1 Introduction

The three key features that I had in mind when designing my program were simplicity, implicit memory management, and being flexible to changing plays.

1.1 Simplicity

I wanted to keep my program as simple as possible, thus, unnecessary design patterns due to their complexity. For example, using design patterns such as the Observer or MVC ease interface changes, but increase the complexity. For a relatively simple game like Straights with only a text-based UI, these design patterns are unneeded.

1.2 Implicit Memory Management

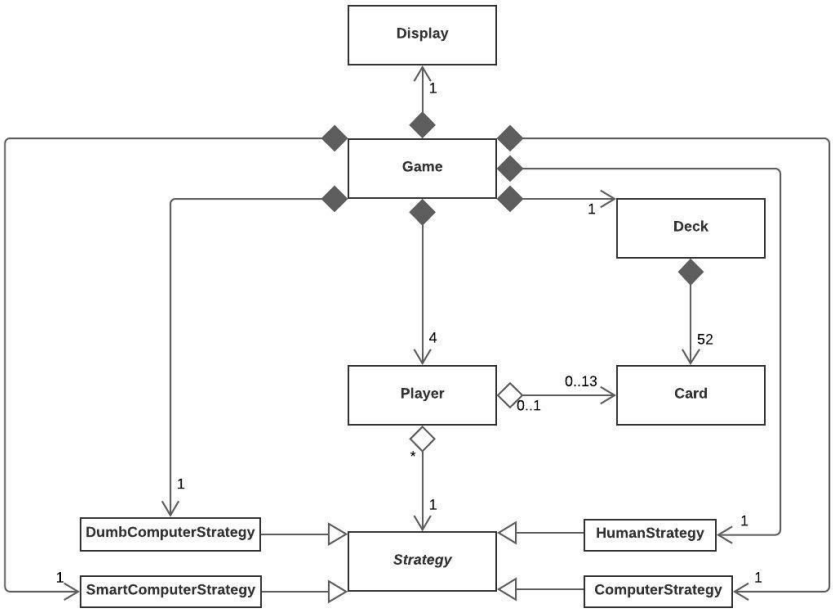
I decided to design my program so that there is no explicit memory management. This was done through the use of composition and aggregation.

1.3 Flexible to Changing Players

One of the key challenges of this project was the human player's ragequit command, as it requires either a transfer of information from a human player to a computer player or the ability to change how a player plays dynamically. For additional features, I wanted to implement other computer players (i.e a dumb computer player) and allow human players to decide which computer player to replace them when they ragequit, a system that allows a player's strategy to be changed dynamically seemed more applicable.

2 Overview

2.1 Summary UML



2.2 Classes

2.2.1 Card

The card class contains the suite and rank of a card which are represented by integers (i.e. KD would be represented by of card with a rank of 13 and a suite of 2). To construct a card the rank and suite of the card is passed to the constructor. The rank and suite of the cards are private but can be accessed and modified with the class' accessors and modifiers.

2.2.2 Deck

A deck is composed of 52 cards, this is implemented as a vector of cards (thus these cards' lifespan is bound to the lifespan of the deck they belong to). To construct a deck a seed, which is used to construct the default_random_generator, is passed to the constructor. The constructor also constructs 52 cards from AC (suite of 1 rank of 1) to KS(suite of 4 rank of 13). The deck has a shuffle method which shuffles the cards using the constructed default_random_generator.

2.2.3 Strategy

Strategy is an abstract class. It has one pure virtual method, Card *makeMove (int name, vector<Card *> &hand, vector<Card *> &discarded, vector<Card *> &legalMoves, Card *&discardedCard). Name is the name of the player which called the makeMove method, hand is the cards in the player's hand, discarded is the cards that have been discarded by the player, legalMoves is the cards that are legal to be played, discardedCard is a Card pointer that will be used to store the card that the player chooses to discard (if the player chooses to discard).

When a card is played, it is removed from hand and the pointer to that card is returned. When a card is discarded, it is removed from the hand, added to discarded, discardedCard is set to the point to that card and a nullptr is returned.

