

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

Ишанина Л.Н.

Преподаватель

Жангиров Т.Р.

Санкт-Петербург

2020

Цель работы.

Разработать и реализовать набор классов: класс игрового поля, набор классов юнитов, удовлетворяющие требованиям, таким как, создание поля произвольного размера, контроль максимального количества объектов на поле. Возможность добавления, удаления объектов, копирования поля. Юниты должны иметь один общий интерфейс, имеют возможность перемещаться по карте.

Задание.

Задание на лабораторную работу в исходной формулировке. Указывается полностью. Указывается вариант задания, если есть.

Разработать и реализовать набор классов:

- Класс игрового поля
- Набор классов юнитов

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

Описание основных классов

Класс Field

Класс, необходимый для создания игрового поля, а также для работы с ним.

Основные требования:

1) Создание поля произвольного размера

Само поле — это двумерный массив указателей на объекты класса `ObjectInterface`. Класс `Field` хранит такие поля как `height`(высота поля), `width`(ширина поля), `controlMax`(переменная, для контролирования максимального количества объектов на поле), `countObjects`(переменная, которая считает объекты на поле), `flag`(переменная-флаг, для контролирования ситуации при слишком маленьком заданном размере поля), и `ObjectInterface** obj`(двумерный массив указателей на объект). Конструктор поля инициализирует его поля и выделяет память. Конструктор представлен на рисунке 1.

```
//----- Конструктор -----//
Field::Field(unsigned height, unsigned width, unsigned controlMax)
: width(width), height(height), controlMax(controlMax) {
    countObjects = 0;
    flag = 0;
    if(height < 3 || width < 3)
    {
        std::cout << "Error! Invalid field size!" << std::endl;
        flag = 1;
    }
    if (flag == 0)
    {
        obj = new ObjectInterface **[width];
        for (int i = 0; i < width; i++)
        {
            obj[i] = new ObjectInterface *[height];
            for (int j = 0; j < height; j++)
            {
                obj[i][j] = nullptr;
            }
        }
    }
}
```

Рисунок 1 — конструктор поля.

Поле создается произвольного размера, а также учитываются неверные случаи, когда размер стороны поля задается меньше трех. Кроме того,

переменные `height` и `width` являются переменными `unsigned`, что помогает создавать поле с заданными только неотрицательными размерами.

Демонстрационный пример создания поля недопустимого размера:

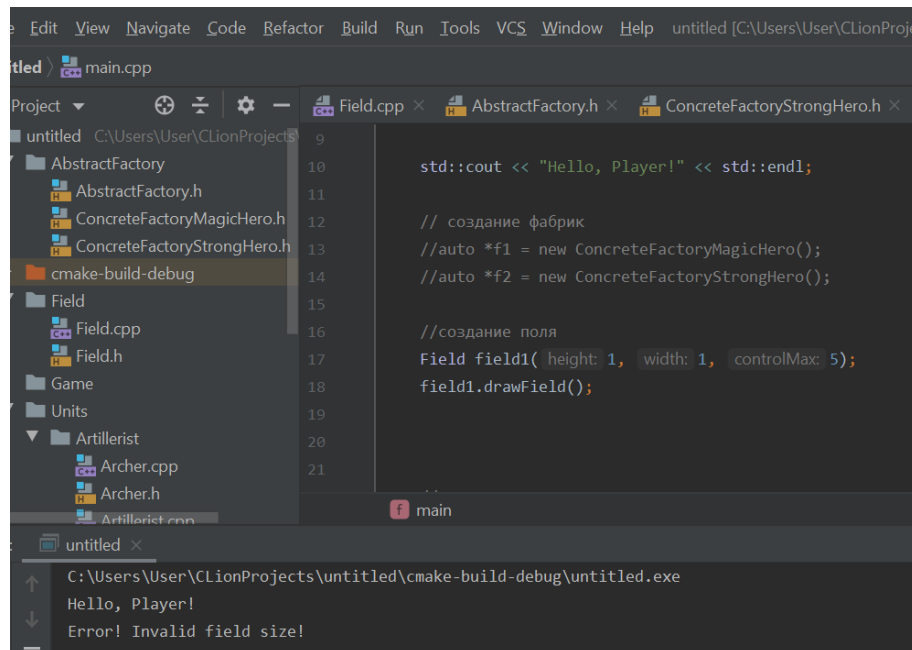


Рисунок 2 – демонстрационный пример создания поля недопустимого размера.

