Straights Implementation

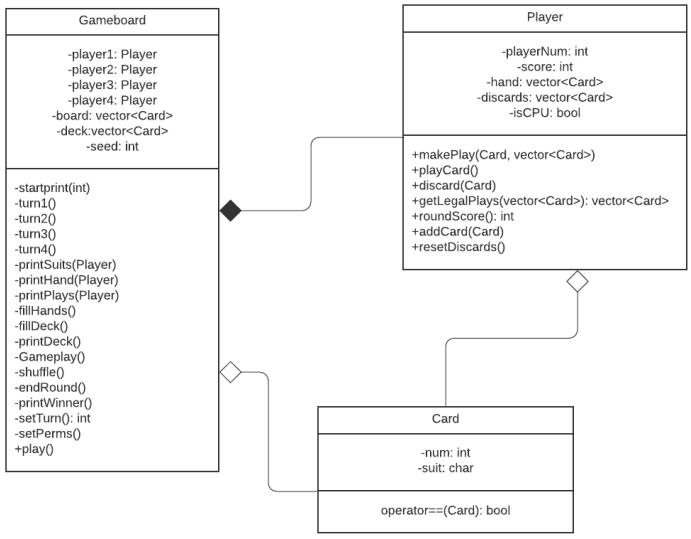
Joey Johnson

Introduction:

For the final project, I chose to create an implementation of the card game Straights. In this document, I will detail the workings of my program, how it came to fruition, and some overall thoughts in regard to the program and the coding process in general.

Overview:

My Straights implementation was structured to follow a loose derivation of the Model View Controller design pattern. The general idea behind implementing this strategy is to have the three main aspects of a program (the data holders, the data updater, and the data presenter) in distinct segments, so that a modification to any of them will not require changes in the others. Though I did not follow this directly, my program was thoroughly inspired by the thesis behind this, and much of my program follows these principles.



My implementation of Straights is broken into three distinct classes, as well as the main function. I chose to structure my program like this because I felt that these three aspects (cards, players, and the gameboard) were the most prominent, distinct aspects of the game. As such, I chose to implement them separately from one another in order to maximize the simplicity of the code. Encapsulation principles are very prevalent here as well. Whenever possible, const accessor methods were opted to be used to keep the implementation details of the related class private. Additionally, a large portion of the methods I created were set to private to further emphasize this, and to ensure that during testing or usage, nothing was being altered that should not have been. I also made plentiful use of the <vector> class whenever possible and found it to

be both easy and effective to work with. The ability to dynamically change the size of the vectors throughout the program and the easiness of both adding and removing elements made it a no-brainer to use, and it did exactly what I was aiming for it to do at all times.

I began my implementation by creating the Card class which, hence the name, represents a standard playing card. Each card object has a number (int) and a suit (char), and cards can easily be created through a constructor that takes in both of these numbers. Some cards proved more difficult to create than others (more on that below), but after some alterations this method proved itself to work. The card class has accessor methods that are used throughout the implementation, as well as an overridden output operator which assists in correctly printing the cards.

The Player class was crafted next, and includes all of the methods that are necessary for a player in Straights. Each player has a player number (int), a score (int), a bool that states whether it is a CPU or a human playing, as well as two vectors of cards to represent their hands, and their discards. There are a multitude of methods that assist in all of the actions players make throughout Straights, and these methods are either standalone, or work in conjunction with others in the class. For example the makePlay(Card, vector<Card>) method calls upon getLegalPlays(vector<Card>) to determine whether the player has any available plays, and, if so, whether the desired play is one of them. The separation of different calculations into different methods was a tactic I used throughout the entire program, and not only does it make the program much easier to read, but it also allows particular aspects of the program to be called individually, which proved to be very helpful.

Finally, the Gameboard class was the last to be implemented, and it helps with the overall gameplay of Straights. Gameboards each have vectors for both the board and the deck, four Players representing the four players of the game, and a seed (unsigned) which is used for shuffling. As this is supposed to serve as the view aspect of MVC, everything related to output is housed in this class. There are numerous distinct methods that serve all parts of output and gameplay, and ultimately make the program run as intended when the play() method is called. The gameboard class makes use of the methods, both accessor and not, of the Card and Player classes too; all of the three classes work in tandem to create a playable game.
Because the gameboard class effectively minimizes coupling and the vast majority of its usability is housed inside it's own class, running the program in the main function is very straightforward. Once the seed is obtained from the command line (if desired), a new gameboard is created with said seed and the play function is run on it, which commences the game.

