МИНОБРНАУКИ 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 классов, методов классов; наследование

Студентка гр. 8383 _____ Максимова А.А.

Преподаватель _____ Жангиров Т.Р.

Санкт-Петербург
2020

Цель работы.

Разработать и реализовать набор классов: класс игрового поля, набор классов юнитов.

Постановка задачи.

Игровое поле является контейнером для объектов представляющим прямоугольную сетку. Основные требования к классу игрового поля:

- Создание поля произвольного размера
- Контроль максимального количества объектов на поле
- Возможность добавления и удаления объектов на поле
- Возможность копирования поля (включая объекты на нем)
- Для хранения запрещается использовать контейнеры из stl

Юнит является объектов, размещаемым на поля боя. Один юнит представляет собой отряд. Основные требования к классам юнитов:

- Все юниты должны иметь как минимум один общий интерфейс
- Реализованы 3 типа юнитов (например, пехота, лучники, конница)
- Реализованы 2 вида юнитов для каждого типа(например, для пехоты могут быть созданы мечники и копейщики)
- Юниты имеют характеристики, отражающие их основные атрибуты, такие как здоровье, броня, атака.
- Юнит имеет возможность перемещаться по карте

Основная идея игры

Игра рассчитана на двух игроков, каждый из которых может выбрать команду (тип юнитов), за которую будет играть (Монстры или Герои). Также есть нейтральная команда, то есть та, которой не может управлять не один из игроков (Люди).

Цель Монстров: истребить всех людей на игровом поле.

Цель Героев: истребить всех монстров, до тех пор, пока они не истребили людей.

Основные классы, реализованные в программе

1. Class GameField

Предназначен для создания контейнера и работы с ним (добавление, перемещение, удаление юнитов).

2. Class Object

Абстрактный класс, от которого на данный момент наследуется только класс юнитов.

3. Class Unit

Абстрактный класс, от которого наследуются классы разных типов юнитов.

4. Class Dialog

Необходим для вывода текстовой информации (правила, описание персонажей).

5. Class FirstGamer

Класс для взаимодействия с игроком (пока с одним игроком).

6. Class AbstractFactory

Класс, для создания юнитов.

Выполнение требований к классу игрового поля.

1. *Создание поля произвольного размера*

В данной работе игровое поле представляет собой двумерный динамический массив указателей на объект.

Для работы с полем был создан отдельный класс GameField, в котором реализуется создание контейнера, а также методы для работы с ним.

Для инициализации полей класса были написаны два конструктора.

- Конструктор по умолчанию, не принимающий аргументов:

```
GameField::GameField()
{
    height = 10;
    width = 10;
    step = 1;
    maxUnits = 6;
}
```

Рисунок 1 - Конструктор по умолчанию

- Конструктор с параметрами:

```
GameField::GameField(int height, int width) : GameField() //делегирующие конструкторы
{
    if (height <= 3 || width <= 3) {
        std::cout << "Ошибка: неверно указан размер поля!\nСоздано поле стандартного размера.\n";
    }
    else {
        this->height = height;
        this->width = width;
        step = 1;
        if (width >= height && (int)((width / 2) * (0.2)) > step && height >= (width / 2)) {
            step = (int)((width / 2) * (0.2));
        }
        maxUnits = 6 * step;
    }
    CreateField();
}
```

Рисунок 2 - Конструктор с параметрами

При создании экземпляра класса, вызывается конструктор, параметры которого определяет пользователь: `int height` – высота, `int width` – ширина игрового поля.

Несмотря на требование создания поля произвольного размера, существуют значения, по которым невозможно его создать (отрицательные значения, и значения меньше 4). Поэтому из конструктора с параметрами вызывается конструктор по умолчанию, который инициализирует поля класса стандартными значениями, и в случае некорректности введенных данных пользователем, значения полей не изменяются на некорректные (делегирование конструкторов).

Затем из конструктора с параметрами вызывается метод создания и инициализации двумерного динамического массива указателей:

```
void GameField::CreateField() {
    field = new Object * *[height];
    for (int i = 0; i < height; i++) {
        field[i] = new Object * [width];
        for (int j = 0; j < width; j++) {
            field[i][j] = nullptr;
        }
    }
}
```

Рисунок 3 - Метод создания и инициализации контейнера

2. Контроль максимального количества объектов на поле

На данный момент в программе учитывается только максимальное количество юнитов, которое возможно разместить на поле для каждой команды, так как других объектов пока нет (возможно позже этот параметр будет пересмотрен).

Количество конкретных юнитов в команде ограничивается целочисленным значением `maxUnits` по умолчанию равным 6, в зависимости от ширины поля это значение может изменяться (для поля ширины 20: `maxUnits` = 12, для поля 30: равно 18), что определяется с помощью формул:

```
step = (int)((width / 2) * (0.2));
maxUnits = 6 * step;
```

где `step` - переменная, для определения шага юнита при перемещении.

Третий тип (class People), предназначенный для добавления юнитов по умолчанию, в любом месте поля, ограничивается удвоенным значением `maxUnits`.

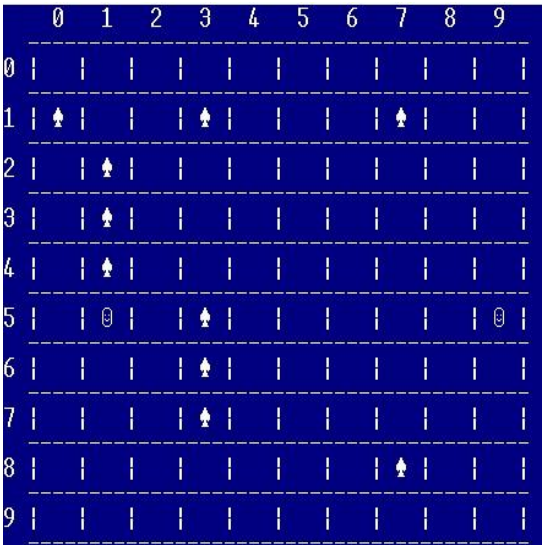


Рисунок 4 - Пример расстановки конкретных юнитов class People, число которых не превышает maxUnits*2

3. Возможность добавления и удаления объектов на поле

Добавление юнита на поле сопровождается его созданием. Виды Monster и Hero добавляются игроками.

1) Игрок вводит команду добавления юнита: три числа, первое - вид юнита (число от одного до четырех), координаты его расположения на поле (x; y)

2) Таким образом, команда вызова метода добавления выглядит так:

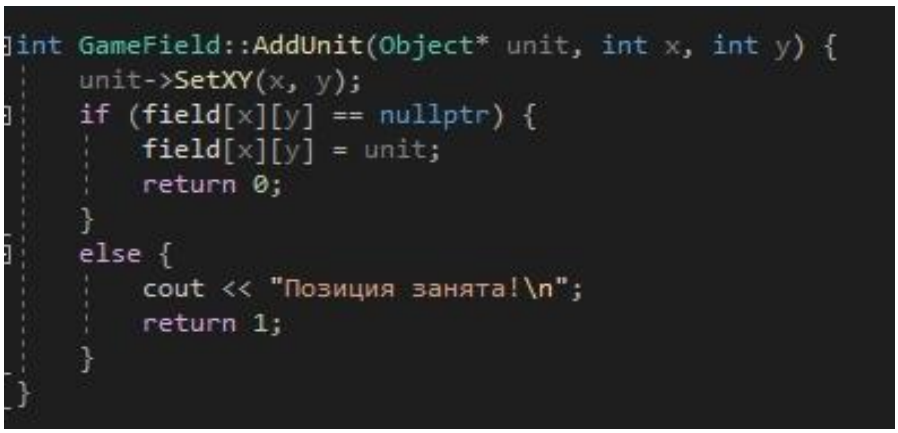
```
field->AddUnit(field->Create('m', 1), x, y));
```

3) Метод `CreateUnit` принимает тип и вид юнита, который нужно создать и возвращает указатель на него. Метод обращается к конкретным фабрикам класса Абстрактная фабрика, описанный в след. разделах.

```
Unit* GameField::CreateUnit(char type, int view) { //man отдельно
    ConcreteFactoryMonster monster;
    ConcreteFactoryHero hero;
    ConcreteFactoryPeople people;
    switch (type) {
    case 'm':
        if (view == 1 || view == 2) return monster.CreateUnit(view);
        break;
    case 'h':
        if (view == 3 || view == 4)  return hero.CreateUnit(view);
        break;
    case 'p':
        if (view == 5 || view == 6)  return people.CreateUnit(view);
        break;
    default:
        cout << "Ошибка: " << view << " <-- такого вида не существует!\n";
        break;
    }
}
```

