Ashna Jagadisan (ID#: 20824887)

Design Document

CS 246 Final Project – Straights

Overview

My implementation of the Straights game consists of two primary design patterns: The Model View Controller (MVC) design pattern and the Strategy design pattern. As the program makes use of an MVC design pattern, it consists of 3 core components: the model, the view, and the controller. Note that the `Table` acts as the model of the MVC and controls the state of the game. To do this, the `Table` has several constituent parts.

Each table owns a `Deck` and four `SuitPiles`. The `Deck` keeps the current state of the deck of `Card`s used in the game and handles shuffling while each `SuitPile` keeps track of the `Card`s in the pile (namely the top and bottom cards). The `Deck` has 53 `Card`s and each `SuitPile` can have up to 13 `Card`s. Each `Card` also has a face and suit.

Moreover, the `Table` has four `Player`s by the default rules defined in the project specification,, however this number can be modified in the `Table`. A `Player` can either be a `Human` or `Computer`, each with a hand of cards, a discard pile, and a collection of legal plays they can make. Each `Computer` player has a `Strategy` they use to make a play as well. To implement the various possible play strategies for `Computer` players, the Strategy design pattern was used. A `Strategy` can be either an `EasyStrategy`, the `BasicStrategy`, or a `HardStrategy`. The `Table` stores the `Difficulty` of the default strategy to be used by `Computer Player`s at any point in the game.

As part of the MVC design pattern, the model and the view use the Observer design pattern where the model is the subject, and the view is the observer. As such, the `Table` is the `Subject` which can have `Observers` and the `TextView` and `View` are `Observers`. To implement the GUI, the `View` uses the GTK+ library and has a `DeckGUI` which contains the graphical representation of the `Card`s.

In order to translate interactions with the view to the model, the `TextView` and `View` have a `Controller` which can access the `Table`.

Design

Since the face and suit of cards have an inherent order, they were implemented with the enumerated classes `Suits` and `Faces`. This allows for iteration through suits and faces in a straightforward way. Similarly, the `Difficulty` level of `Computer` player strategies was also implemented with an enumerated class.

One design challenge involved the inability to use the `std::find` algorithm with Card objects due to the lack of an equality operator defined for the class. To solve this, operator overloading was used for `operator==`. Note that as `operator==` is symmetric, the overloading was implemented as a non-member function. Thus, a `Card` object is a plain-old data type and thereby implemented as a struct.

Another design challenge occurred when deciding how to seed the random number generator to shuffle the `Deck`. Initially, I had been re-seeding the random number generator every time the `Deck` was shuffled; however, the generator should only be seeded once. To solve this, I stored the generator as a data field in the `Deck` and initialized it with a seed specified when a `Deck` object is created. Thus, the generator is only seeded once in the `Deck` constructor and can then be called multiple times (via `Deck::shuffle()`) without being re-seeded, as intended.

The largest design challenges I faced in this project occurred when implementing the `Player` class. Initially, I had planned to simply have a `Player` class with a single `Computer` child class where the parent class has a `virtual` method to be overridden in the child class. However, I failed to realize that by having a pure virtual method, the `Player` class becomes abstract and cannot be instantiated. To solve this, I created another derived class for `Human` players. Yet, I was then faced with another issue as the `Human` derived class must implement the pure virtual method as well in order to not be classified as an abstract class. However, the `Player::makePlay()` pure virtual method should only be called for `Computer` players, not `Human` players. To solve this, I added a Boolean data field in the `Player` class, `isComp`, that allows determining whether the `Player` is a `Computer`. As such, the invariant can be enforced by only calling `makePlay()` after checking that a `Player` is a `Computer`.

Another design challenge I faced when implementing the `Player` class involved transferring `Human` player data to `Computer` players. To solve this, I made a shared pointer to a Computer player that is created with the information of a Human player. Specifically, I wrote the constructor for the `Computer` class such that it takes a `std::shared_ptr<Player>`, which is passed from the `Table`. Then in the `Computer` constructor, I used the `Player` constructor to initialize a `Player` with the inputted player's fields (except for `isComp`, which is set to `true` for `Computer` players). Then, this `std::shared_ptr<Computer>` is stored in place of the old `Human` player in the `Table`. Note that as the `Table` stores a vector of shared pointers to `Player`s, shared pointers to both `Human` and `Computer` objects can be stored due to inheritance. This differs from my initial plan to `cast` `Player`s to new `Computer` players. However, I did use `dynamic_pointer_cast`s to set the strategy field of existing `Computer` players based on the state of the game to implement an enhancement (see the `setDynamicCompDifficulty` method in `table.cc`).

