Министерство образования и науки Российской Федерации Федеральное государственное автономное образовательное

учреждение высшего образования

«ЮЖНЫЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»

(ФГАОУ ВО «ЮФУ»)

Институт компьютерных технологий и информационной безопасности

ОТЧЕТ ПО ЛАБАРАТОРНОЙ РАБОТЕ  
по дисциплине «Объектно ориентированное программирование»

**ДИНАМИЧЕСКИЙ ВЫБОР ТИПА ОБЪЕКТОВ**

Выполнил  
студент группы КТбо2-1 И.А. Быстриченко

Принял

Таганрог 2020

СОДЕРЖАНИЕ

[1 Задание 3](#_Toc22220)

[2 Спецификация классов с комментариями 4](#_Toc24965)

[2.1 Класс Card 4](#_Toc32078)

[2.2 Класс Deck 4](#_Toc19973)

[2.3 Класс Player 4](#_Toc10206)

[2.4 Классы RandomCardPlayer, BiggestCardPlayer и SmallestCardPlayer 5](#_Toc19910)

[2.5 Класс GameTable 5](#_Toc15207)

[2.6 Класс ConsoleInteractor 6](#_Toc10091)

[2.7 Класс DeckLoader 6](#_Toc3029)

[2.8 Класс RandomGenerator 6](#_Toc20384)

[3 Используемые математические зависимости 7](#_Toc12759)

[4 Диаграмма классов 8](#_Toc16478)

[5 Листинг программы 9](#_Toc31605)

1. Задание

Определить классы Карта (Card) и Колода\_карт

(Deck). Поля первого – масть (suit) и достоинство (rank).

Методы класса возвращают масть и достоинство. Второй

включает массив (32) объектов первого. Методы класса:

− перемешивания колоды;

− сравнения 2-х карт по достоинству при условии, что масти

одинаковы;

− создания 4-х мест и раздачи равного количества карт;

− моделирования упрощенного розыгрыша взятки: на стол

выкладываются по одной карте от каждого из 4-х игроков;

первая выложенная карта определяет масть; выигрывает

карта, старшая по достоинству (картинки старше простых

карт; козырной масти нет).

Список карт для инициализации программы хранить в файле.

Требуется разработать механизм виртуальных функций использующийся для создания программы, в которой выбор типа игрока производится во время выполнения программы по запросу пользователя. Спецификация классов с комментариями

Программа включает в себя 10 классов

* 1. Класс Card

Служит для определения карт в игровой колоде и колодах игроков.

Содержит поля \_rank(достоинство) и \_suit(масть), а так же методы для получения значений этих полей.

* 1. Класс Deck

Класс представляет из себя колоду карт с методами для работы с колодой.

В полях класса содержится указатель на динамический массив объектов класса Card, \_deckSize - размер текущей колоды карт, а так же \_bufSize - размер выделенного буфера для хранения карт. Кроме того класс содержит поле \_lastTakedCard для того, чтобы иметь возможность получить последнюю вытащенную карту из колоды.

В конструкторе класса выделяется память дефолтного размера колоды, который может увеличиваться по мере добавления карт.

Класс имеет 3 публичных метода и 1 protected. К публичным относятся AddCard, TakeCard и DeleteCard. То, что они делают, можно понять из названия.

Когда метод AddCard пытается добавить карту при достижении конца буфера, вызывается приватный метод IncreaseBufferSize который создает новый массив удвоенной длины и копирует в него старый, после чего освобождает память, выделенную под старый массив.

* 1. Класс Player

Класс представляет из себя абстрактную модель игрока. Игрок, в понимании программы - колода, умеющая доставать карты по особым правилам и имеющая имя. Именно поэтому этот класс наследуется от класса Deck.

К унаследованным полям и методам класса Deck добавляется поле \_name для хранения имени игрока, а так же метод GetName.

Кроме того в классе содержатся 2 абстрактных метода TakeGameCard и TakeGameCardBySuit. Именно эти 2 метода будут определять стиль игры игрока в классах наследниках.