Рисунок 5 - Получение указателя на юнита

4) Метод `AddUnit` принимает указатель на объект, а также координаты x и y, куда должен быть добавлен юнит. Проверяется, чтобы юнит был помещен на незанятую позицию поля.



```
int GameField::AddUnit(Object* unit, int x, int y) {
    unit->SetXY(x, y);
    if (field[x][y] == nullptr) {
        field[x][y] = unit;
        return 0;
    }
    else {
        cout << "Позиция занята!\n";
        return 1;
    }
}
```

Рисунок 6 - Добавление юнита на поле

5) В классе FirstGamer, из которого осуществляется вызов метода добавления юнитов, предусмотрен метод проверки на некорректные значения bool FirstGamer::CaseError(int command), так в случае если игрок попробует добавить юнита не на свою часть поля, или введет недоступную для него команду добавления, то есть попробует добавить юнита соперника, его команда не будет выполнена, а игрок будет предупрежден.

Добавление юнитов типа People происходит по умолчанию, поэтому первые два пункта заменяются методом:

```
void FirstGamer::AddPeople() {
    cout << "Добавление персонажей по умолчанию\n";
    int view;
    int i = field->GetMaxUnits() * 2;
    srand(time(NULL));
    while (i != 0) {
        view = rand() % 2 + 5; //5 или 6
        x = rand() % field->GetHeight();
        y = rand() % field->GetWidth();
        if (field->AddUnit(field->CreateUnit('p', view), x, y)) {
            i++; //Персонаж не создан!
        }
        else {
            if (view == 5) cout << "Создан охотник\n";
            else cout << "Создан человек\n";
        }
        i--;
    }
    field->PrintCurField();
    CaseAddCommand();
}
```

Рисунок 7 - Метод для вызова добавления юнита по случайно сгенерированным координатам

Удаление юнитов также осуществляется по введенным пользователем координатам x и y, а потому проверяется наличие объекта в заданной позиции, на случая ошибки пользователя. После удаления объекта, позиции присваивается значение nullptr.

```
int GameField::DeleteUnit(int x, int y) {
    if (field[x][y] != nullptr) {
        delete field[x][y];
        field[x][y] = nullptr;
        return 0;
    }
    else {
        cout << "На позиции нет юнита для удаления!\n";
        return 1;
    }
}
```

Рисунок 8 - Удаление юнитов с поля

Вызов метода удаления осуществляется из метода:

```
void FirstGamer::CaseDeleteUnit() { //возможно и не нужна, но пока для приме
    exit = 1;
    while (exit) {
        cout << "Введите координаты юнита, которого хотите удалить.\n";
        cin >> x >> y;
        while (!CaseError() || field->DeleteUnit(x, y)) {
            cout << "Введите снова:\n";
            cin >> x >> y;
        }
        cout << "Юнит удален!\n";
        field->PrintCurField();
        cout << "Введите 0, если хотите прекратить удаление.\n";
        cout << "Введите 1, если хотите продолжить удаление.\n";
        cin >> exit;
    }
}
```

Рисунок 9 - Метод для считывания координат удаляемого юнита

описанного в классе FirstGamer, сопровождаемого обработкой ошибок, и продолжающего свою работу до тех пор, пока игрок не введет значение 0.

4. Возможность копирования поля

Для реализации копирования поля, вместе с объектами, расположенными на нем, был написан конструктор копирования.

```
GameField::GameField(const GameField& other) {
    this->height = other.height;
    this->width = other.width;
    this->step = other.step;

    this->field = new Object**[other.height];
    for (int i = 0; i < other.height; i++) {
        this->field[i] = new Object * [other.width];
        for (int j = 0; j < other.width; j++) {
            this->field[i][j] = other.field[i][j];
        }
    }
}
```

Рисунок 10 - Конструктор копирования

Данный конструктор, создает новый объект через копирование уже существующего объекта. Принимает константную ссылку, `const` – так как ничего изменять в копии не нужно. Вызов конструктора используется в методе

```
void FirstGamer::SaveField() {
    GameField copy(*field);
    copy.PrintCurField();
    cout << "Игровое поле сохранено.\n";
}
```

Рисунок 11 - Вызов конструктора копирования

для сохранение игрового поля игроком.

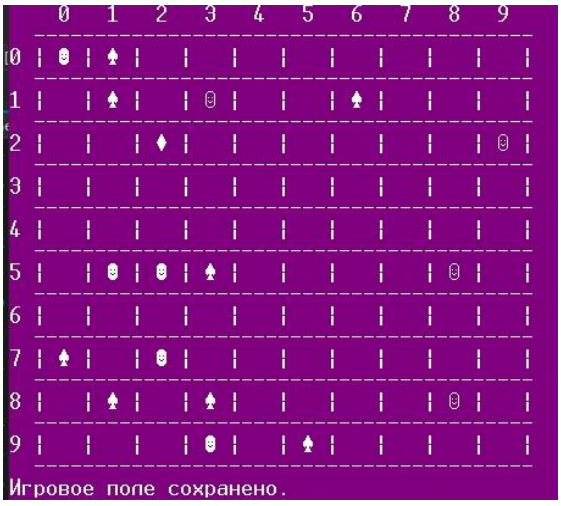


Рисунок 12 - Вывод копии игрового поля

5. Дополнительные методы

- Метод печати поля `void GameField::PrintCurField()`, с рисованием "сетки" и нумерацией строк и столбцов матрицы - для удобства при использовании методов добавления, удаления и перемещения юнитов.
- Геттеры для получения значения приватных полей (используется в других классах):

Рисунок 13 - Геттеры

- **`int GameField::MovingUnit(int x, int y, char course, int steps)`** - метод для перемещения юнитов по полю в любом направление (по вертикали, горизонтали, диагонали). В методе накладываются ограничения, на количество шагов для определенного типа юнитов. Проверяется что позиция с координатами, введенными игроком, свободна, и не выходит за пределы игрового поля. Используется функция swap, меняющая объекты местами.

- Также прописан деструктор для освобождения динамической памяти, выделенной под контейнер, по окончанию работы с ним.

Выполнение требований к набору классов юнитов.

1. Интерфейс

Интерфейс юнитов реализуется посредством наследования от абстрактного класса Unit, наследуемого от абстрактного класс Object.

В классе Unit имеются поля **health, damage, armor** – атрибуты юнитов, поля **x, y** - координаты расположения объекта, **view** – вид юнита, **name[2]** - имя, для печати на поле, переопределенные методы для получения значения полей, метод печати значений этих полей и чисто виртуальный деструктор, делающий класс Unit абстрактным. Таким образом, все типы юнитов, наследуемые от Unit имеют перечисленные поля и методы.

```
class Object {
protected:
    int x;
    int y;
    int view;
    char name[2];

public:
    virtual void SetName(char symbol) = 0; //м
    virtual void SetView(int type) = 0;
    virtual void SetXY(int x, int y) = 0;
    virtual char* GetName() = 0;
    virtual int GetView() = 0;
    virtual int GetX() = 0;
    virtual int GetY() = 0;
};
```

Рисунок 14 - Абстрактный класс Объект

```
class Unit :public Object {        //абстрактный класс
protected:

    int health;         //здоровье
    int damage;         //урон
    int armor;          //броня

public:
    void SetName(char symbol) override {
        name[0] = symbol;
        name[1] = '\0';
    }

    void SetView(int view) override {
        this->view = view;
    }

    char* GetName() override {
        return name;
    };

    void SetXY(int x, int y) override {
        this->x = x;
        this->y = y;
    }

    int GetView() override {
        return view;
    }

    int GetX() override {
        return x;
    }

    int GetY() override {
        return y;
    }

    virtual ~Unit() = 0;  //чисто виртуальный деструктор

    void Print();
};
```

Рисунок 15 - Абстрактный класс Unit

2. Реализация трёх типов юнитов

От класса Unit наследуются три типа юнитов: class Monster, class Hero, class People. См. UML диаграмму классов.

