Design Doc

Levi Shem-Tov
20886218

Overview

I began by working on the Game class as implementing the layout of the game first would make recognizing the necessary classes that needed to be implemented much easier. When implementing the Game class, the first additional class I needed to implement was the Card class. This makes sense as the program is made to replicate a card game. The Game class needed to shuffle and deal a proper deck of cards. I made the Card class hold a char suit and char rank and associated getters and setters that verify the validity of the cards created. The constructor I created sets all cards to have both a suit and rank of '\0' and then overwrites that when the Game class makes the deck. The Game class has between 0 and 52 cards, based on if cards have been played, replicating a real deck of cards.

After the Card class was implemented, I moved on to the Player class. This class is owned by the Game class, and holds the functions and variables for each player, such as their discard sum and hand cards. The players are initialized at the start of the game, and there must be four of them.

With the Player class, I implemented a decorator pattern on its subclasses Human and Computer. The Player class itself is an abstract base class with Human and Computer being the types of players.The Player class communicates its moves with the Turn class. Player has a virtual playTurn function that is then implemented in both Human and Computer. For the Human subclass, this function takes input and returns the given Turn based on what the current player decides to do. For the Computer subclass, this function uses the logic described in the assignment document to decide what Turn it will make. The observer pattern is implemented for the players as a way to update each other on their Turns. The Player class is thus a concrete observer and the game, updating the other players on one's turn, acts as the subject.

The Turn class is a fairly simple class. It is used to communicate between the Game class and Player class. When user input is read in Player, the Game doesn't know what moves each player is making. Using the Turn class, the Player sets both the Card being used on that Turn and the type of Turn it is, ("play", "discard", etc.).

Now that the necessary classes have been implemented for Game, creating the functioning program becomes straightforward. Most of the Game is played out in the runGame function, which handles all the necessary parts of the game. The Game class can be viewed as the manager of the game. From the command line and through main, the seed is passed to runGame and then to shuffleDeck. runGame then initializes the Players and the deck. Once the deck has been made, shuffled, and dealt to the players, the starter (whoever holds the 7 of spades) is found and the game begins. The game continues to take input from the Humans and make Turns for Computers while the discard sum is less than 80. Once this threshold has been reached, the game ends and the winner is decided.

Design

The main way I used design to simplify the creation of this project was using design patterns. Using the implementation of different design patterns learned throughout the class allowed me to create a more intuitive program and now allows other people to better understand how the program works when looking at my code.

Firstly, using an decorator pattern with Player and its subclasses, Human and Computer, makes understanding the difference between the two subclasses much easier to understand and work with. If the decorator pattern was not used in this case, and Human and Computer were treated as two unrelated classes, many of the game mechanics would be much harder to implement. The main one that would be difficult would be “ragequit”. With a decorator pattern, I simply needed to change the humanPlayer boolean and move over the handCards to a new Computer player. However, if the decorator pattern was not used, making this transition becomes far more difficult. Each player could not just be set to Human and Computer anymore. Rather, without the decorator pattern, Humans and Computers would need to be created at the start, and created and deleted when a “ragequit” occurred.

As well, the observer pattern simplified the communication immensely between Player and Game. When using an observer pattern, updating each Player on the Turns another Player makes is not a difficult process. The playedDeck simply needs to be updated and as all Players have access to this, all necessary information is passed on to the next Player through the Game. Without each Player observing the Game, passing this information from Player to Player becomes far more complex.

These were the two main challenges I faced throughout the project and design patterns were a big help. When I began creating the program, I understood the

relationships between each class but was unsure of the simplest way of implementing them. For example, with all Players needing updates on the Game state after each Turn, I was unsure how to have Player and Game relate without creating a circular reference. However, making it so Player observes Game solved this issue quite simply and allowed for an understandable relationship between the two classes that still works to implement the necessary functionality.

Resilience to Change

As this project gave the option of implementing bonus features, I needed to design my program to allow for the option to do so from the start. I was unsure if I would be bonus features at the beginning of creating the program. This meant that even though no bonus features were implemented, the design of the program allows for their implementation fairly easily. As we have learned throughout the course, the best way to create a program that is resilient to change is with the reduction of coupling and the increase of modularity, both of which I have done in my program.

