also responsible for using TextDisplay to print out the necessary output. It determines if the card being played by any player is valid, and the attempted head that it is being placed on actually exists. A map was used to store the heads (see Board.h). Although the original plan was to have a head class and store them in a vector, it quickly became difficult, since if a head was no longer in play the vector would require indexing. A much simpler solution was to utilize a dictionary, where each integer which is the head index is mapped to the head itself.

Player:

The Player class keeps track of each player and the cards that they have. It keeps the draw pile and the discard piles separate in a vector, as well as the current and reserve card in strings. Additionally, the corresponding methods in this class allow for modification of their deck as required, thus, it makes it easier for Play_Game to use this class to make any modification for each player. The number of functions required to implement this class was clearly underestimated. In order to ensure encapsulation, the classes other than Player did not have access to the vector of cards: so, every movement of a card required a method. These include getting the top card, switching the card, getting the card from reverse, changing any specific card, getting the size of their draw and discard piles, etc.

Base_print:
This class is the print class that Board uses to call all the methods from TextDisplay. TextDisplay inherits from Base_print. So, any changes to the printing function are quickly recoverable with minimal modifications.

TextDisplay:
The TextDisplay class is responsible for outputting the correct output in the correct format to the console. Depending on the action, Play_Game calls each separate function to output as desired. In detail, it will print the players, the heads in play, if a player is in winning state then the winning play, and the corresponding actions made by the player that requires input. This class is inherited publicly from Base_print which is associated with the board class (see Base_print.h).

Design
Many techniques were used to make the game in this project. Firstly, to ensure encapsulation between player and it's associated classes, it quickly became relevant to implement new functions in the movement of the cards. After producing the functions in the original UML plan, it became apparent that there are a lot more movements of the cards that are possible. Another technique used was to use map storage in the stl library. The original idea was to have a head class, and the board class would store a vector of those. However, this would introduce the fact that explicit memory management must be used, and the indexing of the heads would cause complications. So, a map was used as a dictionary, to assign the number of the head to each playing hand. This reduced the complication of having to erase each head in the destructor. In general, the Play_Game, Base_print, and the Player classes are associated with Board. Play_Game accesses information from Board to ensure operation of the game, which inherits Base_Print to make sure that the proper functions from TextDisplay are used for the correct output.

Resilience to Change
The design supports the possibility of various changes to the program as each class is closely related (high cohesion). So, any new features implemented would follow the theme of this project. Moreover, each of the classes have low dependence on each other (low coupling). An introduction of a new function or feature would require minimal modifications to the code. Additionally, the inheritance of TextDisplay from Base_print makes it easier to create and maintain new features. Instead of writing new data and functions for printing, TextDisplay can simply inherit from the base class. A similar case follows for Play_Game and the Board classes; Play_Game inherits from Board class. Lastly, the proper documentation of each function in each class makes it easier to implement new code, as one doesn't have to go through each implementation of a function to determine what it does. The usage of smart pointers and no explicit usage of memory management in this project strongly saves it from memory leaks. Thus

due to this design, a change in rules, input syntax, or a new feature ensures minimal reconditioning.

Answers to Questions

Question: What sort of class design or design pattern should you use to structure your game classes so that changing the interface or changing the game rules would have as little impact on the code as possible? Explain how your classes fit this framework.
We could use an observer pattern. In this case, the Board that keeps track of all the cards and implements all the rules would be the concrete subject, and all of its dependencies (observers) would be the player, heads, play_game, and text display. Since these are all classes that are used and dependent upon the board class, they would be concrete observers. As such, any rules or interface change that is implemented has little impact and the program works just as intended.

Question: Jokers have a different behaviour from any other card. How should you structure your card type to support this without special-casing jokers everywhere they are used?
If a player gets a joker in their hand, then we store it as a "joker" string. Since they are not required to decide the value of that card until it is played, we keep it as that string. However, when this card is played, the user must decide on the value. As such, we do not store it as a joker, but as a string of whatever the value chosen is, "A,2,3,4…,J,Q,K". Additionally, to mark this card as joker, we make the second character a "J". So, a joker with value 5 would be "5CJ". With C being the club.

Question: If you want to allow computer players, in addition to human players, how might you structure your classes? Consider that different types of computer players might also have differing play strategies, and that strategies might change as the game progresses i.e. dynamically during the play of the game. How would that affect your structure?

Adding varying computer players with varying play strategies would probably require defining them explicitly before the game has begun. I would make a separate computer class that incorporates different computer players with differing strategies. It would not change much of the structure in our already existing classes except for a few lines of code that calls on the computer class to make decisions instead of requiring input from the user. Depending on the game stage and the current condition of the heads, we would call the corresponding methods.

Question: If a human player wanted to stop playing, but the other players wished to continue, it would be reasonable to replace them with a computer player. How might you structure your classes to allow an easy transfer of the information associated with the human player to the computer player?

We could add a field to the player class that determines if a human or a computer is playing that player. Then, when the play game class calls on a player for a turn, instead of reading from input, we would utilize various methods and play strategies to make the correct play. Ideally, we would use the computer class and call it within the player class instead of requiring input from the console. In this respect, we do not have to transfer any of the information -- but rather, the computer class utilizes the player class to make the correct play.

Extra Credit Features:
A grammar feature was added so that when a player is holding an Ace or an eight, the input says "Player X you are holding an AJ". This wasn't very challenging as it only required a simple if statement to check if the card requires extra grammar. A feature in the -testing mode is also added: one can choose how many cards each player has at the start. This was again, fairly easy to implement, instead of having the vector of cards be 54*num_players, it would simply be num_cards_per_player*num*players.

Last Questions:
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?
I learned the importance of using time management. When working in a group, if a problem arises, it is easy to ask your team-mates. Getting a different perspective on the problem would give you insight on how to solve it. However, when working alone, this becomes very difficult. One can be stuck on a problem for hours if they don't know how to go about solving it. However, taking time away from looking at the code to do something else really assisted in refreshing the mind. Taking these breaks often ensured that the work was done on a consistent pace.

What would you have done differently if you had the chance to start over?
I would definitely have liked to spend a significant time in Part 1 of this project (i.e., the planning). Firstly, I had a very vague schedule for what I wanted to get done each day. I think a more rigorously outlined schedule, highlighting which functions and which features to implement each day would have been much more beneficial to time management. Secondly, I would have liked to spend more time planning the design and thinking more thoroughly about it. When implementing the classes, I quickly realized the new things that must be added to ensure functionality of the program.