2.2.3.1 HumanStrategy

HumanStrategy is a concrete class of the Strategy abstract class. The makeMove method is responsible for getting and processing the players' commands (play <card>, discard <card>, quit, deck, ragequit). Play<card> and discard <card> works as described in the Strategy section. When the quit command is given, the program exits with 0. When the deck command is given, the method throws a DeckException that is handled in the Game class. Similarly, for the ragequit command, the method throws a RagequitException that is handled in the Game class.

2.2.3.2 ComputerStrategy

ComputerStrategy is a concrete class of the Strategy abstract class. The makeMove method functions as described above. As instructed, if there are legal plays it will play the first legal card, if not it will discard the first card in hand.

2.2.3.3 SmartComputerStrategy

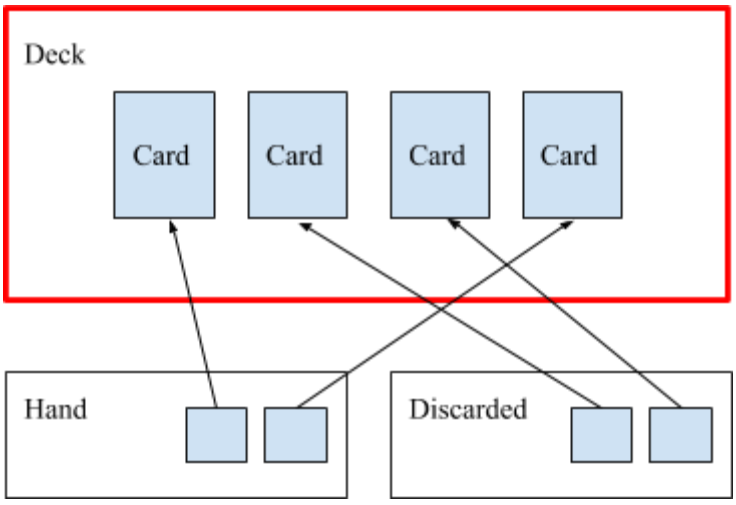
SmartComputerStrategy is a concrete class of the Strategy abstract class. The makeMove method functions as described above. If there are legal plays it will play cards with the rank of 7 last and play other cards from the greatest rank to smallest rank. If there are no legal plays, it will discard the card of the smallest rank.

2.2.3.4 DumbComputerStrategy

DumbComputerStrategy is a concrete class of the Strategy abstract class. The makeMove method functions as described above. If there are legal plays it will play cards with the smallest rank to greatest rank. If there are no legal plays, it will discard the card of the highest rank.

2.2.4 Player

A Player has a Strategy that dictates how they play/behave. It also has the following fields: hand, discarded and scores, strat, and name. Hand and discarded are vectors of card pointers, these pointers point to cards in the deck (see diagram below). Scores stores their score at the end of each round. Strat is a pointer to a Strategy that is owned by Game. When constructing a Player we need to pass a name and the location of a Strategy. There is no need to make different player classes as the type of a Player is dictated by the type of Strategy they have. All fields are private. Accessors (i.e. getName, getHand, ect) and mutators (i.e. addCard, setStrategy, ect) are provided. The makeMove method is called when it is the Player's turn to play. It calls the makeMove method on their Strategy (wraps the makeMove method in Strategy).



2.2.5 Game

Game is composed of a Deck, Display, a vector of Player, and one of each concrete Strategy class. It also has some Booleans which sets the mode of the game. Lastly, it has a vector of sets of integers which represents the state of the table (a set due to its ordering property) and a seed. Game is responsible for the game flow (i.e. inviting the player, shuffling the deck, calling each player to make their move when it's their turn, handling exceptions thrown by a player, etc) and ensuring the game rule. Its only public method is play() which first sets the game mode if the -enableBonus flag is used, invites all players, and begins new rounds until the game has ended. It is not responsible for printing the state of the game, the Display class is responsible for that.

