

# Design Document

# **Straights**

---

Raghav Vishnoi

14th December, 2020

## Introduction

In this project, I have implemented a text-based game called Straights. The rules of the game are the exact same as the ones provided in the Straights document provided by the instructors. To achieve this goal the key features of the course I have used are the following:

- Makefile and compilation
- Preprocessor guards
- Objects, classes (abstract and concrete) and other OOP principles
- Memory Management
- Type Casting
- Simple IO and parsing
- Software Engineering principles
- UML
- Design patterns , mainly the observer pattern

## Overview

This Overview includes elements of the gameplay:

Straights is a game. The game has a deck and a gameboard. The game also has 4 players who are either a computer player or a human player.

- The game starts with inviting players and asks them if they are human or computer players.
- Then the game is setup (the deck and gameboard are prepared)
- Then the cards are shuffled
- After properly shuffling the cards are dealt to the players
- Then the order of the game is decided based on who gets the 7 of Spades upon shuffling.
- Then the true round begins. Since each player has 13 cards each player gets 13 turns.
- The gameplay for a computer is different from the human. Humans can enter the choices as to what they want to do, but the computer is bound to specific actions as described in the Straights document.
- Once the round is completed the individual scores of players are calculated and the winner is displayed.

- The game is packed up and the players leave the game. If they want to play again they restart the game.

## Human Gameplay

These are the following things a Human can do when it's his turn:

These are the specific commands the player needs to use:

play <card>	Plays the appropriate card in his hand if its a legal play. If not he is provoked to choose again.
discard <card>	Discards the card . ie if he has no legal plays. This card goes to his legal pile that will later be used to calculate score in the end
“deck “	(Note: when this command is used the space is included). This reveals the deck of cards after it has been shuffled. This is used for debugging purposes
“quit “	(Note: when this command is used the space is included). This terminates the entire game.

<card> is a string like 7S or 9D etc.

## Computer Gameplay

The computer gameplay is simple.

When it's the computers turn, he plays the first legal option he has. If he has no legal option, then he discards the first card in his hand.

## Design and Implementation

This section takes you through the implementation of the game and the designs used in the implementation

## Class Descriptions and dependencies:

- The Game is implemented as a master class (concrete). It has two compositions , the deck and the gameboard. The Game class is responsible for creating and destroying these compositions. The Game also has an aggregate , namely the Player class. The players can exist without the game, so the game class is not responsible for creating or destroying the players.
- The Player, which is an aggregate of the Game is an abstract class. It's a parent and it has two children that inherit from it. The children are Human and Computer.
- We also have Subject and Observers. Each player is both a subject and an observer. Additionally the gameboard is both a subject and observer. When the player is a subject, it observes the gameboard and when the gameboard is the subject , it observes the players playing the game. (how the observer pattern is used and implemented will be discussed later .)

## Classes and Functions for reference:

These are brief descriptions provided to understand the implementation below

### Game:

make_game()	Sets up the game, attaches observers
<code>shuffle(std::vector&lt;std::string&gt; &amp; cards, std::default_random_engine rng )</code>	Shuffles the deck of cards
deal()	Deals the cards to players
decide_order()	Decides the order the game based on who gets the 7 of spades
decide_winner()	Calculates score and displays winner

### Deck:

print_deck()	Prints the deck of cards. This can be used to check if the deck is shuffled
--------------	---

### Gameboard:

notify () overridden	Uses this function to inform other plays that its state is changed
get_info()	Returns the card recently played .

### Player:

player_choice_card()	Displays the current cards in the hand, discarded cards, legal cards and the cards on the table
score()	Calculates the score of the player based on his discard pile
rearrange()	This function is used to rearrange cards in the legal list to help play implementation
valid()	Tells us if the card played is a valid option
play(game)	This actually portrays the playing of a card on the table.

The human and the computer class inherit most functions from its parent player.

### Implementation: (Very general)

- First the 4 players are declared . Then we move to step 1: inviting the players. In this part the user is questioned whether he wants to be a computer player or human player. Based on that we use casting to point to the child. Memory is allocated on the heap for these objects. (Example : `Player *p1= new human()`)
- Then using the 4 players the game is initialized (using the game constructor)
- After the game is initialized we call the function `make_game()` to attach all observers in the game. The board attaches players and each player attaches the board to it.

