МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО
ОБРАЗОВАНИЯ
«Санкт-Петербургский политехнический университет Петра Великого»
Институт компьютерных наук и технологий

Высшая школа искусственного интеллекта

Направление: 02.03.01 Математика и компьютерные науки

ОТЧЁТ ПО ДИСЦИПЛИНЕ:
«Программирование и алгоритмизация»

Реализация классов StupidPlayer и SmartPlayer для карточной игры «UNO» на языке C++

Студент,
группы 3530201/20002 _____ Золоев Г. А.

Преподаватель,
_____ Сеннов В. Н.

«____»_____ 2023 г.

Санкт-Петербург, 2023

Содержание

1 Постановка задач

Главная цель курсовой работы – разработать класс игрока, который может успешно встроиться в уже написанную игру "UNO". Данный класс является наследником абстрактного класса UnoPlayer. Всё это заключается в реализации виртуальных методов класса-родителя классом-наследником в соответствии с комментариями. Соответственно, задачами является реализовать эти методы, так как они отражают действия игроков во время партий.

Задачи:

– Метод receiveCards (получить карты в колоду);

– Метод playCard (сыграть картой);

– Метод drawAdditionalCard (взять дополнительную карту);

– Метод changeColor (поменять цвет);

2 Описание алгоритмов и реализации

2.1 Математическое описание алгоритмов классов игроков

2.1.1 StupidPlayer

Стратегия игры в классе StupidPlayer состоит в том, что, проверяя свою колоду, игрок будет оценивать каждую карту по очереди, и если ей можно походить, то ей походит. При необходимости поменять цвет игрок выберет его случайно. Если игра предложит игроку взять одну карту, то игрок проверит можно ли ей походить, если да, то походит, если нет, то оставит.

2.1.2 SmartPlayer

Стратегия игры в классе SmartPlayer состоит из нескольких этапов:

1) Если у соперника 3 и менее карты(если нет, то переходим в пункт "2"):
   (a) пытаемся походить картой "+2" любого доступного цвета;
   (b) если пункт "a"не сработал, то пытаемся походить любой картой действия, тем самым замедляя противника;
   (c) если карт действия нет, то переходим в пункт "3".
2) Если в наличии больше, чем 1 карта действия, или есть одна, но цвет этой карты есть в колоде, то мы ходим картами действия, до тех пора, пока они не закончатся. Если таких карт нет, то переходим в пункт "3".
3) Ходим картой максимального номинала текущего цвета в игре. Если карт такого цвета нет, переходим в пункт "4".
4) Считаем какого цвета в колоде у нас максимальное количество, играем такой картой, если её номинал совпадёт с текущим в игре,(если таких несколько, то выбираем с наибольшим номиналом). Если такой нет, то переходим в пункт "5".

5) Пытаемся походить картой, номинал которой совпадает с текущем в игре, причем если таких карт несколько, то ходим максимальной. Если таких карт нет, то переходим к следующему пункту.

6) Ходим дикой картой.

2.2 Реализация

2.2.1 StupidPlayer

Рассмотрим методы, которые были реализованы. (Программный код класса предствален в "Приложении А".)

std::string name():

Этот «геттер-метод» возвращает поле класса name_of_player.

void receiveCards(const std::vector<const Card*>& cards):

В этом методе мы будем получать карты(при начале партии или когда противник сыграет картой «+2» или «+4»).

Чтобы каждый раз при начале новой партии «чистить» массив cards мы проверяем размер передаваемого вектора, если он равен семи, то «чистим» вектор посредством метода clear().

Далее проходимся по передаваемому вектору и добавляем каждый элемент в конец нашего поля класса cards.

const Card* playCard():

Данный метод вызывается, когда приходит очередь ходить. Мы должны вернуть карту, которой хотим походить.

Для начала мы сортируем вектор по возрастанию, потом проходимся циклом по вектору, и поочерёдно сравниваем цвет и значение карты с текущими. Когда найдём соответствие, создаем и записываем в переменную temp текущую карту, потом удалям его из поля класса cards и возвращаем эту карту.

bool drawAdditionalCard(const Card* additionalCard):

Данный метод, вызывается в тех случаях, когда у нас нет карты, чтобы походить. Нам нужно вернуть true, если картой походить можно, иначе false.