* 1. Классы RandomCardPlayer, BiggestCardPlayer и SmallestCardPlayer

Классы RandomCardPlayer, BiggestCardPlayer и SmallestCardPlayer наследуются от класса Player и определяют его абстрактные методы.

Игрок RandomCardPlayer если есть выбор из нескольких карт, вытащит рандомную, BiggestCardPlayer вытащит карту с наибольшим достоинством, SmallestCardPlayer с наименьшим достоинством.

* 1. Класс GameTable

Класс GameTable служит для обеспечения игрового процесса.

У класса в полях имеется указатель на колоду, массив указателей на игроков и количество игроков.

Также класс содержит 8 публичных и 1 приватный метод.

Кпубличным относится функции:

ShuffleDeck перемешивающая колоду карт. Перемешивание колоды описано в разделе “Используемые математические зависимости”

Статические функции CompareCards и CompareCardsRanks о предназначении которых можно понять из названий. CompareCards выбрасывает ошибку при сравнении карт разных мастей.

Функция GiveCards раздающая карты из колоды игрокам

Функции GetDeck, GetPlayer и GetNumberOfPlayers для получения значений полей

Функция Play вызывающая приватную функцию DoPlay, если у игроков есть карты. В противним случае, она выбрасывает исключение.

* 1. Класс ConsoleInteractor

Класс ConsoleInteractor - основной класс взаимодействия с пользователем.

По сути, класс состоит из одного основного публичного метода Start и набора вспомогательных приватных методов для обеспечения работы метода Start.

Метод Start создает игровой стол и запускает бесконечный цикл взаимодействия с пользователем.

* 1. Класс DeckLoader

Класс DeckLoader предназначен для ввода колоды карт с файла, для чего имеет единственный статический метод LoadFromFile.

* 1. Класс RandomGenerator

Класс RandomGenerator предоставляет удобную функцию генерации рандомного числа в указанных пределах.

Класс используется в классах RandomCardPlayer и GameTable

1. Используемые математические зависимости

Перемешивание колоды в классе gameTable осуществляется следующим способом:

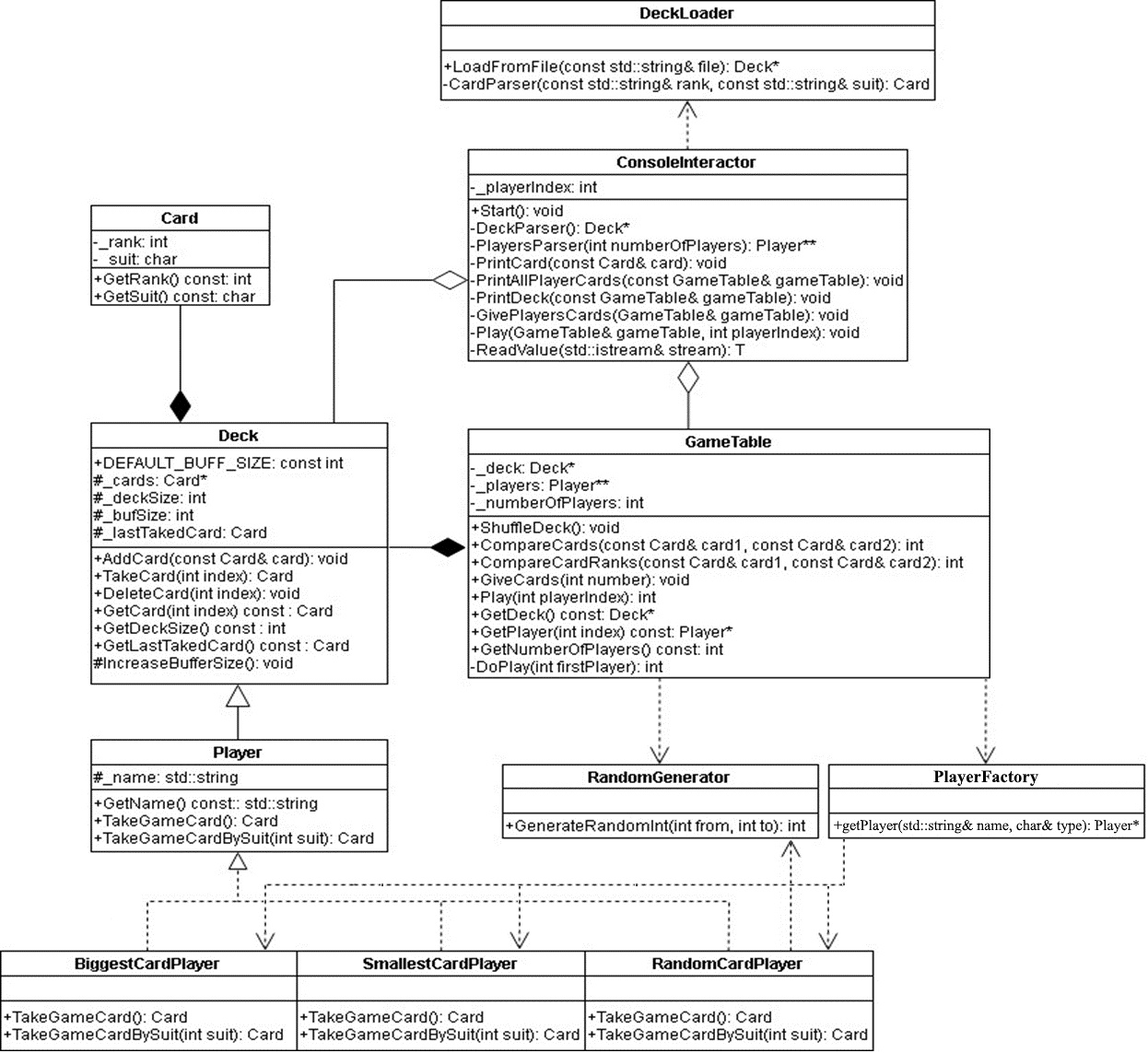
Допустим у нас колода длины N.

