ЕМИНОБРНАУКИ РОССИИ

САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ «ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)

Кафедра МО ЭВМ

ОТЧЕТ

по лабораторной работе №1

по дисциплине «Объектно-ориентированное программирование»

Тема: Создание классов, конструкторов класса, методов класса, наследование

Студент гр. 8303	____________________	Деркач Н.В.
Преподаватель	____________________	Филатов А.Ю.

Санкт-Петербург

2020

Цель работы.

Научиться создавать классы и их конструкторы, реализовать методы классов и познакомиться с наследованием классов.

Ход работы.

1) Были разработаны и реализованы классы

   --- Класс игрового поля Field

   --- Набор классов юнитов

2) Было создано поле field класса Field прямоугольного размера с контролем максимального количества объектов на нем.
3) Добавлены методы добавления и удаления юнитов.
4) Написан конструктор копирования поля.
5) Реализованы 3 класса юнитов имеющих общий интерфейс.
6) Реализованы 2 класса юнитов для каждого из основных типов юнитов
7) Прописаны характеристики юнитов (здоровье, стоимость, сила)
8) Написан метод move, реализующий передвижение юнитов по полю

Выводы.

В ходе выполнения работы были созданы классы поля и юнитов и их конструкторы, реализовано наследование классов юнитов и методов классов.