2) Контроль максимального количества объектов на поле

Пока что максимальное количество объектов на поле задается игроком при вызове конструктора поля. Сам контроль максимального количества объектов на поле происходит в методе добавления юнита на поле.

Демонстрационный пример работы контролирования максимального количества объектов на поле:

представленный на рисунке 5. Данный метод принимает координаты, куда нужно поместить юнит, а также указатель на юнит.

```
//----- Добавление юнита -----//
void Field::appendUnit(unsigned newX, unsigned newY, ObjectInterface *unit) {

    if(!flag)
    {
        if (countObjects < controlMax && newX <= width && newY <= height)
        {
            if(obj[newX][newY] == nullptr)
            {
                countObjects++;
                obj[newX][newY] = unit;
            }
        }
    }
}
```

Рисунок 5 – метод добавления юнита на поле.

В данном методе также происходят три проверки. Первая проверка – если размер поля введен неправильно, то и смысла добавлять юнита нет, так как поле не будет существовать. Вторая проверка – проверка на превышение объектов на поле и выход за границы поля. И третья проверка – является ли клетка пустой, чтобы не ставить юнита на место, где уже расположен юнит.

Удаление юнита с поля происходит с помощью метода void deleteUnit(unsigned newX, unsigned newY), который принимает координаты для удаления юнита с поля. Метод удаления представлен на рисунке 6.

```
//----- Удаление юнита -----//
void Field::deleteUnit(unsigned newX, unsigned newY) {

    if (newX <= width && newY <= height) {
        countObjects--;
        obj[newX][newY] = nullptr;
    }
}
```

Рисунок 6 – метод удаления юнита на поле.

Демонстрационный пример добавления юнита на поле можно увидеть на рисунках 3,4 представленных в пункте 2).

Демонстрационный пример удаления юнита с поля:

Удаляется персонаж `elf_on_unicorn`, координаты которого (0,2).

