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.

Студент гр. 8304 _____ Ястребов И.М.

Преподаватель _____ Размочаева Н.В.

Санкт-Петербург

2020

Цель работы.

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

Задание.

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 объектом, 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

Ход работы.

Разработаны и реализованы классы:

- Класс игрового поля Field

- Набор классов юнитов: Mortar, Catapult, Spearman, Rogue, HorsemenLeader, RegularHorseman

Класс игрового поля Field является контейнером для объектов представляющим собой прямоугольную сетку. Реализована возможность создания поля произвольного размера. Для работы с полем созданы методы удаления и добавления юнитов, передвижения юнитов, атаки юнитов, конструкторы и операторы копирования, перемещения и присваивания.

Все классы юнитов имеют один общий интерфейс, описанный в базовом абстрактном классе Unit. Созданы три подтипа юнитов. Для каждого подтипа создано еще два подтипа.

Выводы.

В ходе выполнения работы были разработаны и реализованы классы игрового поля и набор классов юнитов. Так же получены навыки по созданию абстрактных классов, интерфейсов классов, деструкторов и конструкторов, методов классов и работе с наследованием классов.

Приложение А.

Исходный код.

Файл Unit.cpp:

```
#include "pch.h"

#include <iostream>
#include "Unit.h"
#include "Field.h"
#define DEFAULT 1

Unit::~Unit() = default;

void Unit::attack(Field *f, Unit* target) {
	if (!f) {
		std::cout << "Invalid field passed\n";
		return;
	}

	if (!target) {
		std::cout << "Invalid target passed\n";
		return;
	}

	if (sqrt(pow(abs(this->getPosition().first - target-
>getPosition().first), 2)
		+ pow(abs(this->getPosition().second - target-
>getPosition().second), 2)) <= this->getRange())
	{
		target->actDamaged(this->getDamage());
		return;
	}

	else {
		std::cout << "Out of attack range\n";
		return;
	}

}
```

```
void Unit::move(Field *f, int dX, int dY) {
	if (!f) {
		std::cout << "Invalid field passed\n";
		return;
	}

	if (!(f->moveUnit(this->getPosition().first, this->getPosition().second, dX, dY))) {
		std::cout << "Move failed\n";
		return;
	}
	this->setPosition(this->getPosition().first + dX, this->getPosition().second + dY);
}
```

```
Ballista::~Ballista() = default;

Ballista::Ballista(int health, int dmg, int range, int armor) {
	if (health <= 0) {
		std::cout << "Cannot create negative-health unit, health set to DEFAULT\n";
		this->setHealth(DEFAULT);
	}
	else {
		this->setHealth(health);
	}

	if (dmg <= 0) {
		std::cout << "Cannot create negative-dmg unit, health set to DEFAULT\n";
		this->setDamage(DEFAULT);
	}
	else {
		this->setDamage(dmg);
	}

	this->setRange(range);
```

```
	if (range <= 0) {
		std::cout << "Cannot create negative-attack-range unit, health
set to DEFAULT\n";
		this->setRange(DEFAULT);
	}
	else {
		this->setRange(range);
	}

	this->setArmor(armor);
	if (armor <= 0) {
		std::cout << "Cannot create negative-armor unit, health set to
DEFAULT\n";
		this->setArmor(DEFAULT);
	}
	else {
		this->setArmor(armor);
	}
}



Berserker::~Berserker() = default;

Berserker::Berserker(int health, int dmg, int range, int armor) {
	if (health <= 0) {
		std::cout << "Cannot create negative-health unit, health set to
DEFAULT\n";
		this->setHealth(DEFAULT);
	}
	else {
		this->setHealth(health);
	}

	if (dmg <= 0) {
		std::cout << "Cannot create negative-dmg unit, health set to
DEFAULT\n";
		this->setDamage(DEFAULT);
	}
	else {
		this->setDamage(dmg);
```

```
	}

	this->setRange(range);
	if (range <= 0) {
		std::cout << "Cannot create negative-attack-range unit, health
set to DEFAULT\n";
		this->setRange(DEFAULT);
	}
	else {
		this->setRange(range);
	}

	this->setArmor(armor);
	if (armor <= 0) {
		std::cout << "Cannot create negative-armor unit, health set to
DEFAULT\n";
		this->setArmor(DEFAULT);
	}
	else {
		this->setArmor(armor);
	}
}



Horseman::~Horseman() = default;

Horseman::Horseman(int health, int dmg, int range, int armor) {
	if (health <= 0) {
		std::cout << "Cannot create negative-health unit, health set to
DEFAULT\n";
		this->setHealth(DEFAULT);
	}
	else {
		this->setHealth(health);
	}

	if (dmg <= 0) {
		std::cout << "Cannot create negative-dmg unit, health set to
DEFAULT\n";
		this->setDamage(DEFAULT);
```

```
	}
	else {
		this->setDamage(dmg);
	}

	this->setRange(range);
	if (range <= 0) {
		std::cout << "Cannot create negative-attack-range unit, health set to DEFAULT\n";
		this->setRange(DEFAULT);
	}
	else {
		this->setRange(range);
	}

	this->setArmor(armor);
	if (armor <= 0) {
		std::cout << "Cannot create negative-armor unit, health set to DEFAULT\n";
		this->setArmor(DEFAULT);
	}
	else {
		this->setArmor(armor);
	}
}
```

