САНКТ-ПЕТЕРБУРГСКИЙ ПОЛИТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ им. ПЕТРА ВЕЛИКОГО
Институт компьютерных наук и технологий
Высшая школа искусственного интеллекта

Отчет по курсовой работе

Игра в дурака

Работу выполнил:
Алехичев А. В.
группа 3530201/10001

Проверил:
Курочкин Л. М.

Санкт-Петербург - 2022 г.

Содержание

1 Постановка задачи

1. Разработать алгоритм, который может играть в дурака в качестве игрока по заданным правилам
   - Правила находятся в `/doc/rules.jpg` или по ссылке
2. Реализовать алгоритм на языке C++, в качестве класса `Player`.
   - Для работы с колодой использовать статическую библиотеку `/dealer.lib`.
   - Сопроводительные файлы к лабораторной и библиотека **dealer** доступны по ссылкам: WIN32 или Linux64
3. Подготовить отчёт по работе, соответствующий требованиям
   - Требования находятся в `/doc/kr_req.docx` или по ссылке
   - Задокументировать разработанный алгоритм и написанный по нему код

2 Описание исходных данных задачи

2.1 Алгоритм игры в дурака

Данный алгоритм определяет возможные ходы для двух игроков, то как они взаимодействуют и различные состояния игры. Алгоритм представлен в виде блок-схемы, а также в файле `/main.cpp`, распространяющемся в одном zip-архиве с библиотекой dealer. Блок-схему можно посмотреть в приложении 1

2.2 Статическая библиотека **dealer**

Дана статическая библиотека **dealer**. Эта библиотека отвечает за контроль колоды и стола. К этому относится: хранение, перетасовка колоды, получение информации о колоде, взятие карты из колоды. А также получение информации о козырной масти и столе.

3 Определение терминов

3.1 Ранг карты

Свойство карты.

Ранг карты выражается с помощью целого числа — типа `int`.

С помощью метода `Dealer::RankIndex` можно узнать ранг карты.

Ранг обычной карты можно использовать как индекс в массиве `ranks`, чтобы получить имя ранга.

Ранг особой карты используется как обозначение состояния особых карт.

Данные состояния — «Пас» и «Без карты» выражаются числами 300 и 400 соответственно.
Эти числа указаны в коде как `PAS` и `NOCARD`.

Ранги обычной карты можно сравнить. Величина ранга увеличивается в порядке:
`2 < 3 < 4 < 5 < 6 < 7 < 8 < 9 < 10 < Jack < Queen < King < Ace`.

3.2 Масть карты

Свойство карты.

Масть карты выражается с помощью целого числа — типа `int`.

С помощью метода `Dealer::SuitIndex` можно узнать масть карты.

Масть обычной карты можно использовать как индекс в массиве `suits`, чтобы получить имя масти.

Масть обычной карты можно использовать как индекс в массиве `suitsSymb`, чтобы получить символ масти.

Масть может быть *козырной*. Масть считается козырной, если совпадает
с мастью, выбранной дилером до начала игры.
Эту масть можно узнать с помощью
метода `Dealer::GetTrump`.

3.3 Карта

Обычная карта имеет одну из четырёх мастей и один из тринадцати рангов.

Особая карта может находиться в одном из двух дополнительных особых состояний: «Пас» или «Без карты». В таком случае, её ранг будет равен `PAS` или `NOCARD` соответственно.

Обычная карта может быть *козырной*. Карта считается козырной, если у неё козырная масть.

Карта может быть «атакующей» или «защищающей». Карта, которой играет игрок с инициативой, называется атакующей. Карта, которой его оппонент может согласно правилам может побить атакующую карту, называется защищающая.

«Атакующую» карту можно побить «защищающей». Это возможно в одном из двух случаев:

- Если карты имеют одинаковую масть, то атакующую карту можно побить защищающей в случае если у первой ранг меньше, чем у второй.
- Если карты имеют разную масть, то атакующую карту любого ранга можно побить козырной защищающей картой любого ранга.