Тогда на первом шаге из колоды карт достается рандомная карта в диапазоне [0, N-1] и ставится в конец колоды.

Теперь колода состоит из N-1 неперемешанных карт и одной рандомно полученной на предыдущем шаге.

Повторяем предыдущие шаги вытаскивая на n-ом шаге рандомную карту в диапазоне [0, N-n] до тех пор, пока часть карт, вытащенных рандомно не станет по размеру равной N-1, и получаем перемешанную колоду карт

1. Диаграмма классов



1. Листинг программы

**main.cpp**

#include "ConsoleInteractor.h"

int main()

{

ConsoleInteractor consoleInteractor;

consoleInteractor.Start();

return 0;

}

**Card.h**

#pragma once

class Card

{

public:

Card(int rank = 0, char suit = 0) : \_rank(rank), \_suit(suit) {}

int GetRank() const { return \_rank; }

char GetSuit() const { return \_suit; }

private:

int \_rank;

char \_suit;

};

**Deck.h**

#pragma once

#include <stdexcept>

#include "Card.h"

class Deck

{

public:

const int DEFAULT\_BUFF\_SIZE = 36;

Deck()

{

\_cards = new Card[DEFAULT\_BUFF\_SIZE];

}

~Deck()

{

if (\_cards != nullptr)

{

delete[] \_cards;

}

}

void AddCard(const Card& card);

Card TakeCard(int index);

void DeleteCard(int index);

Card GetCard(int index) const { return \_cards[index]; }

int GetDeckSize() const { return \_deckSize; }

Card GetLastTakedCard() const { return \_lastTakedCard; }

protected:

Card\* \_cards = nullptr;

int \_deckSize = 0;

int \_bufSize = DEFAULT\_BUFF\_SIZE;

Card \_lastTakedCard;

void IncreaseBufferSize();

};

**Deck.cpp**

#include "Deck.h"

void Deck::AddCard(const Card& card)

{

if (\_deckSize == \_bufSize)

{

IncreaseBufferSize();

}

\_cards[\_deckSize] = card;

\_deckSize++;

}

Card Deck::TakeCard(int index)

{

\_lastTakedCard = \_cards[index];

DeleteCard(index);

return \_lastTakedCard;

}

void Deck::DeleteCard(int index)

{

for (int i = index; i < \_deckSize; i++)