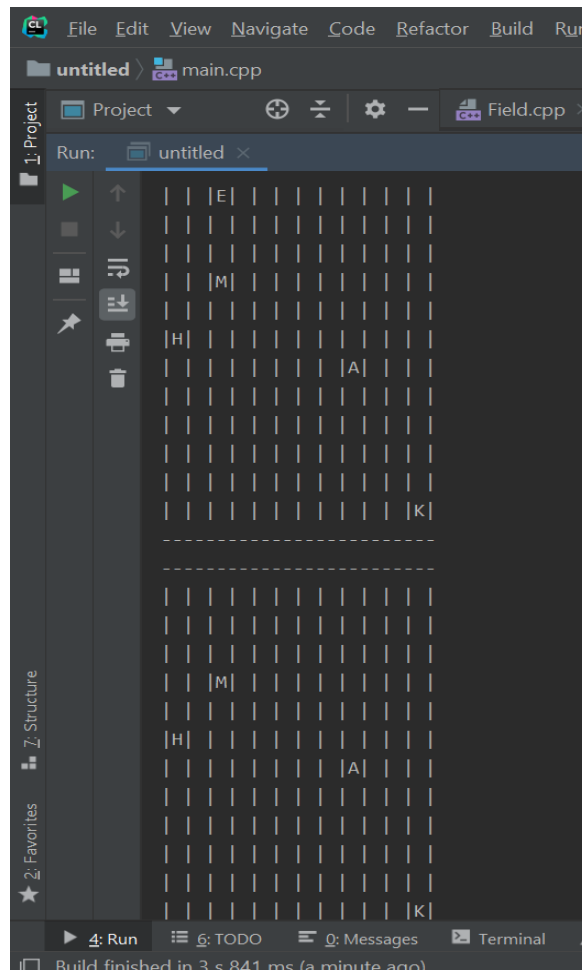
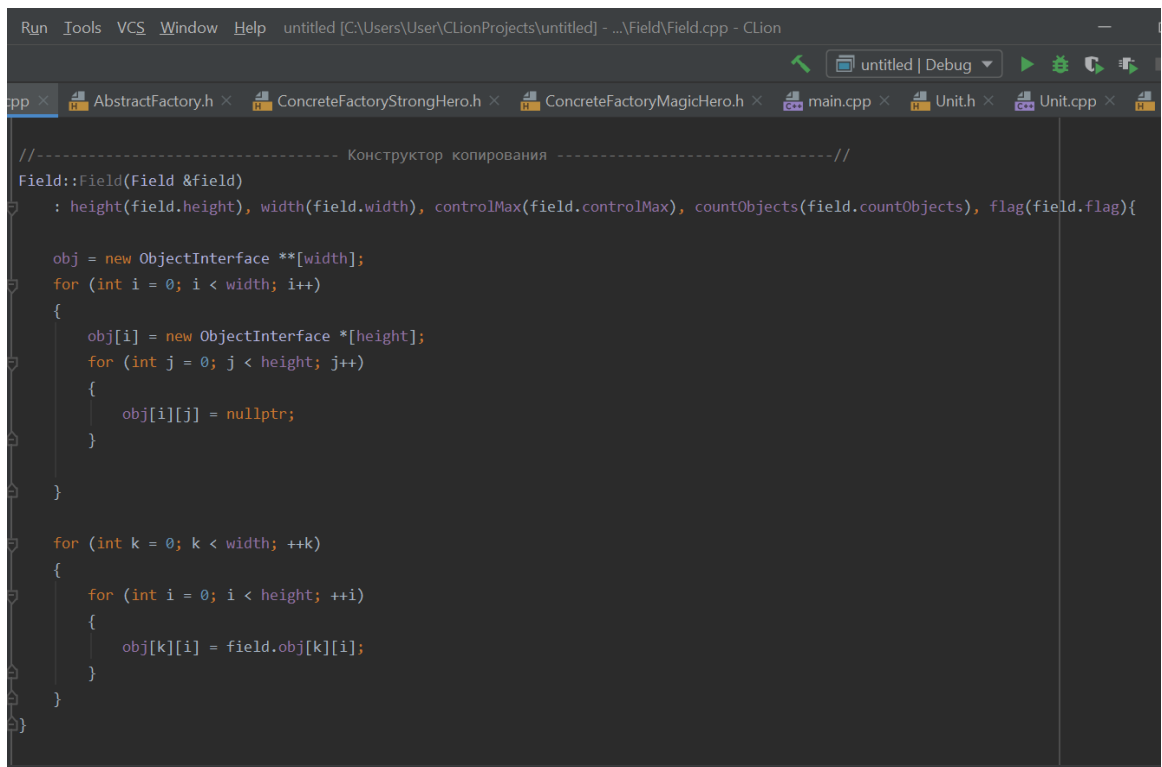


Рисунок 6 –демонстрационный пример удаления юнита на поле.

4) Возможность копирования поля (включая объекты на нем)

Для копирования поля(включая объекты на нем) был создан конструктор копирования и также оператор копирования. Конструктор копирования и оператор копирования см. соответственно на рисунках 7 и 8.