3. Реализация двух видов юнитов для каждого типа

От каждого типа юнита наследуется по два класса вида юнитов. Например от класса People наследуются классы Man и Hunter. См. UML диаграмму классов.

4. Атрибуты

Атрибутами юнитов являются поля **`health, damage, armor,`** описанные в классе Unit, и инициализируемые в конструкторах классов видов и типов юнитов. Дополнительных атрибутов пока нет, но возможно будут добавлены при написании следующих лабораторных.

```
People::People(int damage, int armor, char symbol, int view) {
    this->damage = damage;
    health = 200;
    this->armor = armor;
    SetName(symbol);
    SetView(view);
}

Hunter::Hunter() :People(300, 150, (char)6, 5) {}

Man::Man() : People(150, 50, (char)1, 6) {}
```

Рисунок 16 - Пример инициализации атрибутов для классов Hunter, Man (виды) с вызовом конструктора базового класса People (тип)

5. Возможность перемещения по карте

Перемещение юнитов по карте реализуется с помощью метода, реализованного в GameField, описанного выше. Вызов этого метода происходит из класса FirstGamer, сопровождается проверкой введенных пользователем данных.

```
void FirstGamer::CaseMovUnit() {
    int steps;
    char a;
    cout << "Для того, чтобы переместить юнита на другую позицию, необходимо ввести:\n";
    cout << "Текущие координаты расположения юнита (x, y);\n";
    cout << "Количество шагов (из допустимого).\n";
    cout << "Длина одного шага равна " << field->GetStep() << endl;
    cout << "Допустим количество шагов:\n Вампир : от 1 до 3.\n";
    cout << "Оборотень: от 1 до 2.\n Целитель: 1 шаг.\n Маг: от 1 до 2 шагов.\n";
    exit = 1;
    while (exit) {
    cout << "Направление движения юнитов:\n"; //пока все юниты ходят по всем направлениям
    cout << "\"l\" - влево; \"r\" - вправо; \"u\" - вверх; \"d\" - вниз\n";
    cout << "\"a\" - диагональ вверх-направо; \"b\" - диагональ вверх-налево;\n";
    cout << "\"c\" - диагональ вниз-направо; \"f\" - диагональ вниз-налево.\n";
    cin >> x >> y >> a >> steps;
    while (CaseError(a, steps) || field->MovingUnit(x, y, a, steps)) {
        cout << "Введите снова\n";
        cin >> x >> y >> a >> steps;
    }
    field->PrintCurField();
    cout << "Введите 0, если хотите прекратить перемещение.\n";
    cout << "Введите 1, если хотите продолжить перемещение.\n";
    cin >> exit;
    }
}
```

Рисунок 17 - Вызов метода перемещения

```
bool FirstGamer::CaseError(char a, int steps) {
    if (x < 0 || y < 0 || x > field->GetHeight() - 1 || y > field->GetWidth() - 1) {
        cout << "Ошибка: данной позиции не существует!\n";
        return 1;
    }
    if (a == 'r' || a == 'l' || a == 'u' || a == 'd' || a == 'a' || a == 'b' || a == 'c' || a == 'f') {
    }
    else {
        cout << "Ошибка: команды не существует!\n";
        return 1;
    }
    if (steps < 0 || steps > 3) {
        cout << "Ошибка: недопустимое количество шагов!\n";
        return 1;
    }
    return 0;
}
```

Рисунок 18 - Проверка корректности введенных данных

Демонстрационные примеры.

Полный пример работы программы.

```
Добро пожаловать в игру "Mystic Strategy"
Если вы хотите ознакомиться с правилами игры, введите "y".
Если вы уже знаете правила и хотите их пропустить - введите "n".
y
                Правила игры "Mystic Strategy".
Данная игра рассчитана на двух игроков. Один из игроков играет за монстров (вампиры и
Другой - за героев (целители и маги). Также на поле находятся люди.
Цель монстров - уничтожить или "перевести" на свою сторону людей.
Цель героев - истребить монстров, сохранив при этом хотя бы часть населения.
Выигрывает тот - кто раньше выполнит свою цель.

                Доступные команды:
МОНСТРЫ
Цель команды: истребить или обратить на свою сторону всех людей.
ВАМПИР: команда для добавления "1".
Здоровье: 1000 единиц.
Урон: 450 единиц.
Броня: 400 единиц.
Дополнительные свойства: может обращать людей в вампиров (пополнение вашей армии).

ОБОРОТЕНЬ: команда для добавления "2".
Здоровье: 1000 единиц.
Урон: 650 единиц.
Броня: 500 единиц.
Дополнительные свойства: отсутствуют.
-------------------------------------------------------------------------------

ГЕРОИ
Цель команды: уничтожить всех Монстров, сохранив часть людей (хотя бы одного).
ЦЕЛИТЕЛЬ: команда для добавления "3".
Здоровье: 500 единиц.
Урон: 100 единиц.
Броня: 100 единиц.
Дополнительные свойства: может лечить Охотников и Магов.

МАГ: команда для добавления "4".
Здоровье: 500 единиц.
Урон: 600 единиц.
Броня: 300 единиц
```

Рисунок 19 - Вывод правил игры

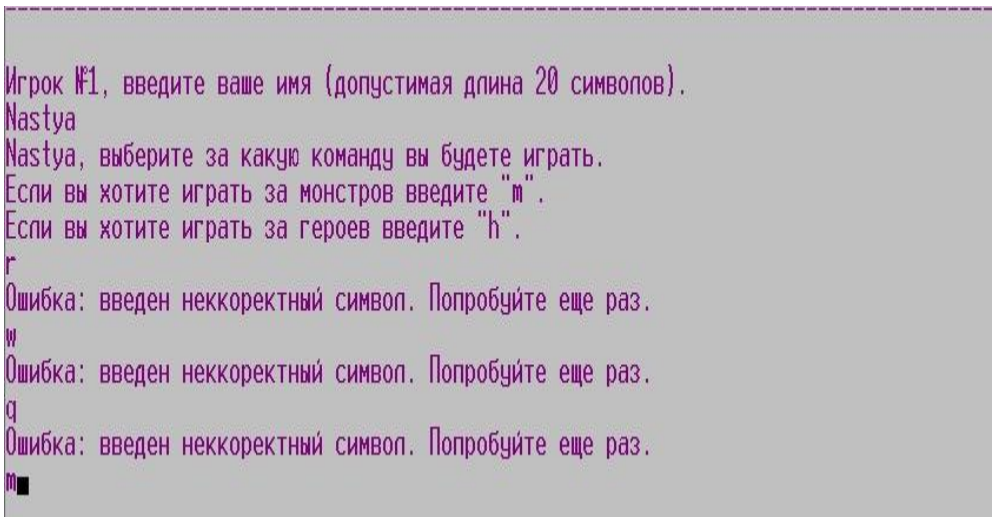


Рисунок 20 - Обработка ошибок при неверно введенной команде

Рисунок 21 - Задание поля произвольного размера

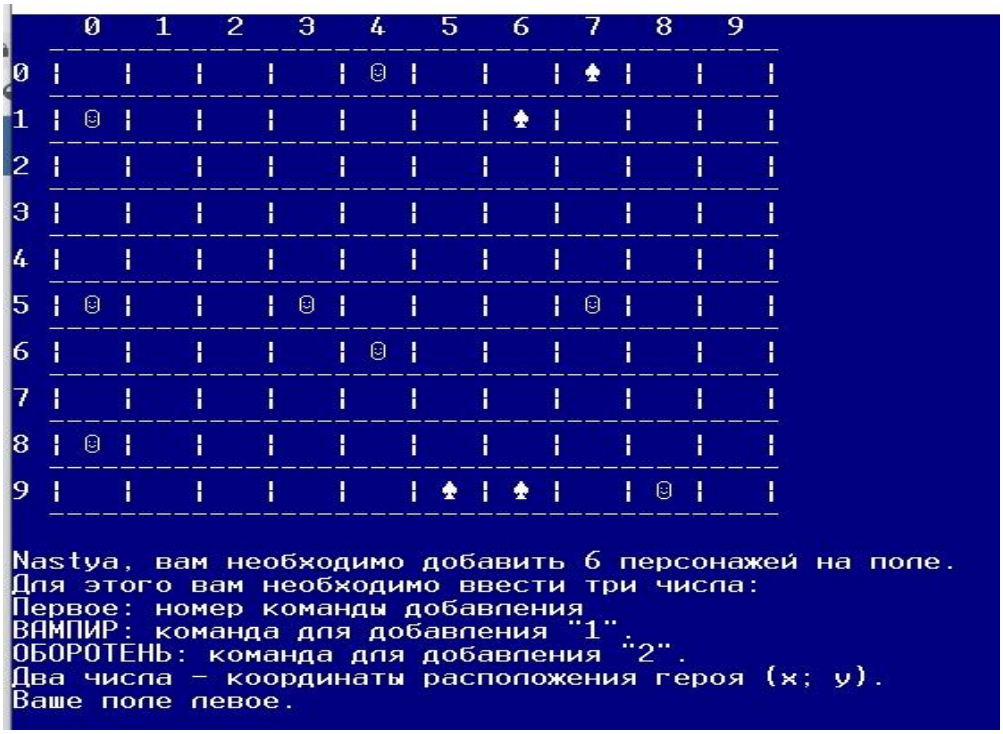


Рисунок 22 - Первоначальный вид поля после рандомного заполнения "людьми"