{

\_cards[i] = \_cards[i + 1];

}

\_deckSize--;

}

void Deck::IncreaseBufferSize()

{

\_bufSize \*= 2;

Card\* temp = new Card[\_bufSize];

for (int i = 0; i < \_deckSize; i++)

{

temp[i] = \_cards[i];

}

delete[] \_cards;

\_cards = temp;

}

**PlayerFactory.h**

#pragma once

#include "RandomCardPlayer.h"

#include "SmallestCardPlayer.h"

#include "BiggestCardPlayer.h"

class PlayerFactory

{

public:

static Player\* getPlayer(const std::string& name, const char& type)

{

Player\* player;

switch (type)

{

case 'r': player = new RandomCardPlayer(name); break;

case 'b': player = new BiggestCardPlayer(name); break;

case 's': player = new SmallestCardPlayer(name); break;

default: throw std::logic\_error("Invalid input. Try again!\nFormat: \"Name type\"");

}

return player;

}

};

**Player.h**

#pragma once

#include "Deck.h"

class Player

{

public:

Player(const std::string& name) : \_name(name) { \_deck = new Deck(); }

std::string GetName() const { return \_name; }

Deck\* GetDeck() { return \_deck; }

Card TakeCard(int index) { return \_deck->TakeCard(index); }

void AddCard(const Card& card) { \_deck->AddCard(card); }

virtual Card TakeGameCard() = 0;

virtual Card TakeGameCardBySuit(int suit) = 0;

protected:

std::string \_name;

Deck\* \_deck;

};

**RandomCardPlayer.h**

#pragma once

#include "Deck.h"

#include "Player.h"

class RandomCardPlayer : public Player

{

public:

RandomCardPlayer(const std::string& name) : Player(name) {}

Card TakeGameCard();

Card TakeGameCardBySuit(int suit);

};

**RandomCardPlayer.cpp**

#include "RandomCardPlayer.h"

#include "RandomGenerator.h"

Card RandomCardPlayer::TakeGameCard()

{

return TakeCard(RandomGenerator::GenerareRandomInt(0, \_deck->GetDeckSize()));

}

Card RandomCardPlayer::TakeGameCardBySuit(int suit)

{

int count = 0;

for (int i = 0; i < \_deck->GetDeckSize(); i++)

{

if (\_deck->GetCard(i).GetSuit() == suit)

{

count++;

}

}

if(count == 0)

{

return TakeGameCard();

}

else

{

int rand = RandomGenerator::GenerareRandomInt(1, count);

int index = -1;

for (int i = 0; i < \_deck->GetDeckSize() && rand > 0; i++)

{

if (\_deck->GetCard(i).GetSuit() == suit)

{

rand--;

if (rand == 0)

{

index = i;

}

}

}

return TakeCard(index);

}

}

**BiggestCardPlayer.h**

#pragma once

#include "Player.h"

class BiggestCardPlayer : public Player

{

public:

BiggestCardPlayer(const std::string& name) : Player(name) {}

Card TakeGameCard();

Card TakeGameCardBySuit(int suit);

};

**BiggestCardPlayer.cpp**

#include "BiggestCardPlayer.h"

Card BiggestCardPlayer::TakeGameCard()

{

int index = 0;

for (int i = 1; i < \_deck->GetDeckSize(); i++)

{

if (\_deck->GetCard(i).GetRank() > \_deck->GetCard(index).GetRank())

{

index = i;

}

}

return TakeCard(index);

}

Card BiggestCardPlayer::TakeGameCardBySuit(int suit)

{

int index = -1;

for (int i = 0; i < \_deck->GetDeckSize(); i++)

{

if (\_deck->GetCard(i).GetSuit() == suit && (index == -1 || \_deck->GetCard(i).GetRank() > \_deck->GetCard(index).GetRank()))

{

index = i;

}

}

if (index == -1)

{

return TakeGameCard();

}

else

{

return TakeCard(index);

}

}

**SmallestCardPlayer.h**

#pragma once

#include "Player.h"

class SmallestCardPlayer : public Player

{

public:

SmallestCardPlayer(const std::string& name) : Player(name) {}

Card TakeGameCard();