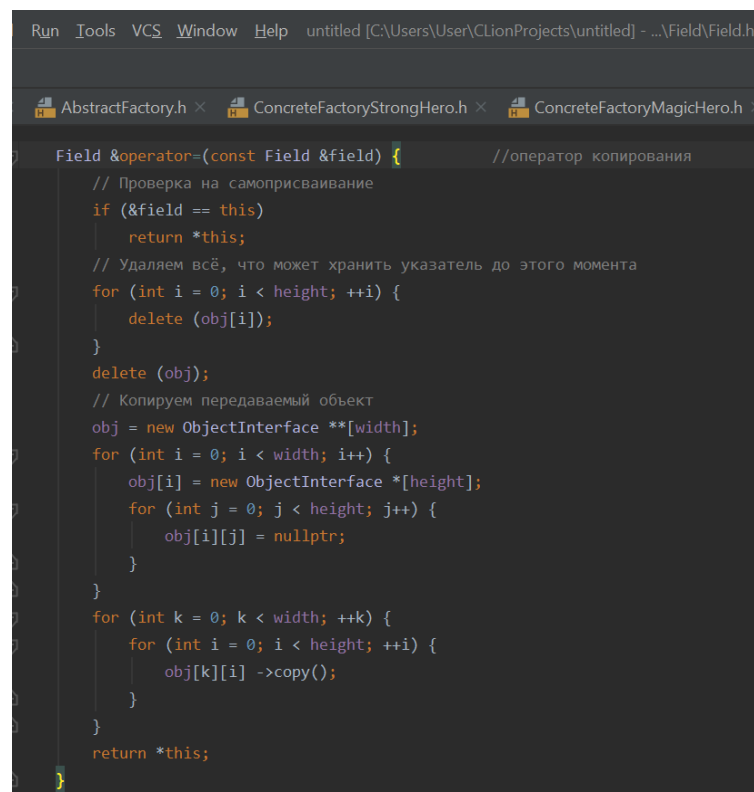
```
Run Tools VCS Window Help untitled [C:\Users\User\CLionProjects\untitled] - ...Field\Field.cpp - CLion
untitled | Debug
AbstractFactory.h ConcreteFactoryStrongHero.h ConcreteFactoryMagicHero.h main.cpp Unit.h Unit.cpp

//----- Конструктор копирования -----//
Field::Field(Field &field)
: height(field.height), width(field.width), controlMax(field.controlMax), countObjects(field.countObjects), flag(field.flag){

    obj = new ObjectInterface **[width];
    for (int i = 0; i < width; i++)
    {
        obj[i] = new ObjectInterface *[height];
        for (int j = 0; j < height; j++)
        {
            obj[i][j] = nullptr;
        }
    }

    for (int k = 0; k < width; ++k)
    {
        for (int i = 0; i < height; ++i)
        {
            obj[k][i] = field.obj[k][i];
        }
    }
}
```

Рисунок 7 – конструктор копирования.



```
Run Tools VCS Window Help untitled [C:\Users\User\CLionProjects\untitled] - ...Field\Field.h
AbstractFactory.h ConcreteFactoryStrongHero.h ConcreteFactoryMagicHero.h

Field &operator=(const Field &field) { //оператор копирования
    // Проверка на самоприсваивание
    if (&field == this)
        return *this;
    // Удаляем всё, что может хранить указатель до этого момента
    for (int i = 0; i < height; ++i) {
        delete (obj[i]);
    }
    delete (obj);
    // Копируем передаваемый объект
    obj = new ObjectInterface **[width];
    for (int i = 0; i < width; i++) {
        obj[i] = new ObjectInterface *[height];
        for (int j = 0; j < height; j++) {
            obj[i][j] = nullptr;
        }
    }
    for (int k = 0; k < width; ++k) {
        for (int i = 0; i < height; ++i) {
            obj[k][i] ->copy();
        }
    }
    return *this;
}
```

Рисунок 8 – оператор копирования.

