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»

**КОНТЕЙНЕРЫ STL**

Выполнил  
студент группы КТбо2-1 И.А. Быстриченко

Принял

Таганрог 2020

СОДЕРЖАНИЕ

[1 Задание 3](#_Toc60086398)

[1.1 Класс Card 3](#_Toc60086399)

[1.2 Класс Deck 3](#_Toc60086400)

[1.3 Класс Player 4](#_Toc60086401)

[1.4 Классы RandomCardPlayer, BiggestCardPlayer и SmallestCardPlayer 4](#_Toc60086402)

[1.5 Класс GameTable 5](#_Toc60086403)

[1.6 Класс ConsoleInteractor 5](#_Toc60086404)

[1.7 Класс DeckLoader 6](#_Toc60086405)

[1.8 Класс RandomGenerator 6](#_Toc60086406)

[1.9 Класс PlayerFactory 6](#_Toc60086407)

[1.10 Класс IdentifierGenerator 6](#_Toc60086408)

[2 Используемые математические зависимости 7](#_Toc60086409)

[3 Диаграмма классов 8](#_Toc60086410)

[4 Листинг программы 9](#_Toc60086411)

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 использовать контейнеры STL для работы с колодой и хранения игроков. Реализовать список игроков, отсортированный по id и по количеству побед игрока.

* 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 содержатся вектор карт и копия последней взятой карты.

Класс имеет 6 публичных методов AddCard, TakeCard, DeleteCard, GetCard, GetDeckSize и GetLastTakedCard. То, что они делают, можно понять из названия.

* 1. Класс Player

Класс представляет из себя абстрактную модель игрока.

Класс содержит 3 поля: \_name, \_deck и \_winnings. То, что в них хранится понятно из названия.

Также в классе содержатся 2 абстрактных метода TakeGameCard и TakeGameCardBySuit и 7 публичных GetName, GetDeck, TakeCard, AddCard, GetWinnings, IncWinnings, DecWinnings. 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 рангом, SmallestCardPlayer с наименьшим рангом.

* 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.

К публичным относится функции:

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

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

Функция GiveCards, раздающая карты из колоды игрокам.

Функция GetNumberOfPlayers и GetPlayersData 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. Класс PlayerFactory

Фабрика классов Player

* 1. Класс IdentifierGenerator

Генератор идентификаторов игроков

1. Используемые математические зависимости

Перемешивание колоды в классе gameTable осуществляется следующим способом:

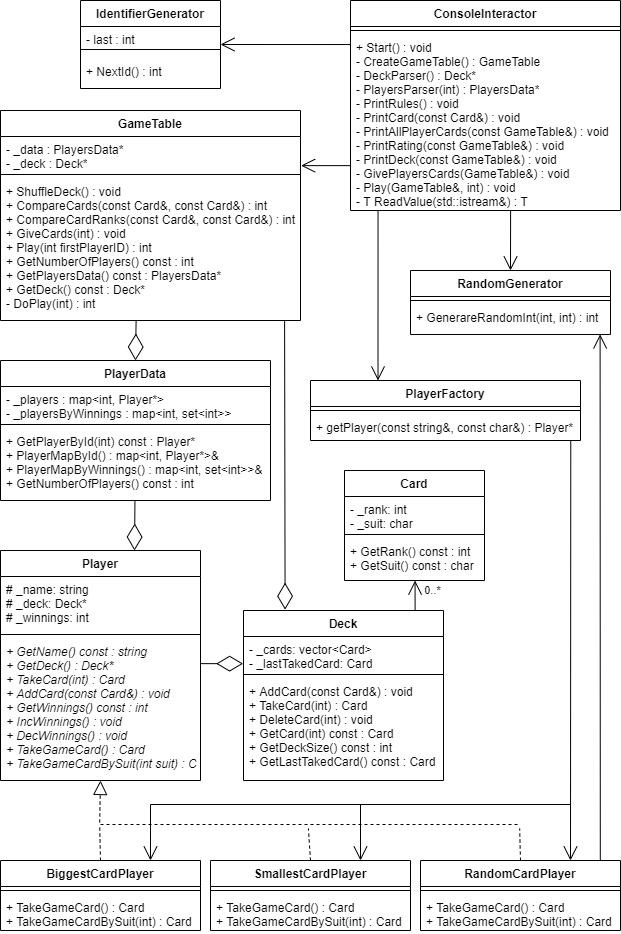
Допустим у нас колода длины N.