Проверяем «дикая» ли карта, если да, то возвращаем true, если нет,

то проверяем сходится её цвет или значение с текущими в игре, если да, то возвращаем true, если нет, то добавляем эту карту в поле класса cards, и возвращаем false.

CardColor changeColor():

Данный метод вызывается в тех случаях, когда игрок бросил дикую карту. Мы должны вернуть один из цветов: Red, Blue, Green, Yellow.

Создаём вектор colors, в котором хранится 4-ре цвета, возвращаем один из этих цветов путём случайного выбора.

2.2.2 SmartPlayer

Рассмотрим методы, которые были реализованы. (Программный код класса представлен в "Приложении Б".)

std::string name():

Этот «геттер-метод» возвращает поле класса name_of_player.

void receiveCards(const std::vector<const Card*>& cards):

В этом методе мы будем получать карты(при начале партии или когда противник сыграет картой «+2» или «+4»).

Чтобы каждый раз при начале новой партии «чистить» массив cards мы проверяем размер передаваемого вектора, если он равен семи, то «чистим» вектор посредством метода clear().

Далее проходимся по передаваемому вектору и добавляем каждый элемент в конец нашего поля класса cards.

const Card* playCard():

Данный метод вызывается, когда очередь игрока ходить. Мы должны вернуть карту, которой хотим походить. Создаётся вектор и переменные:

- vector<const Card*> array; – он приравнивается к возвращённому значению метода array_can_play() (он будет описан в следующих пунктах) – в этом векторе будут лежать карты, которыми возможно походить, причём отсортированные по возрастанию.
- int op_index; – инициализируется индексом нашего соперника, посредством тернарного оператора.
- bool is_another_player_have_three_cards; – у переменной говорящее название, если у соперника три или менее карт, то переменная

инициализируется true, иначе – false.

Далее для удобства перейдём к этапам стратегии, которые описаны в секции с математическими алгоритмами.

1) Если is_another_player_have_three_cards равен true, то тогда проходимся циклом по вектору array и пытаемся найти карту "+2", если нашли, то мы удаляем из исходного вектора cards данную карту (посредством метода вектора erase и метода find_index (он будет описан в следующих пунктах)). После, возвращаем карту. Иначе – снова проходимся циклом по вектору array, пытаясь найти либо карту «действия», либо «дикую» карту. Сначала удаляем её из вектора cards (посредством std::erase и find_index (он будет описан в следующих пунктах)), потом возвращаем её. Если, пройдя цикл, такой карты не нашлось, то пункт 2.

2) Если метод is_more_than_one_action_card(array) возвращает true, то удаляем из исходного вектора cards карту, которая находится на последнем месте в векторе array (array[array.size() -1] (посредством метода вектора std::erase и метода find_index(он будет описан в следующих пунктах). Далее, возвращаем эту карту (которая находится в конце вектора array). Если метод is_more_than_one_action_card(array) вернул false, то пункт 3.

3) Проходимся циклом (с конца) по вектору array, если цвет текущей карты(array[i] равен текущему цвету игры(currentColor)) и значение карты < 10(так мы проверяем, что это обычная карта), то удаляем из исходного вектора cards данную карту array[i] (посредством метода вектора std::erase и метода find_index(будет описан далее)), и возвращаем найденную карту(array[i]). Если, пройдя весь цикл, такая карта не найдена, то пункт 4.

4) Проходимся циклом по вектору array(с конца) и ищем карту, значение которой равно значению последней карты в сбросе, а цвет равен возвратимому значению метода return_max_cnt_color() (метод будет описан в следующих пунктах). Эту карту мы сначала удаляем из исходного вектора cards(посредством метода вектора std::erase и метода find_index(он будет описан в следующих пунктах)), затем возвращаем эту карту(array[i]). Если, пройдя весь цикл, такая карта не найдена, то пункт 5.

