CS246 Final Project Design Document - Straights
By: Simon Zuccherato

Introduction:

When I first started working on this project, I thought it would be simple. Reading over the specification for Straights, I could already see in my mind's eye how I wanted to implement it. So I created my design document and UML, foolishly thinking that I would encounter relatively few problems while coding.

Truth is, this was probably my most difficult assignment this whole year, which is fitting for the final project. I ran into multiple unforeseen problems, which ended up impacting the overall design of the program. Luckily, I managed to get everything working almost a week before the deadline, giving me time to study for my exams and work on the documentation.

Overview:

The basic structure of my program is as follows: there is a Card class which holds the basic information of a card (mostly just Number and Suit), in addition to methods to access those values in differing formats.

The cards are stored in the deck, which belongs to the Game class. This class performs actions related to the game's functioning such as shuffling and checking if a play is valid, in addition to checking if a game is over. The Game class also contains a vector composed of Players.

Player is a class that has a hand of cards, a discard pile of cards, and a score which is an integer. Player is the superclass for the two types of players that can be found in a game of Straights. These are Human and Computer players. Computer players have a method to choose a card that can accommodate multiple levels of AI, in addition to methods to play and discard those cards. Human players are relatively simple compared to Computer players, with basic methods to play and discard cards. Their complexity resides in the Main class.

The Main class contains most gameplay loops. This includes dealing with command-line arguments, setting up which players are Human and which are Computers, receiving input from human players, and printing output.

The structure of my Straights implementation isn't perfect. There are a few different changes that could be implemented, such as making the main gameplay loop happen within the Game class and making a few more methods private. If I were to remake this assignment, I would probably incorporate many of these changes. As it is, however, I think it's a serviceable structure.

Design:

There weren't any real design patterns used in my implementation of Straights. The only instance of a design pattern was the fact that vector iterators were used in order to erase elements from the vectors found throughout the program. However, I myself did not implement the iterators in this case.

Originally, I was planning on including a variable determining which Game each Player belongs to. This Game would help determine whether plays were valid, and would remove the need to pass a card back to the main method in order to play it on both the Player and Game. Unfortunately, this didn't really work out as I had failed to consider that two classes cannot both include each other. This was the biggest problem that I encountered while coding. In fact, the other major design problem I had was a symptom of that original problem. After I had reworked my code once, I realized that even that solution did not remove all references to the original Game object in my Computer code, so I again had to shift around and restructure a few of my methods until this problem was actually, finally, resolved.

The most difficult relationship between classes for me to implement was the inheritance relationship between Player, as the superclass, and both Computer and Human, as the subclasses. As this was the first time I had written .h files for inherited classes from scratch, I found it a bit difficult to get the inheritance working the way I wanted it to.

Specifically, I had included the subclass in the superclass, and as I am used to working with Java I assumed all methods would be virtual. Thankfully, both of these were quite easy to fix once I figured out exactly what the problem was.

One thing that I thought may have been difficult to execute, but that turned out to be surprisingly easy, was the use of RAII. Specifically, players were kept in shared pointers while cards were kept in unique pointers, both within the Game class. Keeping Cards in unique pointers was actually quite simple, as they did not need to be edited after their initial creation and could easily be passed around to the hands of different players or to different functions (such as those to check if the card is valid or to eventually play the card).

Honestly, I find using RAII easier than using heap allocation most of the time, as memory leaks are a major headache and I very rarely manage to get all my deletes correct on the first try. Therefore implementing RAII wasn't just for bonus marks, but also as my personal preference.

Resilience to Change:

Overall, I think my program ended up being quite resilient to change. There are still some small improvements which could probably be made, but changes in the rules are relatively easy to implement.

Even adding in the bonus elements that I did add took relatively little time. Adding the better AI simply required adding a section to the choose method of Computer, and while adding the improved user interface was a bit tougher, requiring changes in both Game and Main, it wasn't hard to figure out and was implemented in only a couple hours.

Although the cohesion of classes may sometimes be a little low, especially in the Game and Main classes, overall the classes are structured so that changes require relatively few edits to different classes in order to be implemented.

One type of change that I think would be difficult is adding a GUI to the game. While I had it on my original schedule, I did not end up including it because I did not end up having enough time, plus I do not currently know how to implement a GUI in C++ and am not motivated enough to learn that concept for a couple extra bonus marks. I do think it would be difficult, however, since multiple classes (both Game and Main) print to std::cout, and both of these classes do a lot of printing. Frankly, there's a lot of text involved in this game, as it uses a text interface. Changing all of this to graphical text would be difficult, and while I changed up the text for the improved UI, figuring out how to make the text work with the GUI would require a lot of effort on my part.

I was actually surprised by how resilient to change my final program ended up being. After I had to restructure my classes due to not being able to include Game in Player, I thought that this would end up ruining my resilience to change. In a few ways it actually made it stronger, however; it allowed me to introduce the choose method to my Computer class which compartmentalizes the AI choice of a Card into one method. Therefore, changing the AI of the program only required changing that one method. Originally, I was going to combine the choose and play methods of Computer, but I feel they work better as separate methods as it also allows the returned value of choose to be used as either a play or a discard depending on the situation.

I think the structure of my program could also apply to many different card games. For example, since I have a stripped-down Card class with a suit and a number, this class could apply to many other card games played with a standard deck of 52 cards. I also think that the basic structure of the Player class and its subclasses are applicable to most other card games, as most games will have both Human and/or Computer players. Player could be similarly extended to include other types of players, such as a Dealer in a card game that uses them.