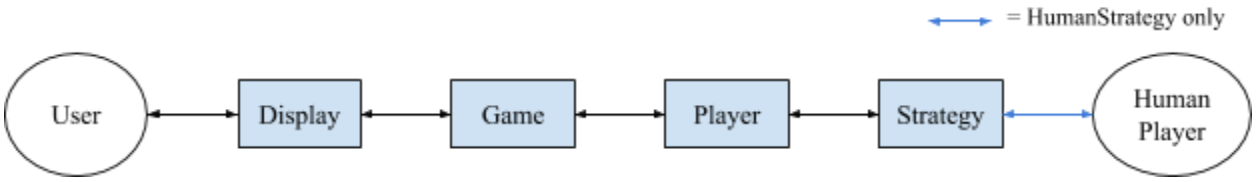
2.2.6 Display

Display is responsible for printing the outputs (i.e. printing the state of the table, the scores, player's hand, player's legal plays, printing the deck when a player asks, ect) and receiving inputs from the user (i.e.

inputs received when inviting players). However, note that this excludes the inputs from a “human” Player, it is handled by HumanStrategy as it is a unique behaviour to “human” Players.

2.3 Data Flow

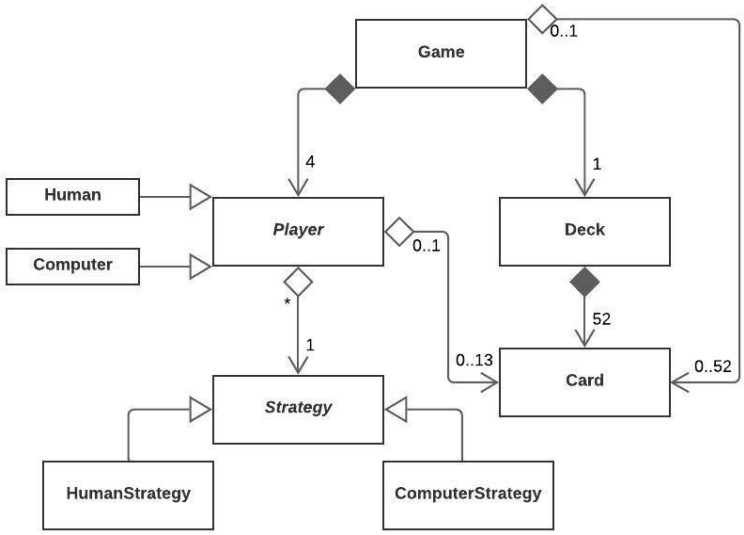
The below diagram shows a rough sketch of the data flow of the program.



3 Design

In the section, I will discuss how I came to my final iteration of my design.

3.1 First Iteration



3.1.1 Distinctive Features

The following features are not kept in the final iteration.

1. *The aggregation relationship between the Game and Card class.*
   This relationship serves the purpose of keeping track of which cards have been played. This is needed to display the table. The Game class has a vector<Card *> for each suite. There are several cons to this implementation. One, it stored a repetition of information. Each Card knows its suite but this information can also be deduced by looking at which vector it was stored in (i.e. if a card was stored in the diamond vector then its suite is diamond). Two, it made it difficult to print the cards in the order of increasing rank (which is how the table is printed). I would have to guarantee that each time I add a card into the vector, the vector would still be ordered by rank (this would have required extra functions since vectors unlike sets are not ordered).

2. *Abstract player class with concrete subclasses Human and Computer.*
   When first designing the project, it seemed to make sense to have an abstract player class with concrete subclasses Human and Computer since there were two distinct types of players that needed to be implemented. With the use of the Strategy design pattern, this was found to be unneeded as the two players only differed in how they play, which was abstracted out.

3. *Local prints.*
   I had designed the system in such a way that each class would be responsible to print some specific portion of the output. For example, Card would be responsible for outputting its suite and rank, Deck would be responsible for printing the deck by calling the print function on each of the cards they own. At the beginning, this strategy made it easier to break down the responsibility and made the outputs of the program less overwhelming. However, this was later changed so that my system would be more resilient to a change of UI.