3.4 Колода

Последовательный набор уникальных не особых карт.

В начале игры колода из каждой возможной обычной карты перемешивается случайным образом. Таким образом до начала игры в колоде 52 карты.
За перемешивание колоды отвечает метод `Dealer::ShuffleDec`

Дилер может доставать из колоды карты.

3.5 Ход

Ход это последовательность действий игроков, начинающаяся началом игры или концом другого хода, и заканчивающаяся тем, что игрок без инициативы успешно защитился от атакующих карт или не смог защититься.

За ход игрок с инициативой может атаковать шестью картами, в согласии с правилами.

3.6 Стол

Стол это место, куда игроки разыгрывают карты в соответствии с правилами.

На столе лежат карты. Стол имеет тип `Card*[6]`.

За один ход игроки могут сыграть по 6 карт каждый.

3.7 Дилер

Дилер отвечает за управление колодой и столом.

Дилер в коде отображён как класс `Dealer`.

Дилер достаёт карты из колоды и раздаёт игрокам в соответствии с правилами игры.
За это отвечает метод `Dealer::GetCard`.

У дилера можно спросить козырную масть.
За это отвечает метод `Dealer::GetTrump`.

У дилера можно спросить число вышедших из колоды карт.
За это отвечает метод `Dealer::getcurrentCard`.

У дилера можно получить указатель на стол.
За это отвечает метод `Dealer::GetheadTrick`.

У дилера можно узнать номер текущей атакующей карты в ходе.
За это отвечает метод `Dealer::GetCurrentHeadTrik`.

У дилера можно узнать можно ли сходить атакующей картой.
За это отвечает метод `Dealer::NextTrikEnable`.

У дилера можно попросить особые карты "пас" и "без карты".
За это отвечают методы `Dealer::GetPas` и `Dealer::GetNocard`.

У дилера можно узнать последнюю атакующую или защищающую карту.
За это отвечают методы `Dealer::GetLastCard` и `Dealer::GetLastDefendCard`.

Только с помощью дилера можно атаковать или защищаться картой.
За это отвечают методы `Dealer::Attack` и `Dealer::Defend`.

Дилер проверяет защиту от атакующих карт на корректность.
За это отвечает метод `Dealer::CheckHeadTrick`.

После каждого хода дилер очищает стол от карт.
За это отвечает метод `Dealer::ClearTable`.

3.8 Игрок

Игрок — алгоритм или человек, принимающий не противоречащие правилам решения в игре.

Один игрок играет лишь за одну сторону.

В коде представлен как класс `Player`, наследующийся от класса `PlayerAbstract`.

Карты игрока, находятся в «руке».

4 Описание реализации класса Player

4.1 Поля класса Player

Определение класса можно найти в приложении.

1. Имя игрока.

   Так как в функции `main` в конструктор класса `Player` зачем-то передаётся имя, то значит надо его хранить. Таким образом, поле на данный момент не выполняет никакой функции. Имя хранится как поле `const char* m_name`.

2. Рука игрока.

   Рука - массив, где игрок хранит все свои карты.
   Рука хранится как поле `Card *m_hand[deckSize]`, где `deckSize` = 52 — размер колоды. Указатели на карты могут храниться в руке без определённого порядка, но должны располагаться подряд от нулевого индекса без пробелов. Местам в массиве, которым не назначены указатели на карты, должны быть назначены нулевые указатели. Привести руку к такому состоянию можно с помощью метода `Player::normalizeHand`.

3. Количество карт в руке.

   Хранится как поле `int m_cardscount`.

4. Состояние игры.

   Состояние игры - структура с информацией об игре, которую игроку нужно знать.

   Хранится как поле `GameState m_state`.

   Тип структуры определён как `struct GameState { int trumpSuit; };`

   На данный момент структура содержит только поле с текущей козырной мастью — поле `trumpSuit`. Это поле хранится тут для удобства.

