Jason Ming Li
20907761
jm25li@uwaterloo.ca
Dec 15, 2021

CS246 Final Design Document

Introduction

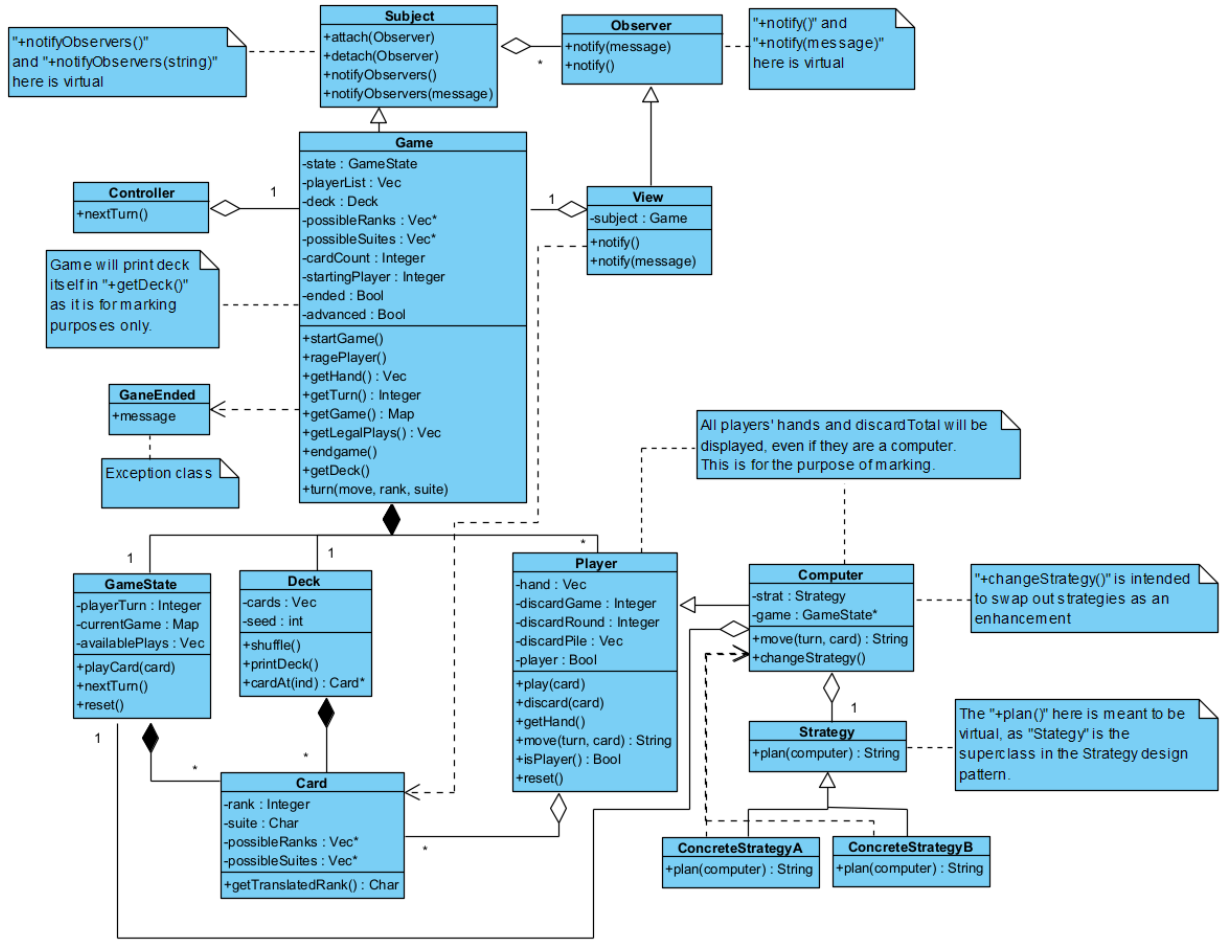
In this program, the game of Straights has been implemented with memory entirely managed by smart pointers. Features include those outlined in the project specifications. That is, the game of Straights itself, the ability to add computer players with a simple AI, the ability to see the deck, and the choice to end games early. Pseudo-random shuffling has also been included as specified, taking in an optional integer seed in the command-line. Thus equal seeds result in equal deck states post shuffle.

