Sanat Bhalla
20908162
CS246 Final Project (Hydra)
Final Design Document
University of Waterloo

INTRODUCTION

The game of Hydra is developed using the principle of Object-Oriented Programming in C++. It uses the MVC (Model View Controller) Design Pattern. My model represents one with High Cohesion, Low Coupling and promotes Encapsulation.

The game of Hydra consists of more than one player (which the game takes as an input) and X number of 54 card decks (52 normal card and 2 Jokers) where X is the number of players. Each player has a draw pile, discard pile, reserve card and a play card. At the start, the 54 times X cards are shuffled together and is distributed equally among the X players. Each player would now have exactly 54 cards in their draw pile before the game begins.
To begin the game, the first player removes the topmost card from their draw pile and sets it as the first head. Each player then plays their turn in the order following the rules of the game.
The first player to have an empty draw pile (no card present), an empty discard pile and an empty reserve card wins the game.

Overview

To accommodate the MVC design pattern of the game, it contains the following classes –

- Game & Player class (Model)
- Print class (View)
- GameController (Controller)

The main function creates an object of GameController class with number of players and a Boolean value grammar (bonus). This GameController object calls the play method by passing another Boolean argument for testing. The play method under GameController is responsible for calling methods from the Game and the Print class to introduce logic to the game.

Player

The Player class contains information of each player in the game. It consists of 2 vectors of strings – draw pile & discard pile (where each string is a card consisting of a card value and suite value), a string representing the reserve card of the player (initially set to “”) and another string representing the play card of the player (topmost card of the draw pile). The player also contains a Boolean whether the player is a computer or not.

The game logic called by the play method in GameController class keeps on updating the fields of each player. The wincCheck method in the Player class returns a Boolean whether the player has won the game or not and proceeds accordingly.

Game

The Game class keeps track of the heads (game board) and all the players. The heads are stored as a STL (Standard Template Library) Map, where the first element is an int (ID) and the second is a vector of strings (card pile/head pile). The map was a useful tool to identify the head that had to be cut off with the int ID and also kept track of the head size.
The players are stored in the Game class as shared_ptrs. This made memory management a lot easier as I did not have to free any allocated memory manually.
This class triggered various methods on each player object stored in the memory according to the logic called by the GameCrontroller class.

GameController

The GameController is the most important class. It stores pointers to a Game and a Print objects. This class consists of the play method that triggers the methods of the Game and Print classes according to the rules of the game. It also consists of a testing method that sets the card and suit value manually from the input. This class is responsible for accommodating all the rules of the game and thus any addition/deletion of a rule would just affect this class.

Print

This class is responsible for all the output statements of the program. It consists of 3 major functions –

- Print heads – prints the status of the heads (ie no. of heads, topmost card on that pile and the number of cards in that head pile)
- Print Player Information before anyone's turn – prints out the size of the draw pile and the discard pile of each player
- Print Player Information during the turn – prints out the size of the draw pile and discard pile of each player as well as well as the number of cards in hand, number of cards remaining to be played and the number of reserve cards of the player whose turn it is

The GameController class triggers these functions on the pointer to the player class it holds before every move and also when the turn shifts from one player to the other.
All output statements are stored here so any modifications in them would just affect this class.

Design

My Design Pattern changed a lot from DD1. Initially, I had planned on implementing a strategy design pattern on the Card Type (Normal and Joker), but as I started implementing it, I realized that there was no significant difference in the behaviour of both the card types. Also keeping the card as a class would just be extra work since the only fields it would have is the card value and the suit value. This would have been a sign of low cohesion which is not a good thing. Moreover, I would have had to make my main function as the game controller but this would have violated the principles of encapsulation.

In order to accommodate encapsulation, improve cohesion and reduce coupling, I decided to implement the MVC (Model View Controller) design pattern. Here, the Game and the Player class would be a part of the Model, Print class as the View and the GameController class as the Controller. This increased the efficiency of coding allowing one class to just undertake a single responsibility.