Определения методов класса можно найти в приложении.
Блок-схемы, если имеются, находятся в другом документе.

4.2 Метод Player::YouTurn

1. Метод принимает аргумент `bool flag` — признак того, что на этом ходу атакует этот игрок.

2. Метод ничего не возвращает.

3. Метод ничего не делает и находится в классе лишь для соответствия интерфейсу класса `PlayerAbstract`.

4.3 Метод Player::PutCard

1. Метод не принимает аргументов.
2. Метод не возвращает значений.
3. Метод вынуждает игрока атаковать картой (или спасовать). По большей части метод выбирает лишь атаковать или нет, а не чем атаковать.
4. Если в руке нет карт, то атаковать нечем, кладём карту «без карты».
   Если есть, с помощью метода `Player::chooseAttackingCard` выбираем карту, которой можно атаковать.
   Если на этом ходу мы уже атаковали, то пасуем.
   Если нет, атакуем выбранной картой (`Dealer::Attack`).
   Достаём карту из колоды (`Player::popCard`).

4.4 Метод Player::TakeCards

1. Метод не принимает аргументов.
2. Метод не возвращает значений.
3. Метод берёт все карты со стола.
4. Получаем указатель на стол у дилера (`Dealer::GetheadTrick`).
   В цикле проходим по картам стола.
   Если карта — обычная, то берём её с помощью метода `Player::TakeOneCard`.

4.5 Метод Player::GetHeadTrick

1. Метод не принимает аргументов.
2. Метод не возвращает значений.
3. Метод вынуждает игрока защититься картой или спасовать.
4. Если последняя атакующая карта была «пас» или «без карты», то кладём «без карты». Это значит что защищаться не нужно, противник не атакует.
   Если не осталось карт в руке, то кладём «без карты».
   Принимаем решение, какой картой защищаться (`Player::chooseDefendingCard`).
   Если метод вернул константу `unchosen` — это признак паса.
   В остальных случаях метод возвращает индекс карты, которой надо защититься. Достаём карту по индексу и защищаемся.

4.6 Метод Player::TakeOneCard

1. Метод принимает аргумент `Card*&nc` — ссылку на карту, которую надо взять в руку.
2. Метод не возвращает значений.
3. Метод помещает карту в руку.
4. В руке в вышеуказанном состоянии, `m_cardscount` — индекс элемента массива, куда надо записать карту.
   Записываем туда данный указатель на карту и увеличиваем на 1 значение счётчика `m_cardscount`, т.к. карт прибавилось на одну.

4.7 Метод Player::ShowCards

1. Метод не принимает аргументов.
2. Метод не возвращает значений.
3. Метод печатает карты в руке.
4. С помощью цикла проходим по всем картам, узнаём их масть и ранг с помощью `Dealer::SuitName` и `Dealer::RankName`, печатаем на экране в формате `<символ масти><первая буква ранга>` через пробел.

   Примечание: ранг 10 печатается как 1.

4.8 Метод Player::INeedCard

1. Метод не принимает аргументов.
2. Метод возвращает значение типа `bool` - признак того, что игроку не хватает карт до полной руки.
3. Метод узнаёт, нужны ли карты игроку.
4. Если количество карт в руке < 6 и у дилера в колоде остались карты, то возвращает истину.

4.9 Метод Player::GetCardNum

1. Метод не принимает аргументов.
2. Метод возвращает значение типа `int` — количество карт в руке игрока.
3. Метод возвращает поле класса `m_cardscount`.

Дальнейшие методы класса не принадлежат к интерфейсу `PlayerAbstract`

4.10 Метод Player::normalizeHand

1. Метод не принимает аргументов.
2. Метод не возвращает значений.
3. Метод приводит руку к вышеуказанному виду, сдвигает карты влево по руке.
4. Копируем руку в буфер.
   В цикле проходим по буферу, и ставим найденные карты в крайнее левое положение в руку.
   Остальную часть руки «зануляем» и устанавливаем в счётчик `m_cardscount` сколько карт мы нашли.