Card TakeGameCardBySuit(int suit);

};

**SmallestCardPlayer.cpp**

#include "SmallestCardPlayer.h"

Card SmallestCardPlayer::TakeGameCard()

{

int index = 0;

for (int i = 1; i < \_deck->GetDeckSize(); i++)

{

if (\_deck->GetCard(i).GetRank() < \_deck->GetCard(index).GetRank())

{

index = i;

}

}

return TakeCard(index);

}

Card SmallestCardPlayer::TakeGameCardBySuit(int suit)

{

int index = -1;

for (int i = 0; i < \_deck->GetDeckSize(); i++)

{

if (\_deck->GetCard(i).GetSuit() == suit && (index == -1 || \_deck->GetCard(i).GetRank() < \_deck->GetCard(index).GetRank()))

{

index = i;

}

}

if (index == -1)

{

return TakeGameCard();

}

else

{

return TakeCard(index);

}

}

**GameTable.h**

#pragma once

#include "Deck.h"

#include "Player.h"

class GameTable

{

public:

GameTable(Player\*\* players, int numberOfPlayers, Deck\* deck) : \_players(players), \_numberOfPlayers(numberOfPlayers), \_deck(deck) {}

~GameTable();

void ShuffleDeck();

static int CompareCards(const Card& card1, const Card& card2);

static int CompareCardRanks(const Card& card1, const Card& card2) { return card1.GetRank() - card2.GetRank(); };

void GiveCards(int number);

int Play(int playerIndex);

Deck\* GetDeck() const { return \_deck; }

Player\* GetPlayer(int index) const { return \_players[index]; }

int GetNumberOfPlayers() const { return \_numberOfPlayers; }

private:

Deck\* \_deck;

Player\*\* \_players;

int \_numberOfPlayers;

int DoPlay(int firstPlayer);

};

**GameTable.cpp**

#include "GameTable.h"

#include "DeckLoader.h"

#include "RandomGenerator.h"

GameTable::~GameTable()

{

delete \_deck;

for (int i = 0; i < \_numberOfPlayers; i++)

{

delete \_players[i];

}

delete[] \_players;

}

void GameTable::ShuffleDeck()

{

for (int i = \_deck->GetDeckSize() - 1; i > 0; i--)

{

\_deck->AddCard(\_deck->TakeCard(RandomGenerator::GenerareRandomInt(0, i)));

}

}

int GameTable::CompareCards(const Card& card1, const Card& card2)

{

if (card1.GetSuit() == card2.GetSuit())

{

return CompareCardRanks(card1, card2);

}

else

{

throw std::logic\_error("Ńomparison of cards of different suits");

}

}

void GameTable::GiveCards(int number)

{

if (number < 0 || number > \_deck->GetDeckSize() / \_numberOfPlayers)

{

throw std::exception("Wrong number of cards!");

}

for (int i = 0; i < number \* \_numberOfPlayers; i++)

{

\_players[i / number]->AddCard(\_deck->TakeCard(0));

}

}

int GameTable::Play(int playerIndex)

{

if (\_players[0]->GetDeck()->GetDeckSize() == 0)

{

throw std::logic\_error("Players don't have cards");

}

else

{

return DoPlay(playerIndex);

}

}

int GameTable::DoPlay(int firstPlayer)

{

Card biggestCard = \_players[firstPlayer]->TakeGameCard();

char suit = biggestCard.GetSuit();

int winnerIndex = firstPlayer;

for (int i = 0; i < \_numberOfPlayers; i++)

{

if (i == firstPlayer)

{

continue;

}

Card playerCard = \_players[i]->TakeGameCardBySuit(suit);

if (playerCard.GetSuit() == suit && CompareCards(playerCard, biggestCard) > 0)

{

biggestCard = playerCard;

winnerIndex = i;

}

}

return winnerIndex;

}

**RandomGenerator.h**

#pragma once

#include <random>

class RandomGenerator

{

public:

static int GenerareRandomInt(int from, int to)

{

std::random\_device rd;

std::mt19937 gen(rd());

std::uniform\_int\_distribution<> dist(from, to);

return dist(gen);