Демонстрационный пример работы конструктора копирования:


```
Run Tools VCS Window Help untitled [C:\Users\User\CLionProjects\untitled] - ...main.cpp - CLion
untitled | D
AbstractFactory.h ConcreteFactoryStrongHero.h ConcreteFactoryMagicHero.h main.cpp Unit.h

//создание поля 1
Field field1( height: 12, width: 12, controlMax: 5);

//создание юнитов
Unit* magician = f1->CreateArtillerist();
Unit* archer = f2->CreateArtillerist();
Unit* elf_on_unicorn = f1->CreateCavalryman();
Unit* knight_on_horse = f2->CreateCavalryman();
Unit* gnome = f1->CreateInfantryman();
Unit* knight = f2->CreateInfantryman();

//Добавление объектов на поле
field1.appendUnit( newX: 3, newY: 2, magician);
field1.appendUnit( newX: 6, newY: 8, archer);
field1.appendUnit( newX: 11, newY: 11, knight);
field1.appendUnit( newX: 0, newY: 2, elf_on_unicorn);
field1.appendUnit( newX: 5, newY: 0, knight_on_horse);
field1.appendUnit( newX: 6, newY: 0, gnome); //для демонстрационного примера добавляется шестой юнит,
// но при выводе будет видно, что он не добавится, т к
// уже добавлено максимальное количество юнитов на поле

std::cout << "Первое поле:" << std::endl;
field1.drawField();
auto* field2 = new Field(field1);
std::cout << "Второе поле:" << std::endl;
field2->drawField();
```

Рисунок 9 – демонстрационный пример.

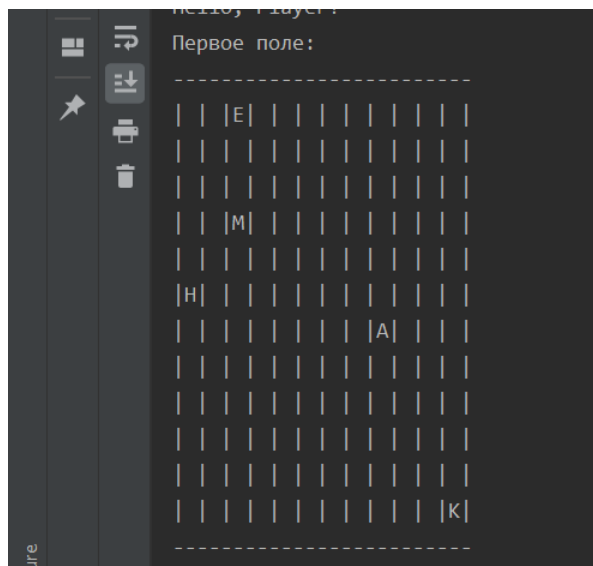


Рисунок 10 – вывод игрового поля1.

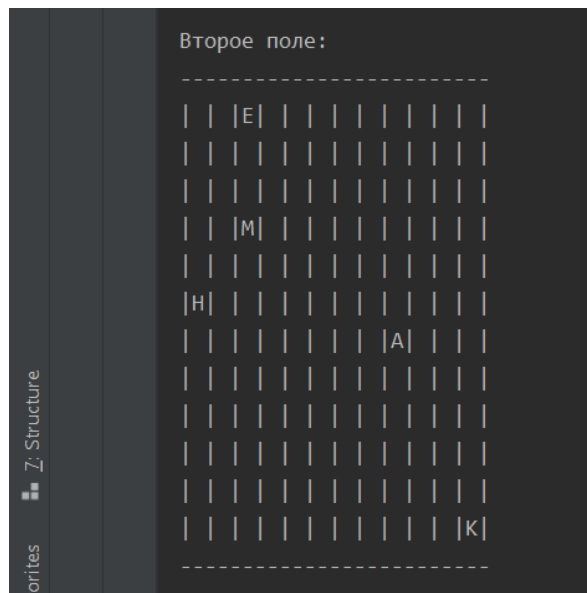


Рисунок 11 – вывод копии игрового поля1.