Design

A large focus throughout this course and specifically for this process was to ensure that you were "maximizing cohesion and minimizing coupling". This concept means that elements within a unit of code should be closely related and have a single purpose, whereas the different units of code should be relatively distinct. This idea was at the forefront of my mind throughout the entire process, and it can be seen vehemently in my code. Each class in my implementation

has all of the necessary attributes to serve all of their respective purposes. All of the main actions are executed in unique functions, to ensure that changes can be made to each without disrupting the flow of the others. As for the coupling aspect, the functioning of each class is almost entirely separate from the others, which achieves what is desired in this area too. Each object was consciously exposed to others as minimally as possible, and any alteration to the functionality or nature of an object can be implemented almost entirely in it's own class with only minor changes likely being needed in the others. For example, if a change was made to the game rules in relation to the legality of cards available to be played, only the getLegalPlays(vector<Card>) method in the Player class would need to be changed. Though this method is used occasionally in other places, changes only need be made here, and the functionality will remain the same.

Another overarching topic throughout the course was encapsulation. The value of making sure that details can only be manipulated in certain ways was very clear to me right away, and as such I ensured that this was used immensely throughout my project. Any time a field needs to be accessed or manipulated, it is done so through some accessor or manipulator method.
For any information that needed to be accessed or manipulated only within a class, these methods were made private to ensure that only within the class could they be used, and nowhere else. It is much more seamless for each class to only know that their desired command is being executed as they wish; there is no need for them to know exactly what is happening behind the scenes. The importance of both of these concepts was made extremely prevalent in this course, and thus I felt it a necessity to include in my Straights implementation.

As for specific challenges I encountered while designing my program, one that was a major issue at first was in the storing of Card objects. While initially designing the layout of my program, I failed to realize that some cards have two char aspects (e.g. TH or QS), and some have an int and a char aspect (3H or 5D). In an attempt to uncover a method to work around this, I did some intense brainstorming and external research on my own. After much trial and error I eventually uncovered a solution: when reading in cards from input, I would read in two char values as the fields. I would then build the card in a helper function by converting the char representing the number to its corresponding int value (e.g. 'T' to 10), and, if the char was already an int, then I would execute "- '0'" on the number to convert it to an int (this command took a decent amount of research). To output the cards, a similar reverse operation is executed that converts each int to its corresponding char. Dealing with this was a process that took a long time and took many different attempts, but ultimately I was able to overcome it. There were a few other instances similar to this that came up throughout my coding, and thankfully, due to the strategies learned in this course, I was able to develop a solution to them.

Resilience to Change

The ability to change my program as a result of various changes to the gameplay was something we were tasked with keeping in mind, and so I made sure to structure my code in a way that best accounted for this. As mentioned above, the biggest way I adapted to this was to employ a MVC-esque design pattern in my programming. All of the different aspects of the

program were stored in distinct, separate classes in order to limit coupling and make a significant change to the nature of the program a not-too-daunting task. For example, a change to the output and display of the gameplay would only require changes in the Gameboard class, which solely focuses on tracking and displaying the current state of the board. Similarly, if there was a change to which cards could be played at certain times, this would only require a change in the Player class. If the definition of a playing card was changed, then only the Card class would require adjustments. Further, within these classes, all major functions or aspects of the gameplay were made in separate, bare-bones functions, ensuring that any sort of detail that may be changed can be adjusted with little impact on the other functions in the class/program.

Employing this MVC related strategy was, I believe, the best way to account for any change in the nature of the program, simply because it decreases the most amount of code that must be changed as possible. Additionally, the encapsulation used throughout only further drives this home, as I tried to mainly use accessor methods when garnering some sort of data from another class. As a result of this, the other classes will not have any indication that the method of attaining a certain piece of data was altered and the program will continue as usual, exactly as desired. The MVC pattern coupled with the encapsulation used is, I believe, a phenomenal one-two punch to overcome any sort of change that may need to be implemented. Though I am grateful that I did not need to account for any change when creating this program, I know that, given my code structure, it would not be too much of an issue to account for.

Answers to Questions

What sort of class design or design pattern should you use to structure your game classes so that changing the user interface from text-based to graphical, or changing the game rules, would have as little impact on the code as possible? Explain how your classes fit this framework.

For my program I am using the MVC (model + view + controller) design pattern. The MVC pattern is a design pattern wherein the program state, presentation logic, and control logic are all separated, and can therefore be edited and manipulated independently. This means that the different areas have little knowledge of the structure of the various functions possessed by the others. I believe that this choice of design pattern perfectly accounts for this as, given that the three main aspects of the program are separate, a significant alteration to one of the aspects should not affect the others. Changing the user interface from text-based to graphical is a change that is related to the view aspect of MVC, and so only that area will need to be adjusted, and the others can remain as they were: in my case, the Gameboard class would need to be altered. Similarly, changing the game rules would likely only require an adjustment to the model part of MVC, with the others not requiring any adjustments: in this case, likely only the Player class would require change. This design pattern promotes independence amongst different aspects of the code, and therefore has the smallest impact on overall changes to the program. For these reasons, I believe that it is the most effective route to go for this program, and it is the strategy I have employed.