5) Проходимся циклом по вектору array(с конца) и ищем карту, значение которой равно значению последней карты в сбросе. Удаляем эту карту из вектора cards (посредством метода вектора std::erase и метода find_index(он будет описан в следующих пунктах)), возвращаем эту карту (array[i]). Если, пройдя весь цикл, карты не нашлось, то пункт 6.

6) Остается лишь вариант, что игрок может походить «дикой» картой. Мы удаляем из исходного вектора cards карту, которая находится на последнем месте в векторе array(array[array.size() -1]) (посредством метода вектора std::erase и метода find_index(он будет описан в следующих пунктах). Далее, возвращаем эту карту(которая в конце вектора array).

bool drawAdditionalCard(const Card* additionalCard):

Данный метод, вызывается в тех случаях, когда у нас нет карты, чтобы походить. Нам нужно вернуть true, если картой походить можно, иначе false.

Проверяем «дикая» ли карта, если да, то возвращаем true, если нет, то проверяем сходится её цвет или значение с текущими в игре, если да, то возвращаем true, если нет, то добавляем эту карту в поле класса cards, и возвращаем false.

CardColor changeColor():

Данный метод, вызывается в тех случаях, когда игрок бросил дикую карту. Мы должны вернуть из цветов: Red, Blue, Green, Yellow.

Создаём вектор colors, в котором хранится 4-ре цвета, возвращаем один из этих цветов путём случайного выбора.

std::vector<const Card*> array_can_play():

Данный метод будет возвращать вектор карт, которыми возможно сыграть.

Объявляем вектор:

- std::vector<const Card*> array; – позже именно его будем возвращать.

Проходимся циклом по вектору cards, и если цвет или значение карты под текущей итерацией равны цвету и значению последней карты в сбросе, но эта карта не равна «дикой», то добавляем в вектор array эту карту. Пройдя цикл, проверяем равен ли размер вектора array нулю. Если да, то опять проходимся по циклу и добавляем в array «дикие» карты(если они есть). Пройдя оба цикла, сортируем вектор array по значению карт по возрастанию и возвращаем array.

CardColor return_max_cnt_color():

Данный метод будет возвращать из цветов: Red, Blue, Green, Yellow, которого в нашей колоде (векторе cards) больше всего.

Объявляем векторы:

- std::vector<const Card*> array; – позже именно его будем возвращать.
- std::vector<const Card*> red_arr; – будет отвечать за карты красного цвета;
- std::vector<const Card*> blue_arr; – будет отвечать за карты синего цвета;
- std::vector<const Card*> green_arr; – будет отвечать за карты синего цвета;
- std::vector<const Card*> yellow_arr; – будет отвечать за карты жёлтого цвета;
- std::vector<const Card*> mx_size; – для удобства;

Проходимся циклом по вектору cards и посредством конструкции switch case до добавляем карты в соответствующие им по цвету векторы. Пройдя цикл, снова проходимся циклом по вектору arr(используя конструкцию for (std::vector<const Card*> arr: red_arr, blue_arr, green_arr, yellow_arr), то есть arr поочерёдно будет равен векторам в фигурных скобках). Если размер текущего вектора больше размера вектора mx_size, то текущий вектор приравнивается к mx_size. Тем самым, в mx_size по итогу цикла, будет лежать цикл, имеющий наибольший размер. Далее, проверим равен ли mx_size нулю, если да, то случайно выбираем один из четырех цветов, если нет то возвращаем цвет нулевого элемента, лежащего в mx_size.

int find_index(const Card* temp):

Данный метод в основном используется, чтобы корректно удалять элементы из вектора cards. Метод возвращает индекс переданной карты в векторе cards.

Проходимся циклом по вектору cards, и если значение и цвет карты temp равны значению и цвету текущей карты(cards[i]), то возвращаем этот индекс i, под которым лежит карта.

bool is_more_than_one_action_card(std::vector<const Card*> array):

Данный метод вернет true в двух случаях: если карт «действия», которыми мы можем сыграть больше, чем одна. Если есть только одна карта «действия», которой мы можем сыграть, но в array есть ещё обычная карта такого же цвета.