Another large design challenge I faced occurred when trying to structure classes to allow for different types of play strategies for `Computer` players. Initially, I had thought to use derived classes under the `Computer` class for the different types of computer players, however I found the strategy design pattern to be a better solution. To implement the design pattern, I used the `Computer` class as the Context class. This class has a `Strategy` abstract base class who's derived classes contain overridden `play()` methods that are called based on the type of `Strategy` the `Computer` player has. One challenge I faced when trying to implement the strategy design pattern was figuring out how to connect the `Computer` player to the derived strategy classes. To solve this, I passed the `Computer` player (using `this`) to the `Strategy::play()` method from `Computer::makePlay()`. As a result, the overridden `play()` methods could access a pointer to the Computer object and use it's player data to make the appropriate move.

Upon implementing the above components, the remainder of the program (apart from the GUI) was relatively straightforward. One challenge I faced when implementing the `TextView` and `Controller`

was figuring out when to notify the view from `Table`. Initially, I had evidently used the `notify()` method incorrectly as the final game message was printed several times before the program ended. To solve this, I precisely evaluated exactly when I would need to update the view. I then decided to only `notify` the view in the `Table::nextPlayer()` method for any of the following three cases; the round ends, a new round begins, or it is the next player with a *non-empty hand*'s turn.

Another challenge I faced was determining whether the `Computer` player played or discarded a card from the `TextView` and `Table`. To solve this, I checked whether the `Computer` player has any legal plays after the either played or discarded card is obtained. Note that since the `legalPlays` of each player are updated only once at the start of each turn, they would not yet be modified to reflect the current `Computer` player's move. Thus, if the `Computer` has legal plays, the appropriate 'play' message is output in TextView and the Controller would tell the Table to add the played card to the appropriate suit pile. Otherwise, the 'discard' message is output.

When first implementing the GUI, I faced an issue regarding how to store the various widgets in the window. Initially, I had started to create a separate widget for each button and card, however this became very tedious given the large number of widgets displayed at any given time. Moreover, it would be difficult to update each widget individually every time the `update()` method is called. To solve this, I used vectors of pointers to widgets. This allows for easily iterating through and updating related widgets. For instance, when it is a new player's turn, one can simply iterate through the `handButtons` vector to update the display and show the new player's hand. Note that the `Gtk::manage()` method is used to properly manage the lifetimes of these pointers.

Resilience to Change

Throughout the planning and implementation of my program, I largely focused on enforcing resilience to change. To easily facilitate changes to the program specification, I planned my program to have low coupling and high cohesion. For instance, most classes follow the single responsibility principle; rather than the entire game model being implemented in the `Table` class, it is spit into several constituent parts. The `Deck` class implements creating and shuffling of the deck, the `SuitPile` class deals with checking and adding to a suit pile, the `Player` class handles player actions, and the `Strategy` classes each implement a single play strategy. Moreover, simply by utilizing the MVC design pattern the single responsibility principle is inherently applied; on top of `Table` handling the game state, `TextView` and `View` each handle a presentation state while the Controller handles the control logic. By designing classes in this way, changes to any one module can be made without modifying the others, thereby allowing for easily changing various aspects of the program.

Thus, my design supports the possibility of various changes to the program specification. For instance, if the program specification were to change the `gameEndScore` from 80 to 100, this can be specified in the `Table` constructor (for which `gameEndScore = 80` is a default parameter). Similarly, if the specification were to require a different number of players, starting card, or default computer strategy, these could easily be set in the Table constructor as well. This is because I implemented several data fields in the Table class that allow for customization such as `gameEndScore`, `numOfPlayers`, `startingCard`, and `defaultStrategy`. On top of that, if the specification were to change the input syntax, this would only require minor modifications to the `TextView`/`View` class.

Ashna Jagadisan (ID#: 20824887)

Answers to Questions

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

To ensure changing the interface or game rules would have as little impact on the code as possible, the Model View Controller (MVC) design pattern should be used to structure the game classes. This is because the MVC design pattern separates the program state (model), presentation state (view), and control logic (controller).

As a result, changing the interface of the game would be simply require changing the view. Similarly, changing the game rules would only require changes to the program state, while the rest of the program stays as is.

The classes in my implementation of this program fit this framework with the `Table`, `Controller`, and `TextView` classes. The `Table` class is the model as it manages the state of the game; the class handles all `Player`, `Deck`, and `SuitPile` data as well as their modifications. The `Controller` class is the controller in the MVC design pattern as it manages interactions with the view to modify data. Based on user input, `Controller` maps user actions from the to actions that take place in the model, or `Table`. The `TextView` class is the view as it manages the interface to present the data; any data that must be output to the user is handled by the `TextView` class.

Note that as part of the MVC design pattern, the Observer design pattern is also at play. The `Table` and `TextView` implement the Observer design pattern where the `Table` is the Subject and the `TextView` is the Observer.