Также в классе представлены методы вывода поля на экран - `void drawField()` и перемещения юнита по полю - `void move(int x, int y, int newX, int newY)`. (Представлены на рисунках 12 и 13 соответственно.)

```

Run Tools VCS Window Help untitled [C:\Users\User\CLionProjects\untitled]
AbstractFactory.h ConcreteFactoryStrongHero.h ConcreteFactoryWeakHero.h
void Field::drawField() {
    if(!flag){
        for (int k = 0; k < width * 2 + 1; k++)
            std::cout << '-';
        std::cout << std::endl;
        for (int i = 0; i < width; ++i){
            std::cout << '|';
            for (int j = 0; j < height; ++j){
                if (obj[i][j] != nullptr){
                    obj[i][j]->whatYouName();
                    std::cout << '|';
                }
                else
                {
                    std::cout << " ";
                    std::cout << '|';
                }
            }
            std::cout << std::endl;
        }
        for (int k = 0; k < width * 2 + 1; k++)
            std::cout << '-';
        std::cout << std::endl;
    }
}
}
  
```

Рисунок 12 – метод вывода поля на экран.

```
//----- Передвижение юнита -----//  
void Field::move( int x, int y, int newX, int newY)  
{  
    if (newX <= width && newY <= height)//проверка, что в пределах поля  
    {  
        if(obj[newX][newY] == nullptr) //проверка, что там никто не стоит  
        {  
            obj[newX][newY] = obj[x][y];  
            deleteUnit(x, y);  
        }  
    }  
}
```

Рисунок 13 – метод перемещения юнита.

Демонстрационный пример перемещения юнита на поле представлен на рисунках 14,15:

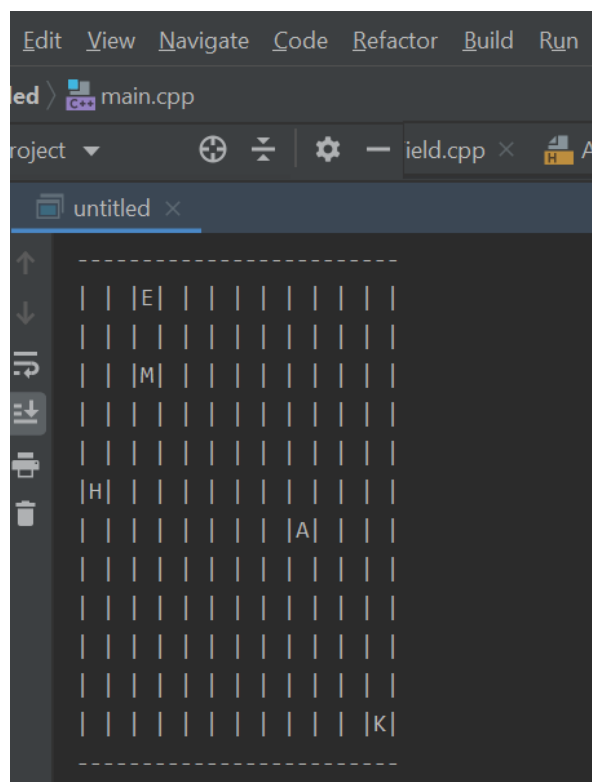


Рисунок 14

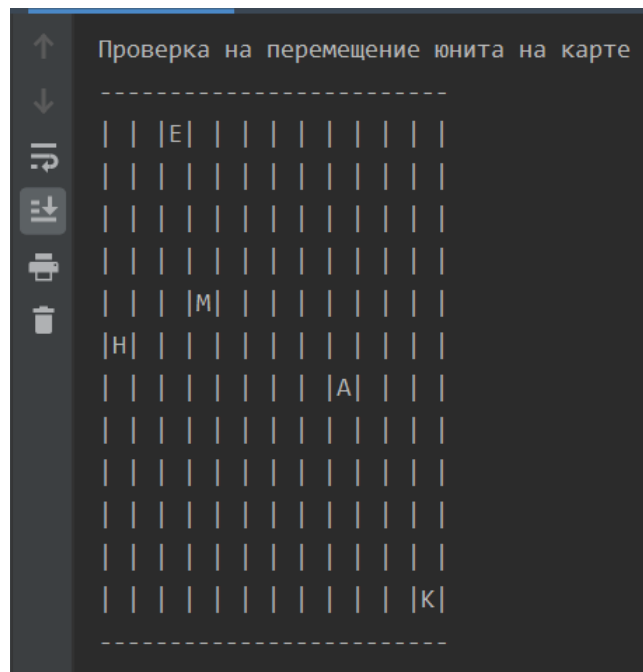


Рисунок 15

Класс Field

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

- Все юниты должны иметь как минимум один общий интерфейс
- и
- Реализованы 3 типа юнитов (например, пехота, лучники, конница)
- и
- Реализованы 2 вида юнитов для каждого типа (например, для пехоты могут быть созданы мечники и копейщики)