Заводим переменные:

- int cnt = 0 – она будет отвечать количество карт «действия».

- Cardcolor temp; – будет отвечать за пункт б);
- bool is_more = false – то, что мы возвращаем;

Проходимся циклом по вектору array, если карта под текущей итерацией является картой «действия», то cnt++, а temp приравняется к цвету этой карты. Если, пройдя цикл, cnt >= 2, то is_more = true, а если cnt == 1, то снова проходимся циклом по тому же вектор, и если, найдется обычная карта такого же цвета как temp, то is_more = true, выходим из цикла. Возвращаем is_more.

3 Результаты

В исходном коде игры "Uno"есть функция statisticTest(), с помощью которой можно узнать, как часто выигрывает игрок. Поставим ограничение на 1000 игр. В игре 1 на 1 SmartPlayer в среднем выигрывает StupidPlyaer'а в 95% случаев, что является хорошим результатом.

Вывод

В результате работы были успешно реализованы классы игроков. Все поставленные задачи были выполнены при помощи понятий и принципов объектно-ориентированного программирования, изученных во время освоения дисциплины «Программирование и алгоритмизация».

На это было потрачено около 7 часов. Работа выполнялось в среде разработке Xcode 13 с помощью компилятора Apple Clang версии 14.0.0.

Полученные опыт и знания будут использоваться в дальнейших работах и проектах.

Приложение А. Исходный код StupidPlayer

StupidPlayer.hpp

```
#pragma once

#include "events.h"
#include "uno_game.h"

using std::string;
using std::vector;
using std::sort;

class StupidPlayer: public UnoPlayer{
    vector<const Card*> cards;
    string name_of_p;
public:
    StupidPlayer(string name);

    /// @brief Игрок возвращает свое имя.
    /// @return имя игрока.
    virtual std::string name() const;

    /// @brief Игрок получает на руки карты.
    /// @param cards массив карт.
    virtual void receiveCards(const std::vector<const Card*>
                                                & cards);

    /// @brief Игрок возвращает карту, которую он сыграет
    ///  (положит всброс).
    /// @return карта, которую игрок положит в сброс.
    virtual const Card * playCard();

    /// @brief Если у игрока нет подходящих карт, он тянет
    /// дополнительную карту и говорит, хочет ли он ее
    /// сразу положить или нет.
```

```
    /// @param additionalCard дополнительная карта.
    /// @return true, если игрок сразу же кладет эту карту,
    /// иначе false.
    virtual bool drawAdditionalCard(const Card * additionalCard);

    /// @brief Если игрок положил "Закажи цвет" или
    /// "Возьми четыре", то
    /// игра запросит у него новый цвет.
    /// @return новый цвет.
    virtual CardColor changeColor();
};
```

A.2. StupidPlayer.cpp

```
#include "StupidPlayer.hpp"

StupidPlayer::StupidPlayer(string name){
    this->name_of_p = name;
}

std::string StupidPlayer::name() const{
    return name_of_p;
};


void StupidPlayer::receiveCards(const std::vector<const Card*>&
                                cardsss){
    if (cardsss.size() == 7){
        cards.clear();
    }

    for (int i = 0; i < cardsss.size(); i++){
        this->cards.push_back(cardsss[i]);
    }
};

bool CompareCardsByValue(const Card* a, const Card* b) {
    return a->value < b->value;
}

const Card * StupidPlayer::playCard(){
    sort(cards.begin(), cards.end(), CompareCardsByValue);
    const Card* temp;
    for (int i = 0; i < cards.size(); i++){
        if ((cards[i]->color == this->game()->currentColor() ||
            cards[i]->value == this->game()->topCard()->value))
        {
            const Card* temp = cards[i];
            cards.erase(cards.begin() + i);
```

```
            break;

        }
    }

    return temp;
}
bool StupidPlayer::drawAdditionalCard(const Card *
                                      additionalCard){
    if (additionalCard->is_wild()){
        return true;
    }
    if (additionalCard->color == this->game()->
                                 currentColor() ||
        additionalCard->value == this->game()->
                                 topCard()->value){
        return true;
    }

    cards.push_back(additionalCard);
    return false;
};
CardColor StupidPlayer::changeColor(){
    srand((unsigned) time(NULL));
    vector<CardColor> colors {Red, Blue, Green, Yellow};
    int x = rand() % 4;
    return colors[x];};
```