If one were to change the game rules, only the `Table` class must be modified. For instance, to change the score required for the game to end, `Table::gameEndScore` would be changed from the originally specified value of 80.

Furthermore, if one were to change the interface of the game, the `TextView` class would need to be changed. For example, to implement a GUI, a `View` class can be created. The user can specify that they would like to play with a graphical interface via a command line flag, which the `main` function will process and create a `View` object for the view rather than a `TextView` object.

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 (dynamically during the play of the game). How would that affect your structure?

To allow computer players alongside human players, inheritance can be used; A general `Player` base class could have `Human` and `Computer` derived class where the role of the `Computer` class is to implement all computer player functionality.

To allow computer players with differing play strategies that may change as the game progresses, the Strategy design pattern can be used where the `Computer` class is the context class. The `Computer` class will have a `Strategy` on which it can call the `Strategy::play(Computer *c)` method. Note that the `Strategy` class is an abstract base class, so the overridden method in the derived strategy class corresponding to the `Computer` player's strategy will be run. For instance, suppose a `Computer` player uses an `EasyStrategy`. Then, when `Player::makePlay()` is called by the controller, `EasyStrategy::play(Computer *c)` will end up being called. Thus, allowing different types of `Computer` players with different play strategies.

If the strategies of `Computer` players were to change as the game progresses, `Computer` player strategies can be handled in the `Table` class (which manages the state of the game). Then, based on certain aspects of the current game state, the `Table` can modify the strategy of a `Computer` player using an mutator method. Note that the MVC design pattern is still maintained when making these changes as all changes occur in the model module.

The class structure in my implementation uses inheritance for `Computer` players as well as the strategy design pattern for `Computer` player strategies and can thereby implement these changes.

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?

In order to easily transfer the information associated with a human player to a computer player, inheritance can be used. The classes can be structured such that an abstract `Player` class has derived `Human` and `Computer` classes. Moreover, the `Computer` constructor should take a `Player shared_ptr` which is can then use to initialize a new `Player` in the member initialization list. Then, a `Table::newCompPlayer()` method can create a `shared_ptr` to a `Computer` player using the information from the current `Human` player.

Moreover, a type check field can be implemented in the `Player` class to allow the `Table` and `Controller` to know which methods to call based on the type of player. In my implementation, the `Player` class has a Boolean field, `Player::isComp`, that is true if and only if the `Player` is a `Computer` player. This field can then be accessed by the model and the view using accessor methods.

For instance, if a player were to rage quit and be replaced with a computer player, the `Table` would create a `shared_ptr` to a `Computer` object initialized with the data from the current player and the `Player::isComp` field set to true. The `Table` could then replace the current player with this new `Computer` player. This change occurs in the `Table::newCompPlayer()` method of my implementation.

Extra Credit Features

For this project, I implemented 3 main extra credit features: house rules, smarter computer players, and a graphical user interface.

House Rules

The user has the option of playing with the house rules by specifying the command-line argument `-houserules`. When playing with house rules, the game ending score is 45 instead of 80, the number of players is 2 instead of 4, and the starting card is a king of hearts rather than a seven of spades.

This feature was implemented by creating the `gameEndScore`, `numOfPlayers`, and `startingCard` data fields in the `Table` class which can then be initialized to the above house rules when creating the `Table` in `main`. Note that default values aligning with the original program description are specified for these parameters in the `Table` constructor.

Smarter Computer Players

This feature was implemented in two different ways: different types of computer players and dynamic computer players.

The user has the option of specifying the computer player difficulty with command-line arguments `-easy` or `-hard`, which signify computer players with an easy and hard difficulty, respectively. If the user specifies the easy difficulty, the computer players will utilize a "play low, discard high" strategy in which the lowest legal play is made or, if there are no legal plays, the highest card in the player's hand is discarded. As scores are calculated using player discards, this will result in the computer players accumulating a higher score much faster, thus making them easier opponents.

If the user specifies the hard difficulty, the computer players will utilize a "play high, discard low" strategy in which the highest legal play is made or, if there are no legal plays, the lowest card in the player's hand is discarded. This will result in the computer players accumulating a higher score much slower, thus making them more difficult opponents.

The challenge faced when implementing this extra credit feature was figuring out how to structure classes to implement different types of Computer players. As explained in the design section, a Strategy design pattern was used to solve this. Thus, when the controller calls `Player::makePlay()`, which will call the defined method in the Computer class, a call to `Strategy::play(Computer *c)` will be made. This will in turn run one of the overridden functions in either the `EasyStrategy`, `BasicStrategy`, or `HardStrategy` class, depending on the strategy of the player (which is specified as a data field in the `Computer` class).