}

};

**ConsoleInteractor.h**

#pragma once

#include "Card.h"

#include "Player.h"

#include "GameTable.h"

class ConsoleInteractor

{

public:

void Start();

private:

int \_playerIndex = 0;

static GameTable CreateGameTable();

static Deck\* DeckParser();

static Player\*\* PlayersParser(int numberOfPlayers);

static void PrintCard(const Card& card);

static void PrintAllPlayerCards(const GameTable& gameTable);

static void PrintDeck(const GameTable& gameTable);

static void GivePlayersCards(GameTable& gameTable);

static void Play(GameTable& gameTable, int playerIndex);

template < typename T >

static T ReadValue(std::istream& stream);

};

**ConsoleInteractor.cpp**

#include "ConsoleInteractor.h"

#include <sstream>

#include <iostream>

#include "BiggestCardPlayer.h"

#include "GameTable.h"

#include "DeckLoader.h"

#include "RandomCardPlayer.h"

#include "SmallestCardPlayer.h"

#include "PlayerFactory.h"

void ConsoleInteractor::Start()

{

GameTable gameTable = CreateGameTable();

std::cout <<

"\nCommands:\n"

"1 - Shuffle deck\n"

"2 - Print deck\n"

"3 - Print player cards\n"

"4 - Give players cards\n"

"5 - Play\n";

while (true)

{

try

{

std::cout << "Enter next command:\n>> ";

int temp = ReadValue<int>(std::cin);

switch (temp)

{

case 1: gameTable.ShuffleDeck(); break;

case 2: PrintDeck(gameTable); break;

case 3: PrintAllPlayerCards(gameTable); break;

case 4: GivePlayersCards(gameTable); break;

case 5: Play(gameTable, \_playerIndex); \_playerIndex = (\_playerIndex + 1) % gameTable.GetNumberOfPlayers(); break;

default: throw std::logic\_error("Illegal command input");

}

std::cout << std::endl;

}

catch (std::exception err)

{

std::cout << err.what() << std::endl << std::endl;

}

}

}

GameTable ConsoleInteractor::CreateGameTable()

{

Deck\* deck = DeckParser();

int numberOfPlayers = -1;

while (numberOfPlayers <= 0)

{

try

{

std::cout << "\nEnter the number of players:\n>> ";

numberOfPlayers = ReadValue<int>(std::cin);

if (numberOfPlayers <= 0)

{

throw std::logic\_error("For game you should have 1 or more players!");

}

}

catch (std::exception err)

{

std::cout << err.what() << std::endl;

}

}

Player\*\* players = PlayersParser(numberOfPlayers);

return GameTable(players, numberOfPlayers, deck);

}

Deck\* ConsoleInteractor::DeckParser()

{

Deck\* deck = nullptr;

while (deck == nullptr)

{

try

{

std::string deckFile;

std::cout << "Enter the deck file:\n>> ";

std::cin >> deckFile;

deck = DeckLoader::LoadFromFile(deckFile);

}

catch (std::exception err)

{

std::cout << err.what() << std::endl << std::endl;

}

}

return deck;

}

Player\*\* ConsoleInteractor::PlayersParser(int numberOfPlayers)

{

Player\*\* players = new Player \* [numberOfPlayers];

std::cout << "\nEnter players \"Name type\"" << std::endl;

std::cout << "The type of player determines his way of choosing the card to play\n"

"Types:\n"

"r - Random card\n"

"b - Biggest card\n"

"s - Smallest card\n";

int i = 0;

while (i < numberOfPlayers)

{

std::string name;

char type;

std::cout << ">> ";

try

{

name = ReadValue<std::string>(std::cin);

type = ReadValue<char>(std::cin);

players[i] = PlayerFactory::getPlayer(name, type);

i++;

}

catch (std::exception err)

{

std::cout << err.what() << std::endl << std::endl;

}

}

return players;

}

void ConsoleInteractor::PrintCard(const Card& card)

{

if (card.GetRank() >= 2 && card.GetRank() <= 10)

{

std::cout << card.GetRank() << " " << card.GetSuit() << std::endl;

