CS246 / Michael Zheng / 20885687 - Straights Final Design Document

Introduction

My Straights game for the CS246 final project was built according to the specifications highlighted in the given Straights document. Not only was correctness the focus of the project, but also the design of my code and how each object within the game interacted with each other. Overall, I felt like this was a fun and challenging final project to build.

Overview

From a high level point of view, this game was built using the observer pattern along with an abstract class and class inheritance to create the players. I also used separate classes for the deck, cards, and piles. From the observer pattern, my subject in this case was the board, which is a class inheriting from the subject which holds a vector of players and the piles, along with the playing deck which is meant to represent the board that the game is being played on (holds the data for the game). For the observer, I created a text observer that inherited from the observer class, which is meant to represent an observer that outputs text based on what's happening in the game.

In terms of differences between my initial design and my final design, there is not much difference. Only thing that changed was that I built out some extra public methods that I did not anticipate in the initial design, as along the way I realized that certain objects needed more functionality. Moreover, in my final design, I have the board (subject) own one deck object which was not in the initial design. However, in terms of the design aspects, for the most part I stuck with my original plan, which was to have an observer pattern for the game subject and a text observer to display text, as well as have the player types inherit from an abstract base class and have all objects be instantiations of a given class.

A large inspiration for why I created my game like this was because I wanted to ensure abstraction and encapsulation, along with ensuring high cohesion and low coupling. Since each aspect of the game, from the players to the cards, essentially have their own class which contains the methods and data that that object needs to hold, it allows other objects to interact only through the given class's interface and using its public methods, which reduces the chance of erroneous errors and ensures that users can't wrongly access certain fields within the class. For instance, I implemented getters and setters for each class where a given user can only access / change a field within the object through a given method, thereby greatly reducing the probability that a field is incorrectly changed. Moreover, having each aspect represented as a class means that I'm able to write specific methods that are related to each aspect, thereby increasing cohesion since a given object's methods are relevant to itself. Coupling is also reduced since each object can only be manipulated through its given methods, and since the given object's methods hold the function logic, the object does not need to depend on the user to manipulate itself, thus reducing reliance on the user and lowering coupling. In terms of data abstraction and encapsulation, abstraction is achieved since the fields of an object are essentially "hidden" from the user and only retrieved and manipulated through its getter / setter methods, and encapsulation is achieved due to the fact that all key aspects and objects of the game are represented by classes which holds its respective fields and methods.

I understand that coupling and cohesion are important design topics, so I will dive into this topic further. Firstly, I will discuss how my program works to reduce coupling. Given that coupling is defined as the interdependence of modules with each other, in order to ensure that my program had low coupling, I had to make sure that there was low dependence among all my objects (e.g deck, cards, pile) with each other. In order to do this, one of the methods I employed was using getter and setter methods for each class. This way, the methods of the class are responsible for manipulating the class's data fields, thus ensuring that there is no reliance on outside functions in order to do so. Moreover, I ensured that each class had an interface (.h file in this case) which had all the public methods available to the client, which abstracts away the functions that the method undergo to achieve the desired effect and by having only the private fields accessible through the public method interface, there won't be any unexpected manipulation by the client. Thus, low coupling is achieved by ensuring that each class has a public method interface which the client has access to in order to make the necessary effects, along with getter and setter methods to ensure that private fields are contained and only manipulated when wanted. In terms of high cohesion, I achieved this through the use of classes for each object along with their relevant data fields and methods. For example, my card objects are instantiations of my card class, which contain private fields suit and value, which is necessary data that the card is supposed to have, along with getter and setter methods for the suit and value data fields. By having classes represent each important aspect of the game and the objects, this means that I am able to encapsulate the necessary data to represent each object in its own class, along with having only the necessary methods that the object needs in its class. Moreover, since I utilized the observer design pattern, I even have classes for the board (which is meant to represent the game, inherits from subject class) and the text observer, which splits the responsibility for outputting text and game logic into two separate classes, thus further achieving high cohesion since each class has a sole responsibility and so its data fields and methods work together to achieve that one sole purpose.

Design

When designing this project, there were certain problems that arrived, and techniques I utilized in order to solve them, which included:

Having centralized game logic. This was a big problem that I was debating before I even began coding. I recognized that in order for the game to work properly and for the game data (e.g each player's game score, each player's cards left) to be recorded properly, I needed some way to hold all this data in one place and be able to manipulate it as the game went on. The solution I came up with to this problem was to use the observer design pattern. I recognized that this game needed to output text, and with my issue of having to have a "centralized" object to hold game logic, I thought that it would be a good idea to make the game board a subject in the design pattern, which is meant to hold all of the data of a given game. This way, I could keep all of my game data in a “board” object which is the class that inherits from the subject class within the observer design pattern.

