Overview

The project has 8 classes, which includes 1 enum class and 1 abstract class. The enum class is called *Suit*, which enumerates the different types of suits present in a deck of playing cards, namely spades, hearts, clubs, and diamonds. The next class is *Card*, whose objects will represent each of the fifty-two cards of a deck. The relationship between *Card* and *Suit* is that of composition, i.e. *Card* is composed of *Suit*. The next and final class in this tier is *Deck*, whose objects will be the four piles of different suits in the game, the main deck of 52 cards and three hands (namely, hand, legal plays, and discards) of each of four players. The relationship between *Deck* and *Card* is that of aggregation, i.e. *Deck* is aggregated by *Card*. A *Deck* object can contain anywhere from zero to fifty-two cards.

In the next tier, we have four classes. Firstly, we implement the **strategy design pattern**. The *Strategy* abstract class forms a relationship of aggregation with the *Player* class. A *Player* object has exactly one *Strategy* object associated with it. The *Strategy* sub-classes are: *Human* and *Simple_computer*. The object of these sub-classes will determine who will play the game (human or computer). These sub-classes are a specialization of *Strategy* class. A *Player* class will make objects which will play the game. The relationship between *Deck* and *Player* is that of aggregation, i.e. *Player* is aggregated by *Deck*. A *Player* object can contain anywhere from zero to three deck objects.

In the final tier, we have only one class, *Game*. A *Game* object starts and plays and finishes the game. *Game* has a relationship of aggregation with *Deck* and *Player*. A G*ame* object contains 5 *Deck* objects and 4 *Player* objects.

You can see the overall structure of my classes in my final uml PDF.

Design

We will be going through each .h files. Firstly we will see the members defined and then talk about any specific techniques used in it.
*'None' means that there is nothing, not even void.

card.h

This interface file contains two classes, namely *Suit* and *Card*.

The first class, *Suit*, is an enum class with the following possibilities: spades, hearts, diamonds, and clubs.

The second class, *Card*, has the following member variables:

Type	Name	Description
Char	value	Stores the value of the card (eg. A, 2, 10, J, etc)
Suit	suit	Stores the suit of the card

Card has the following member functions:

Return type	Function name	Argument type	Description
None	Card	Char, Suit	constructor
char	getValue	none	Accessor function for value
char	getSuit	none	Returns the first character of the suit
int	give_value	none	Returns the value of the card
void	printCard	none	Prints the card (eg. KS, 7S)

There are no special techniques used in the implementation of this interface file. For card number 10 of any suit, the *value* member variable is set to ':' since ':' - '0' is 10, but wherever the card is print, 10 is outputted with the respective suit.

deck.h

The class, *Deck*, has the following member variables:

Type	Name	Description
std::vector<std::shared_ptr<Card>>	deck	Is a vector of shared pointers to card objects. Stores the cards in the deck object.

Deck has the following member functions:

Return type	Function name	Argument type	Description
void	showDeck	none	Displays the cards in the deck to the standard output
bool	empty	none	Return true if deck does not have any cards, else false
void	remove	none	Removes any duplicate elements(cards) from the vector *deck*
int	deck_size	none	Returns the number of cards in the deck
void	clear_deck	none	Removes all the cards from the deck
void	showCard	none	Displays the cards in the deck without the suit to the standard output
void	insertCard	std::shared_ptr<Card>	Inserts the passed card at the back of the deck
void	insertFront	std::shared_ptr<Card>	Inserts the passed card at the front of the deck
void	shuffle	String = “”	Shuffles the deck according to given seed
std::shared_ptr<Card>	out	int i	Returns the pointer to the card at position i in the deck
std::shared_ptr<Card>	find_card	char, char	Returns the pointer to the card whose value and suit was passed as parameters if found in deck, else nullptr
void	card_remove	std::shared_ptr<Card>	Removes the passed card from the deck

Since the STL container vector is used, most of these are wrapper functions for manipulating the vector by using functions defined for vector STL. Member functions *shuffle* and *remove* use functions such as *std::shuffle* and *std::remove* defined in the *algorithm* library. Functions *card_remove, showDeck, remove, clear_deck, showCard, and find_card* uses the iterator design pattern to traverse in the vector.

strategy.h

This interface file contains an abstract class, *Strategy*. It does not have any member variable. It has the following member functions:

Return type	Function name	Argument type	Description
None	~Strategy	none	destructor
std::shared_ptr<Card>	play	std::shared_ptr<Deck>	Is a pure virtual method
std::shared_ptr<Card>	discard	std::shared_ptr<Deck>	Is a pure virtual method

human.h

This interface contains the class *Human*, which inherits from *Strategy*. It has no member variables. The following are its' member functions :

Return type	Function name	Argument type	Description
std::shared_ptr<Card>	play_discard (**private**)	std::shared_ptr<Deck>	Returns the card searched in the passed deck if found, else nullptr
std::shared_ptr<Card>	play	std::shared_ptr<Deck>	Calls play_discard with the passed deck
std::shared_ptr<Card>	discard	std::shared_ptr<Deck>	Calls play_discard with the passed deck