ПРИЛОЖЕНИЕ А. ИСХОДНЫЙ КОД

```
#INCLUDE <QCOREAPPLICATION>
#include <QMediaPlayer>
#include <QDir>
#include <QUrl>
#include <iostream>
#include <cstdio>
#include <cstdlib>
#include <conio.h>

using namespace std;

#define COUNT_OF_UNITS_TYPE 6
#define UP_ARROW 72
#define LEFT_ARROW 75
#define DOWN_ARROW 80
#define RIGHT_ARROW 77

class Unit{
protected:
public:
    int attackRange;
    int cost;
    int hp;
    int force;
    char sym;
    struct id{                     //будет двузначным числом, указывающим положение
юнита в массиве,
        int code;                  //чтобы метод move мог легко достать из массива
нужного юнита
        int index;                 // и поменять ему координаты
    }id;
    int x, y;
    bool compFlag = 1;
    // virtual void interaction() = 0; // виртуальная функция для взаимодействия
юнитов с нейтральными объектами. В каждом классе отряда перегружу её

};

class NitralObject{
public:
    char sym;
};

class Stone: public NitralObject{
public:
    Stone(){
        this->sym = 's';
    }
};

class Gold: public NitralObject{
public:
    Gold(){
        this->sym = 'g';
    }
};

class ForceWell: public NitralObject{
```

```cpp
public:
    ForceWell(){
        this->sym = 'f';
    }
};

class LifeWell: public NitralObject{
public:
    LifeWell(){
        this->sym = 'l';
    }
};

/* взаимодействие с нейтральными объектами с помощью паттерна "Стратегия" */

class Compression{
public:
    virtual bool compress(Unit* unit) = 0;
   // virtual ~Compression();
};

class StoneCompression: public Compression{
public:
    bool compress(Unit *unit){
        return 0;
    }
};

class GoldCompression: public Compression{
public:
    bool compress(Unit* unit){
        return 1;
    }
};

class ForceWellCompression: public Compression{
public:
    bool compress(Unit* unit){
        if (unit->id.code == 0 || unit->id.code == 1){
            unit->force += 50;
        }
        if (unit->id.code == 1 || unit->id.code == 2){
            unit->force += 100;
        }
        if (unit->id.code == 5){
            unit->force += 1;
        }
        unit->compFlag = 0;
        return 0;
    }
};

class LifeWellCompression: public Compression{
public:
    bool compress(Unit* unit){
        if (unit->id.code == 0 || unit->id.code == 1){
            unit->hp += 100;
        }
        if (unit->id.code == 1 || unit->id.code == 2){
            unit->hp += 50;
        }
        if (unit->id.code == 5){
            unit->hp += 1;
        }
        unit->compFlag = 0;
        return 0;
    }
};

class Compressor{
private:
    Compression* p;
public:
    Compressor(Compression* comp) : p(comp) {}
    bool compress (Unit* unit){
        return p->compress(unit);
    }
    ~Compressor(){
        delete p;
    }
};

/* ландшафт и паттерн "Прокси" */

class Landscape{
public:
  //  friend class Forest;
  //  friend class Plain;
  //  friend class Swamp;
    char sym;
    virtual bool interaction(Unit* unit) = 0;
  //  virtual ~Landscape();
};

class Forest: public Landscape{

public:
    Forest(){
        this->sym = '|';
    }
    bool interaction(Unit* unit){
        if (unit->sym == 'G'){
            return 0;
        }
        return 1;
    }
};

class Plain: public Landscape{
public:
    Plain(){
        this->sym = ' ';
    }
    bool interaction (Unit* unit){
        if (unit->sym == 'G'){
            unit->force += 25;
            return 1;
        }
        return 1;
    }
};
```

```
class Swamp: public Landscape{
public:
    Swamp(){
        this->sym = '_';
    }
    bool interaction(Unit* unit){
        if (unit->sym == 'G'){
            unit->force -=25;
            return 1;
        }
        if (unit->sym == 'S'){
            return 0;
        }
        return 1;
    }
};

class Warrior: public Unit
{
private:

public:

};

class Mage: public Unit
{
public:

};

class Saboteur: public Unit
{
public:
};

class Gladiator: public Warrior{
public:
};

class Gunslinger: public Warrior{
public:
};

class Healer:public Mage{
public:
};

class Wizard:public Mage{
public:
};

class Jew:public Saboteur{
public:
};

class Kamikadze:public Saboteur{
public:
};
```

```
// юниты игрока 1

class Player1Gladiator: public Gladiator{
public:
    Player1Gladiator(int x, int y){
        this->id.code = 0;
        this->x = x;
        this->y = y;
        this->hp = 400;
        this->force = 50;
        this->attackRange = 1;
        this->sym = 'G';
        this->cost = 100;
    }
};

class Player1Gunslinger: public Gunslinger {
public:
    Player1Gunslinger(int x, int y){
        this->id.code = 1;
        this->x = x;
        this->y = y;
        this->hp = 250;
        this->force = 30;
        this->attackRange = 3;