Displaying text. Although this seemed simple at first, taking into the account that I needed good design and I couldn’t just have everything output text (because what if I needed to add graphical output?) I needed a solution that meant the base code didn’t output text itself and have an observer of some sort take that responsibility. Again, this is another reason that I utilized the observer design pattern. Inheriting from the observer class was my text observer class, which had the sole responsibility of outputting text. This way, I had one object which was responsible for text output, which also meant that if I wanted to add a graphical observer, I could.

Creating different types of players. The given assignment outline talks about two types of players, computers and humans, but what if I wanted to add more types? Thus, I couldn’t just make another concrete class each time I wanted to add a different type of player. Instead, my solution was to make an abstract parent class, called player, in which each different type of player class could inherit from. This way, each type of player class could have the same homogeneous set of methods to ensure that the code works for whatever different player type. Moreover, this made it much easier to store different types of players in the same data structure (e.g a vector).

Creating multiple objects. Since the game utilizes multiple cards and piles, I figured that I would need to utilize classes in order to encapsulate the logic and data of each object within the game and to ensure homogeneity within objects.

Resilience to Change

I believe that my program is flexible, and if there was to be a change needed in the program, my code doesn’t need much editing. Some highlighted aspects of my program that are flexible to change:

Cards. For instance, say that each card needed extra elements added to them (e.g each card needs to hold an extra data field), given that I utilized a card class in order to build each card object off of, a change to the data fields of the card class allows me to easily add extra data to each card object if needed.

Deck. If more / less cards are to be present in a deck, or maybe if a fifth suit of cards is needed to be added to the deck, I can make that work given that I have a deck class and the instantiation of the cards within the deck are only done in one place, which is when the board object is instantiated. Thus, these edits would just require a change to the cards inserted in the deck in the instantiation of the deck, and since each given card is a card class object and homogenous, the code already in place should respond correctly to the edited deck of cards. Moreover, the deck is an object of a deck class, so the cards stored in the deck won’t be erroneously manipulated.

Players. Both my human and computer classes inherit from an abstract parent player class. Thus, if I wanted to add / delete types of players, I would just need to build / remove classes that inherit from the parent class. This way, I can build out as many types of players as I want without having to worry about other aspects of my program changing, as each type of player is really just seen as a child of the player class.

Observers. Since I utilize the observer design pattern, it is easy for me to create another observer, say a graphical observer, which outputs other types of displays. All I would need to do would be to create a new observer class off of the given observer class, and have it take a reference to the board, which contains the game data. I would then write the respective methods for the observer which allow it to display the game data, without interfering with the other processes within the program.

Piles. I built a pile class to represent the 4 piles that the players can place their cards on. The pile class also includes a method which outputs a vector of legal plays that the player can make. If you wanted to change what cards constituted legal plays, an edit to the legal play method in the class is all you would need as the rest of the game logic is just built around a call to the method. Or, maybe if you wanted to have a pile representing multiple suits, you can just edit the suit data field in order to indicate that during instantiation.

Board. The board class which inherits from the subject class is flexible to change as it contains data fields and methods which hold all the game data, and if you wanted to change a certain aspect of how the game itself is supposed to be run, all that would require would be edits to the class's methods in order to do so.

A large focus of the project was to ensure that the project was quite flexible to change and a change in the program requirements would only result in minimal code edits. With this in mind, to ensure program flexibility I made full use of classes and design patterns within my program (as highlighted above) in order to ensure the four principles of OOP are present (data abstraction, encapsulation, polymorphism, inheritance) and utilized so that each module of the program acts independently and has its own responsibilities, which also ties back to the idea of high cohesion and low coupling. Overall, I believe that my use of observer pattern, abstract base classes, and separate classes and interfaces for each aspect of the game makes it so that a required change for a given element of the game only requires minimal edits to the code (edits to the respective independent classes).

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?

Answer: For the first half of the question, which is changing the user interface, the design pattern that I used to resolve this problem is to use the observer design pattern. Right now, since the project specified the use of text output, I have a text observer which has the responsibility of outputting the text needed. However, if I were to change that into a graphical observer, I can instead create a graphical observer class inheriting from the observer class and have that attached to the board (subject), and give it the responsibility of displaying graphics. This way, the rest of the code doesn't have to change and just the type of observer and its responsibility changes.