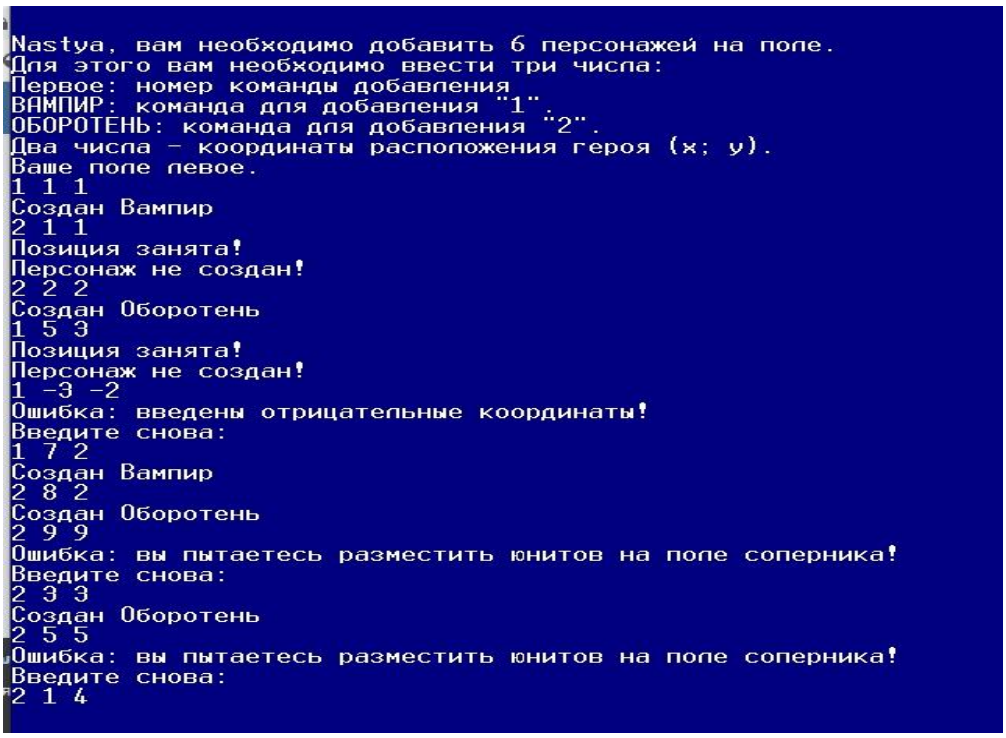


Рисунок 23 - Добавление юнитов на поле с обработкой ошибок

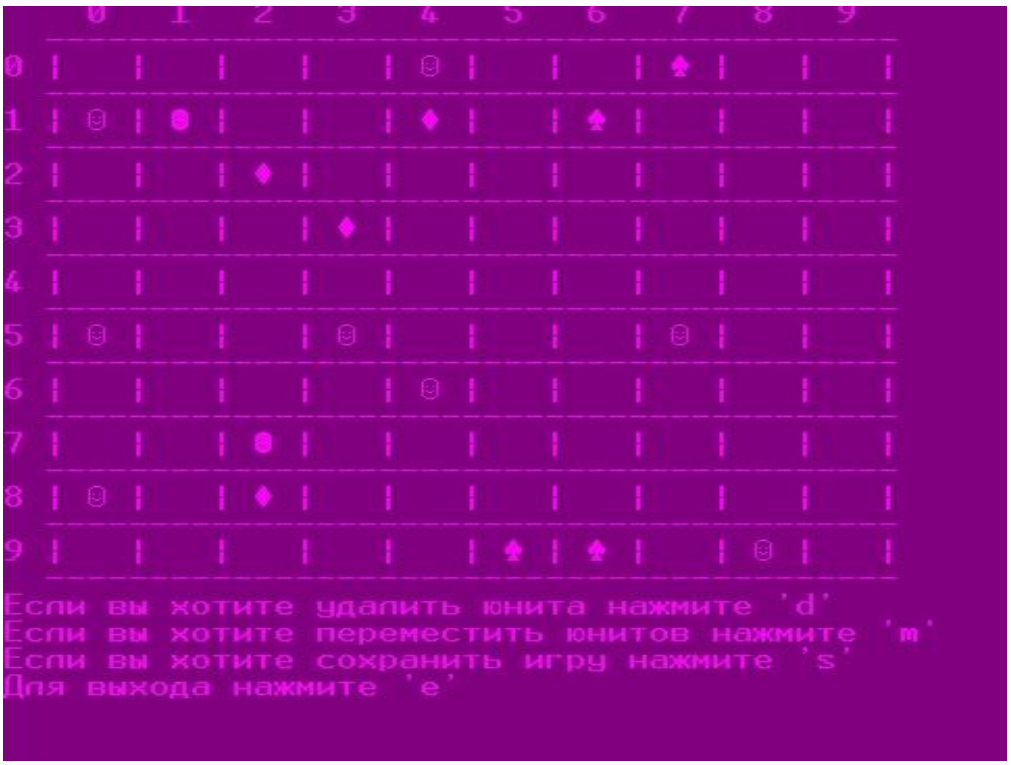


Рисунок 24 - Полученное заполнение поля

Рисунок 25 - Удаления юнита с позиции 1 0

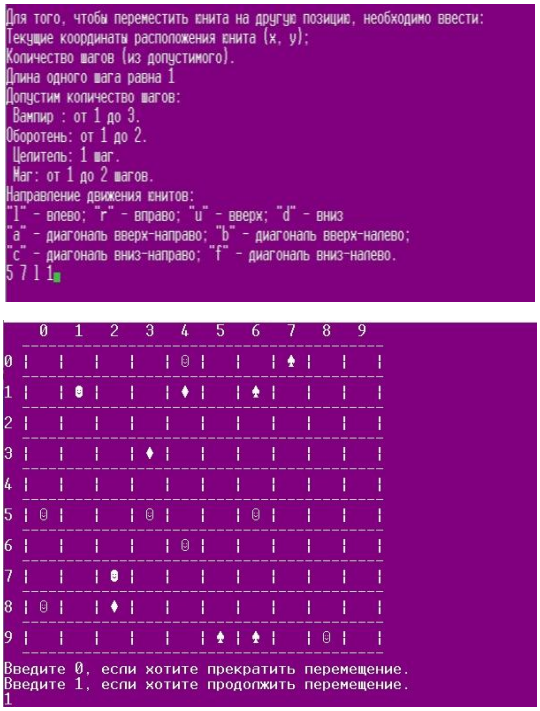


Рисунок 26 - Перемещение юнита с позиции 5 7 на позицию 5 6

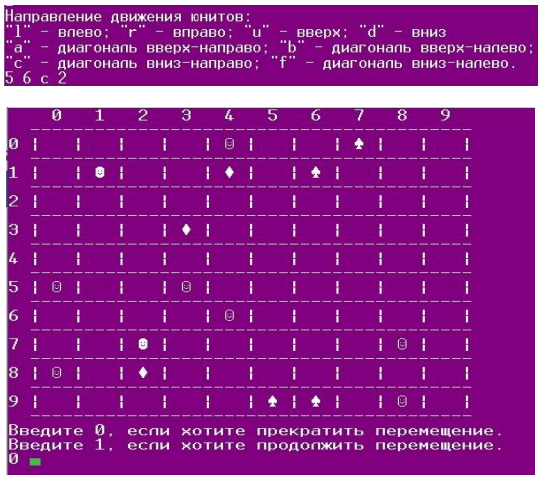


Рисунок 27 - Перемещение юнита с позиции 5 6 на позицию 7 8

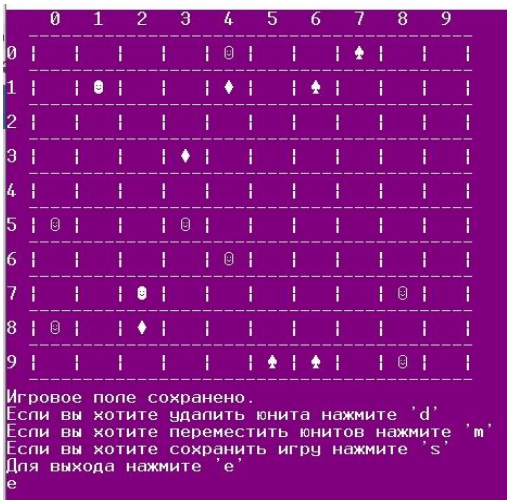


Рисунок 28 - Завершение работы программы

Конструктор перемещения.

Принимает не константную ссылку (rvalue reference). Используется для переноса данных из одного объекта в другой, не копирование. Таким образом, при использовании функции swap, происходит перенос данных из одного объекта в другой.

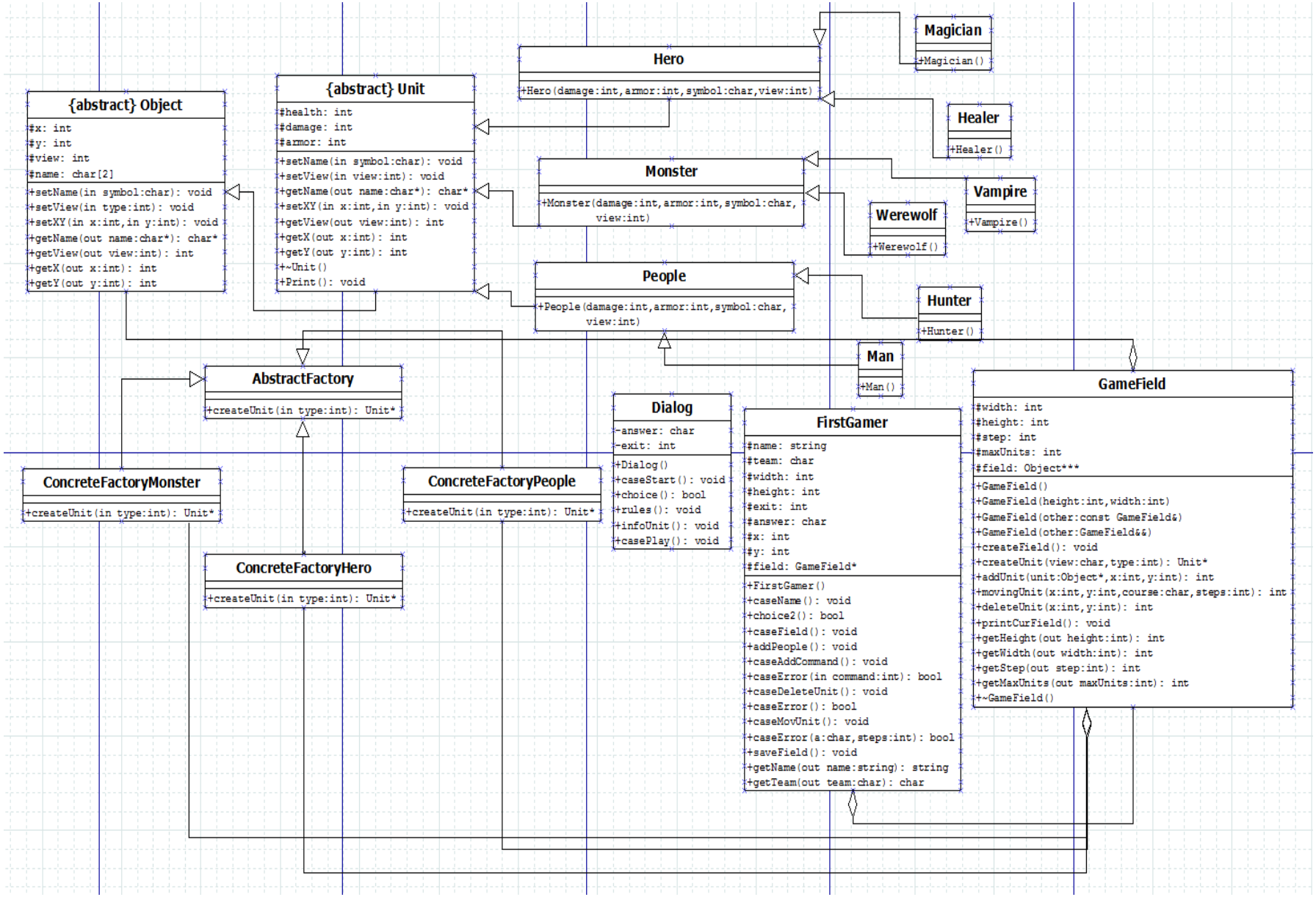
```
GameField::GameField(GameField&& other):field(nullptr), height(NULL), width(NULL), step(NULL) {

    swap(field, other.field);
    swap(height, other.height);
    swap(width, other.width);
    swap(step, other.step);

}
```

Рисунок 29 - Конструктор перемещения

Абстрактная фабрика.

Реализован класс Абстрактная Фабрика и наследуемые от него классы Конкретных Фабрик, в которых описаны методы, создающие экземпляры конкретных видов юнитов. Обращение к фабрикам происходит в методе Unit* GameField::createUnit(char type, int view). См. UML диаграмму классов.