4.11 Метод Player::less

1. Метод принимает два аргумента – карты. `Card* first, Card* secnd`
2. Метод возвращает `bool` — признак того, что первая карта «меньше» второй.
3. Метод сравнивает две карты.
4. Одна карта «меньше» второй, если у первой меньше ранг, или первая уступает в козырности.

4.12 Метод Player::canDefend

1. Метод принимает два аргумента – карты. `Card* first, Card* secnd`
2. Метод возвращает `bool` — признак того, что первой картой можно побить вторую.
3. Первой картой можно побить вторую, если одно выполняется:
   (a) вторая меньше первой и масти совпадают.
   (b) вторая уступает в козырности.

4.13 Метод Player::popCard

1. Метод принимает один аргумент `int index` — индекс карты, которую надо достать.
2. Метод возвращает карту, которую надо было достать.
3. Метод достаёт карту из колоды, и поправляет руку.
4. На место нужной карты, ставим `nullptr`, поправляем колоду и возвращаем карту.

4.14 Метод Player::chooseAttackingCard

1. Метод не принимает аргументов.
2. Метод возвращает индекс (`int`) карты в руке, которой можно атаковать.
3. Метод выбирает карту, которой можно атаковать, которую меньше всего жалко. Сам метод не атакует.
4. В цикле проходим по руке и методом `Player::less` выбираем «наименьшую».

4.15 Метод Player::chooseDefendingCard

1. Метод не принимает аргументов.
2. Метод возвращает либо индекс (`int`) карты, которой можно побить атакующую, либо `unchosen` — признак паса.
3. Метод выбирает пасовать или защищаться, и если защищаться, то выбирает карту, которой это можно сделать.
4. В цикле выбираем карту наименьшую карту, которой можно защититься от атакующей карты.

5. Если выбрали козырь, то пасуем, возвращая `unchosen`, если не конец игры. В конце игры можно будет использовать козырей.
   Возвращаем индекс найденной карты.

5 Описание тестирования

- Класс игрока `Player` способен играть в соответствии с предоставленными правилами и алгоритмом игры в файле `main.cpp`.
- Библиотека `VLD` не находит утечек памяти. Это вероятно связано с тем, что я не использую динамически выделенную память.
- Тестирование считаю достаточным.

6 Заключение

6.1 Изучено

- В процессе реализации алгоритма я изучил методологию «Объектно–Ориентированного» программирования на данном примере.
- Изучена система вёрстки LaTeX.
- Столкнувшись с исключениями возникающими внутри библиотеки, мне пришлось познакомиться с методами реверс-инжиниринга. В частности с программами gdb, ghidra и objdump. Эти инструменты помогли мне локализовать мою проблему; решить её мне помог исходный код библиотеки.

6.2 Освоено

- Освоена методология ООП, разработка алгоритмов с её помощью.
- Освоены методы вёрстки документов с помощью LaTeX.