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 class structure?

In this scenario, changes would need to be made to the Player class in order to account for varying strategies. The Player (when isCPU is equal to true of course) would need to develop much more flexibility in the makePlay() function, as well as in the helper functions associated with it. Depending on the current strategy, what cards have already been played, and what strategies have already been tried, different cards will likely be chosen to be played or discarded, rather than in the current implementation where it is always the first card. These methods will need to be developed much more, and possess the ability to change their strategy dynamically based on the stage of the game. Based on the current structure of my program it is unlikely that my class structure will need to be adjusted, however alterations to the internal methods of the Player class would be certainly needed in the event that the strategy changed, specifically to the getLegalPlays(vector<Card>) method. Due to the design implementation that I described above, though, it is fortunate that this is likely the only aspect that would require changes.

How would your design change, if at all, if the two Jokers in a deck were added to the game as wildcards i.e. the player in possession of a Joker could choose it to take the place of any card in the game except the 7S?

Adding jokers to the deck would require some fairly significant changes as it largely alters the nature of the game. First of all, the initial dealing process would need to be reconsidered, as either two players will have two extra cards, or two cards would need to be set aside for the round. Additionally, the entire gameplay would need to be altered to account for the possibility of the same card being on the board or involved in the game twice. It is certain that the getLegalPlays(vector<Card>) method would need to be changed to account for jokers always being a valid play. The play() function would also need to be adjusted to account for the two additional turns that will occur, and thus the while loop will need to be increased by two iterations. For most of the other loops, though, I focused on using the accessor methods to attain certain sizes and thus iterations, which won't require any changes. Joker's do not have a suit or value and so they do not fit the mold of the card vector either, though dummy values representing these items could simply be used to account for them. All in all, adding jokers to the game is no small feat and would require a decent amount of redesign. That being said, my program is structured in the most effective way possible to successfully adapt to it.

Extra Credit Features

I added the ability for the program to disregard invalid input given to it. This was achieved by recursively running the turn*() methods over when an unrecognized command was given.

Final Questions

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?

By far, the most important lesson I learned in this project was the importance of laying out a plan prior to beginning to code. Though I initially dreaded this step of the project and longed to begin coding right away, I severely underestimated how appreciative I would be when I ultimately started building it. It was so much easier being able to follow a guideline when implementing the various functions and classes, and made the overall coding process much more enjoyable. Though changes were of course necessary from my initial plan, they were much more manageable and less aggravating than if I was freewriting the code with no plan whatsoever. I made sure to keep my plan dynamic as well and adjust it to any issues I encountered early on, and therefore was able to follow some sort of guideline throughout the entire process, which helped immensely.

Another very helpful lesson I learned was to not be too attached to one idea, and be willing to make adjustments if something looks unfeasible. It is extremely frustrating when something you layed out doesn't work how you intended, and it can be very tempting to keep changing elements in hopes of finding some way for it to work. Though this can be effective sometimes, it is often a better approach to take a step back, admit defeat, and instead think of a different method to solve the problem. There is no use in beating a dead horse and repeatedly trying the same process over and over again when it seems clear that nothing will change. It may take some extra work, but ultimately it is much more effective to problem-solve and determine another method that will certainly work, and go that route. These two concepts became very clear to me as I worked on my project, and I am certain that I will employ them throughout all of my future coding endeavours.

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

Given the chance to start the process over, one change I would make would be to place a larger focus managing my time better. Though I did finish the planning process and start the coding stage when I intended to, I underestimated the time that it would take to create a working, well-crafted program, and complete all of the other necessary documents. Further, while finishing this I had to balance studying for four other final exams, which meant I couldn't devote as much time each day to this project as I would have liked. Though everything did work out this time, if something catastrophic came up during this period, it is likely that I would not have finished everything on time. Next time, I will make sure to be conscious of what other responsibilities I have when initially planning the project out, and give myself plenty of time to account for anything that may come up during the process. By doing this, along with the tactics mentioned in the question above, I think I will give myself the highest chance of success in the future.

Conclusion:

This project was an extremely rewarding journey to be a part of. This task was initially very intimidating to me: I was worried that I was in too deep, and impostor syndrome feelings were

plentiful. As I began to work through it, though, to my surprise the project became more and more manageable, until, lo-and-behold, I had a working project. I feel extremely proud that I was able to finish my Straights implementation in such a cohesive way, and, though it may sound untrue, gracious that I was tasked with completing it.