        this->sym = 'S';
        this->cost = 125;
    }
};

class Player1Healer: public Healer {
public:
    Player1Healer(int x, int y){
        this->id.code = 2;
        this->x = x;
        this->y = y;
        this->hp = 100;
        this->force = 40;
        this->attackRange = 2;
        this->sym = 'H';
        this->cost = 150;
    }
};

class Player1Wizard: public Wizard {
public:
    Player1Wizard(int x, int y){
        this->id.code = 3;
        this->x = x;
        this->y = y;
        this->hp = 50;
        this->force = 100;
        this->attackRange = 3;
        this->sym = 'W';
        this->cost = 200;
    }
};

class Player1Jew: public Jew {
public:
    Player1Jew(int x, int y){
        this->id.code = 4;
        this->x = x;
        this->y = y;
        this->hp = 1;
        this->force = 1;
        this->attackRange = 1;
        this->sym = 'J';
        this->cost = 1000;
    }
};

class Player1Kamikadze: public Kamikadze {
public:
    Player1Kamikadze(int x, int y){
        this->id.code = 5;
        this->x = x;
        this->y = y;
        this->hp = 50;
        this->force = 1000;
        this->attackRange = 3;
        this->sym = 'K';
        this->cost = 50;
    }
};

// юниты игрока 2

class Player2Gladiator: public Gladiator{
public:
    Player2Gladiator(int x, int y){
        this->id.code = 0;
        this->x = x;
        this->y = y;
        this->hp = 400;
        this->force = 50;
        this->attackRange = 1;
        this->sym = 'G';
        this->cost = 100;
    }
};

class Player2Gunslinger: public Gunslinger {
public:
    Player2Gunslinger(int x, int y){
        this->id.code = 1;
        this->x = x;
        this->y = y;
        this->hp = 250;
        this->force = 30;
        this->attackRange = 3;
        this->sym = 'S';
        this->cost = 125;
    }
};

class Player2Healer: public Healer {
public:
    Player2Healer(int x, int y){
        this->id.code = 2;
        this->x = x;
        this->y = y;
        this->hp = 100;
        this->force = 40;
        this->attackRange = 2;
        this->sym = 'H';
        this->cost = 150;
    }
};

class Player2Wizard: public Wizard {
public:
    Player2Wizard(int x, int y){
        this->id.code = 3;
        this->x = x;
        this->y = y;
        this->hp = 50;
        this->force = 100;
        this->attackRange = 3;
        this->sym = 'W';
        this->cost = 200;
    }
};

class Player2Jew: public Jew {
public:
    Player2Jew(int x, int y){
        this->id.code = 4;
        this->x = x;
        this->y = y;
        this->hp = 1;
        this->force = 1;
        this->attackRange = 1;
        this->sym = 'J';
        this->cost = 1000;
    }
};

class Player2Kamikadze: public Kamikadze {
public:
    Player2Kamikadze(int x, int y){
        this->id.code = 5;
        this->x = x;
        this->y = y;
        this->hp = 50;
        this->force = 1000;
        this->attackRange = 3;
        this->sym = 'K';
        this->cost = 50;
    }
};

/* отдел для абстрактной фабрики */

class ArmyFactory {
public:
    virtual Gladiator* createGladiator(int x, int y) = 0;
    virtual Gunslinger* createGunslinger(int x, int y) = 0;
    virtual Healer* createHealer(int x, int y) = 0;
    virtual Wizard* createWizard(int x, int y) = 0;
    virtual Jew* createJew(int x, int y) = 0;
    virtual Kamikadze* createKamikadze(int x, int y) = 0;
   // virtual ~ArmyFactory();
};

class Player1Factory: public ArmyFactory{
```

```
public:
    Gladiator * createGladiator(int x, int y) {
        return new Player1Gladiator(x, y);
    }
    Gunslinger * createGunslinger(int x, int y){
        return new Player1Gunslinger(x, y);
    }
    Healer * createHealer(int x, int y){
        return new Player1Healer(x, y);
    }
    Wizard * createWizard(int x, int y){
        return new Player1Wizard(x, y);
    }
    Jew * createJew(int x, int y){
        return new Player1Jew(x, y);
    }
    Kamikadze * createKamikadze(int x, int y){
        return new Player1Kamikadze(x, y);
    }
};

class Player2Factory: public ArmyFactory{
public:
    Gladiator * createGladiator(int x, int y) {
        return new Player2Gladiator(x, y);
    }
    Gunslinger * createGunslinger(int x, int y){
        return new Player2Gunslinger(x, y);
    }
    Healer * createHealer(int x, int y){
        return new Player2Healer(x, y);
    }
    Wizard * createWizard(int x, int y){
        return new Player2Wizard(x, y);
    }
    Jew * createJew(int x, int y){
        return new Player2Jew(x, y);
    }
    Kamikadze * createKamikadze(int x, int y){
        return new Player2Kamikadze(x, y);
    }
};

/* отдел базы */

class Base{
public:
    int hp;
    int x, y;