Additionally, extra options can be enabled to go beyond specifications. Additional features include playing with a number of players outside the default of 4 and the option to have computers with “advanced” AI replace rage quitters. Note, however, in the case a factor of 52 (the number of cards in a standard deck) players are playing, then there’s a chance the 7 of spades isn’t drawn, thus no one can ever “play” a card.

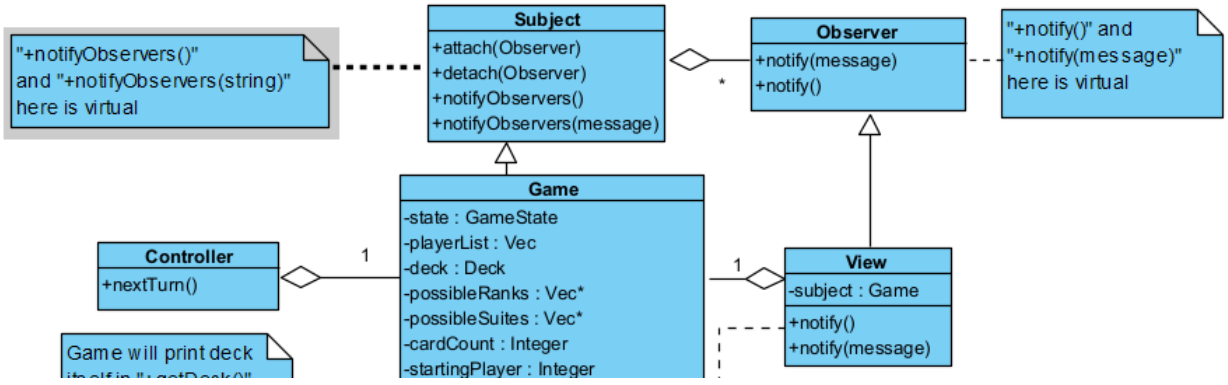
This document will cover in depth the implementation details of this program, and my takeaways from this final assignment. For details on how to run the program itself, please refer to “demo.pdf”. You may find it helpful to also have “uml-finall.pdf” open, specifically looking at the enhanced version of the program.

(Without images or page breaks, this document is only 7 pages.)

Overview



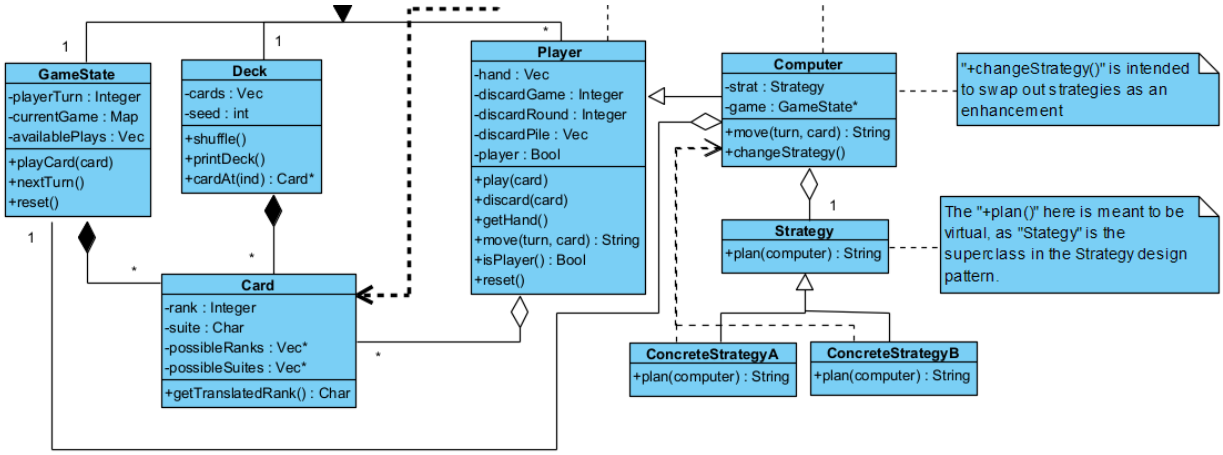
The overall structure of this program (shown above) hasn't changed much since the iteration presented on due date 1. No additional classes (outside of some error handling headers) have been added, and the relationships have only changed slightly.