Изначально был создан класс-интерфейс объекта на поле ObjectInterface, в том числе и юнита, в котором содержатся виртуальные методы: whatYouName() – метод для вывода на поле имя юнита, copy() – метод для копирования юнита, этот метод необходим для копирования поля с объектами на нем. На рисунке 16 показано содержание файла ObjectInterface.

```
//----- Интерфейс объекта -----//
class ObjectInterface{
public:

    virtual void whatYouName() = 0; //метод для отображения юнита на поле как специальной буквы
    virtual ObjectInterface* copy() = 0; //метод копирования
};
```

Рисунок 16 – ObjectInterface.h

От ObjectInterface наследуется класс юнита Unit и уже от этого класса наследуются 3 типа юнита:

- Артиллерист – Artillerist.h/Artillerist.cpp
- Кавалерист – Cavalryman.h/Cavalryman.cpp
- Пехотинец – Infantryman.h/Infantryman.cpp

От этих трех классов наследуются по два вида юнита. То есть от Артиллериста наследуется: Archer и Magician, от Кавалериста: KnightOnHorse и ElfOnUnicorn, от Пехотинца: Knight и Gnome.

- Юниты имеют характеристики, отражающие их основные атрибуты, такие как здоровье, броня, атака.

У каждого конкретного юнита определены свои характеристики: здоровье, броня, урон, а также имя юнита. На рисунках 17-22 показано конструкторы каждого из юнитов, где для каждой характеристики присваиваются свои значения.

```
Archer::Archer() : Artillerist() {
    Artillerist::health = 100;
    Artillerist::armor = 80;
    Artillerist::damage = 45;
    Artillerist::name = 'A';
}
```

Рисунок 17 – Archer.cpp

```
Magician::Magician() : Artillerist() {  
    Artillerist::health = 100;  
    Artillerist::armor = 75;  
    Artillerist::damage = 50;  
    Artillerist::name = 'M';  
}
```

Рисунок 18 – Magician.cpp

```
KnightOnHorse::KnightOnHorse(): Cavalryman() {  
    Cavalryman::health = 100;  
    Cavalryman::armor = 50;  
    Cavalryman::damage = 50;  
    Cavalryman::name = 'H';  
}
```

Рисунок 19 – KnightOnHorse.cpp

```
ElfOnUnicorn::ElfOnUnicorn(): Cavalryman() {  
    Cavalryman::health = 100;  
    Cavalryman::armor = 30;  
    Cavalryman::damage = 70;  
    Cavalryman::name = 'E';  
}
```

Рисунок 20 – ElfOnUnicorn.cpp

```
Knight::Knight(): Infantryman() {  
    Infantryman::health = 150;  
    Infantryman::armor = 50;  
    Infantryman::damage = 35;  
    Knight::name = 'K';  
}
```

Рисунок 21 – Knight.cpp

```
Gnome::Gnome(): Infantryman() {
    Infantryman::health = 150;
    Infantryman::armor = 30; //т.к.
    Infantryman::damage = 50;

    Infantryman::name = 'G';
}
```

Рисунок 22 – Gnome.cpp

- Юнит имеет возможность перемещаться по карте

Юнит перемещается по полю с помощью метода `move()`, который содержится в файле `Field.cpp`. Он принимает координаты текущего положения на карте юнита, и координаты куда его нужно переместить.

```
//----- Передвижение юнита -----//
void Field::move( int x, int y, int newX, int newY)
{
    if (newX <= width && newY <= height) //проверка, что в пределах поля
    {
        if(obj[newX][newY] == nullptr) //проверка, что там никто не стоит
        {
            obj[newX][newY] = obj[x][y];
            obj[x][y] = nullptr;
        }
    }
}
```

Рисунок 23 – Метод передвижения юнита.