    int countOfUnits = 0;
    int gold = 500;
    char sym = 'B';
    Unit* createUnit(int code, ArmyFactory& factory){
        if (code == 0){
            Unit* unit = factory.createGladiator(x+1, y);
            if (unit->cost <= this->gold)
            {
                this->gold -= unit->cost;
                countOfUnits++;
                return unit;
            }
            else{
```

```
            cout << "\nNot enough gold\n";
            delete unit;
            return nullptr;
        }
    }
    if (code == 1){
        Unit* unit = factory.createGunslinger(x+1, y);
        if (unit->cost <= this->gold)
        {
            this->gold -= unit->cost;
            countOfUnits++;
            return unit;
        }
        else{
            cout << "\nNot enough gold\n";
            delete unit;
            return nullptr;
        }

    }
    if (code == 2){
        Unit* unit = factory.createHealer(x+1, y);
        if (unit->cost <= this->gold)
        {
            this->gold -= unit->cost;
            countOfUnits++;
            return unit;
        }
        else{
            cout << "\nNot enough gold\n";
            delete unit;
            return nullptr;
        }
    }
    if (code == 3){
        Unit* unit = factory.createWizard(x+1, y);
        if (unit->cost <= this->gold)
        {
            this->gold -= unit->cost;
            countOfUnits++;
            return unit;
        }
        else{
            cout << "\nNot enough gold\n";
            delete unit;
            return nullptr;
        }
    }
    if (code == 4){
        Unit* unit = factory.createJew(x+1, y);
        if (unit->cost <= this->gold)
        {
            this->gold -= unit->cost;
            countOfUnits++;
            return unit;
        }
        else{
            cout << "\nNot enough gold\n";
            delete unit;
            return nullptr;
        }
    }
    if (code == 5){
```

```
                Unit* unit = factory.createKamikadze(x+1, y);
                if (unit->cost <= this->gold)
                {
                    this->gold -= unit->cost;
                    countOfUnits++;
                    return unit;
                }
                else{
                    cout << "\nNot enough gold\n";
                    delete unit;
                    return nullptr;
                }
            }
        }
};

/* отдел поля */

struct Cell{
    Unit* unit = nullptr;
    Base* base = nullptr;
    NitralObject* nObject = nullptr;
    Landscape* landScape = nullptr;
};

class Field{
private:
    int countOfDiffUnits = COUNT_OF_UNITS_TYPE;
    int rows;
    int columns;
    int countOfObjects;
    int maxNumOfObjects;
    Cell*** field;
    Unit*** units;                              // либо пофиксить эту матрицу и
сделать её нормальным стеком, либо мараться с каждой функцией, ищущей юнита
public:
    Field(int M, int N, int maxNumOfObjects){
        rows = M;
        columns = N;
        this->maxNumOfObjects = maxNumOfObjects;
        field = new Cell**[this->rows];
        for (int i = 0; i < rows; i++){
            field[i] = new Cell*[columns];
            for(int j = 0; j < columns; j++){
                field[i][j] = new Cell;
                field[i][j]->base = nullptr;
                field[i][j]->unit = nullptr;
                field[i][j]->nObject = nullptr;
                if (i == 3 && j == 7){
                    field[i][j]->nObject = new Stone();
                }
                if (i == 6 && j == 4){
                    field[i][j]->nObject = new Gold();
                }
                if (i == 11 && j == 15){
                    field[i][j]->nObject = new LifeWell();
                }
                if (i == 15 && j == 5){
                    field[i][j]->nObject = new ForceWell();
                }
             //   if ((i != 6 && j != 4) || (i != 3 && j != 7) || (i != 9 && j !
= 5) || (i != 15 && j != 10)){
```

```
            //    }
                if (i > 3 && i < 9){             // потом сделаю рандомизированно
                    if (j > columns/2-3 && j < columns/2 + 3){
                        field[i][j]->landScape = new Forest;
                        continue;
                    }
                }
                if (i > rows/2+2 && i < rows/2 + 8){
                    if (j > columns/2-1 && j < columns/2 + 5){
                        field[i][j]->landScape = new Swamp;
                        continue;
                    }
                }
                field[i][j]->landScape = new Plain;
            }
        }
        units = new Unit**[countOfDiffUnits];
        for(int i = 0; i< countOfDiffUnits; i++){
            units[i] = new Unit*[maxNumOfObjects];