UML диаграмма классов



Выводы.

Была написана программа на языке программирования C++, реализующая класс игрового поля и набор классов юнитов, а также классы, для взаимодействия игрока с полем и юнитами.

ПРИЛОЖЕНИЕ А

Main.cpp

```
#include <iostream>
#include <ctype.h>
#include "Dialog.h"
#include "FirstGamer.h"
using namespace std;

int main() {
	setlocale(LC_ALL, "ru");
	Dialog now;
	now.caseStart();
	return 0;
}
```

ПРИЛОЖЕНИЕ Б

GameField.h

```
#pragma once
#include "Object.h"
#include "Unit.h"
#include "AbstractFactory.h"

class GameField {
protected:
	int width, height;
	int step;
	int maxUnits;
	Object*** field;                                    //двумерный
массив указателей
public:
	GameField();
	GameField(int height, int width);
	GameField(const GameField& other);      //конструктор
копирования
	GameField(GameField&& other);           //конструктор
переноса
	void createField();
	Unit* createUnit(char view, int type);
	int addUnit(Object* unit, int x, int y);
	int movingUnit(int x, int y, char course, int steps);
	int deleteUnit(int x, int y);
	void printCurField();
	int getHeight();
	int getWidth();
	int getStep();
	int getMaxUnits();
	~GameField();

};
```