4. *Explicit Memory Management Needed.*
   The challenge with using the Strategy design pattern was that because of the aggregation relationship between the Player and Strategy class is who will be responsible in creating and deleting the strategies used. In this design it is unclear of how this will be handled so that no explicit memory management will be needed.

3.1.2 Kept Features

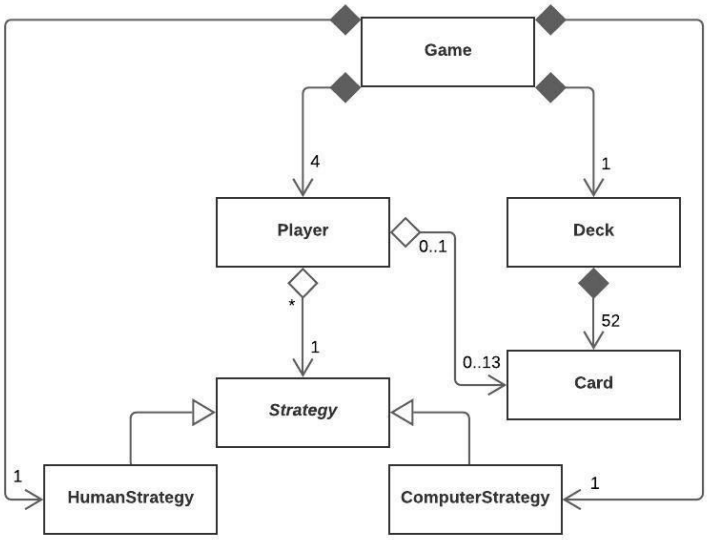
The following features were kept in final iteration.

1. *The Game is composed of four Player and a Deck.*
   This composition relationship makes logical sense as it models a real life game which is composed of four players and a deck. Modelling the system to a real life game makes the design easier to understand. In addition, if the Game constructs the Player objects and Deck object and stores them as fields, the Player objects and Deck object will have the lifetime of the Game. Once the Game is destroyed its Player objects and Deck object will be destroyed as well. This way there is no need to explicitly manage the memory.

2. *The Deck is composed of fifty-two Card*
   The reasoning is similar to that of the above point. There would be no need to explicitly manage the memory and it improves readability as it models a real life game.

3. *The aggregation relationship between Player and Card*
   The reasoning is similar to that of the above point. There would be no need to explicitly manage the memory since the Card is owned by the Deck, therefore there would be no issues storing these cards as pointers in a vector since we know that at the end the Deck is responsible for destroying the Cards (which as said above is done so without `delete`). In addition as said above, improves readability as it models a real life game (players have cards that they can play).

3.2 Second Iteration



3.1.1 Distinctive Features

The following features were not kept in the final iteration.

1. *Local prints.*
   This was the greatest weakness of this design. I had originally implemented this design as I reasoned that since for my additional feature I was going to focus on implementing additional computer players and not UI thus how I output does not matter that much. Furthermore, at the beginning overwhelmed with the complexity of the program and its output, I found it easier to grasp if I divided the outputs and just focused on implementing them one by one. Therefore each class was still responsible for different parts of the outputs as described above. However as I was finishing up my implementation I noticed that this design had made my Game class bloated with outputting the states. It was hard to seperate the logical operations from the outputting. In addition due to the system being relatively large, I would forget which class printed what and had a difficult time piecing them together. Overall the readability of my code became poor.