Resilience to Change

My design followed the principle of Object-Oriented Programming. Implementing the MVC lowered the coupling between 2 or more classes thus allowing change in one part of the program not affect the other. I had different classes to perform different functions, thus reducing the interdependency between them.
The Game class keeps a track of the heads and stored all players as shared pointers. Any change in the game was called by the GameController to the Game class, and it updated the heads/players. Any change in setting up the cards, change in the minimum number of players required or the heads can directly take place heare without affecting the other classes.
The Player class holds the information of each player. It is not triggered directly by the GameController, but the Game class calls methods of the Player class to update it. Any change in the draw pile, discard pile, reserve or play card can directly take place here without affecting other classes. The GameController to not directly have access to the Player class is a good sign of low coupling, encapsulation and information hiding.
The GameController class stores pointers to objects of Game and Print class. Since the GameController is just responsible to implement the rules and logic of the game, the addition/deletion of any rule could just be accommodated by changing the play function in this class and would not affect any other class.
The Print class is just responsible for the output statements. Any change in the structure or design of the output statements will be handled by this class and any change made here will not affect the rest of the classes.

Questions

1) *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.*

DD1 –

I would be implementing a Strategy Design Pattern to make the code more elegant, reduce the coupling and increase the cohesion. Changing the rules of the game would just affect the Game class which can be fixed easily. This would not have any impact on classes such as Player or CardType. Again, changing the interface of the game would just affect either the Player or the Game class, leaving the CardType unchanged. This can be accommodated in my design pattern due to the minimal interdependency of each class on one another (low coupling).

DD2 –

My design pattern changed a lot since DD1. I decided to implement an MVC (Model-View Controller) rather than a Strategy Design Pattern. This is a more efficient design pattern as having the Card class was just a sign of low cohesion since it would contain just 2 fields. Any change in the interface/rules would affect the Game or GameController class depending on the change.

My classes fit this design pattern perfectly as explained earlier. The Game and the Player class are a part of the Model, The Print class is a part of the View and the GameController is the Controller.

2) *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?*

DD1 –

Since I have a class CardType which extends a Joker class, I would be saving all my cards in every pile as a CardType instead of a string. This would allow me to make any changes to the CardType (eg. removing instances of Jokers) without actually special-casing Jokers everywhere. All I need to do is to modify the CardType class.

DD2 –

I did not use the CardType class in order to increase cohesion. I special-cased Jokers everywhere since it was an easier choice and it fit my current design pattern. The Joker is instated in the Game class and just has conditions in the GameController class as well. So, if we ever decide to remove the Joker card from our game logic, we would just have to modify the Game class.

3) *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?*

DD1 –

To allow computers to play along with human players, I inherited Computer class from Players along with Human class. Human class is responsible for all the objects and functionality. Computer class can extend various subclasses to define different strategies to make the move and win the game. So whenever the Computer object is called, we can select the complexity of the strategy based on the round of the game (eg. earlier rounds have easier strategies, but as the number of heads increased, the strategies applied would be more complex).

DD2 –

In order to accommodate Computer players along with Human players, I would change the logic of DD1 a bit. I would have a Boolean field in the Player class isComputer that would be true if the player was a computer and false if the player was a human. I would set this while setting the game class.

Essentially, the human and computer players have the same fields, methods and attributes but different behaviour so separating them on the basis of their attributes wouldn't make sense rather I could separate them in the GameController class by implementing the computer logic and triggering it if the Player was a computer.

4) *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?*

DD1 –

If we further had to enhance the game where a human left the game and the computer took over, we would just implement the Computer class as a subclass of the Player class. In order to transfer the information from a Human object to a Computer, we can implement a move operator (constructor and assignment). This would allow a person to leave the game, and would transfer all of its information to the computer object.

DD2 –

If a human player wanted to leave the game, we could directly change the isComputer value of that player from false to true and it would follow the strategy of the computer from the next move. This

would be very efficient and less confusing as compared to calling the move operator as mentioned in DD1.