Таким образом, все основные требования выполнены.

Дополнительные требования

**Созданы конструкторы копирования и перемещения*

Конструктор копирования уже был продемонстрирован в 4 пункте.

Конструктор перемещения находится в файле Field.cpp, оператор перемещения в файле Field.h(рис.16,17)

```
Field::Field(Field &&field)//конструктор перемещения
:width(field.width), height(field.height), obj(field.obj), controlMax(field.controlMax), countObjects(field.countObjects)
{
    field.obj = nullptr;
    field.width = 0;
    field.height = 0;
}
```

Рисунок 16 – Конструктор перемещения

```
//----- Оператор перемещения -----//

Field &operator=(Field &&field) {

    if (&field == this)
        return *this;
    for (int i = 0; i < this->width; i++) {
        delete (this->obj[i]);
        for (int j = 0; j < this->height; j++) {
            delete &this->obj[i][j];
        }
    }
    delete this->obj;

    this->height = field.height;
    this->width = field.width;

    this->obj = field.obj;
    return *this;
}
```

Рисунок 17 – Оператор перемещения

Также выполнено дополнительное требование:

**Все методы принимают параметры оптимальным образом (то есть, отсутствует лишнее копирование объектов)*

Выполнено требование:

**Для создания юнитов используются паттерны “Фабричный метод” / “Абстрактная фабрика”*

Абстрактная фабрика.

Абстрактная фабрика — это порождающий паттерн проектирования, который позволяет создавать семейства связанных объектов, не привязываясь к конкретным классам создаваемых объектов.

В данной реализации герои игры делятся на магических и сильных героев. То есть существуют волшебный артиллерист — маг, волшебный кавалерист — эльф на единороге и волшебный пехотинец — гном. Также существуют сильный артиллерист — лучник, сильный кавалерист — рыцарь на коне и сильный пехотинец — рыцарь.

Класс абстрактной фабрики: (файл AbstractFactory.h)

```
class AbstractFactory {  
  
    public:  
        virtual Artillerist *CreateArtillerist() const = 0;  
        virtual Cavalryman *CreateCavalryman() const = 0;  
        virtual Infantryman *CreateInfantryman() const = 0;  
  
};
```

ConcreteFactoryMagicHero.h

```

class ConcreteFactoryMagicHero : public AbstractFactory {
public:
    Artillerist *CreateArtillerist() const override {
        return new Magician();
    }
    Cavalryman *CreateCavalryman() const override {
        return new ElfOnUnicorn();
    }
    Infantryman *CreateInfantryman() const override {
        return new Gnome();
    }
};

```

ConcreteFactoryStrongHero.h

```

class ConcreteFactoryStrongHero : public AbstractFactory {
public:
    Artillerist *CreateArtillerist() const override {
        return new Archer();
    }
    Cavalryman *CreateCavalryman() const override {
        return new KnightOnHorse();
    }
    Infantryman *CreateInfantryman() const override {
        return new Knight();
    }
};

```

Демонстрационный пример использования абстрактной фабрики в main.cpp:

```
// создание фабрик
auto *f1 = new ConcreteFactoryMagicHero();
auto *f2 = new ConcreteFactoryStrongHero();

//создание поля 1
Field field1( height: 12, width: 12, controlMax: 5);
//создание юнитов
Unit* magician = f1->CreateArtillerist();
Unit* archer = f2->CreateArtillerist();
Unit* elf_on_unicorn = f1->CreateCavalryman();
Unit* knight_on_horse = f2->CreateCavalryman();
Unit* gnome = f1->CreateInfantryman();
Unit* knight = f2->CreateInfantryman();
```

Выводы.

В ходе лабораторной работы была реализована программа, удовлетворяющая необходимым и некоторым дополнительным требованиям.