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"

#include <vld.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 <vector>

#include "Card.h"

class Deck

{

public:

void AddCard(const Card& card) { \_cards.push\_back(card); }

Card TakeCard(int index);

void DeleteCard(int index);

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

int GetDeckSize() const { return \_cards.size(); }

Card GetLastTakedCard() const { return \_lastTakedCard; }

private:

std::vector<Card> \_cards;

Card \_lastTakedCard;

};

**Deck.cpp**

#include "Deck.h"

Card Deck::TakeCard(int index)

{

\_lastTakedCard = \_cards[index];

DeleteCard(index);

return \_lastTakedCard;

}

void Deck::DeleteCard(int index)

{

for (int i = index; i < \_cards.size() - 1; i++)

{

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

}

\_cards.pop\_back();

}

**PlayerFactory.h**

#pragma once

#include "RandomCardPlayer.h"

#include "SmallestCardPlayer.h"

#include "BiggestCardPlayer.h"

#include <stdexcept>

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"

#include <string>

class Player

{

public:

Player(const std::string& name) : \_name(name), \_winnings(0) { \_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); }

int GetWinnings() const { return \_winnings; }

void IncWinnings() { \_winnings++; }

void DecWinnings() { \_winnings--; }

virtual Card TakeGameCard() = 0;

virtual Card TakeGameCardBySuit(int suit) = 0;

protected:

std::string \_name;

Deck\* \_deck;

int \_winnings;

};

**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() - 1));

}

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"

#include "PlayersData.h"

class GameTable

{

public:

GameTable(PlayersData\* data, Deck\* deck) : \_data(data), \_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 firstPlayerID);

int GetNumberOfPlayers() const { return \_data->GetNumberOfPlayers(); }

PlayersData\* GetPlayersData() const { return \_data; }

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

private:

PlayersData\* \_data;

Deck\* \_deck;

int DoPlay(int firstPlayerID);

};

**GameTable.cpp**

#include "GameTable.h"

#include "DeckLoader.h"

#include "RandomGenerator.h"

#include <map>

#include <set>

#include <stdexcept>

#include <iostream>

GameTable::~GameTable()

{

delete \_deck;

delete \_data;

}

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("Сomparison of cards of different suits");

}

}

void GameTable::GiveCards(int number)

{

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

{

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

}

std::map<int, Player\*> map = \_data->PlayerMapById();

for (auto i = map.begin(); i != map.end(); i++)

{

for (int j = 0; j < number; j++)

{

i->second->AddCard(\_deck->TakeCard(0));

}

}

}

int GameTable::Play(int firstPlayerID)

{

Player\* player = \_data->GetPlayerById(firstPlayerID);

if (player->GetDeck()->GetDeckSize() == 0)

{

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

}

else

{

return DoPlay(firstPlayerID);

}

}

int GameTable::DoPlay(int firstPlayerID)

{

Card biggestCard = \_data->GetPlayerById(firstPlayerID)->TakeGameCard();

char suit = biggestCard.GetSuit();

std::map<int, Player\*>& map = \_data->PlayerMapById();

std::map<int, Player\*>::iterator winner = map.find(firstPlayerID);

for (auto i = map.begin(); i != map.end(); i++)

{

if (i == map.find(firstPlayerID))

{

continue;

}

Card playerCard = i->second->TakeGameCardBySuit(suit);

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

{

biggestCard = playerCard;

winner = i;

}

}

std::map<int, std::set<int>>& wmap = \_data->PlayerMapByWinnings();

wmap[winner->second->GetWinnings()].erase(winner->first);

winner->second->IncWinnings();

wmap[winner->second->GetWinnings()].insert(winner->first);

return winner->first;

}

**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 "GameTable.h"

class ConsoleInteractor

{

public:

void Start();

private:

static GameTable CreateGameTable();

static Deck\* DeckParser();

static PlayersData\* PlayersParser(int numberOfPlayers);

static void PrintRules();

static void PrintCard(const Card& card);

static void PrintAllPlayerCards(const GameTable& gameTable);

static void PrintRating(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"

#include "IdentifierGenerator.h"

#include <map>

#include <set>

enum Comands

{

rules, shuffle, pdeck, pplayercards, prating, givecards, play

};

void ConsoleInteractor::Start()

{

GameTable gameTable = CreateGameTable();