GameField.cpp

```
#include "GameField.h"
#include <iostream>
#include "AbstractFactory.h"
#include "GameField.h"
using namespace std;

GameField::GameField()                    //конструктор       "по
умолчанию"
{
     height = 10;
     width = 10;
     step = 1;
     maxUnits = 6;
}

GameField::GameField(int  height,  int  width)  :  GameField()
//делегирующие конструкторы
{
     if (height <= 3 || width <= 3) {
          std::cout   <<   "Ошибка:   неверно   указан   размер
поля!\nСоздано поле стандартного размера.\n";
     }
     else {
          this->height = height;
          this->width = width;
          step = 1;
          if (width >= height && (int)((width / 2) * (0.2)) >
step && height >= (width / 2)) {
               step = (int)((width / 2) * (0.2));
          }
          maxUnits = 6 * step;
     }
     createField();
```

```
}

GameField::GameField(const GameField& other) {
	this->height = other.height;
	this->width = other.width;
	this->step = other.step;

	this->field = new Object**[other.height];
	for (int i = 0; i < other.height; i++) {
		this->field[i] = new Object * [other.width];
		for (int j = 0; j < other.width; j++) {
			this->field[i][j] = other.field[i][j];
		}
	}
}

GameField::GameField(GameField&& other):field(nullptr),
height(NULL), width(NULL), step(NULL) {

	swap(field, other.field);
	swap(height, other.height);
	swap(width, other.width);
	swap(step, other.step);

}

void GameField::createField() {
	field = new Object * *[height];
	for (int i = 0; i < height; i++) {
		field[i] = new Object * [width];
		for (int j = 0; j < width; j++) {
			field[i][j] = nullptr;
		}
	}
}
```

```
Unit* GameField::createUnit(char type, int view) {
	ConcreteFactoryMonster monster;
	ConcreteFactoryHero hero;
	ConcreteFactoryPeople people;
	switch (type) {
	case 'm':
		if (view == 1 || view == 2) return
monster.createUnit(view);
		break;
	case 'h':
		if (view == 3 || view == 4) return
hero.createUnit(view);
		break;
	case 'p':
		if (view == 5 || view == 6) return
people.createUnit(view);
		break;
	default:
		cout << "Ошибка: " << view << " <-- такого вида не
существует!\n"; //доработать момент
		break;
	}
}

int GameField::addUnit(Object* unit, int x, int y) {
	unit->setXY(x, y);
	if (field[x][y] == nullptr) {
		field[x][y] = unit;
		return 0;
	}
	else {
		cout << "Позиция занята!\n";
		return 1;
	}
```

```
    }

    int GameField::movingUnit(int x, int y, char course, int
steps) { //'r' - право 'l' - лево 'd' - вниз 'u' - вверх
         switch (field[x][y]->getView()) {
         case 1:
              if (steps > 3) {
                   cout << "Максимальное кол-во шагов для данного
персонажа = 3.\n";
                   return 1;
              }
              break;
         case 2:
              if (steps > 2) {
                   cout << "Максимальное кол-во шагов для данного
персонажа = 2.\n";
                   return 1;
              }
              break;
         case 3:
              if (steps > 1) {
                   cout << "Максимальное кол-во шагов для данного
персонажа = 1.\n";
                   return 1;
              }
              break;
         case 4:
              if (steps > 2) {
                   cout << "Максимальное кол-во шагов для данного
персонажа = 2.\n";
                   return 1;
              }
              break;
         }
         steps *= step;
```

```
          if (course == 'r') {
               if (field[x][y + steps] == nullptr && (y + steps) <
width) { //пока можно перемещаться только на свободные позиции
                    swap(field[x][y], field[x][y + steps]);
                //можно перемещаться на поле противника
               }
          }
          else if (course == 'l') {
               if (field[x][y - steps] == nullptr && (y - steps) >
-1) {
                    swap(field[x][y], field[x][y - steps]);
               }
          }
          else if (course == 'd') {
               if (field[x + steps][y] == nullptr && (x + steps) <
height) {
                    swap(field[x][y], field[x + steps][y]);
               }
          }
          else if (course == 'u') {
               if (field[x - steps][y] == nullptr && (x - steps) >
-1) {
                    swap(field[x][y], field[x - steps][y]);
               }
          }
          else if (course == 'a') {
               if (field[x - steps][y + steps] == nullptr && (y +
steps) < width && (x - steps) > -1) {
                    swap(field[x][y], field[x - steps][y + steps]);
               }
          }
          else if (course == 'b') {
               if (field[x - steps][y - steps] == nullptr && (y -
steps) > -1 && (x - steps) > -1) {
```

```
                    swap(field[x][y], field[x - steps][y - steps]);
               }
          }
          else if (course == 'c') {
               if (field[x + steps][y + steps] == nullptr && (y +
steps) < width && (x + steps) < height) {
                    swap(field[x][y], field[x + steps][y + steps]);
               }
          }
          else if (course == 'f') {
               if (field[x + steps][y - steps] == nullptr && (y -
steps) > -1 && (x + steps) < height) {
                    swap(field[x][y], field[x + steps][y - steps]);
               }
          }
          else {
               cout << "Невозможно выполнить перемещение на
заданную позицию!\n";
               return 1;
          }
          return 0;
     }

     int GameField::deleteUnit(int x, int y) {
          if (field[x][y] != nullptr) {
               delete field[x][y];
               field[x][y] = nullptr;
               return 0;
          }
          else {
               cout << "На позиции нет юнита для удаления!\n";
               return 1;
          }
     }
```

```
void GameField::printCurField() {
	int k = 0;
	system("cls");
	system("color 5F");
	cout << "  ";
	for (int i = 0; i < width; i++) {
		if (i < 10) cout << "  " << i << " ";
		else if (i == 10) cout << "  " << i << "  ";
		else cout << i << "  ";
	}
	cout << "\n" << "  -";
	for (int i = 0; i < width; i++) {
		cout << "----";
	}
	cout << endl;
	for (int i = 0; i < height; i++) {
		if (k <= 9) cout << k++ << " ";
		else cout << k++;
		for (int j = 0; j < width; j++) {
			if (field[i][j] == nullptr) {
				cout << "|   ";
			}
			if (field[i][j] != nullptr) {
				cout << "| " << field[i][j]->getName() <<
" ";
			}
			if (j == width - 1) cout << "|";
		}

		cout << "\n" << "  -";
		for (int i = 0; i < width; i++) {
			cout << "----";
		}
		cout << endl;
	}
```

```
    }

    int GameField::getHeight() {
        return height;
    }

    int GameField::getWidth() {
        return width;
    }

    int GameField::getStep() {
        return step;
    }

    int GameField::getMaxUnits() {
        return maxUnits;
    }

    GameField::~GameField() {
        for (int i = 0; i < height; i++) {
            delete[] field[i];
        }
        delete[] field;
}
```

ПРИЛОЖЕНИЕ В

Unit.h

```
#pragma once
#pragma once
#include <iostream>
using namespace std;

class Unit :public Object {            //абстрактный класс
protected:

     int health;              //здоровье
     int damage;              //урон
     int armor;               //броня

public:
     void setName(char symbol) override {
          name[0] = symbol;
          name[1] = '\0';
     }

     void setView(int view) override {
          this->view = view;
     }

     char* getName() override {
          return name;
     };

     void setXY(int x, int y) override {
          this->x = x;
          this->y = y;
     }

     int getView() override {
```

```
          return view;
     }

     int getX() override {
          return x;
     }

     int getY() override {
          return y;
     }

     virtual ~Unit() = 0;  //чисто виртуальный деструктор

     void Print();
};


class Hero :public Unit {          //тип № 1 -- Герой
public:
     Hero(int damage, int armor, char symbol, int view);
};

class Healer :public Hero {        //вид Целитель
public:
     Healer();      // вызов конструктора базового класса
};

class Magician :public Hero { //вид Маг
public:
     Magician();
};


class Monster :public Unit {            //тип № 2 --  Монстр
public:
```

```
        Monster(int damage, int armor, char symbol, int view);
    };

    class Vampire :public Monster {         //вид Вампир
    public:
        Vampire();
    };

    class Werewolf :public Monster {        //вид Оборотень
    public:
        Werewolf();
    };


    class People :public Unit {             //тип Человек
    public:
        People(int damage, int armor, char symbol, int view);
    };

    class Hunter :public People {           //вид Охотник
    public:
        Hunter();
    };

    class Man :public People {                  //вид      Обычный
человек
    public:
        Man();
};
```

Unit.cpp

```
    #include "Object.h"
    #include "Unit.h"

    Unit::~Unit(){}
```

```
void Unit::Print() {
	cout << "ОПИСАНИЕ ПЕРСОНАЖА:\n";
	cout << "Имя: " << name << "\n";
	cout << "Здоровье: " << health << " единиц.\n";
	cout << "Урон: " << damage << " единиц.\n";
	cout << "Броня: " << armor << " единиц.\n";
}

Hero::Hero(int damage, int armor, char symbol, int view) {
	this->damage = damage;
	health = 500;
	this->armor = armor;
	setName(symbol);
	setView(view);
}

Healer::Healer() :Hero(200, 100, (char)3, 3) {}		// 
вызов конструктора базового класса

Magician::Magician() : Hero(600, 300, (char)15, 4) {}

Monster::Monster(int damage, int armor, char symbol, int 
view) {
	this->damage = damage;
	health = 1000;
	this->armor = armor;
	setName(symbol);
	setView(view);
}

Vampire::Vampire() :Monster(450, 400, (char)2, 1) {}

Werewolf::Werewolf() : Monster(650, 500, (char)4, 2) {}
```

```
People::People(int damage, int armor, char symbol, int view)
{
	this->damage = damage;
	health = 200;
	this->armor = armor;
	setName(symbol);
	setView(view);
}

Hunter::Hunter() :People(300, 150, (char)6, 5) {}

Man::Man() : People(150, 50, (char)1, 6) {}
```