            for(int j = 0; j < maxNumOfObjects; j++){
                units[i][j] = new Unit;
                units[i][j] = nullptr;
            }
        }
    }

    Field(const Field& field) : rows(field.rows), columns(field.columns),
countOfObjects(field.countOfObjects), maxNumOfObjects(field.maxNumOfObjects){
// конструктор копирования
        cout << "Call the construktor of copy\n";
        this->field = new Cell**[rows];
        for (int i = 0; i < field.rows; i++){
            this->field[i] = new Cell*[columns];
            for(int j = 0; j < columns; j++){
                this->field[i][j] = new Cell;
                if (field.field[i][j]->landScape){
                    if (field.field[i][j]->landScape->sym == '|'){
                        this->field[i][j]->landScape = new Forest;
                    }
                    if (field.field[i][j]->landScape->sym == '_'){
                        this->field[i][j]->landScape = new Plain;
                    }
                    if (field.field[i][j]->landScape->sym == ' '){
                        this->field[i][j]->landScape = new Swamp;
                    }
                }
                if (field.field[i][j]->base){
                    this->field[i][j]->base = new Base;
                    this->field[i][j]->base->hp = field.field[i][j]->base->hp;
                    this->field[i][j]->base->sym = field.field[i][j]->base->sym;
                }
                if (field.field[i][j]->nObject){
                    this->field[i][j]->nObject = new NitralObject;
                    this->field[i][j]->nObject->sym = field.field[i][j]-
>nObject->sym;
                }
                if (field.field[i][j]->unit){
                    this->field[i][j]->unit = new Unit;
                    this->field[i][j]->unit->x = field.field[i][j]->unit->x;
                    this->field[i][j]->unit->y = field.field[i][j]->unit->y;
                    this->field[i][j]->unit->id.code = field.field[i][j]->unit-
>id.code;
```

```
                    this->field[i][j]->unit->id.index = field.field[i][j]->unit-
>id.index;
                    this->field[i][j]->unit->sym = field.field[i][j]->unit->sym;
                }
                this->field[i][j]->landScape->sym = field.field[i][j]-
>landScape->sym;
            }
        }

        this->units = new Unit**[COUNT_OF_UNITS_TYPE];
        for (int i = 0; i < COUNT_OF_UNITS_TYPE; i++){
            this->units[i] = new Unit*[maxNumOfObjects];
            for (int j = 0; j < maxNumOfObjects; j++){
                if (field.units[i][j]){
                    this->units[i][j] = new Unit;
                    this->units[i][j]->x = field.units[i][j]->x;
                    this->units[i][j]->y = field.units[i][j]->y;
                    this->units[i][j]->id.code = field.units[i][j]->id.code;
                    this->units[i][j]->id.index = field.units[i][j]->id.index;
                    this->units[i][j]->sym = field.units[i][j]->sym;
                }
            }
        }


    }

    Field& operator= (const Field& field){          // оператор присваивания для
копирования
        if (&field == this)
            return *this;
        cout << "new operator =\n";
        for (int i = 0; i < this->rows; i++){
            for (int j = 0; j < this->columns; j++){
                if (this->field[i][j]->base)
                    delete this->field[i][j]->base;
                if (this->field[i][j]->unit)
                    delete this->field[i][j]->unit;
                if(this->field[i][j]->nObject)
                    delete this->field[i][j]->nObject;
            //    if (this->field[i][j]->landScape)
              //      delete this->field[i][j]->landScape;
            }
            delete [] this->field[i];
        }
        for (int i = 0; i < COUNT_OF_UNITS_TYPE; i++){
            delete [] this->units[i];
        }
        delete[] this->field;
        delete[] this->units;
        this->field = new Cell**[rows];
        for (int i = 0; i < field.rows; i++){
            this->field[i] = new Cell*[columns];
            for(int j = 0; j < columns; j++){
                this->field[i][j] = new Cell;
                if (field.field[i][j]->landScape){
                    if (field.field[i][j]->landScape->sym == '|'){
                        this->field[i][j]->landScape = new Forest;
                    }
                    if (field.field[i][j]->landScape->sym == '_'){
                        this->field[i][j]->landScape = new Plain;
                    }
                    if (field.field[i][j]->landScape->sym == ' '){
                        this->field[i][j]->landScape = new Swamp;