Приложение В. Исходный код SmartPlayer

В.1. SmartPlayer.hpp

```
#pragma once

#include "events.h"
#include "uno_game.h"
#include <algorithm>
#include <iostream>

using std::string;
using std::vector;
using std::sort;

class SmartPlayer: public UnoPlayer{
    vector<const Card*> cards;
    string name_of_player;
public:
    SmartPlayer(string name);
    /// @brief Игрок возвращает свое имя.
    /// @return имя игрока.
    virtual std::string name() const;

    /// @brief Игрок получает на руки карты.
    /// @param cards массив карт.
    virtual void receiveCards(const vector<const Card*>& cards);

    /// @brief Игрок возвращает карту, которую он сыграет
    (положит в сброс).
    /// @return карта, которую игрок положит в сброс.
    virtual const Card * playCard();

    /// @brief Если у игрока нет подходящих карт, он тянет
    дополнительную
    /// карту и говорит, хочет ли он ее сразу положить или нет.
```

```
    /// @param additionalCard дополнительная карта.
    /// @return true, если игрок сразу же кладет эту карту,
    иначе false.
    virtual bool drawAdditionalCard(const Card * additionalCard);

    /// @brief Если игрок положил "Закажи цвет" или
    "Возьми четыре", то
    /// игра запросит у него новый цвет.
    /// @return новый цвет.
    virtual CardColor changeColor();

    CardColor return_max_cnt_color();
    vector<const Card*> array_can_play();
    int find_index(const Card* temp);
    bool is_more_than_one_action_card(vector<const Card*> array);
};
```