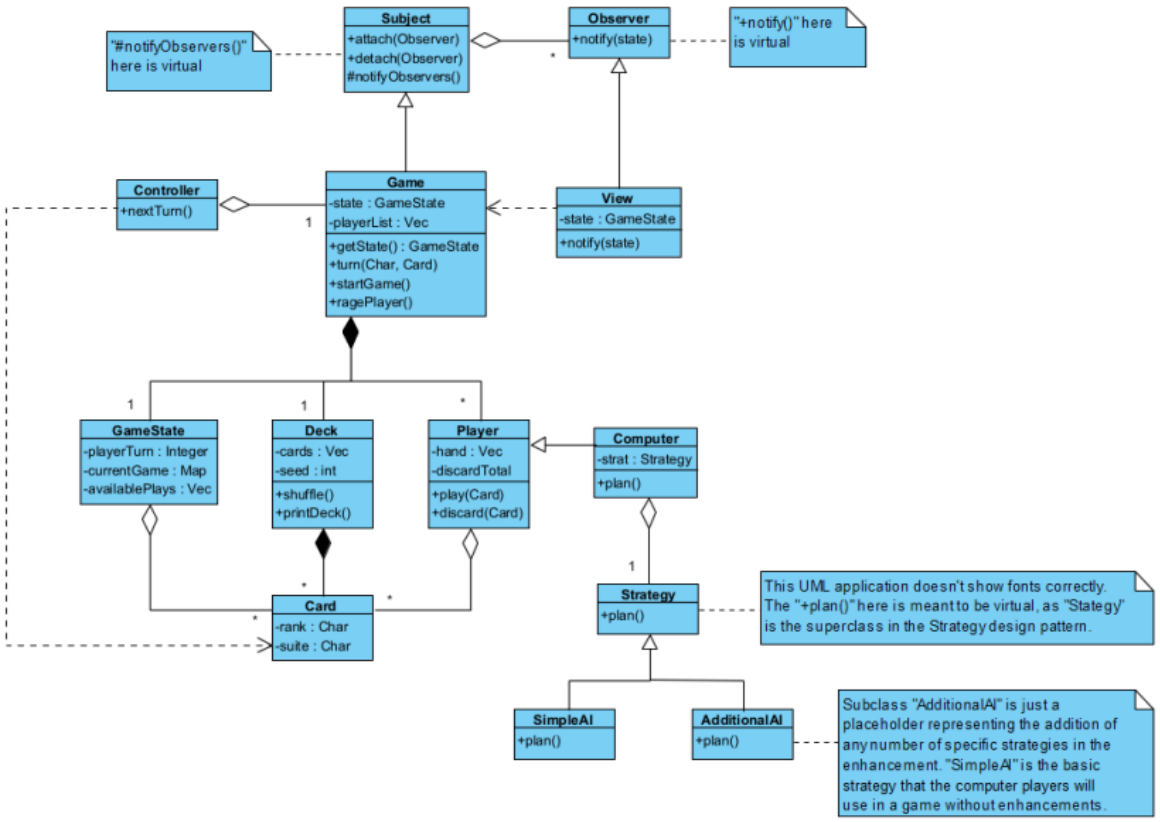
The first division in classes still follows the MVC model. The text-based View still remains as an observer of the Game in an observer design pattern. Prior to starting due date 2, the plan was simply for View to pull information from Game, but now there is also the option to

push statements from Game to View in light of a few problems discussed in the *Design* and *Resilience to Change* sections.

Inside the model, Game remains as the central hub for the functioning of the game. The rules of the game, what happens with each action and how the game of Straights proceeds is processed within the Game. Information on the state of the game is kept in GameState, the state of the deck is kept in Deck, which also owns the Cards held by the Players. And the Players keep information about their own hand and their different discard stats.

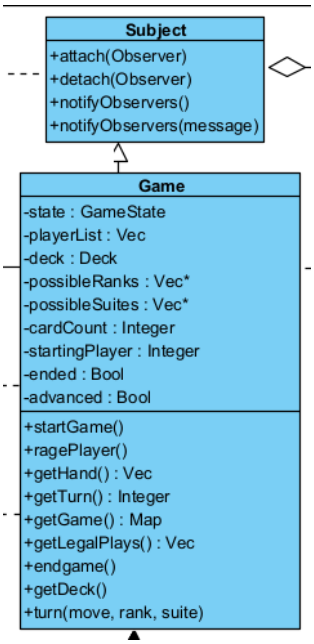


Computer players are still a subclass of Players and still utilizes the same strategy design pattern for swapping out between different Strategies. What changed here, though, is the direct access to the game's GameState for reasons discussed in *Design*.