The user also has the option of playing with *dynamic* computer players by using the command-line argument -dynamic. Dynamic computers are `Computer` players with play strategies that change dynamically during the play of the game. More specifically, if a `Computer` player has more than 9 cards in their hand, they will use the `EasyStrategy` and if they have less than 5 cards in their hand, they will use the `HardStrategy`. Otherwise, the `BasicStrategy` outlined in the program specification

will be used. This will cause for Computer players with increasing difficulty levels as each round progresses.

To implement this feature, a `setDynamicCompDifficulty` method as well as a Boolean `dynamicStrategy` data field were created in the `Table` class. The method takes a shared pointer to a Player and modifies its strategy field using `dynamic_pointer_cast`ing according to the number of cards in its hand if it is a `Computer` player and `dynamicStrategy` is true. This function is called when a `newRound()` starts, a `newCompPlayer()` is added, and in `nextPlayer()`. Moreover, when the `-dynamic` argument is used, `dynamicStrategy` is initialized to true when the `Table` is created in `main`.

Graphical User Interface

The user can play the game with a graphical user interface rather than the text user interface outlined in the program specification by using the command-line argument `-gui`.

As I had implemented the rest of the program using the MVC design pattern, implementing the GUI involved creating another view. Note that alongside the `View` class, the `DeckGUI` class is also used to display the card images. Although this makes the process of implementing the GUI very straightforward as it logically follows from the `TextView` and does not require changes to any other modules, I still found this extra feature to be the most difficult of the three to implement. This is largely because I implemented the GUI using the GTK+ graphics library, which I have no experience using. Moreover, my experience with implementing GUIs in general is very limited.

The majority of the time spent implementing the GUI was in the `View` constructor. Once I was more familiar with how to use the GTK+ library, implementing the remaining methods was quite straightforward as it was similar to the implementation of `TextView`. A more specific challenge I faced when implementing this feature was getting the message dialog boxes to close every time the "OK" button is pressed (as they would sometimes stay open). To solve this, I created a `View::showDialog(string message)` method which initializes a message dialog box, sets the message, sets the default response, and runs the box. As a result, the dialog box goes out of scope and is thereby closed when the response is received and the method is exited.

I predominantly designed the GUI based on the sample executable provided however, I made some changes for my implementation. For instance, I found that one feature missing from the sample text UI in the sample GUI was displaying the legal plays for the current player. Thus, I implemented this in my GUI by making any cards in the current player's hand that are legal plays highlighted in green. This was done by individually editing each card image and setting up a second array of png files to the images in the `DeckGUI` class (these images can be found in the legal_img folder). I then modified the `View::update()` method such that for any legal cards in the players hand, the image displayed on the GUI is that of the second array.

Note that when testing the code without the GUI implementation (that is, without the GTK+ library, the `View` class, and the `DeckGUI` class), valgrind reports no memory leaks or errors and that all heap blocks have been freed. However, valgrind reports 'potentially lost' and 'still reachable' memory with the GUI implementation. These leaks are strictly due to the graphics library and not the program code, in which all memory is cleaned up.

Ashna Jagadisan (ID#: 20824887)

Final Questions

What lessons did you learn about writing large programs?

First and foremost, I learned the importance of planning programs prior to writing them. I spent a significant time planning my program before writing any code which I feel helped a great deal in terms of gaining clarity of what each module is supposed to do and how they work together. In particular, I found that creating a UML diagram is very useful as I frequently referenced it while writing the code. Although my implementation deviated slightly from the initial UML diagram I created, it was still very helpful in the initial stages. I also wrote snippets of pseudocode for any method or class implementations I was unsure of. This was also very helpful and frequently referenced when writing code for the implementation.

Moreover, I learned the importance of having clear goals in mind during every step of the implementation process. For instance, I knew I wanted to implement a GUI for my program and allow easy adaptability to any changes in the project specification from the beginning. In doing so, I subconsciously wrote classes to align with these goals which resulted in an easier implementation process for these extra features.

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

If I were able to start over, I would focus more on the correctness of my plan for the project. While I did already spend a significant time on planning the program, I was not fully confident in certain aspects of my plan prior to writing the code. For instance, I was not fully certain on whether my solution to introducing new and different types of Computer players would work before I started coding. This resulted in a large time sink during the implementation of the program as my initial plan would not work. As such, I was counterintuitively looking through design patterns for a solution in the middle of coding. If I were to further analyse my plan prior to coding, I could have potentially caught flaws such as this, which would greatly improve the ease of implementing the project.

Furthermore, I would also consider using proper encapsulation from the start of the planning step. As a result of not focusing on encapsulation, I ended up making all class fields private and writing accessors and mutators at the end of my implementation. This was quite tedious as I had to filter through all the files and change any references to class fields with accessors or mutators. Had I been more strategic with my planning, this could have been avoided altogether.