Файл Unit.h:

```
#pragma once

#include <cmath>
#include <utility>
#include <memory>

class Field;

class location {
public:
	location(int x, int y) {
		this->position.first = x;
		this->position.second = y;
	}

	explicit location(std::pair<int, int> pos) {
		this->position = pos;
	}

	location() = default;
	~location() = default;

	void setPosition(int x, int y) {
		this->position.first = x;
		this->position.second = y;
	}

	void setPosition(std::pair<int, int> pos) {
		this->position = pos;
	}

	std::pair<int, int> getPosition() const {
		return position;
	}

private:
```

```
	std::pair<int, int> position;


};


class Health
{
public:
	Health() {
		health = 0;
	}
	~Health() = default;

	explicit Health(int h) {
		this->health = h;
	}

	int getHealth() {
		return health;

	}

	void setHealth(int h) {
		this->health = h;
	}

	void actDamaged(int dmg) {

		this->health -= dmg;
	}

	void actHealed(int heal) {
		this->health += heal;
	}

private:
	int health;

};
```

```
class Damage
{
public:
	Damage() {
		damage = 0;
		range = 0;
	}
	~Damage() = default;

	Damage(int d, int r) {
		this->damage = d;
		this->range = r;
	}

	int getDamage() const {
		return damage;

	}

	void setDamage(int d) {
		damage = d;
	}

	int getRange() const {
		return range;

	}

	void setRange(int r) {
		range = r;
	}

	void actDecreaseDmg(int debuff) {

		damage -= debuff;
	}

	void actIncreaseDmg(int buff) {
		damage += buff;
```

```
	}

private:
	int damage;
	int range;

};


class Armor
{
public:

	Armor() {
		armor = 0;
	}
	~Armor() = default;

	explicit Armor(int a) {
		this->armor = a;
	}

	int getArmor() const {
		return armor;

	}

	void setArmor(int a) {
		armor = a;
	}

	void actDecreaseArmor(int debuff) {

		armor -= debuff;
	}

	void actIncreaseArmor(int buff) {
		armor += buff;
	}

private:
```

```
    int armor;

};


class Unit :public Armor, public Health, public Damage, public location
{
public:
    Unit() = default;

    virtual Unit* clone() = 0;

    virtual ~Unit() = 0;

    void move(Field *f, int dX, int dY);

    void attack(Field *f, Unit* target);
};



class Ballista :public Unit
{
public:
    Ballista() = default;

    explicit Ballista(int health, int dmg, int range, int armor);

    ~Ballista() override = 0;

};



class Mortar : public Ballista {

public:
    Mortar() : Ballista(500, 50, 10, 15)
    {};

    Unit* clone() final {
```

```
            auto tmp = new Mortar;
            tmp->setHealth(this->getHealth());
            tmp->setDamage(this->getDamage());
            tmp->setRange(this->getRange());
            tmp->setArmor(this->getArmor());
            return (Unit*)tmp;
      }
      ~Mortar() final = default;
};

class Catapult : public Ballista {
public:
      Catapult() : Ballista(300, 30, 10, 8)
      {};
      Unit* clone() final {
            auto tmp = new Catapult;
            tmp->setHealth(this->getHealth());
            tmp->setDamage(this->getDamage());
            tmp->setRange(this->getRange());
            tmp->setArmor(this->getArmor());
            return (Unit*)tmp;
      }
      ~Catapult() final = default;
};
```

```
class Berserker : public Unit {

public:

	Berserker() = default;

	explicit Berserker(int health, int dmg, int range, int armor);

	~Berserker() override = 0;

};



class Spearman : public Berserker {

public:
	Spearman() : Berserker(1400, 75, 2, 35)
	{};

	Unit* clone() final {
		auto tmp = new Spearman;

		tmp->setHealth(this->getHealth());

		tmp->setDamage(this->getDamage());

		tmp->setRange(this->getRange());

		tmp->setArmor(this->getArmor());

		return (Unit*)tmp;

	}

	~Spearman() final = default;
};
```

```
class Rogue : public Berserker {

public:
	Rogue() : Berserker(900, 40, 5, 25)
	{};

	Unit* clone() final {
		auto tmp = new Rogue;

		tmp->setHealth(this->getHealth());

		tmp->setDamage(this->getDamage());

		tmp->setRange(this->getRange());

		tmp->setArmor(this->getArmor());

		return (Unit*)tmp;

	}

	~Rogue() final = default;
};



class Horseman : public Unit {

public:

	Horseman() = default;

	explicit Horseman(int health, int dmg, int range, int armor);

	~Horseman() override = 0;
};



class HorsemenLeader : public Horseman {
```

```
public:
	HorsemenLeader() : Horseman(800, 60, 8, 20)
	{};

	Unit* clone() final {
		auto tmp = new HorsemenLeader;

		tmp->setHealth(this->getHealth());

		tmp->setDamage(this->getDamage());

		tmp->setRange(this->getRange());

		tmp->setArmor(this->getArmor());

		return (Unit*)tmp;

	}

	~HorsemenLeader() final = default;
};


class RegularHorseman : public Horseman {

public:
	RegularHorseman() : Horseman(600, 55, 8, 20)
	{};

	Unit* clone() final {
		auto tmp = new RegularHorseman;

		tmp->setHealth(this->getHealth());

		tmp->setDamage(this->getDamage());

		tmp->setRange(this->getRange());

		tmp->setArmor(this->getArmor());

		return (Unit*)tmp;
```

```
	}

	~RegularHorseman() final = default;
};
```