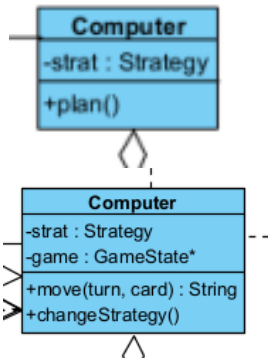


There are a significant number of additions in terms of class functions and parameters that are best seen in comparing “uml.pdf” (shown above if unaccessible) with “uml-final.pdf”. These are simply corrections to under-complications in the original plan and missed project specifications, or for quality-of-life.

Design



The first of many problems was the question of how Observers would detect immediate actions which are never tracked in Game without severely weakening its encapsulation. Unlike the state of the game, Game never keeps a history of player actions. While the tracking of state with GameState is essential to the functioning of the game, actions don’t need to be known for longer than a turn. Therefore, the idea of introducing and managing numerous parameters to track the last action, by who, and with what card would be cumbersome. It is far more reasonable, however, to push this all information to the Observers when a move is made.



The next challenge faced occured when developing the Computer player. While the Game knows what the human player wants to do from the input from the Controller, the original Computer player had no way of interacting with the game or knowing anything about the game. This led to the decision of letting the Computer player directly access the GameState. Specifically, this allows the Computer player to tell the GameState a Card is played without the unnecessary overhead of repeated validity checking through the Game and, more importantly, without exposing a public function to the outside controllers that allows for unchecked plays.

```
void Game::startGame() {
        if (ended) throw GameEnded{};
        deck->shuffle();
        state->reset();

        // redistributing cards
        int hand_size = possibleRanks
```

This required a change in how GameState is managed however, especially when it comes to starting a new round. The original plan was to reinitialize the GameState. This way the round looks just like when the game started, but the Players still remember their necessary stats. Deallocating the old GameState whilst allocating a new GameState, though, invalidates the GameState pointers held by the Computer players. Thus, the reset function had to be introduced to the GameState to return it to its state at the beginning of the game without the need to reinitialize it.

Recall the motivation for using the observer design pattern between the Game and View. We'll cover deeper in *Resilience to Change* that keeping the Game/Controller relationship one-way as it is now allows us to have multiple controllers interacting with the Game in a single game. However, the problem is the original intention to have the Game loop through the turns inside the Game itself. In other words, when the game starts and a move is made, the Game calls the controller for the next input before the controller instructs the Game again. The intention here was to have Game handle the progression of the game entirely on its own once a function is called once to start the game. This original idea was foiled by the consideration mentioned prior, though. Obviously, the Game can't demand inputs from the controller itself if it never knows which controller to expect the next input from. As a result, the game loop isn't handled by the game itself anymore, and the Game simply steps through the game.

```
// game loop
try {
        g->startGame();
        while(!g->getEnded()) {
                c->nextTurn();
        }
} catch (GameEnded ge) {
```

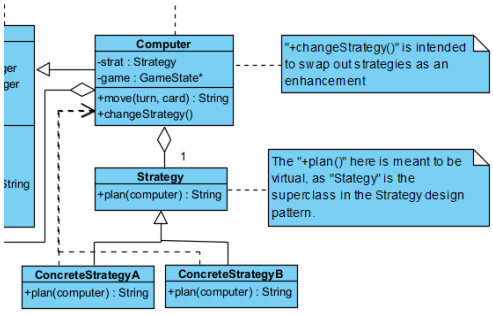
But now, we don't know when to break the game loop. The original solution involved returning status codes. 0 for if the game was still in progress, not 0 otherwise. However, if utilizations outside of Game don't check for these status codes, they could still call "turn()" and mutate the model in unintended ways. To resolve this, I've taken inspiration from how input streams are handled and included a fail-bit-equivalent, "ended", to the Game. Now, with every iteration through the game,

the game loop can check if the Game has ended or not. Additionally, mutating functions in Game itself such as “turn()” check if “ended” is false, otherwise it throws.