PrintRules();

auto firstPlayer = gameTable.GetPlayersData()->PlayerMapById().begin();

while (true)

{

try

{

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

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

switch (temp)

{

case rules: PrintRules(); break;

case shuffle: gameTable.ShuffleDeck(); break;

case pdeck: PrintDeck(gameTable); break;

case pplayercards: PrintAllPlayerCards(gameTable); break;

case prating: PrintRating(gameTable); break;

case givecards: GivePlayersCards(gameTable); break;

case play: Play(gameTable, firstPlayer->first);

firstPlayer++;

if (firstPlayer == gameTable.GetPlayersData()->PlayerMapById().end())

{

firstPlayer = gameTable.GetPlayersData()->PlayerMapById().begin();

}

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;

}

}

PlayersData\* data = PlayersParser(numberOfPlayers);

return GameTable(data, 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;

}

PlayersData\* ConsoleInteractor::PlayersParser(int numberOfPlayers)

{

std::map<int, Player\*> players;

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;

IdentifierGenerator gen;

while (i < numberOfPlayers)

{

std::string name;

char type;

std::cout << ">> ";

try

{

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

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

players.insert(std::pair<int, Player\*>(gen.NextId(), PlayerFactory::getPlayer(name, type)));

i++;

}

catch (std::exception err)

{

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

}

}

return new PlayersData(players);

}

void ConsoleInteractor::PrintRules()

{

std::cout <<

"\nCommands:\n" <<

rules << " - Print rules\n" <<

shuffle << " - Shuffle deck\n" <<

pdeck << " - Print deck\n" <<

pplayercards << " - Print player cards\n" <<

prating << " - Print leaderboard\n" <<

givecards << " - Give player cards\n" <<

play << " - Play\n";

}

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)

{

std::map<int, Player\*>& map = gameTable.GetPlayersData()->PlayerMapById();

for(auto i = map.begin(); i != map.end(); i++)

{

std::cout << i->second->GetName() << ":" << std::endl;

for(int j = 0; j < i->second->GetDeck()->GetDeckSize(); j++)

{

PrintCard(i->second->GetDeck()->GetCard(j));

}

}

}

void ConsoleInteractor::PrintRating(const GameTable& gameTable)

{

std::map<int, std::set<int>>& wmap = gameTable.GetPlayersData()->PlayerMapByWinnings();

for (auto i = wmap.rbegin(); i != wmap.rend(); i++)

{

std::set<int> list = i->second;

for (auto j = list.begin(); j != list.end(); j++)

{

std::cout << gameTable.GetPlayersData()->GetPlayerById(\*j)->GetName() << ":" << i->first << std::endl;

}

}

}

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)

{

PlayersData\* data = gameTable.GetPlayersData();

int winner = gameTable.Play(playerIndex);

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

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

auto i = data->PlayerMapById().find(playerIndex);

do

{

std::cout << i->second->GetName() << ": ";

PrintCard(i->second->GetDeck()->GetLastTakedCard());

i++;

if (i == data->PlayerMapById().end())

{

i = data->PlayerMapById().begin();

}

}

while (i->first != playerIndex);

std::cout << "Winner: " << data->GetPlayerById(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);

}

**IdentifierGenerator.h**

#pragma once

class IdentifierGenerator

{

public:

IdentifierGenerator() : last(1) {}

int NextId() { return last++; }

private:

int last;

};

**PlayersData.h**

#pragma once

#include <map>;

#include <set>;

#include <list>

#include "Player.h"

class PlayersData

{

public:

PlayersData(const std::map<int, Player\*>& players) : \_players(players)

{

\_playersByWinnings.insert(std::pair<int, std::set<int>>(0, std::set<int>()));

for (auto i = \_players.begin(); i != \_players.end(); i++)

{

\_playersByWinnings[0].insert(i->first);

}

};

Player\* GetPlayerById(int id) const;

std::map<int, Player\*>& PlayerMapById() { return \_players; }

std::map<int, std::set<int>>& PlayerMapByWinnings() { return \_playersByWinnings; }

int GetNumberOfPlayers() const { return \_players.size(); }

private:

std::map<int, Player\*> \_players;

std::map<int, std::set<int>> \_playersByWinnings;

};

**PlayersData.cpp**

#include "PlayersData.h"

Player\* PlayersData::GetPlayerById(int id) const

{

auto it = \_players.find(id);

if (it == \_players.end())

{

throw std::exception("Wrong player id");

}

return it->second;

}