I found reducing the coupling of my program to be the more difficult task of the two but still designed the program in such a way that there is little overlap. Each class only accesses a different class in one way the majority of the time. While there is still overlap in the uses of some classes such as Card and Turn, the overall structure of the program makes the relationships between different classes fairly distinct, thus allowing for necessary changes to one class to have a small or non-existent impact on the other classes.

Additionally, modularity was a big factor in the design of the project. I found the UML extremely helpful in visualizing the division of classes and what needed to be implemented where. I often go into a coding project or assignment with some idea of how I want the program classes to be divided. However, being required to create this visualization beforehand simplified the process of dividing the classes immensely. Separating the functionality of each class and having a parent class handle the overlap between the two classes (such as in Human, Computer, and Player) would allow the addition of more features with little external change or added Player types with change only being needed in the parent Player class.

Answers to Questions

Question: 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.

Initial Answer: I've divided the game into a Game class which handles most of the game and a Player class that deals with an individual's moves. Doing so allows for easy changes to the interface and game rules. If a legal game move needs to be changed, I only need to change part of the Player class to allow for this. Similarly, to handle the interface of the game, I need only implement an observer and create multiple forms of interface and the rest of the program remains the same. Splitting classes into smaller and more specific classes reduces coupling and allows for fewer coding changes to be made to implement a change to the game itself.

Changes: No changes.

Question: 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 structures?

Initial Answer: All types of players inherit from the Player class and so implementing different computer strategies would not be difficult. All information needed for a given strategy is stored in the Player class. Implementing multiple computer subclasses with different playing strategies would require implementing new subclasses of Player and would not change the Player class itself. For the computer players to dynamically change based on the progression of the game, only a change to the given computer player subclass needs to be made. This change will allow the computer player to dynamically change their strategy without affecting the parent Player class.

Changes: To dynamically change a computer's strategy, different Player subclasses beyond the base Computer one must be created. Then, the implementation of that Computer's strategy would go into that subclass. Then, within each type of Computer class, a function would check the game state and decide the strategy that should be changed to. Lastly, the Computer would notify the Game through Turn, and the Game would then change the Computer's strategy by changing which Player subclass handles the Turns, similar to how the handleRage function works.

Question: 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?

Initial Answer: I would make a change to the function that checks which cards can be played in the Turn class. As the Joker can replace any card (aside from the 7s), overriding the legal playing card function would be easy, and allow the given player to treat the card as any of the legally playable cards. A change would also need to be

made to the function that plays a card, where the player that possesses the wildcard would need to indicate which card they are playing as when it is played.

Changes: The Player class now checks what Cards are allowed to be played so the change when holding a Joker would need to be implemented in the Player class. The change that would be made to Turn would indicate if the Player plays a Joker, and then which Card they are replacing. As well, a change would be needed to be added to all Player subclasses. For Human, an updated way of reading input would need to be implemented. For Computer, the strategy would need to be tweaked to account for Jokers being played as any legal Card.

Final Questions

Question: 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?

Answer: As I worked alone on this project, I will be answering the second question. The main lessons I learned about writing large programs is the importance of planning the structure of the program beforehand as well as being able to adapt from the initial estimated completion dates of certain tasks.

In terms of planning the structure of the program, creating the UML and plan before starting to work on the program was a big help. Knowing the layout of the program and focusing on replicating the UML when beginning the project allowed me to rarely get stuck on issues. If I couldn't solve an issue I was stuck with right away, I had the ability to switch and work on a different function or class without wasting a lot of time and come back with a clear idea of how to implement a solution.

Adapting to the initial time estimates was also very important in finishing this program. My first few time estimates were finished much faster than expected but rather than doing no work for time being and sticking to the estimated completion dates, I shifted the implementation and testing of the program to an earlier date. This helped me finish the program on time because both of those things took longer than the estimated time I had left for them in my initial plan.

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

Answer: If I were to start over, I would have been less concerned about sticking exactly to my UML when I ran into issues with the program. Recognizing that I couldn't realistically foresee some of the issues I ran into and be more willing to move away from the UML design when necessary would have saved me a lot of time. Obviously not every issue can be predicted beforehand, and my UML ran into multiple problems with

the relationships between classes. If I were to start over, being more flexible with changes and trusting my instincts to make them would be the biggest difference in how I would approach the project.