3.1.2 Kept Features

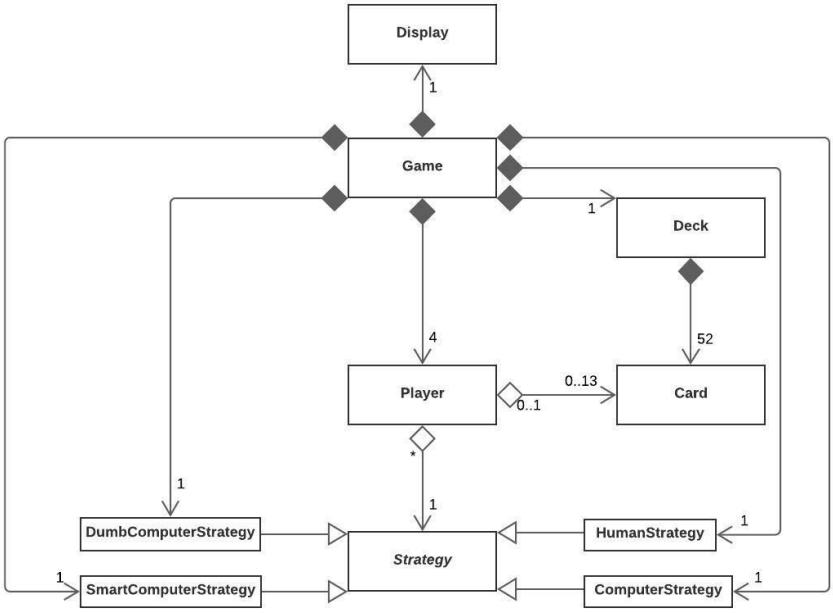
The following features were kept in final iteration.

1. *Player is a concrete class with no subclasses.*
   As mentioned above, when trying to implement the first iteration, I found that by using the Strategy design pattern and abstracting how a player plays, I do not need to have a Human class and a Computer class. A human player will just be a Player with the HumanStrategy and a computer player will just be a Player with the ComputerStrategy.

2. *The concrete Strategy classes are owned by Game*
   One of the concerns with the first iteration was that there would need explicit memory management. By creating a composition relationship between Game and the concrete Strategy classes, the lifetime of the concrete Strategy class objects are binded to the lifetime of the Game object. Thus no need for explicit memory management.

3. *Removal of the aggregation relationship between Game and Deck.*
   As mentioned above, there was a problem of keeping a vector<Card *> sorted in increasing rank and there was a repetition of information (the suite of a card can be found in two different ways). In the iteration the Game keeps track of what's on the table with an array of set<int>, called table. That is each index of the array corresponds to a suite (the set<int> stored at table[0] are cards with diamond suite, ect). I choose to use a set as it is always sorted.

3.3 Final Iteration



3.1.1 Distinctive Features

The following features were added to the final iteration.

1. *The Display class.*
   As mentioned above, the main downfall of the second iteration was the local output functions. By abstracting the outputting into the Display class, that was no longer an issue. In addition, by this point I was familiar with the game and also had a working program which I can add on top so this abstraction did not seem as daunting as it did in the beginning.

2. *Additional concrete Strategy classes.*
   The additional concrete Strategy classes, SmartComputerStrategy and DumbComputerStrategy were added for the additional feature portion of this project.

4 Resilience to Change

The following section will explore how my program supports various changes to the game specification. A sample scenario will be given in each section to clarify the situation described by the section title and a vague solution will be given.

4.1 Legal Plays

Sample scenario: The following cards are legal. A 7 of any suite starts a new pile on the table. A card with the same suite and higher rank as another card that has already been played.

All that needs to be changed is the private getLegalMoves method in the Game class. The getLegalMoves method is responsible for informing the player which cards in their hand are legal to play. This information, in the form of vector<Card *> is passed to the player which is passed to their strategy. If the way each player plays is unchanged and only what dictates a legal move was changed then we only need to change the private getLegalMoves method in the Game class.

4.2 Deck

Sample scenario: Four Jokers have been added to the deck.

Need to change the deck class. In the deck constructor, we will change it so that the new cards are added into the deck. As long as the number of cards is divisible by the number of players(four) no additional changes are needed.

4.3 Number of Players

Sample scenario: This is now a two player game.

Since the number of players is stored as a constant, `NUM_OF_PLAYERS`, in Game class and used in the Game methods. In the sample scenario, `NUM_OF_PLAYERS` need to change that from 4 to 2.

4.4 Player Behaviour

Sample scenario: The computer player instead of playing the first legal card, it plays the last legal card.