}

else if(card.GetRank() >= 11 && card.GetRank() <= 14)

{

switch (card.GetRank())

{

case 11: std::cout << 'J'; break;

case 12: std::cout << 'Q'; break;

case 13: std::cout << 'K'; break;

case 14: std::cout << 'A'; break;

}

std::cout << " " << card.GetSuit() << std::endl;

}

}

void ConsoleInteractor::PrintAllPlayerCards(const GameTable& gameTable)

{

for(int i = 0; i < gameTable.GetNumberOfPlayers(); i++)

{

std::cout << gameTable.GetPlayer(i)->GetName() << ":" << std::endl;

for(int j = 0; j < gameTable.GetPlayer(i)->GetDeck()->GetDeckSize(); j++)

{

PrintCard(gameTable.GetPlayer(i)->GetDeck()->GetCard(j));

}

}

}

void ConsoleInteractor::PrintDeck(const GameTable& gameTable)

{

Deck\* deck = gameTable.GetDeck();

if (deck->GetDeckSize() == 0)

{

throw std::logic\_error("Deck is empty!");

}

for(int i = 0; i < deck->GetDeckSize(); i++)

{

PrintCard(deck->GetCard(i));

}

}

void ConsoleInteractor::GivePlayersCards(GameTable& gameTable)

{

std::cout << "Enter number of cards: ";

int number = ReadValue<int>(std::cin);

gameTable.GiveCards(number);

}

void ConsoleInteractor::Play(GameTable& gameTable, int playerIndex)

{

int winner = gameTable.Play(playerIndex);

std::cout << "Player " << gameTable.GetPlayer(playerIndex)->GetName() << " moves first." << std::endl;

std::cout << "Player moves:" << std::endl;

int i = playerIndex;

do

{

std::cout << gameTable.GetPlayer(i)->GetName() << ": ";

PrintCard(gameTable.GetPlayer(i)->GetDeck()->GetLastTakedCard());

i = (i + 1) % gameTable.GetNumberOfPlayers();

}

while (i != playerIndex);

std::cout << "Winner: " << gameTable.GetPlayer(winner)->GetName() << std::endl;

}

template <typename T>

T ConsoleInteractor::ReadValue(std::istream& stream)

{

T res;

std::string input;

stream >> input;

std::istringstream sin(input);

if (!(sin >> res))

{

throw std::logic\_error("Illegal value input");

}

char c;

if (sin >> c)

{

throw std::logic\_error("Garbage after value input");

}

return res;

}

**DeckLoader.h**

#pragma once

#include <string>

#include "Deck.h"

class DeckLoader

{

public:

static Deck\* LoadFromFile(const std::string& file);

private:

static Card CardParser(const std::string& rank, const std::string& suit);

};

**DeckLoader.cpp**

#include "DeckLoader.h"

#include "Deck.h"

#include <fstream>

#include <iostream>

Deck\* DeckLoader::LoadFromFile(const std::string& file)

{

std::fstream fin;

fin.open(file);

Deck\* deck = new Deck();

if (fin.is\_open())

{

while(!fin.eof())

{

std::string rank;

std::string suit;

if (fin >> rank >> suit)

{

Card temp = CardParser(rank, suit);

deck->AddCard(temp);

}

}

}

else

{

throw std::runtime\_error("File not found exception");

}

return deck;

}

Card DeckLoader::CardParser(const std::string& rank, const std::string& suit)

{

int cardRank = -1;

char cardSuit = -1;

if(rank == "10")

{

cardRank = 10;

}

else if(rank.length() == 1)

{

if(rank[0] >= '2' && rank[0] <= '9')

{

cardRank = rank[0] - '0';

}

else

{

switch (rank[0])

{

case 'J': cardRank = 11; break;

case 'Q': cardRank = 12; break;

case 'K': cardRank = 13; break;

case 'A': cardRank = 14; break;

}

}

}

if(suit.length() == 1 && suit[0] >= '1' && suit[0] <= '4')

{

cardSuit = suit[0];

}

if(cardRank == -1 || cardSuit == -1)

{

throw std::runtime\_error("Card parser exception");

}

return Card(cardRank, cardSuit);

}