Файл FieldIterator.cpp:

```
#include "pch.h"


#include "FieldIterator.h"
#include "Unit.h"
#include "Field.h"


Iterator::Iterator(const Field& gameField) : gameField(gameField)
{
	this->i = 0;
	this->j = 0;
```

```
	this->width = gameField.getSize().width;
	this->height = gameField.getSize().height;
}


bool Iterator::hasNext() const
{
	return i < gameField.getSize().height && j <
gameField.getSize().width;

}


void Iterator::first()
{
	j = 0;
	i = 0;
}


const Iterator& operator++(Iterator& it)
{
	if (it.j + 1 < it.width) {
		++it.j;
	}
	else {
		++it.i;
		it.j = 0;
	}

	return it;
}

const Iterator& operator--(Iterator& it)
{
	if (it.j - 1 >= 0) {
		--it.j;
	}
	else {
		--it.i;
		it.j = it.width - 1;
```

```
	}

	return it;
}


Unit* Iterator::operator*() const
{
	return ((gameField.GetHead())[i * this->width + j]);
}
```

Файл FieldIterator.h:

```
#pragma once

#ifndef OOP_FIELDITERATOR_H
#define OOP_FIELDITERATOR_H

#include <cstdio>


class Field;
class Unit;

class Iterator
{
public:
	explicit Iterator(const Field& gameField);

	bool hasNext() const;
	void first();
	friend const Iterator& operator--(Iterator& it);
	friend const Iterator& operator++(Iterator& it);
	Unit* operator*() const;


private:
	size_t i;
	size_t j;
	size_t width;
	size_t height;
	const Field& gameField;
};
```

```
#endif //OOP_FIELDITERATOR_H
```