B.2. SmartPlayer.cpp

```
#include "SmartPlayer.hpp"

SmartPlayer::SmartPlayer(string name){
    name_of_player = name;
}

std::string SmartPlayer::name() const{
    return name_of_player;
};


void SmartPlayer::receiveCards(const std::vector<const Card*>&
                                                    cardsss){
    if (cardsss.size() == 7){
        cards.clear();
    }

    for (int i = 0; i < cardsss.size(); i++){
        cards.push_back(cardsss[i]);
    }
};



const Card * SmartPlayer::playCard(){
    vector<const Card*> array = array_can_play();
    int op_index = game()->activePlayerIndex() == 1 ? 0:1;
    bool is_another_player_have_two_cards = game()->
                numberOfCards()[op_index] <= 3 ? 1:0;



        if (is_another_player_have_two_cards){
            for(int i = static_cast<int>(array.size()) - 1; i >= 0;
```

```
                                                    i--){
        if(array[i]->value   == 10){
            cards.erase(cards.begin() +
                                    find_index(array[i]));
            return array[i];
        }
    }

    for(int i = static_cast<int>(array.size()) - 1;
                                    i >= 0 ; i--){
        if((array[i]->is_action()) ||
                                array[i]->is_wild()){
            cards.erase(cards.begin() +
                                find_index(array[i]));
            return array[i];
        }
    }

};

if(is_more_than_one_action_card(array)){
    cards.erase(cards.begin() + find_index(
                    array[array.size() - 1]));
    return array[array.size() - 1];
}

for(int i = static_cast<int>(array.size()) - 1; i >= 0;
                                                  i--){
    if(array[i]->color == game()->currentColor() &&
       array[i]->value < 10){
        cards.erase(cards.begin() + find_index(array[i]));
        return array[i];
    }
}

for(int i = static_cast<int>(array.size()) - 1; i >= 0 ;
                                                    i--){
    if(array[i]->color == return_max_cnt_color() &&
```

```
                array[i]->value == game()->topCard()->value)
            {
                cards.erase(cards.begin() + find_index(array[i]));
                return array[i];
            }
        }

        for(int i = static_cast<int>(array.size()) - 1; i >= 0 ;
                                                            i--){
            if(array[i]->value == game()->topCard()->value)
            {
                cards.erase(cards.begin() + find_index(array[i]));
                return array[i];
            }
        }




    cards.erase(cards.begin() + find_index(
                                    array[array.size() - 1]));
    return array[array.size() - 1];
}


bool SmartPlayer::drawAdditionalCard(const Card * additionalCard){
    if (additionalCard->is_wild()){
        return true;
    }

    if (additionalCard->color == this->game()->currentColor() ||
        additionalCard->value == this->game()->topCard()->value){
        return true;
    }

    cards.push_back(additionalCard);
    return false;
```

```
};


CardColor SmartPlayer::changeColor(){
    return return_max_cnt_color();
};

bool CompareCardsByValue_(const Card* a, const Card* b) {
    return a->value < b->value;
}

vector<const Card*> SmartPlayer::array_can_play(){
    vector<const Card*> array;

    for (int i = 0; i < cards.size(); i++){
        if ((cards[i]->color == game()->currentColor() &&
                            cards[i]->is_wild() == false) ||
            (cards[i]->value == game()->topCard()->value &&
                        game()->topCard()->is_wild() == false) ||
            cards[i]->value == 13)

        {
            array.push_back(cards[i]);
        }
    }

    if (array.size() == 0){
        for (int i = 0; i < cards.size(); i++){
            if (cards[i]->value == 14)
            {
                array.push_back(cards[i]);
            }
        }
    }
    sort(array.begin(), array.end(), CompareCardsByValue_);
    return array;
};
```

```
CardColor SmartPlayer::return_max_cnt_color(){
    vector<const Card*> red_arr;
    vector<const Card*> blue_arr;
    vector<const Card*> green_arr;
    vector<const Card*> yellow_arr;

    vector<const Card*> mx_size;

    for(int i = 0; i < cards.size(); i++){
        switch (cards[i]->color) {
            case Red:
                red_arr.push_back(cards[i]);
                break;
            case Blue:
                if (cards[i]->is_wild() == false){
                    blue_arr.push_back(cards[i]);
                }
                break;
            case Green:
                green_arr.push_back(cards[i]);
                break;
            case Yellow:
                yellow_arr.push_back(cards[i]);
                break;
        }
    }

    for (vector<const Card*> arr: {red_arr, blue_arr,
                                   green_arr, yellow_arr}) {
        if (arr.size() > mx_size.size()){
            mx_size = arr;
        }

    }

    if (mx_size.size() == 0){
        srand((unsigned) time(NULL));
```

```
        vector<CardColor> colors {Red, Blue, Green, Yellow};
        int x;
        while(true){
            x = rand() % 4;
            if (colors[x] != game()->currentColor()){
                break;
            }
        }
        return colors[x];
    }

    return mx_size[0]->color;
};



int SmartPlayer::find_index(const Card* temp){

    for (int i = 0; i < cards.size(); i++){
        if (cards[i]->value == temp->value &&
            cards[i]->color == temp->color){
            return i;
        }
    }

    return 0;
}

bool SmartPlayer::is_more_than_one_action_card(vector<const Card*>
                                                        array){
    bool is_more = false;
    int cnt = 0;
    CardColor temp;

    for(int i = static_cast<int>(array.size()) - 1; i >= 0;
                                                        i--){
        if (array[i]->is_action()){
            cnt++;
```

```
            temp = array[i]->color;
        }
    }

    if (cnt >= 2){
        is_more = true;
    }

    else if(cnt == 1 && array.size() > 1){
        for(int i = static_cast<int>(array.size()) - 1; i >= 0;
                                                        i--){
            if (array[i]->is_action() == false && array[i]->
                                            color == temp){
                is_more = true;
                break;
            }
        }
    }

    return is_more;
}
```