The second half of the question, which addresses changing the game rules, ties into my program's use of separate classes for each key aspect of the game. For instance, if I wanted to change what cards are constituted as legal plays based on what was currently in the 4 piles in the game, since I have a pile class with a method that retrieves the legal plays for that given pile, I would just need to change how that method functioned. Or, if I wanted to add / reduce the amount of players or piles in the game, I have data structures which hold this information. Or, if I wanted to add / remove cards from the deck, I have a deck class which encapsulates the data of what cards are in the deck and so in the instantiation of the deck object, I could just change what cards are being added to the deck object. Overall, due to the use of separate classes for each object and the encapsulation and abstraction that using classes provides, I can easily change the game rules by modifying the methods of the classes while keeping the rest of the program code unchanged.

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?

Answer: Given that we need to have computer players with different strategies, our class structures can change such that specific computer classes can be generated for differing strategies. For instance, I could have a different strategy computer inherit from the player parent class, or I can have it be a subclass of the current computer class with its "play" method changed in order to match the strategy it wants to play. This way, it is still a child of the player class and can be stored without changing the current code, but at the same time be unique in that it has its own designated style of play which comes from an edit to its methods compared to the generic computer class.

Moreover, the board (which is the subject) stores the game data, which can be accessed through getters and setters. Thus, in order to satisfy the fact that the computer may implement a strategy that changes as the game goes on, the computer is able to retrieve game data during its turn and depending on the game state (how many cards are in the 4 piles, how many cards are left in its hand, etc), change its strategy as defined in its play method.

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

Answer: If two jokers were to be added to the deck, a few changes would be made. Firstly, in the instantiation of the deck, I would push two jokers cards to the deck in order to reflect the addition of the new cards. Moreover, I would make a slight change to the data field and methods of the card class in order to store the fact that this is a joker card, and also be able to communicate with other users of the card class that the specified object is to be identified as a joker card and have the property that it is able to be played as any card other than the 7S. In order to make it so that it has the property of replacing almost any card in the game, we can have the piles signal that it is always a legal play to play a joker, since each pile has a legal plays method which returns which cards would be considered a legal play given the cards currently in the pile.

Extra Credit Features

Some extra credit features that I implemented in my program:

Handling end of file input. Within my program, if you were to exit with an end of file (ctrl + d), the program handles it and ensures that memory is properly freed. I did this by having the program see if cin has an eof error, and if so, would print out an error message and exit the program with all allocated memory freed. This problem was challenging because I had to ensure that when eof was detected, all memory was freed. I accomplished this by having all my allocated memory within the board object, and so when the board destructor was called, all of the memory stored along with it was freed.

Handling of invalid input. If the user were to type a command which is invalid, instead of not being able to handle this, my program just tells the user that they entered an invalid command and prompts them to try and enter a command again. I accomplished this by ensuring that my harness that handled input was within a while loop and so when an invalid command was detected (input did not match any command), then I would not break the loop and instead print an error message and allow the user to give another input. This was a challenge since I had to ensure that I was using a while loop and that my control flow logic made sense so that whenever a command that wasn't understood was inputted, my program knew that it did not match a command and continued the loop.

Final Questions

a) From this project, I learned many lessons about writing large programs. Firstly, I learned about how important proper preparation is prior to actually coding out the program. If I hadn't fully thought out my program's design and realized midway through that I had a critical design error that meant the program couldn't function properly, by then I would've wasted a lot of time and already written enough code to the point where editing the entire thing for a different design pattern would take a lot of work. Moreover, I learned that high level overview thinking of how the program will function and how each aspect of the game will interact with each other is a good mindset to have when building out a large program. This way, I can make sure that each component of my program is properly interacting with each other and ensures that I focus on the bigger picture and make sure my program functions properly instead of being stuck on small details. Furthermore, I learned how important good organization skills and proper file management is. Sometimes, I would forget which files I needed to include and where to include them, which ties back to having clean code and making sure that I know what files I have and how they’re sorted. Overall, my experience from creating this large program gave me a different perspective on how software is developed and the mindset I should have when coding.

b) If I had the chance to start over, there are a few things that I would do differently. Firstly, I would try to use unique pointers from the start. Given that I attempted assignment 5 and had somewhat a grasp on how to use unique pointers, I think that if I began using them from the beginning, it would have been easier for me to implement them into my program. Also, I would focus more on the small details when I’m planning out my program. I realized that when I was in the act of coding out my solution, there were certain small details of the program that I did not consider too much before starting. If I could do it over, I would make sure to make my plan more thorough to take into account the small details that are key to the program functioning properly.

Conclusion

Overall, I had a lot of fun making this project and learned a lot along the way. I believe that my program does a good job of including the essential OOP topics, along with ensuring high cohesion and low coupling. I also think that my program is quite flexible in the case that any changes to the way the game functions is needed, and is built to the required specifications.