Extra Credit Features

1) Grammar –

   An extra credit feature in the Print class that prints “an” instead of “a” before 8 and Ace cards. The program behaves normally unless the user inputs “ --grammar ” as a command-line argument. This is just a modification to the Print class and due to our MVC Design Pattern, this change would not affect any other class due to low coupling.

```
cout << "Player " << turn << " you are holding a";

if (this->grammar && (playCard[0] == 'A' || playCard[0] == '8')) {
    cout << "n";
}
cout << " " << playCard << ". Your move?" << endl;
```

2) Pure House –

   A Pure House is a condition where the same card (same value and suit) is played on the other. For example, “5C” is played on “5C” or “8H” on “8H”. In a normal scenario, the player would end their turn if a same card is played on the other, returning the remaining cards to the top of the draw pile. In this scenario, the player plays all the remaining cards on the current head irrespective of their value. For example if a player had 6 remaining cards and the heads was “3H” and the player’s card in hand was also a “3H”, in the normal scenario, the player would put the 3H on the 3H and would return the 6 remaining cards to its draw pile and end his turn, but in this scenario, the player would play the 3H on the 3H plus all the 6 remaining cards on top of it until the remaining cards go out or the player wins. This feature (new rule) is added to the GameController without disturbing any other class as wished by the Design Pattern.

   This rule can be triggered using “ --pure ” as the command line argument.

```
if (headCardInt == playCardInt) {
    if (headCard.back() == playCard.back() && this->game->getPure()) {
        cout << "== Pure House ==" << endl;
        while (headSize > 0) {
            // win check
            if (this->game->playerWinCheck(turn)) {
                this->print->playerWin(turn);
                return;
            }
            this->game->attachToPreviousHead(move, this->game->removeTopCard(turn));
            --headSize;
        }
    } else {
        this->game->attachToPreviousHead(move, playCard);
    }
    break;
} else if (playCardInt < headCardInt || headCardInt == 1 || playCardInt == 1) {
```

3) Computer Move –

In this extra credit, as asked in one of the questions, the player has an option to leave the game and the computer takes over for the player. The computer implements a basic strategy when the heads cannot be cut, the computer finds the largest head possible and puts the card on it. If the heads can be cut, the computer cuts the oldest head.

Use the " –comp " command line argument to activate this feature.

```
int Game::compMove(string playCard) {
    int largestHead = this->cardToInt(this->heads.begin()->second.back());
    int retVal = this->heads.begin()->first;
    int playCardInt = this->cardToInt(playCard);
    int equalHeadRetVal = 0;
    // if it has to cut, then it will cut off the oldest head (ie 1st in the map)
    if (!this->hasToCut(playCardInt)) {
        // if not cut then find the largest head possible
        for (auto it = this->heads.begin(); it != this->heads.end(); ++it) {
            int tempHead = this->cardToInt(it->second.back());
            if (tempHead == playCardInt || tempHead == 1) {
                equalHeadRetVal = it->first;
            } else if (tempHead > largestHead) {
                largestHead = tempHead;
                retVal = it->first;
            }
        }
    }

    if (retVal == this->heads.begin()->first && equalHeadRetVal != 0) {
        retVal = equalHeadRetVal;
    }
    return retVal;
}
```

All these extra credit features can go together in any permutation and combination.

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?*

I learnt how to write large programmes such as Hydra. I understood how the planning/strategy, design and unit tests should be configured prior to start developing the program. One of the most important things I learnt was the importance of having a good plan and following it throughout with minimal changes to be made as we proceed on developing. This would reduce the confusion later on, increase the efficiency and would give you ample time for developing. I also learnt the practice of following a particular design pattern while developing since it would lead to well organized code. Since we had used the observer pattern in A4q3, whose implementation was similar to the MVC, it gave me a good reference to build on. Also as stated in DD1 Plan of Action, I followed my technique of testing while developing. I wrote small unit tests for each rule and tested it as and so I wrote the code for that particular

rule. This made my program bug free and I did not face any issues compiling it towards the end. Debugging was also a lot easier this way.

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

If I had an option of starting over, I would implement the heads in the Game class in a different way. Instead of using STL map and storing everything on the stack, I would implement it as a linked list and store the cards as a vector of strings on the heap. This way I would also be able to implement the Iterator Design Pattern in addition with the MVC. Storing elements on the heap is a better practice for large programmes such as this and I would have been able to follow it.