ПРИЛОЖЕНИЕ Г

FirstGamer.h

```
#pragma once
#include <iostream>
#include <string>
#include "GameField.h"

using namespace std;

class FirstGamer {// далее будет добавлен класс для второго игрока - пока демонстрационый вариант
protected:
	string name;
	char team;
	int width;
	int height;
	int exit;
	char answer;
	int x, y;
	GameField* field;
public:
	FirstGamer();
	void caseName();
	bool choice2();
	void caseField();
	void addPeople();
	void caseAddCommand();
	bool caseError(int command);
	void caseDeleteUnit();
	bool caseError();
	void caseMovUnit();
	bool caseError(char a, int steps);
	void saveField();
	string getName();
```

```
        char getTeam();
};
```

FirstGamer.cpp

```
#include "FirstGamer.h"
#include <ctime>
FirstGamer::FirstGamer() {
        name = '0';
        team = '0';
        width = 0;
        height = 0;
        exit = 0;
        x = 0;
        y = 0;
        answer = '0';
}

void FirstGamer::caseName() {
        cout << "Игрок №1, введите ваше имя (допустимая длина 20 символов).\n";
        cin >> name;
        cout << name << ", выберите за какую команду вы будете играть.\n";
        cout << "Если вы хотите играть за монстров введите \"m\".\n";
        cout << "Если вы хотите играть за героев введите \"h\".\n";
        if (!choice2()) team = 'h';
        else team = 'm';
        caseField();
}

bool FirstGamer::choice2() {
        do {
                exit = 0;
```

```
            cin >> answer;
            answer = tolower(answer);

            switch (answer) {
            case 'm':
                return 1;
                break;

            case 'h':
                return 0;
                break;

            default:
                cout << "Ошибка: введен неккоректный символ. Попробуйте еще раз.\n";
                exit = 1;
            }
        } while (exit == 1);
    }

    void FirstGamer::caseField() {
        system("cls");
        system("color 79");
        cout << endl << "Перед началом игры введите высоту и ширину игрового поля.\n";
        cin >> height >> width;
        field = new GameField(height, width);
        cout << "Построено поле высоты: " << field->getHeight() << " и ширины: " << field->getWidth() << ".\n";
        addPeople();
    }
    void FirstGamer::addPeople() {
        cout << "Добавление персонажей по умолчанию\n";
        int view;
        int i = field->getMaxUnits() * 2;
```

```
            srand(time(NULL));
            while (i != 0) {
                  view = rand() % 2 + 5; //5 или 6
                  x = rand() % field->getHeight();
                  y = rand() % field->getWidth();
                  if (field->addUnit(field->createUnit('p', view), x,
y)) {
                        i++; //Персонаж не создан!
                  }
                  else {
                        if (view == 5) cout << "Создан охотник\n";
                        else cout << "Создан человек\n";
                  }
                  i--;
            }
            field->printCurField();
            caseAddCommand();
      }
      void FirstGamer::caseAddCommand() {
            int command;
            //system("cls");
            system("color 1F");
            cout << endl << name << ", вам необходимо добавить " <<
field->getMaxUnits() << " персонажей на поле. \n";
            cout << "Для этого вам необходимо ввести три числа:\n";
            cout << "Первое: номер команды добавления\n";
            if (team == 'm') {
                  cout << "ВАМПИР: команда для добавления \"1\".\n";
                  cout   <<   "ОБОРОТЕНЬ:   команда   для   добавления
\"2\".\n";
            }
            else {
                  cout << "ЦЕЛИТЕЛЬ: команда для добавления \"3\".\n";
                  cout << "МАГ: команда для добавления \"4\".\n";
            }
```

```
            cout << "Два числа - координаты расположения героя (x;
y).\n";
            if (team == 'm') cout << "Ваше поле левое.\n";
            else cout << "Ваше поле правое\n";

            cin >> command >> x >> y;
            int i = field->getMaxUnits();
            while (i != 0) {
                  while (!caseError(command)) {
                        cout << "Введите снова:\n";
                        cin >> command >> x >> y;
                  }

                  switch (command) {
                  case 1:
                        if  (!field->addUnit(field->createUnit('m',  1),
x, y)) {
                              cout << "Создан Вампир\n";
                        }
                        else {
                              cout << "Персонаж не создан!\n";
                              i++;
                        }
                        break;

                  case 2:
                        if  (!field->addUnit(field->createUnit('m',  2),
x, y)) {
                              cout << "Создан Оборотень\n";
                        }
                        else {
                              cout << "Персонаж не создан!\n";
                              i++;
                        }
                        break;
```

```
                case 3:
                    if  (!field->addUnit(field->createUnit('h',  3),
x, y)) {
                        cout << "Создан Целитель\n";
                    }
                    else {
                        cout << "Персонаж не создан!\n";
                        i++;
                    }
                    break;

                case 4:
                    if  (!field->addUnit(field->createUnit('h',  4),
x, y)) {
                        cout << "Создан Маг\n";
                    }
                    else {
                        cout << "Персонаж не создан!\n";
                        i++;
                    }
                    break;
                }
                i--;
                if (i) cin >> command >> x >> y;
            }
            field->printCurField();
        }

        bool FirstGamer::caseError(int command) {

            if (team == 'm') {
                if (command < 1 || command > 2) {
                    cout << "Ошибка ведена неверная команда!\n";
                    return 0;
```

```
                }
                if (y > (field->getWidth() - 1) / 2) {
                    cout << "Ошибка: вы пытаетесь разместить юнитов на поле соперника!\n";
                    return 0;
                }
            }
            else if (team == 'h') {
                if (command < 3 || command > 4) {
                    cout << "Ошибка ведена неверная команда!\n";
                    return 0;
                }
                if (y < (field->getWidth() - 1) / 2) {
                    cout << "Ошибка: вы пытаетесь разместить юнитов на поле соперника!\n";
                    return 0;
                }
            }
            if (x < 0 || y < 0) {
                cout << "Ошибка: введены отрицательные координаты!\n";
                return 0;
            }
            else if (x > field->getWidth() - 1 || y > field->getHeight() - 1) {
                cout << "Ошибка: координаты выходят из области поля!\n";
                return 0;
            }
            return 1;
        }

        void FirstGamer::caseDeleteUnit() { //возможно и не нужна, но пока для примера удаления
            exit = 1;
```

```
          while (exit) {
               cout << "Введите координаты юнита, которого хотите
удалить.\n";
               cin >> x >> y;
               while (!caseError() || field->deleteUnit(x, y)) {
                    cout << "Введите снова:\n";
                    cin >> x >> y;
               }
               cout << "Юнит удален!\n";
               field->printCurField();
               cout   <<   "Введите   0,   если   хотите   прекратить
удаление.\n";
               cout   <<   "Введите   1,   если   хотите   продолжить
удаление.\n";
               cin >> exit;
          }
     }

     bool FirstGamer::caseError() {

          if (team == 'm') {
               if (y > (field->getWidth() - 1) / 2) {
                    cout  <<  "Ошибка:  вы  пытаетесь  удалить  юнита
соперника!\n";
                    return 0;
               }
          }
          else if (team == 'h') {
               if (y < (field->getWidth() - 1) / 2) {
                    cout  <<  "Ошибка:  вы  пытаетесь  удалить  юнита
соперника!\n";
                    return 0;
               }
          }
          if (x < 0 || y < 0) {
```

```
            cout << "Ошибка: введены отрицательные
координаты!\n";
            return 0;
        }
        else if (x > field->getWidth() - 1 || y > field-
>getHeight() - 1) {
            cout << "Ошибка: координаты выходят из области
поля!\n";
            return 0;
        }
        return 1;
    }

    void FirstGamer::caseMovUnit() {
        int steps;
        char a;
        cout << "Для того, чтобы переместить юнита на другую
позицию, необходимо ввести:\n";
        cout << "Текущие координаты расположения юнита (x,
y);\n";
        cout << "Количество шагов (из допустимого).\n";
        cout << "Длина одного шага равна " << field->getStep() <<
endl;
        cout << "Допустим количество шагов:\n Вампир : от 1 до
3.\n";
        cout << "Оборотень: от 1 до 2.\n Целитель: 1 шаг.\n Маг:
от 1 до 2 шагов.\n";
        exit = 1;
        while (exit) {
        cout << "Направление движения юнитов:\n"; //пока все
юниты ходят по всем направлениям
        cout << "\"l\" - влево; \"r\" - вправо; \"u\" - вверх;
\"d\" - вниз\n";
        cout << "\"a\" - диагональ вверх-направо; \"b\" -
диагональ вверх-налево;\n";
```

```
          cout << "\"c\" - диагональ вниз-направо; \"f\" -
диагональ вниз-налево.\n";
          cin >> x >> y >> a >> steps;
          while (caseError(a, steps) || field->movingUnit(x, y, a,
steps)) {
               cout << "Введите снова\n";
               cin >> x >> y >> a >> steps;
          }
          field->printCurField();
          cout << "Введите 0, если хотите прекратить
перемещение.\n";
          cout << "Введите 1, если хотите продолжить
перемещение.\n";
          cin >> exit;
          }
     }

     bool FirstGamer::caseError(char a, int steps) {
          if (x < 0 || y < 0 || x > field->getHeight() - 1 || y >
field->getWidth() - 1) {
               cout << "Ошибка: данной позиции не существует!\n";
               return 1;
          }
          if (a == 'r' || a == 'l' || a == 'u' || a == 'd' || a ==
'a' || a == 'b' || a == 'c' || a == 'f') {
          }
          else {
               cout << "Ошибка: команды не существует!\n";
               return 1;
          }
          if (steps < 0 || steps > 3) {
               cout << "Ошибка: недопустимое количество шагов!\n";
               return 1;
          }
          return 0;
```

```
    }

    void FirstGamer::saveField() {
        GameField copy(*field);
        copy.printCurField();
        cout << "Игровое поле сохранено.\n";
    }

    string FirstGamer::getName() {
        return name;
    }

    char FirstGamer::getTeam() {
        return team;
}
```