                    }
                }
                if (field.field[i][j]->base){
                    this->field[i][j]->base = new Base;
                    this->field[i][j]->base->hp = field.field[i][j]->base->hp;
                    this->field[i][j]->base->sym = field.field[i][j]->base->sym;
                }
                if (field.field[i][j]->nObject){
                    this->field[i][j]->nObject = new NitralObject;
                    this->field[i][j]->nObject->sym = field.field[i][j]-
>nObject->sym;
                }
                if (field.field[i][j]->unit){
                    this->field[i][j]->unit = new Unit;
                    this->field[i][j]->unit->x = field.field[i][j]->unit->x;
                    this->field[i][j]->unit->y = field.field[i][j]->unit->y;
                    this->field[i][j]->unit->id.code = field.field[i][j]->unit-
>id.code;
                    this->field[i][j]->unit->id.index = field.field[i][j]->unit-
>id.index;
                    this->field[i][j]->unit->sym = field.field[i][j]->unit->sym;
                }
                this->field[i][j]->landScape->sym = field.field[i][j]-
>landScape->sym;
            }
        }
        this->units = new Unit**[COUNT_OF_UNITS_TYPE];
        for (int i = 0; i < COUNT_OF_UNITS_TYPE; i++){
            this->units[i] = new Unit*[maxNumOfObjects];
            for (int j = 0; j < maxNumOfObjects; j++){
                if (field.units[i][j]){
                    this->units[i][j] = new Unit;
                    this->units[i][j]->x = field.units[i][j]->x;
                    this->units[i][j]->y = field.units[i][j]->y;
                    this->units[i][j]->id.code = field.units[i][j]->id.code;
                    this->units[i][j]->id.index = field.units[i][j]->id.index;
                    this->units[i][j]->sym = field.units[i][j]->sym;
                }
            }
        }
        return *this;
    }

    Field(Field&& field) : rows(field.rows), columns(field.columns),
countOfObjects(field.countOfObjects), maxNumOfObjects(field.maxNumOfObjects){
// конструктор перемещения
        cout << "Call the construktor of relocation\n";
        field.rows = 0;
        field.columns = 0;
        field.countOfObjects = 0;
        field.maxNumOfObjects = 0;
        this->field = field.field;
        for (int i = 0; i < rows; i++){
            for(int j = 0; j < columns; j++){
                field.field[i][j]->landScape = nullptr;
                if (field.field[i][j]->base){
                    field.field[i][j]->base = nullptr;
                }
                if (field.field[i][j]->nObject){
                    field.field[i][j]->nObject = nullptr;
                }
                if (field.field[i][j]->unit){
                    field.field[i][j]->unit = nullptr;
                }
            }
            field.field = nullptr;
        }

        this->units = field.units;
        for (int i = 0; i < COUNT_OF_UNITS_TYPE; i++){
            for (int j = 0; j < maxNumOfObjects; j++){
                if (field.units[i][j]){
                    field.units[i][j] = nullptr;
                }
            }
        }
        field.units = nullptr;
    }

    Field& operator=(Field&& field){            // оператор присваивания для
перемещения
        if (&field == this)
            return *this;
        cout << "new relocation operator =\n";
        for (int i = 0; i < this->rows; i++){
            for (int j = 0; j < this->columns; j++){
                if (this->field[i][j]->base)
                    delete this->field[i][j]->base;
                if (this->field[i][j]->unit)
                    delete this->field[i][j]->unit;
                if(this->field[i][j]->nObject)
                    delete this->field[i][j]->nObject;
             //   if (this->field[i][j]->landScape)
               //     delete this->field[i][j]->landScape;
            }
            delete [] this->field[i];
        }
        for (int i = 0; i < COUNT_OF_UNITS_TYPE; i++){
            delete [] this->units[i];
        }
        delete[] this->field;
        delete[] this->units;
        field.rows = 0;
        field.columns = 0;
        field.countOfObjects = 0;
        field.maxNumOfObjects = 0;
        this->field = field.field;
        for (int i = 0; i < rows; i++){
            for(int j = 0; j < columns; j++){
                field.field[i][j]->landScape = nullptr;
                if (field.field[i][j]->base){
                    field.field[i][j]->base = nullptr;
                }
                if (field.field[i][j]->nObject){
                    field.field[i][j]->nObject = nullptr;
                }
                if (field.field[i][j]->unit){