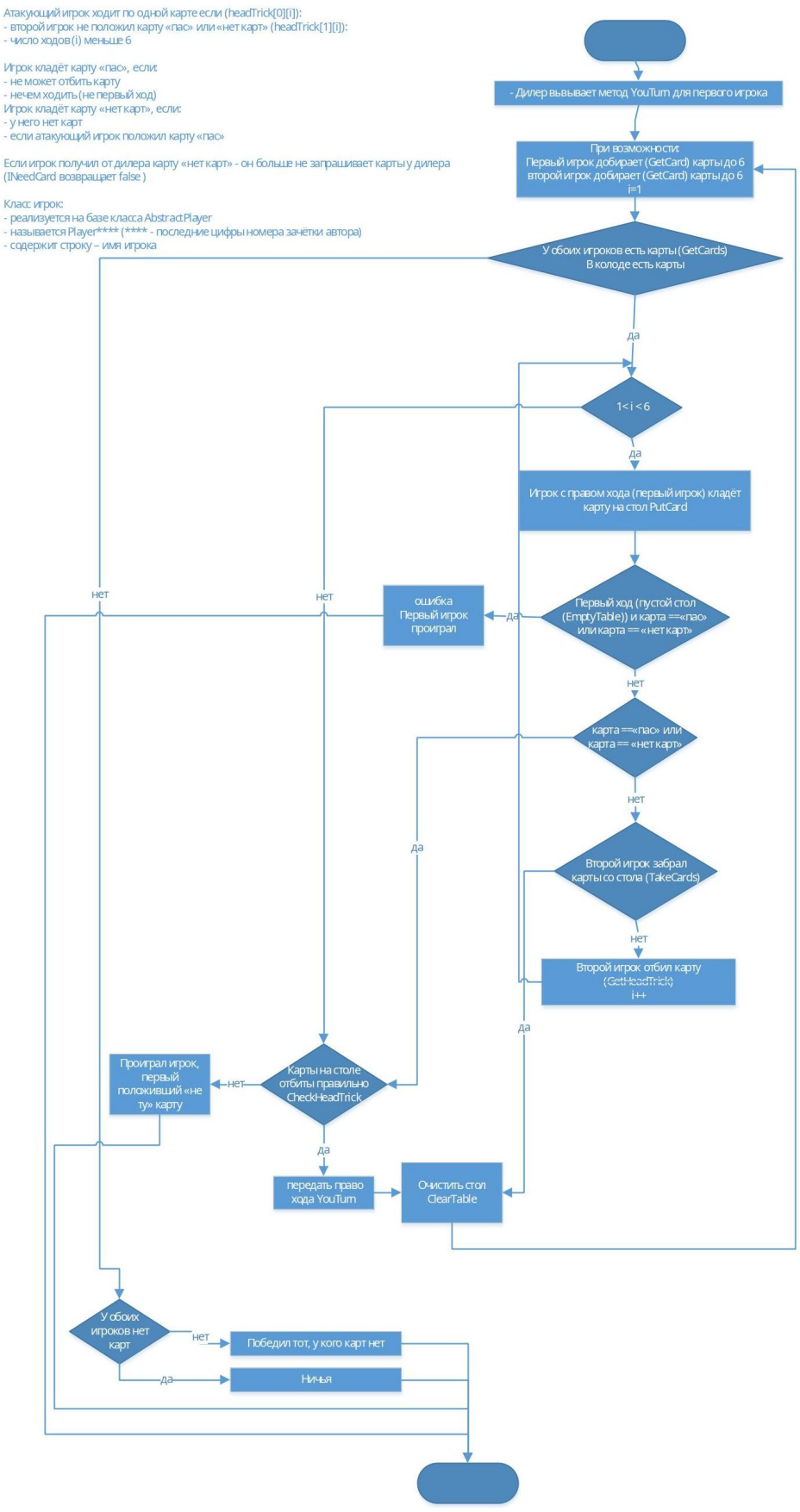
6.3 Реализовано

- Реализован алгоритм игры в дурака по указанным правилам. Алгоритм представлен в качестве класса `Player`.
- Составлен отчёт в соответствии с требованиями.
- Программа, симулирующая 100 игр в дурака между копиями реализованного алгоритма. Таким образом реализация алгоритма тестируется.

Всё что реализовано до неопределённого времени доступно в git-репозитории https://github.com/deutherity/durak.

7 Приложения

7.1 Блок-схема алгоритма игры в дурака (/main.cpp)