ПРИЛОЖЕНИЕ Д

Dialog.h

```
#pragma once
#include "GameField.h"
#include "FirstGamer.h"

class Dialog {      //для взаимодействия с игроками и вызова
методов
private:
     char answer;
     int exit;

public:

     Dialog();
     void caseStart();
     bool choice();
     void rules();
     void infoUnit();
     void casePlay();
};
```

Dialog.cpp

```
#include <iostream>
#include "Dialog.h"
using namespace std;

Dialog::Dialog() {
     answer = '0';
     exit = 0;
}

void Dialog::caseStart() {
     system("color 75");
     cout << "Добро пожаловать в игру \"Mystic Strategy\"\n";
```

```
          cout << "Если вы хотите ознакомиться с правилами игры,
введите \"y\".\n";
          cout << "Если вы уже знаете правила и хотите их
пропустить - введите \"n\".\n";
          if (choice()) rules();
          else infoUnit();
     }

     bool Dialog::choice() {
          do {
               exit = 0;
               cin >> answer;
               answer = tolower(answer);

               switch (answer) {
               case 'y':
                    return 1;
                    break;

               case 'n':
                    return 0;
                    break;

               default:
                    cout << "Ошибка: введен неккоректный символ.
Попробуйте еще раз.\n";
                    exit = 1;
               }
          } while (exit == 1);
     }

     void Dialog::rules() {
          cout << "\t\tПравила игры \"Mystic Strategy\".\n";
          cout << "Данная игра рассчитана на двух игроков. Один из
игроков играет за монстров (вампиры и оборотни).\n";
```

```
        cout << "Другой - за героев (целители и маги). Также на
поле находятся люди.\n";
        cout << "Цель монстров - уничтожить или \"перевести\" на
свою сторону людей.\n";
        cout << "Цель героев - истребить монстров, сохранив при
этом хотя бы часть населения.\n";
        cout << "Выигрывает тот - кто раньше выполнит свою
цель.\n";
        infoUnit();
    }

    void Dialog::infoUnit() {
        cout << "\n\t\tДоступные команды:\n";
        cout << "МОНСТРЫ\n";
        cout << "Цель команды: истребить или обратить на свою
сторону всех людей.\n";
        cout << "ВАМПИР: команда для добавления \"1\".\n";
        cout << "Здоровье: 1000 единиц.\n";
        cout << "Урон: 450 единиц.\n";
        cout << "Броня: 400 единиц.\n";
        cout << "Дополнительные свойства: может обращать людей в
вампиров (пополнение вашей армии).\n";
        cout << endl;
        cout << "ОБОРОТЕНЬ: команда для добавления \"2\".\n";
        cout << "Здоровье: 1000 единиц.\n";
        cout << "Урон: 650 единиц.\n";
        cout << "Броня: 500 единиц.\n";
        cout << "Дополнительные свойства: отсутствуют.\n";
        cout << "-----------------------------------------------
-------------------------------------\n\n";
        cout << "ГЕРОИ\n";
        cout << "Цель команды: уничтожить всех Монстров, сохранив
часть людей (хотя бы одного).\n";
        cout << "ЦЕЛИТЕЛЬ: команда для добавления \"3\".\n";
        cout << "Здоровье: 500 единиц.\n";
```

```
          cout << "Урон: 100 единиц.\n";
          cout << "Броня: 100 единиц.\n";
          cout << "Дополнительные свойства: может лечить Охотников
и Магов.\n";
          cout << endl;
          cout << "МАГ: команда для добавления \"4\".\n";
          cout << "Здоровье: 500 единиц.\n";
          cout << "Урон: 600 единиц.\n";
          cout << "Броня: 300 единиц.\n";
          cout << "Дополнительные свойства: отсутствуют.\n";
          cout << "-----------------------------------------------
-------------------------------------\n\n";
          casePlay();
     }

     void Dialog::casePlay() {
          FirstGamer gamer;
          gamer.caseName();
          char change = 'n';
         system("color 5D");
          while (change == 'n') {

               cout << "Если вы хотите удалить юнита нажмите
\'d\'\n";
               cout << "Если вы хотите переместить юнитов нажмите
\'m\'\n";
               cout << "Если вы хотите сохранить игру нажмите
\'s\'\n";
               cout << "Для выхода нажмите \'e\'\n";

               cin >> change;
               switch (change) {
               case 'd':
                    gamer.caseDeleteUnit();
                    change = 'n';
```

```
                    break;
               case 'm':
                    gamer.caseMovUnit();
                    change = 'n';
                    break;
               case 's':
                    gamer.saveField();
                    change = 'n';
                    break;
               default:
                    break;
               }
          }
}
```

ПРИЛОЖЕНИЕ Е

Object.h

```
#pragma once

 class Object { //абстрактный класс
protected:
	int x;
	int y;
	int view;
	char name[2];

public:
	virtual void setName(char symbol) = 0;
	virtual void setView(int type) = 0;
	virtual void setXY(int x, int y) = 0;
	virtual char* getName() = 0;
	virtual int getView() = 0;
	virtual int getX() = 0;
	virtual int getY() = 0;
};
```

ПРИЛОЖЕНИЕ Ж

AbstractFactory.h

```
#pragma once
#include "Unit.h"

class AbstractFactory {
public:
     virtual Unit* createUnit(int type) = 0;
};

class ConcreteFactoryMonster :public  AbstractFactory {
public:
     Unit* createUnit(int type) override {
          if (type == 1) {
               return new Vampire();
          }
          else if (type == 2) {
               return new Werewolf();
          }
     }
};

class ConcreteFactoryHero :public  AbstractFactory {
public:
     Unit* createUnit(int type) override {
          if (type == 3) {
               return new Healer();
          }
          else if (type == 4) {
               return new Magician();
          }
     }
};
```

```
class ConcreteFactoryPeople :public  AbstractFactory {
public:
     Unit* createUnit(int type) override {
          if (type == 5) {
               return new Hunter();
          }
          else if (type == 6) {
               return new Man();
          }
     }
};
```