Both functions *play* and *discard* calls *play_discards*. The only difference between the two is that the passed deck are different, i.e. *play* is passed the *legal_plays* deck of a player, whereas *discard* is passed the *hand* deck of the player. In *play_discard*, the function calls the *find_card* function on the deck passed with the value passed from standard input, and returns accordingly.

simplecomputer.h

This interface contains the class *Human*, which inherits from *Strategy*. It has no member variables. The following are its' member functions :

Return type	Function name	Argument type	Description
std::shared_ptr<Card>	play_discard (**private**)	std::shared_ptr<Deck>	Returns the first card in the passed deck if found, else nullptr
std::shared_ptr<Card>	play	std::shared_ptr<Deck>	Calls play_discard with the passed deck
std::shared_ptr<Card>	discard	std::shared_ptr<Deck>	Calls play_discard with the passed deck

Both functions *play* and *discard* calls *play_discards*. The only difference between the two is that the passed deck are different, i.e. *play* is passed the *legal_plays* deck of a player, whereas *discard* is passed the *hand* deck of the player. In *play_discard*, the function calls the *out* function on the deck with argument 0 to return the card at the beginning of the deck.

player.h

The *Player* class has the following member variables:

Type	Name	Description
int	total = 0	Stores total score(sum of value of discarded cards) of each player
int	id	Stores the id number of player (1,2,3,4)
char	player	Stores the type of player (h or c)

std::shared_ptr<Deck>	hand	Is the deck which contains the cards in the players' hand at the moment
std::shared_ptr<Deck>	legal_plays	Is the deck which contains the cards of the player which are legal moves according to the current state of game
std::shared_ptr<Deck>	discards	Is the deck which contains the cards which the player has discarded. Clears after every round .
std::shared_ptr<Strategy>	strategy_	Is the pointer to the strategy object of the player.

Player has the following member functions:

Return type	Function name	Argument type	Description
void	set_strategy (**private**)	std::shared_ptr<Strategy>	Changes the strategy_ of the player to the passed argument
none	Player	int, char	Constructor
void	new_total (**private**)	none	Reevaluate the total score of the player
int	getTotal	Bool = false	Returns the total score of the player
void	clear_deck	std::shared_ptr<Card>	Fills the *hand* of the player with cards after shuffling
void	showHand	none	Displays the cards in the *hand* deck to the standard output
void	showLegalHands	none	Displays the cards in the *legal_plays* deck to the standard output
int	getID	none	Returns ID
bool	seven_of_spades	none	Return true if 7S present in the *hand*
void	rage_quit	none	Changes strategy from human to computer
void	clear_legal_plays	None	Clear *legal_plays*
char	getPlayer	None	Returns player type
shared_ptr<Card>	play	String = ""	Plays the game
bool	has_no_cards	None	Returns false if *hand* is not empty
void	make_legal_plays	shared_ptr<Deck>	Makes legal plays accrding to the passed deck

The *Game* has the following functions:

Return type	Function name	Description
bool	play	Changes the strategy_ of the player to the passed argument
void	shuffle	Constructor
void	playerGeneration	Reevaluate the total score of the player
void	gameDeckGeneration	Returns the total score of the player
bool	check_score	Fills the *hand* of the player with cards after shuffling
bool	next_round	Displays the cards in the *hand* deck to the standard output
void	make_legal_hands	Displays the cards in the *legal_plays* deck to the standard output
void	showGameDeck	Returns ID
void	handMaker	Return true if 7S present in the *hand*
void	tell_winners	Outputs winner

Resilience to Change

Any new specification to the program can be implemented in the "game.cc" file as a new function and then put in the *play* function of the same class at the appropiate place.

If we want to implement a more aggressive computer player, we can simple make a new concrete class of the *Strategy* class and change the Player constructor to set the strategy_ pointer to an object of this new class if specified at any point in time of the game.

Answer to Questions

The class designed implemented in my project (can be seen in the UML) has only one place where code has to be slightly changed and/or just added more to adjust any changes to interface and rules. That class is *Game*. Since the game is played in the *Game* object, all the other classes are unaffected by any change in its' code, only the *game.cc* will be recompiled.

I used the ***strategy design pattern*** to accommodate human and computer players. For dynamically changing the strategies of the computer player during the game, we would have to add concrete classes to *Strategy*, add a function in player which changes strategies by checking its' total and then changing strategy_ to whatever we wanted to according to the situation.

Due to the ***strategy design pattern***, we just need to call *rage_quit*, and the strategy_ ptr of the player leaves pointing at a *Human* object and starts pointing to the computer strategy object. So there was no transfer of information; just the strategy was changed.

Extra Credit Features

I have completed the entire project, without leaks, and without explicitly managing the memory. I handled all memory management via STL containers and smart pointers; and hence, I have no *delete* statements in my project and no raw pointers too. They were challenging because of the segmentation faults which made the memory leaked. It was specially

difficult because sometimes the program would run without any fault, but sometimes it would crash. I debugged it using valgrind but more importantly, gdb.

Final Questions

This project taught me how to manage time while writing big programs and to take my time to come back and debug the programs. Even though I started late, I learnt how to plan the whole interface by myself and complete it on time.

I would have used more exception handling if I get to start over. I believe my code can be made more robust than it already is.