7.2 Код файла player.h

```
#pragma once

#include "dealer.h"

class PlayerAbstract
{
protected:


public:
    virtual ~PlayerAbstract() {};
    //получает признак "мой ход"
    virtual void YouTurn(bool) = 0;
    //игрок кладёт карту на стол (headTrick[0][*])
    virtual void PutCard() = 0;
    // забирает все карты со стола
    virtual void TakeCards() = 0;
    // отбивает карту (кладёт карту в (headTrick[1][*])
    virtual void GetHeadTrick() = 0;
    //взял одну карту
    virtual void TakeOneCard(Card * &nc) = 0;
    // вывел свои карты на экран ранг (один/два символа, масть - символ)
    virtual void ShowCards() = 0;
    // возвращает истину, если на руках карт меньше 6
    virtual bool INeedCard() = 0;
    // возвращает число карт на руках
    virtual int GetCardNum() = 0;
};

struct GameState
{
    // Козырь в текущей игре
    int trumpSuit;
};

constexpr int deckSize = 52;

class Player : public PlayerAbstract
{
// Реализуйте интерфейсы абстрактного класса
// Доступные методы из класса Dealer можно увидеть в файле dealer.h
public:

    Player(const char* name);
    ~Player() = default;

    // см. выше
    void YouTurn(bool flag) final;
    void PutCard() final;
```

```
    void TakeCards() final;
    void GetHeadTrick() final;
    void TakeOneCard(Card * &nc) final;
    void ShowCards() final;
    bool INeedCard() final;
    int GetCardNum() final;

private:

    // Сдвигает указатели в руке влево до упора.
    void normalizeHand();
    // Проверяет, имеет ли первая карта меньший ранг, или уступает по козырно
    bool less(Card* first, Card* secnd) const;
    // Проверяет, можно ли первой картой побить вторую.
    bool canDefend(Card* first, Card* secnd) const;
    // Достаёт карту из руки
    Card * popCard(int index);
    // Выбор атакующей карты
    int chooseAttackingCard() const;
    // Выбор защищающей карты
    int chooseDefendingCard() const;

    // Имя игрока
    const char* m_name;
    // Рука с картами
    Card* m_hand[deckSize];
    // Количество карт в руке.
    int m_cardscount = 0;
    // Состояние игры
    GameState m_state;

};

using Player0408 = Player;
```