                    field.field[i][j]->unit = nullptr;
                }
            }
            field.field = nullptr;
        }
        this->units = field.units;
        for (int i = 0; i < COUNT_OF_UNITS_TYPE; i++){
            for (int j = 0; j < maxNumOfObjects; j++){
                if (field.units[i][j]){
                    field.units[i][j] = nullptr;
                }
            }
        }
        field.units = nullptr;
        return *this;
    }

    int checkEmptySpace(int code){
        int in = 0;
        while(units[code][in]){
            in++;
        }
        return in;
    }

    void setUnit(Unit* unit){
        if (unit->x < this->rows && unit->y < this->columns && this-
>countOfObjects < this->maxNumOfObjects){
            int in = checkEmptySpace(unit->id.code);
            units[unit->id.code][in] = unit;
            unit->id.index = in;
            field[unit->y][unit->x]->unit = unit;
        }
    }

    void deleteUnit(Unit* unit){
        field[rows/2][0]->base->gold += unit->cost/2;
        field[rows/2][0]->base->countOfUnits--;
        units[unit->id.code][unit->id.index] = nullptr;
        field[unit->y][unit->x]->unit = nullptr;
        delete unit;
    }

    Unit* changeUnit (Unit* unit){                      // need to be fixed
        for (int i = 0; i < 6; i++){
            for (int j = 0; j < maxNumOfObjects; j++){
                if (units[i][j] && units[i][j] != unit){
                    return units[i][j];
                }
            }
        }
    }

    void move(Unit* unit, int x, int y){                        // передвижение
юнитов
        if (field[y][x]->unit){
            return;
        }
        else{
            if (field[y][x]->landScape->sym != field[unit->y][unit->x]-
>landScape->sym){
                if (!field[y][x]->landScape->interaction(unit)){
                    return;
                }
            }
            if (field[y][x]->nObject){
                if (field[y][x]->nObject->sym == 'f'){
                    if (unit->compFlag){
                        Compressor* p = new Compressor(new
ForceWellCompression);
```

```
                        p->compress(unit);
                    }
                    return;
                }
                if (field[y][x]->nObject->sym == 'l'){
                    if (unit->compFlag){
                        Compressor* p = new Compressor(new LifeWellCompression);
                        p->compress(unit);
                    }
                    return;
                }
                if (field[y][x]->nObject->sym == 's'){
                    return;
                }
                if (field[y][x]->nObject->sym == 'g'){
                    NitralObject* g = field[y][x]->nObject;
                    field[y][x]->nObject = nullptr;
                    delete g;
                    field[rows/2][0]->base->gold += 200;
                }
            }
            field[unit->y][unit->x]->unit = nullptr;
            units[unit->id.code][unit->id.index]->x = x;
            units[unit->id.code][unit->id.index]->y = y;
            unit->x = x;
            unit->y = y;
            field[y][x]->unit = unit;
        }
    }

    void createBase(int x, int y){
        field[y][x]->base = new Base;
        field[y][x]->base->x = x;
        field[y][x]->base->y = y;
    }

    void setUnitFromBase(ArmyFactory& factory, int x, int y, int code){
        Unit* unit;
        unit = field[y][x]->base->createUnit(code, factory);
        if (unit == nullptr){
            return;
        }
        field[unit->y][unit->x]->unit = unit;
        int in = checkEmptySpace(code);
        units[code][in] = unit;
    }

    Unit* getLastUnit(int code){     // чето ломается
        Unit* unit = new Unit;
        int i = 0;
        while (units[code][i+1]){
            i++;
        }
        if (units[code][i]){
            unit = units[code][i];
            return unit;
        }
        else {
            return nullptr;
        }
        /*
        int code1 = code;                     // второй вариант
        Unit* unit = new Unit;
        for (int i = 0; i <= code1; i++){
            for (int j = 0; j < checkEmptySpace(code1); j++){
                if (units[code1][i]){
                    unit = units[code1][i];
                    return unit;
                }
            }
        }
        return nullptr;
        */
    }

    void printField(Unit* unit){
        for (int i = 0; i < rows; i++){
            for (int j = 0; j < columns; j++){
                if (field[i][j]->unit){
                    cout << field[i][j]->unit->sym;
                    continue;
                }
                if (field[i][j]->base){