Resilience to Change

Firstly, let's discuss the MVC structure that separates the Controller, Game, and View. The motivation for this structure, as presented in lectures, is to enforce the single responsibility principle. That is, the model shouldn’t change if the way inputs are made changes, and it shouldn’t be affected if we want to display it a different way. Similar is for the Controller which doesn’t care about the rules or what the game looks like, and the View which doesn’t care how the model got to where it’s at to begin with. This separation between program state, presentation logic, and control logic enables my program to accommodate for different, additional outputs, inputs, and gamerules.

The observer design pattern connecting the model and view allows for a single game to communicate changes to multiple views, and views of different types, in a modular way without changing any code. This is the usual implementation presented in lectures. The controller, though, simply aggregates the single model, and the model keeps no record of the controllers it receives from. Why? Much in the same way I anticipate the addition of numerous, unique views, I also anticipate the addition of numerous, unique controllers. Say we don’t want to receive human input from standard input. Well, we’ll need a new controller now. That will simply translate the new format for human input into input the model can understand, and send it to the model. But now, the model doesn’t care about who sent the request, simply that the request is made and something should happen. These changes are relayed to the views, not the controller. Thus, it is one way. Additionally, if multiple controllers are communicating to the model at a time, it shouldn’t be the model’s job to decide where to get input from; that’s not very cohesive to the model's job of managing program state. That decision should be delegated to some master controller or other overhead code depending on the reasons for multiple controllers.



Next, lets address the strategy design pattern implemented for the Computer’s AI. There are a number of ways to play the game correctly, but a Computer must decide upon a move when it is it’s turn. The computer may also want to change strategies in the middle of a game. The implementation of the strategy design pattern is perfect for this scenario, as we simply have to write a new subclass to strategy instructing the Computer how to make moves, the add conditions in Computer to know when to switch out the current strategy with an instance of the new strategy. This allows us to continue adding AI’s without making major changes. To highlight this, I’ve added an optional second strategy for Computer players to adopt. Only Computers replacing rage quitting players will have this AI if enabled (I’ve chosen this for showcase purposes as I’m not sure when is a good time to start employing this

strategy or if this is even a good strategy). This additional strategy was implemented after the rest of the software had been compiled and tested. Outside of the actual strategy implementation itself, I only had to add 2 lines of code across the entire program to fit it in; one statement in Computer to switch the strategies, and another in Game to call a switching upon a rage quit if enabled.

Outside of these groups of classes, changes to accommodate for new specifications would typically be isolated to individual classes. In the later half of *Answers to Questions: Designing to minimize code change*, I discuss the encapsulation of cohesive classes resulting in this. And to that end, as I've mentioned in my original project plan, the way I've divided up the major classes under Game is transferable to many other card games with different rules and potentially a different GameState depending on the format of the game.

Lastly, I want to touch upon the configuration options in the existing program. The original specification is detailed to handle 4 players, each with 13 cards. But what happens if we only want to play with 2 or 3 players? I suppose you may consider this as an enhancement, but accommodated for any number of players above 0. The GameState appropriately keeps track of whos turn it is (so in the case of 3 players, it goes 1->2->3->1), and the deck is distributed evenly to all players by Game (so in the case of 3 players, each player receives 17 cards, and 1 card isn't given to anyone to be fair. There is the possibility, however, that the card that isn't distributed is a 7 of spades, in which case nobody can ever play a card, and everyone should be discarding on their turn. That's the game's rules, so it's an intended outcome). It doesn't just stop at the number of players in the game either. States such as Deck and Card are flexible containers that work for any ranks, suites, and deck size. Handling of cards in Game and GameState also already support changes to such specifications. This is done by every relevant class possessing a pointer to a vector of suites and a vector of ranks. Thus, it can dynamically generate the possible cards, know which cards are valid, and the resulting deck size when Game is generated.

```
// ranks and suites found in the deck
std::vector<char> ranks = {'A', '2', '3', '4', '5', '6', '7', '8', '9', 'T', 'J', 'Q', 'K'};
std::vector<char> suites = {'C', 'D', 'H', 'S'};
```

```
// starting the game
auto g = std::make_unique<Game>(&ranks, &suites, seed, computerPlayers, advanced);
```

Answers to Questions

Designing to minimize code change:
I've designed to separate my code in a way to maximize cohesion while maintaining a reasonable level of coupling. I've talked in *Resilience to Change* and *Overview* how I've broken up the game loop using MVC. This is a very common implementation for this sort of objective. That is if the View and Controller can remain ignorant to how the Game processes information, so long as it inputs and outputs values in the same format. Similarly, the Game can remain ignorant to how the View displays or the Controller gets information, so long as the input or output the same information respectively.