7.3 Код файла `player.cpp`

```
#include <iostream>
#include "dealer.h"
#include "player.h"

template <typename T>
void swap(T& a, T& b)
{
    T& tmp = a;
    a = b;
    b = tmp;
}

bool Player::less(Card* first, Card* secnd) const
```

```
{
    auto s1 = Dealer::SuitIndex(first);
    auto s2 = Dealer::SuitIndex(secnd);

    auto r1 = Dealer::RankIndex(first);
    auto r2 = Dealer::RankIndex(secnd);

    auto trumpSuit = m_state.trumpSuit;

    if (s1 != s2)
    {
        // Если масти различаются, то надо проверить козырная ли масть
        if (s1 == trumpSuit)
            return false;
        if (s2 == trumpSuit)
            return true;
        return r1 < r2;
    }
    return r1 < r2;
}


Player::Player(const char* name): m_name(name)
{
    for (int i = 0; i < deckSize; ++i)
        m_hand[i] = nullptr;
    m_state.trumpSuit = Dealer::SuitIndex(Dealer::GetTrump());
}

void Player::normalizeHand()
/*
 * Сдвигает карты влево
 * 0 0 1 0 0 2 0 3 4  --->  1 2 3 4 0 0 0 0 0
 * 1 2 3 0
 */
{
    Card * cards[deckSize];
    for (int i = 0; i < deckSize; ++i)
        cards[i] = m_hand[i];

    int j = 0;
    for (int i = 0; i < deckSize; ++i)
        if (cards[i] != nullptr)
            m_hand[j++] = cards[i];
    m_cardscount = j;
    for (; j < deckSize; ++j)
        m_hand[j] = nullptr;
}

void Player::YouTurn(bool flag) {}
```

```
int Player::chooseAttackingCard() const
// Найти индекс карты, которой хочется сходить
{
    // Найти карту наименьшего достоинства
    Card *minimal = m_hand[0];
    int minimal_index = 0;
    for (int i = 1; i < m_cardscount; ++i)
    {
        if (less(m_hand[i], minimal))
        {
            minimal = m_hand[i];
            minimal_index = i;
        }
    }
    return minimal_index;
}

constexpr int unchosen = -1;

void Player::PutCard()
// Атака
{
    if (m_cardscount == 0)
        Dealer::Attack(Dealer::GetNocard());
    int index = chooseAttackingCard();
    Card * chosenCard = m_hand[index];
    if (Dealer::GetCurrentHeadTrik() != 0)
    {
        // if (Dealer::SuitIndex(chosenCard) == m_state.trumpSuit)
        {
            chosenCard = Dealer::GetPas();
            index = unchosen;
        }
    }
    if (index != unchosen)
    // Значит надо убрать карту из руки и сдвинуть карты в руке влево
        popCard(index);
    Dealer::Attack(chosenCard);
}

Card * Player::popCard(int index)
{
    // m_cardscount -= 1;
    Card* card = m_hand[index];
    m_hand[index] = nullptr;
    normalizeHand();
    return card;
}
```

```
void Player::TakeCards()
{
    const auto table = Dealer::GetheadTrick();
    int end = Dealer::GetCurrentHeadTrik();
    for (int i = 0; i < end; ++i)
        for (int j = 0; j < 2; ++j)
        {
            Card *card = table[j][i];
            if (card != nullptr &&
                Dealer::RankIndex(card) != PAS &&
                Dealer::RankIndex(card) != NOCARD)
                TakeOneCard(table[j][i]);
        }
}

void Player::GetHeadTrick()
// Защита
{
    auto rank = Dealer::RankIndex(Dealer::GetLastCard());
    if (rank == PAS || rank == NOCARD)
        return Dealer::Defend(Dealer::GetNocard());
    if (m_cardscount == 0)
        return Dealer::Defend(Dealer::GetNocard());
    int index = chooseDefendingCard();
    if (index == unchosen) // Если нечем крыть - пасуем
        return Dealer::Defend(Dealer::GetPas());
    // Если есть чем - кроем
    // Надо убрать карту из руки и сдвинуть карты в руке
    Dealer::Defend(popCard(index));
}

bool Player::canDefend(Card* first, Card* secnd) const
{
    auto s1 = Dealer::SuitIndex(first);
    auto s2 = Dealer::SuitIndex(secnd);

    return less(secnd, first) && (s1 == s2 || s1 == m_state.trumpSuit);
}

int Player::chooseDefendingCard() const
{
    Card *attackingCard = Dealer::GetLastCard();
    // Карта, которую нужно покрыть
    bool use_trump = Dealer::getcurrentCard() == deckSize;
    // если конец игры, то используй козыри
    int minimal_index = unchosen;
    // Индекс карты, которой мы хотим покрыть

    for (int i = 0; i < m_cardscount; ++i)
    {
```

```
        Card* card = m_hand[i];
        bool is_trump_card = Dealer::SuitIndex(card) == m_state.trumpSuit;
        if (canDefend(card, attackingCard) &&
            (!is_trump_card || use_trump))
            // Если такой картой можно защититься
            // и если это не козырь или мы можем использовать козыри
        {
            if (minimal_index == unchosen)
            // Если ещё не выбрали карту
                minimal_index = i;
            else if (less(card, m_hand[minimal_index]))
            // или нашли карту поменьше
                minimal_index = i;
                // то выбираем эту карту
        }
    }
    return minimal_index;
}


void Player::TakeOneCard(Card*&nc)
{
    m_hand[m_cardscount++] = nc;
}

void Player::ShowCards()
{
    for (int i = 0; i < m_cardscount; ++i)
    {
        Card * card = m_hand[i];
        std::cout << Dealer::RankName(card)[0] <<
#ifdef _WIN32
        suitsSymb[Dealer::SuitIndex(card)]
#else
        Dealer::SuitName(card)[0]
#endif
         << ' ';
    }
    std::cout << std::endl;
}

bool Player::INeedCard()
{
    return m_cardscount < 6 && Dealer::getcurrentCard() < 52;
}

int Player::GetCardNum()
{
    return m_cardscount;
}
```