CS 246 Final Design

Edmond Zhang

Introduction

Straights is a game played with 4 players where players take turns discarding cards or adding cards to 4 different piles separated by suit. As the game progresses, players accumulate points and a victor is determined by the player with the lowest score once a player hits 80 points. This timeless game was recreated using C++ and enables a player to play with friends or against computers.

Overview (describe the overall structure of your project)

Straight could be broken down into several key objects. Firstly, the card object emulates a playing card. The card is a component of other objects. The deck is made up of a default 52 standard playing cards and has methods such as shuffle() that emulate shuffling in real life but also includes methods such as saveState() which allows pseudo-random shuffling after each round. The player object represents the players of the game. Since there are different types of players, abstraction was used to increase cohesion. This will be further elaborated in the design section. A player object has a vector of cards called “hand” that represents the hand of cards a player has. Similarly, there are vectors for the discard pile and legal plays available. The pile object represents the piles organized by suit that players add to. They hold a vector of cards to represent the current pile and 2 integers to give the current valid cards that can be added to the pile.

Design

MVC

I used a version of the Model-View-Controller architectural design. The model and view of the game is handled by the class: game. This class facilitates the changes in the game board including decks, piles, and player hands. The player class facilitates how a user can interact

with the model. If one chooses their player to be a human, they are only limited to a certain number of legal plays. If one enters an invalid play or invalid syntax, the system will continue running until a valid play is made. This improves robustness.

Abstraction

Abstraction was used to organize classes and make changes to certain functionalities depending on which object is of concern. For example, the 2 types of players: humans and computers have different ways of playing cards. Thus, a base class of player was declared with virtual methods. This enabled humans and computers to implement their own variation of playing a card.

Encapsulation

Throughout the implementation of Straights, there are many details/invariants that we want to keep hidden from the user so that they don't make any unwanted changes. An example of this would be the rank of standard playing cards or the number of playing cards in a deck. Thus, encapsulation was used to enforce this. Data fields and methods that were not intended to be used/changed by the user were kept hidden through the use of private/protected declarations and summoned through accessor methods.

Resilience to Change

Single Responsibility Principle

This implementation of straights follows a modal-view-controller architecture. Due to this, each class has its own purpose, leading to high cohesion. Each class is supposed to replicate the characteristics of its own real-life counterpart and nothing else (with maybe some useful fields that help with sorting and identifying such objects). Since every class has its own functionality, this makes it easy to change certain parts of the implementation without having to change existing code.

Abstraction

As stated before, many of the key classes of the game can be further abstracted to accommodate any changes. The game class can be abstracted to accommodate changes to game rules, the player class can be abstracted to create new types of players, and other classes such as deck, pile, and card can be abstracted to accommodate more elements/fields.

Answers to Questions

Changing game rules

All rules of the game Straights, from dealing, to playing, to calculating score are all implemented using class methods. The method playGame() in the game class then handles when each method is called and in what order. Thus, if one were to make any changes to how the game is played, they would take advantage of inheritance. This means they would simply create a subclass of class straights (which contains all the rule logic) and implement the rules they desire. This makes it easy to swap out which ruleset you want to use because all you have to do is call playGame() on your new ruleset(which is a subclass of game). This makes it so that no previous code implementations need to be altered to add new rules as they will simply just exist and can be re-implemented if desired. To add different views, a user should use the observer pattern. This is because we want to support multiple different views (graphical and text) and the observer pattern allows us to extract key changes from the model in order to change the view to the user.

Players

Since all players were implemented using a base class, if one were to add a different type of player, maybe a more difficult computer, all they would have to do is create a subclass of player (or maybe even a subclass of computer), and implement the algorithm they desire for this new computer. If we wanted a computer that changes strategies dynamically as the game

progresses, we could set up multiple play() methods that offer different strategies. Next we can create the if/else conditions that need to be satisfied in order to use each different play() method (For example, if there are more than x number of cards in computers hand use play1() which is good for trying to mitigate discarding score, else use play2() which is good for saving up important cards for later).

Cards

Since every standard playing card was implemented using a base class card with the standard fields of rank, suit, value, name, etc. it would be simple to add different cards to the deck. If one were to add a wildcard such as the Joker, they would simply have to add a method that takes input so that a user can choose what rank/suit they want their wildcard to be and calls all the setter methods of card to create a card that can become any card.

Final Questions

What lessons did you learn about writing large programs?

I learned that writing large programs requires a considerable amount of planning effort. It is not only necessary to lay out which classes are connected to which but to describe in detail how they interact with each other. If one lists a description of each class method, how it works, and how the arguments are used, and any techniques/design patterns used, it makes the actual implementation easier. Since I was working alone, I had to keep track of what part of the code did so when I would step away from the project for extended amounts of time I usually came back confused at what I was looking at. The more detail I put in the design plans, the easier it was to get back into the zone of what I was previously doing. Additionally, another lesson I learned was to expect plans to change. My initial UML and plan of attack are drastically different from the final code. This is because when I first started planning, I did not take into account which methods could be simplified or whether a method even needs to exist. As I started

actually coding, I started to realize these and even found more ways to utilize techniques such as encapsulation to refactor the code.

What would you have done differently if you could start over?

If I could start over, I would have looked over implementations of different card games to get an idea of how to model objects such as cards, decks, game boards, discard piles, etc. and how to replicate real-life actions such as dealing and shuffling. This was my first time programming a card game so I had trouble wrapping my mind around how to simulate such objects and it caused me to neglect including certain data fields that would have simplified methods that used such objects. For example, I initially only kept track of a card's rank and suit because I thought those were the only characteristics of a card. After further consideration, my final design included data fields such as name (which simplified reading inputs and identifying cards), and value (which simplified determining whether or not a card was a valid play for a certain pile).