The main changes that would have to take place if the rules of the game were switched would be within the Game and Computer methods, as which cards are valid and how cards are played changes dramatically between games. The Game class would also likely need to change its field variable in order to accommodate a different style of field from the four rows of Straights.

As for the methods that would need to be edited, the major one would be checkValid in Game. This is the method that determines whether a play is valid, and as in different card games different plays are considered valid, this would likely need a major overhaul.

Another method that would need to be changed in Game is play; this method plays the card on the correct part of the field. As the field itself would likely be changed, this method would need major changes. Additionally, because most card games will likely not use the comparison style of Straights, this method would almost have to be rewritten from scratch for a new card game.

In Computer, the biggest change would be in the AI. A different card game needs a different type of AI that can understand what plays need to be made and choose a play that matches the situation. Therefore the AI would also need to be revamped if the game itself was changed.

Another change that is very likely if there were major changes in the structure of the game is a change in Main, specifically in the interface to the player. In another card game, there might not be a loop of exactly 52 cards to be played between each round. Rounds might be truncated early, or there might need to be a reshuffle within the current round.

Therefore, the exact structure of the main game loop would likely need to be changed in order to accommodate the different rules. These changes might require separating the check to see if a player wins from the for loop that runs to ensure that there is a play or discard for every card in the deck; it would be a simple restructure, however, likely only making slight changes to the loop as it exists

There might also need to be other commands to be accepted other than the simple ones that are needed for Straights. These would be relatively simple to change, only requiring another else if in the main game loop in order to receive the command and call the correct method.

Overall I feel confident that if I were asked to implement an entirely different card game using this code base, I could accomplish it in only a day or two, which would take much less time than coding the entire thing from scratch. Therefore, I think I have succeeded in coding a card game that is resilient to change.

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.

Answer: The class design that would probably make the most sense would be Observer, in order to ensure that the hand is updated when the field is updated and vice versa. My classes do not fit this framework. I'm going to be upfront about this because as the game developed I realized that it would be easiest to not use such a design pattern; instead I simply called both the update methods for the game and player from the Main class. While this gets the job done, it isn't quite as efficient as the Observer design pattern, as it requires two calls from the Main class instead of one.

I probably would have worked harder to implement this design pattern if I were to redo this project. As I worked on the project things just kept slipping further away from any idealized design pattern as bugs popped up and I switched around how things were done. I don't think the end product's perfect by any means, but I'm proud of it and it's mine. That's something, even if it doesn't fit any pattern.

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?

Answer: I structured my classes using inheritance so that both human and computer players share common attributes even though they do still have some differences. Additionally, it allows both of them to be stored in the same vector. Although my game does not currently support different play strategies for different players, as I have implemented different levels of AI this would be quite simple to implement; simply call a different type of AI for each separate computer player.

Changing the AI strategy dynamically would also be simple; as the AI is stored as an integer within the Computer class, changing it would simply require mutating that field. This could be performed at any point during the runtime of the game, as a simple check could determine if the conditions for change have been met, with the AI update following as a consequence if they had.

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?

Answer: In order to facilitate a simple transfer, I structured my Player class so that each of its fields (hand, discard, and score) had an accessor method. While this improved flexibility in the Game and

Main classes, it also proved to be a key solution to this problem. Creating a new Computer player from another given Player is as simple as setting the hand, discard, and score of the new player to be equal to those of the old one. This constructor is versatile enough that it does not necessarily need to be used on a Human player; if necessary, it could also be used to construct a new Computer player from an old one, though I cannot think of a use for that functionality.

The new Computer player is then added to the vector of current players in the correct spot and the old Player is erased from the vector. It's a simple replacement.

Extra Credit Features:

The main extra credit feature that I included was completing the project without explicitly managing my own memory. As described in the Design section, this was a choice I made and I honestly found it easier than using explicit memory management. I did end up using raw pointers on a few different occasions, but they were used to pass around values, and they allowed cards to exist in multiple different places at once. The cards were kept primarily in the deck, but they were lent out to the players' hands, discard piles, and fields using the raw pointers to the cards. Since these all went out of scope when the Game and therefore the deck went out of scope, there were no problems with the memory management.

Another extra credit feature I included was a higher quality AI for the Computer players. This level of AI goes further than simply playing the first legal play or discarding the first card in the hand. This AI plays the highest numbered card out of all the legal plays, except in the case the highest legal card is a 7. If that is true, it only plays a 7 if the only legal plays are 7s.

Additionally, this AI discards the lowest numbered card in the hand. As soon as I implemented this AI, I saw results as the computer players got much better at the game, discarding fewer cards and generally putting up better scores compared to my score as a human player. Therefore, I would say this implementation was a success.

The final extra credit feature I added was an improved UI that prints out each card with its own border. This UI makes readability much easier for the players, and it makes each card of the same number but different suits line up with each other vertically on the field. Overall, it's a major visual improvement on the initial UI that the project document specifies.

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?

Through this project, I learned how important it is to be flexible when working on a large program. On a large program, it's good to start out with a plan, but things will usually come up that require revising that plan. It's important to not be afraid of changing the program and to be willing to iterate on the initial design repeatedly until things work the way they're supposed to.

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

If I had the chance to start over, I would probably structure my program to make better use of the Observer design pattern and to have higher cohesion between my classes. I would also just clean up the code in general and make it more efficient. Overall, however, I think I did a pretty good job on this assignment and there isn’t all that much I would change.

Conclusion:

I’m glad I chose Straights to implement. It was simple enough that it didn’t take too much time away from my other assignments, but it also had enough complexity to keep me engaged and working on it for a couple weeks.

I really enjoyed CS246 as a course. Thank you for providing an engaging and challenging course, and I’m looking forward to future CS courses!