Thus, if I intend to switch the user interface to something different, whether if be a fancier text-based display or one that utilizes a graphical library, I fully anticipated this in allowing the new view become a subclass of Observers and attach itself to Game much in the same way the current View does. We may even handle the fact that, sometimes, View is pushed full statements to display. This can be achieved by parsing the message as a string stream, looking up for keywords such as whose turn it is after the word "Players," and specific action words such as "discard" or "plays" or "invalid," just as a few examples. The new view itself may have to be built from the ground up, independently from the current View, but that is inevitable since it needs to process information from Game differently to accommodate its new display type.

The classes within the model themselves are also impacted little by the changes in other classes within the model. Because each class has a number of getters and setters, they can easily communicate values across classes using similar formats despite varying in implementation. If the rules of the game are changed, only the Game has to change its rules, and possibly the Computer strategy as well. GameState itself is told by Game how to look. Deck simply provides the Cards. Players only care about what happens to their hand when they play or discard a card. None of these 3 classes really care about the rules or how it is implemented, simply what they are told to do with their state at the moment. Thus, adjusting rules in Game is trivial, as Game itself is orchestrating the game.

The gist of it is, with everything encapsulated and implementations hidden from one another, classes depending on other classes only care about whether the input and output have similar results and similar formats. If so, it doesn't care about how it's done.

Structure for switching computer strategies:
As I have shown in my UML diagram, the strategy design pattern is most appropriate for this. Doing so doesn't impact the code outside of the Computer at all. All there is to be added is a subclass to Strategy that implements that particular strategy, and an operation in Computer to allow for switching between the various strategies conditionally. Just as I have discussed in the question prior, the Player only really cares about itself, and the GameState as well. Our strategy simply takes the place of the Game and instructs GameState itself what is happening. But, again, it's encapsulated and GameState doesn't care about who's giving the instructions, simply that the instructions are accepted.

Handling jokers:
Handling jokers would require a few more changes across different classes, simply for the logistical issue of knowing what the joker card should represent. Firstly, Card itself should now be constructed with a bit dictating if it is a joker or not. If it is a joker, it is seen as a joker regardless of what the rank and suite is since jokers are independent of that. Next, the Controller should anticipate an additional field after it is stated a Player will be using a joker card. Then Game should anticipate 2 arguments for cards; one argument will be what that actual card is, so all cards with jokers included, while the second argument will be what the card represents, so itself if it isn't a joker, otherwise it is a non-joker card. Game will check if the actual card is in the player's hand, then if the representative card is an available play. Additionally, when displaying all legal plays, the Game should realize that jokers can represent any card, thus all plays are possible if a joker is in hand. If we've implemented this correctly in all these classes, we shouldn't have to make changes to the Player, the Deck, the GameState, or any views. Computer will have to change in the sense that it too should know what it can do with a joker card, but that is all there is.

Extra Credit Features

As I had mentioned in the intro, I've additionally added the compatibility to play with a different number than 4 players, human or otherwise, compatibility with different decks, and I added the option to enable Computers to replace rage quitting Players, but with "advanced" AI opposed to the simplistic AI outlined in the project guidelines.

The first two features I've talked extensively about in the last paragraph of *Resilience to Change*. The challenge here was how to keep information consistent between all the classes that require it in the model. Things such as the size of the field, the possible available plays, the cards in the players hands, how many cards per hand, and many more are affected by changes in deck details and changes to player count. Some solutions were easy, such as dynamically calculating how many cards were to be distributed to each player after getting the deck size and player count. Other things weren't so easy, such as keeping rank and suite information consistent.

I resolved this by requiring pointers to a vector of ranks and a vector of suites in the Game constructor. This pointer is initialized to all the classes that require it, thus saving on space (not requiring to save entire vectors) and updating overhead (updates to the original vector are seen by all included classes). Therefore, all classes are referring to the same set of ranks and suites, and don't hold onto stagnant, potentially obsolete, or pre-programmed data.

The third was relatively simple, carrying over the ideas from the lectures on the topic of the strategy design pattern. I simply have the Computer player point to the singular superclass of Strategy, and then have logic to point to a different instance of Strategy subclasses when needed. The main challenge was simply creating a new strategy and knowing when a Computer player should use it.