Файл Field.cpp:

```
#include "pch.h"

#include <iostream>
#include "Field.h"
#include "Unit.h"



Field::~Field() {
	for (int i = 0; i < size.width * size.height; ++i)
		delete head[i];

	delete[] head;
}

//constructor copy
//operator=

Field::Field(const Field& field) {
	if (this != &field) {

		this->max = field.max;

		this->currentQuantity = field.currentQuantity;

		this->size = field.size;

		delete[] head;

		this->head = new Unit *[field.size.width * field.size.height];
		for (int i = 0; i < this->size.width * this->size.height; ++i) {
			head[i] = nullptr;

			for (int i = 0; i < field.size.width * field.size.height; +
+i) {
```

```
                        if (!field.head[i])
                              continue;
                        else
                              this->head[i] = field.head[i]->clone();
                  }
            }
      }
}

Field::Field(int width, int height) {
      if ((width <= 0) || (height <= 0)) {
            std::cout << "Empty/Negative size field cannot be created\n";
            return;
      }

      this->head = new Unit*[width * height];
      this->size.width = width;
      this->size.height = height;

      for (int i = 0; i < this->size.width * this->size.height; ++i) {
            head[i] = nullptr;
      }

}

Field::Field(Field&& other) noexcept {

      this->currentQuantity = other.currentQuantity;
      this->size = other.size;
      this->max = other.max;

      this->head = other.head;
      other.head = nullptr;

}


Field& Field::operator=(const Field &field) {

      if (this != &field) {
```

```
            this->max = field.max;

            this->currentQuantity = field.currentQuantity;

            this->size = field.size;

            delete[] head;

            this->head = new Unit*[field.size.width * field.size.height];

            for (int i = 0; i < field.size.width * field.size.height; ++i)
                  this->head[i] = field.head[i]->clone();

      }
      return *this;
}


bool Field::moveUnit(int xPos, int yPos, int dX, int dY) {
      if (((dX > (this->getSize().width - 1) - xPos) || (dY > (this-
>getSize().height - 1) - yPos))
            || ((xPos + dX < 0) || (yPos + dY < 0))) {
            std::cout << "Cant step out of field's borders\n";
            return false;
      }

      if (head[yPos * size.width + xPos] == nullptr ||
            head[(yPos + dY) * size.width + xPos + dX] != nullptr) {
            std::cout << "Can't move non-existent object / Can't move to a
taken position\n";
            return false;
      }

      head[(yPos + dY) * size.width + xPos + dX] = head[yPos * size.width +
xPos];
      head[yPos * size.width + xPos] = nullptr;
      return true;
}


/*bool Field::attackUnit(int xAtt, int yAtt, int xTarg, int yTarg) {
```

```
	if(head[yAtt * size.width + xAtt] == nullptr ) {
		std::cout << "Invalid position of the attacker unit\n";
		return false;
	}
	if(head[yTarg * size.width + xTarg] == nullptr){
		std::cout  << "Invalid position of the target unit\n";
		return false;
	}
	((Unit*)head[yAtt * size.width + xAtt])->attack(this,
(Unit*)head[yTarg * size.width + xTarg]);
	return true;
}*/


void Field::addObj(Unit* object, int xPos, int yPos) {
	if (object == nullptr) {
		std::cout << "If you want to delete an object use removeObject
instead\n";
		return;
	}
	if (currentQuantity + 1 <= max) {
		if (head[size.width * yPos + xPos] == nullptr) {
			head[size.width * yPos + xPos] = object;
			++currentQuantity;
			((Unit *)object)->setPosition(xPos, yPos);
		}
		else {
			std::cout << "This position is already taken by another
unit\n";
			return;
		}
	}
	else {
		std::cout << "Field contains maximum quantity of objects\n";
	}

}



void Field::removeObj(location pos) {
```

```
	if (head[(pos.getPosition().second - 1) * this->size.width +
pos.getPosition().first] == nullptr) {
		std::cout << "Invalid position of unit\n";
		return;
	}
	head[(pos.getPosition().second - 1) * this->size.width +
pos.getPosition().first] = nullptr;
	--currentQuantity;
}



f_size Field::getSize() const {
	return this->size;
}


int Field::getMax() const {
	return this->max;
}


void Field::setLimit(int lim) {
	if (lim < 0) {
		std::cout << "Field cannot contain negative quantity of objects\
n";
	}
	this->max = lim;
}


int Field::getQuantity() const {
	return this->currentQuantity;
}

Unit **Field::GetHead() const {
	return this->head;
}


Iterator* Field::getIterator() const {
```

```
	return (new Iterator(*this));
}
```

Файл Field.h:

```
#pragma once
#include <variant>
#include <utility>
#include <memory>
#include "Unit.h"
#include "FieldIterator.h"
#include <cstring>


typedef struct fieldSize {
	int width, height;

	fieldSize(int width, int height) {
		this->width = width;
		this->height = height;
	}

	fieldSize(const fieldSize &f) {
		this->width = f.width;
		this->height = f.height;
	}

	fieldSize() = default;

}f_size;


class Field
{
public:

	Field() = default;

	~Field();

	Field(const Field &f);
```

```
        Field(int width, int height);

        Field(Field&& other) noexcept;

        Field& operator=(const Field &field);

        bool moveUnit(int xPos, int yPos, int dX, int dY);

        //   bool attackUnit(int xAtt, int yAtt, int xTarg, int yTarg);

        void addObj(Unit* object, int xPos, int yPos);

        void removeObj(location pos);

        f_size getSize() const;

        int getMax() const;

        void setLimit(int lim);

        int getQuantity() const;

        Unit** GetHead() const;

        Iterator* getIterator() const;


    private:
        f_size size = { 0, 0 };

        Unit** head = nullptr;

        int max = 15;

        int currentQuantity = 0;

    };
```

Файл Game.cpp:

```
#include "pch.h"

#include <iostream>
#include "Unit.h"
#include "Field.h"
#include "FieldIterator.h"

void doNothing(Field fi) {
	//hi

};

int main()
{

	Field f(10, 10);

	Catapult* danilkin = new Catapult;
	HorsemenLeader* vanyusha = new HorsemenLeader;

	f.addObj((Unit*)danilkin, 1, 1);
	f.addObj((Unit*)vanyusha, 2, 2);

	danilkin->attack(&f, vanyusha);

	danilkin->move(&f, -1, -1);

	Iterator it(f);

	it.first();

	++it;
	--it;

	auto tmp = (*it);

	doNothing(f);

	return 0;
}
```