Through the use of the Strategy design pattern a player's behavior can be easily changed by changing the concrete Strategy classes. For the sample scenario, we would only need to change the makeMove method in the ComputerStrategy class.

5 Answers to Questions

See DD1 submission for a more in depth general response. My following response will be in the context of my final design. My response is short because the questions were answered above in the design portion.

Question 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.*

Section 2.3 shows a visual representation of the flow of information that may be useful. Although I have not used any design patterns for this I have abstracted outputs and input into the Display class (with the exception of the outputs for HumanStrategy). This makes changing the interface easy.

The Game class is responsible for enforcing the game flow and rules thus if the game rules are changed, the Game class will be changed.

Question 2: *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?*

I implemented the Strategy design pattern which abstracts the behaviour of a player. In addition, my Player has a setStrat method which when called will change the strategy of the player. Due to this, the strategy for a player can easily be changed dynamically. Changing a player's strategy is done in the Game class since the Game class owns all of the concrete Strategy class. Thus the logic of when a computer player changes its strategy would be implemented in the Game class. To add a new strategy, a new concrete subclass of Strategy would be created and have a composition relationship with Game. To create a player with this new strategy, just pass in the location of the strategy object when constructing the player or when setStrat is called.

Question 3: *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?*

As mentioned above, by implementing the Strategy design pattern we can change the strategy of a player dynamically. To implement the ragequit option for the human players (players who have the HumanStrategy), the Game calls setStrat onto the human player and passes the location of a ComputerStrategy object. This way, there is no need to transfer information from a human player to a computer player since a computer player is just a player with the ComputerStrategy and the human player is a player with the HumanStrategy (that is they are one type, player).

6 Extra Credit Features

For extra features I implemented two different computer player strategies: DumbComputerPlayer and SmartComputerPlayer. Too see it at work, refer to demo. This was not difficult since I had designed by project from the beginning so that additional strategies can be easily implemented.

7 Final Question

What lessons did you learn about writing large programs?

1. *The importance of a well thought out plan.*
   Unlike smaller programs where I can plan while implementing for a larger program like this, doing that will leave me confused with alot of broken methods. Due to the complexity, it is crucial to have a well thought out plan before implementation. This way when implementing, I only need to focus on implementation and don't need to worry about where this method will fit in, as that has been planned out. When I do get confused, I just need to take a look at my plan to figure out where I am in my implementation and how what I have worked on will fit into the bigger picture.

2. *Start from the basics and build on it.*

   It is easy to be overwhelmed by the complexity of such a large program. I was when I first read the guideline. I was given the advice to start by building the simplest version of the game then adding on bits by bits until it is what is required. This was a great advice since adding on the more complex functionality is a lot easier when there is a basic game. In addition, it's easier to pinpoint bugs when the program is built little by little. If I tried to implement the whole thing then debug that would be alot of trouble as the bug could be anywhere in the code. If we build little by little and debug right away the bugs are much easier to catch.

3. *Sometimes it's best to start over.*
   When I was implementing my final iteration, I started by trying to fix my implementation for my second iteration. This was quite difficult because at that point my code had become easy and hard to read. It was also hard for me to remember what each method did and how they were implemented. I just started over (not totally from scratch because I had my old implementation to refer to). This was a faster process since it allowed me to traverse through all methods thus they were fresh in my mind when I started to implement the new functionalities.

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

1. *Spend more time planning before implementing.*
   I would have spent more time planning my design so that I didn't have to go through so many iterations of my design. This would include sketching out what methods and fields each class have and how they fit together before implementing the classes. I only had a rough idea of what each class is responsible for, thus when I was implementing them I found myself getting confused quite often.

2. *Find a way to implicitly manage memory without the composition relationship between Game and concrete Strategy classes.*
   One of the flaws that still exists in my design is that Game needs to own all of the concrete Strategy class so that implicit memory management is possible. This could get tricky if there are alot more concrete Strategy classes. If I have the chance to start over or be given more time I would want to find a design which solves this issue.