However, the game of Straights, as detailed in the project guidelines, is quite restrictive in the sense one has to play if they can play, otherwise they discard, and the game itself is pretty simple. My options for how the Computer player should move and when to change strategies were limited.

Thus I chose to have the Computer utilize the “advanced” AI when a player rage quits. The reasoning here is because, in a normal game, a player rage quits if they are unhappy with their position in the game, likely because they have a bad hand. So the “advanced” Computer player replacing them aims to play as best as they can instead of using the aimless “simple” AI. It’s not exactly having Computer players analyze the situation and changing strategies on the fly, but then again I’m not even sure when it makes sense to change strategies in a simple game like this myself.

Here's the overview of the strategy if you are interested. The computer looks through their hand for a card in the list of legal plays it has to play. If it has such a card, they play the one with the highest rank. This aims to remove the potential of discarding high-scoring cards. If it has no legal plays, it discards the lowest ranked card in their hand, minimizing score gained.

Final Questions

Takeaways:
I’ve worked on a number of equally-sized if not larger projects in the past, and everytime it surprises me how cumbersome it becomes to share necessary data around whilst minimizing coupling.

And though it may not be perfect this time around, I found I was a lot more efficient in managing this after having drafted a visual plan using UML. The ability to see where certain information is contained and how data flows at a glance has been invaluable for navigating my files without the division into directories.

The complication of minimizing coupling still remains a struggle, though. If the time permitted, I would love to create more detailed documentation for my functions and better plan the formats of their outputs. For example, coming up with a structure for representing the cards on the field in GameState took some time, considering 2D vectors and various other options before settling with a map of vectors as it provides an intuitive table-like structure with the key, suites, for rows and ranks in the vector for columns.

A big pain for me had been the aspect of communication, particularly on the receiving end. Though I did work alone, it’s equally important for me to fully understand the criteria when developing any program. With a large program such as this, any misunderstandings scale with each other. I found that out when I had to redo much of my model after rereading the specifications on how the game is played and how score is scored. There's a number of additional functions and parameters in each class between my two submitted UML diagrams and unlisted private functions, and that is mostly in part to this problem.

The last takeaway I want to discuss is modular testing. While I was lucky in that I had very few issues after I had compiled the entire program for the first time, I felt quite anxious proceeding with the construction of the entire program without ensuring individual classes worked as intended. It would be much more efficient in the long run to create test suites for individual classes in the future so my issues are already isolated to one class and my focus is more directed to developing one class opposed to the entire program.

Second chance wishes:
First thing I would do if I had a second chance: error handling. In past large projects, error handling had been something I dreaded as a confusing mess. And I tried to get away without error handling in this project. Ultimately though, my aversion to proper error handling has led to messy nested if-statements that are not only hard to read and comprehend, but also tedious to navigate in the console.
Secondly, documenting my code and functions first. Having felt confident in my understanding of my intended implementation, I thought I could get through this in one fell swoop without referring to any documentation. I almost had it, but with the onset of new functions being thrown into the mix, I quickly started forgetting how things were implemented and slowed down quite a bit in trying to understand them again. Also, if I want to make changes or accommodate new specifications down in the future, it would be very difficult for me to do at a glance.

Lastly, lessen the discrepancy between how human and computer players are handled in game and in the controller. I quite dislike the way I have to use a boolean value to distinguish between a human and a computer and then require parameters for one's "move()" while none for the other. It feels at that point I might as well have two different classes opposed to creating Computers as a subclass to Player. I had tried, and failed, at incorporating the visitor design pattern for this case. However I couldn't get around needing the boolean in the controller and carried on with it. The end result is bulky code in the Game and additional functions to accomodate for the Computer players functioning that I personally am not happy with and knows there are more elegant alternatives.

Conclusion

This concludes my design considerations for this project. It was an interesting challenge to create a large object-oriented program with this much consideration and pre-planning to the overall network of classes. At first, the design choices felt trivial, such as the case for MVC and strategy. However, as the project progressed, I began to realize and appreciate the amount of work that goes into communicating efficiently between classes while minimizing overhead and maintaining strong encapsulation. Though my project isn't perfect design wise, I'm delighted to have a functioning program complete in the time frame given.