                    cout << field[i][j]->base->sym;
                    continue;
                }
                if (field[i][j]->nObject){
                    cout << field[i][j]->nObject->sym;
                    continue;
                }
                if(field[i][j]->landScape){
                    cout << field[i][j]->landScape->sym;
                    continue;
                }
            }
            cout << '\n';
        }
        cout << '\n';
        cout << "Gold of your base: " << field[rows/2][0]->base->gold << " | 
Count of your units: " << field[rows/2][0]->base->countOfUnits;
        cout << '\n';
        if (unit)
            cout << "\nForce of current unit: " << unit->force << "\t\tHp of 
current unit: " << unit->hp << endl;

    }
    ~Field(){
        for (int i = 0; i < this->rows; i++){
            for (int j = 0; j < this->columns; j++){
                if (this->field[i][j]->base)
                    delete this->field[i][j]->base;
                if (this->field[i][j]->unit)
                    delete this->field[i][j]->unit;
                if(this->field[i][j]->nObject)
                    delete this->field[i][j]->nObject;
              //  if (this->field[i][j]->landScape)
                //    delete this->field[i][j]->landScape;
            }
            delete [] this->field[i];
        }
        for (int i = 0; i < COUNT_OF_UNITS_TYPE; i++){
            delete [] this->units[i];
        }
        delete[] this->field;
        delete[] this->units;
    }
};

int main(int argc, char *argv[])
{
    QCoreApplication a(argc, argv);

    int rows, columns;
    cout << "Set the field size(rows and columns across the space): " << endl;
    cin >> rows;
    cin >> columns;
    Field gameField(rows, columns, rows*columns);
    Unit* unit = nullptr;
    gameField.createBase(0, rows/2);
    int keyGet;
    /* музло */

    /*

    QMediaPlayer *chelhok;
    chelhok =new QMediaPlayer;
    const QString putii = "D:/proga/C++/OOP/OOP/battle.mp3";
  //  const QString putii = "D:/proga/C++/OOP/OOP/peace.mp3";
  //  const QString putii = "D:/proga/C++/OOP/OOP/ww1.mp3";
    QUrl adresok(QFileInfo(putii).absoluteFilePath());
    chelhok->setMedia(adresok);
    chelhok->setVolume(50);
    chelhok->play();

    */

    do{
        system("cls");
        cout << "Use 'u' to create some unit, use 'f' to delete chosen unit, use
arrows to move last unit and use 'q' to quit\n\n";
        if (unit)
            gameField.printField(unit);
        keyGet = getch();
       // if (keyGet == 'b'){
         //   gameField.createBase(columns+2, rows/2);
           // continue;
       // }

        if (keyGet == 'u'){
            cout << "choose the type of unit (from 0 to 5):" << endl;
            int code;
            cin >> code;
            if(code < 0 || code > 5){
                cout << "Wrong, moron. Can you read at least, are you? I said
FROM 0 TO 5. Try again:" << endl;
                cin >> code;
            }
            if(code < 0 || code > 5){
                cout << "Okey, I just give up, you are so stupid donkey, that it
actually look like harassment from my side.\n I'll write instead of you, don't
please me\n" << endl;
                code = 4;
            }
            Player1Factory factory;
            gameField.setUnitFromBase(factory, 0, rows/2, code);
```

```
                if(gameField.getLastUnit(code)){
                    unit = gameField.getLastUnit(code);
                    continue;
                }
                else {
                    continue;
                }
            }
            if (keyGet == 9){
                unit = gameField.changeUnit(unit);
            }
            if (keyGet == 'f'){
                Unit* unit2;
                unit2 = unit;
                unit = gameField.changeUnit(unit2);
                gameField.deleteUnit(unit2);
            }
            if (keyGet == 224){
                keyGet = getch();
                switch(keyGet){
                case UP_ARROW:
                    gameField.move(unit, unit->x, unit->y-1);
                    break;
                case DOWN_ARROW:
                    gameField.move(unit, unit->x, unit->y+1);
                    break;
                case LEFT_ARROW:
                    gameField.move(unit, unit->x-1, unit->y);
                    break;
                case RIGHT_ARROW:
                    gameField.move(unit, unit->x+1, unit->y);
                    break;
                default:
                    break;
                }
            }

        }while (keyGet != 'q');
       // delete chelhok;
        cout << "\nGame Over!";
        return a.exec();
}
```