- After doing that we use the code given in shuffle.cc and shuffle the deck of cards. This is done by function `shuffle(std::vector<std::string> & cards, std::default_random_engine rng)` . Note that this function uses the *reference* to the deck of cards. This is so that when another round starts we have the shuffled deck of cards to start with. This part ensures that section 5.3 of the straights document is properly implemented. Note that additional argument to ./straights can be passed -> the seed that makes shuffling more random and can be used by the testers efficiently.
- Once the deck of cards are shuffled, we deal the cards to the players using `deal()` in the game class.
- After dealing the cards the function `decide_order()` is called to decide in which order the players are going to take turns. This depends on which player got the 7 of Spades after shuffling. This order is stored in a list called `PlayerList` in `Game` that will be used below.
- Now the game play starts. (As in the rounds). We have a list of players that portrays the order in which they are going to play. Since each player has 13 cards , there will be 13 steps in the game. In each of the steps 4 players will take turns to play their card.
- For each player we check if he is a human or computer since the way they play differ. If it's a computer we call `play(game)` which is overridden in the child classes. Before the play and after the play I display the players card to ensure the play has been made correctly. If the player is a human, I take user input and store the input in strings. Based on the input I implement the step. For example, if the play types "deck " the deck is shown etc.
- After 13 steps in the game, each player no more has any cards and so I use the game class' function called `decide_winner()` to compute each player's individual score and display it to the user.

## Implementation (Specific)

**This section only includes some specifics that would be required by the user**

- Deck is a class with an attribute called `card_deck`. This is a vector of strings that stores 52 strings representing cards. The reason a vector container was used in this was because : the vector can easily be traversed.
- A player has many vectors as attributes: `player_hand`, `discard_list`, `legal_list`, `club_list`, `diamond_list`, `spade_list`, `heart_list`. All these vectors are vectors of strings. Vectors were used here because its easy to add elements using `emplace_back`, its easy to remove elements , find size of list, check if they are empty etc

**This section is where I explain key elements of the game play (Includes observer design pattern).**

- When it's a player's turn , we first check the type of the player, whether he is a computer player or human player.
- If it's a computer player, we call the `play` method of the computer player. This method checks if the legal list is empty, if yes discards the first element in the player hand. If not , then he notifies the gameboard (his observer) that I am playing this card, change how the game board looks. When the gameboard is notified of this move, it in turn notifies its observers (other players) that a card has been played , change your state (legal list) if necessary.
- A similar thing happens if its a human player, except a human can now choose what he wants to do. As in, user input can be taken in.
- The key here is the observer pattern. The way I use it, matches real world gameplay . The player plays the card, the board is notified of the move and changes how it looks. When that happens it notifies other players so they can think about their legal options. Perhaps its over generalised.

## Resilience to Change:

To be really honest, this program supports a lot of changes in specifications but there are situations that will be highlighted that depict failure to do so.

- Even if the way the game is scored changes, that will only affect the function `score()` in the player class. Since the human and computer class inherit from this class, not a lot of change will be made.
- The `play()` functions overridden in the computer and human class respectively include the element of the rules of the game. If the rules were to change a small tweak in those polymorphic functions would do the job.
- One element of failure I would like to admit to is that if a player wanted to change from human to computer player- my program would not easily be able to do that due to heap allocation. I would have to keep track of the delete and new again and again to make it possible and avoid leakage of memory.


My response to this failure would be to use shared pointers instead.

## 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?

- I have taken the advice of a mentor Joseph Istead who taught me a little bit of how softwares are developed in teams. He recommended me to focus on the smallest element of the project, test it thoroughly and build on that. This really helped me because I am not used to thinking this way. Usually for assignments and psets, I think of the whole solution, code it and spend hours scrambling, debugging and testing. This time I broke the project into the smallest bits, and accomplished tasks, tested them and built features on top. This gave me ease and comfort and allowed me to finish before time.





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

- I initially did not take advice and spent two days thinking about the perfect design. But i realise now that any plausible design could work and I could build on that,
- Additionally, I think I could spend more time encapsulating data from the client and also handle errors and invalid input. This would make my game more robust and user friendly.