

**МИНОБРНАУКИ РОССИИ**  
**САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ**  
**ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ**  
**«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)**  
**Кафедра МО ЭВМ**

**ОТЧЕТ**  
**по лабораторной работе №2**  
**по дисциплине «Объектно-ориентированное программирование»**  
**Тема: Интерфейсы, полиморфизм**

Студент гр. 0383:

\_\_\_\_\_

Бояркин Н.А.

Преподаватель:

\_\_\_\_\_

Жангиров Т.Р.

Санкт-Петербург 2021

### **Цель работы.**

Изучить механизм работы интерфейсов и реализовывать функции, соблюдая при этом принцип полиморфизма.

### **Задание.**

Могут быть три типа элементов располагающихся на клетках:

1. Игрок - объект, которым непосредственно происходит управление. На поле может быть только один игрок. Игрок может взаимодействовать с врагом (сражение) и вещами (подобрать).
2. Враг - объект, который самостоятельно перемещается по полю. На поле врагов может быть больше одного. Враг может взаимодействовать с игроком (сражение).
3. Вещь - объект, который просто располагается на поле и не перемещается. Вещей на поле может быть больше одной.

### **Требования:**

- Реализовать класс игрока. Игрок должен обладать собственными характеристиками, которые могут изменяться в ходе игры. У игрока должна быть прописана логика сражения и подбора вещей. Должно быть реализовано взаимодействие с клеткой выхода.
- Реализовать три разных типа врагов. Враги должны обладать собственными характеристиками (например, количество жизней, значение атаки и защиты, и.т.д. Желательно, чтобы у врагов были разные наборы характеристик). Реализовать логику перемещения для каждого типа врага. В случае смерти врага он должен исчезнуть с поля. Все враги должны быть объединены своим собственным интерфейсом.

- Реализовать три разных типа вещей. Каждая вещь должна обладать собственным взаимодействием на ход игры при подборе. (*например, лечение игрока*). При подборе, вещь должна исчезнуть с поля. Все вещи должны быть объединены своим собственным интерфейсом.
- Должен соблюдаться принцип полиморфизма

*Потенциальные паттерны проектирования, которые можно использовать:*

- *Шаблонный метод (Template Method) - определение шаблона поведения врагов*
- *Стратегия (Strategy) - динамическое изменение поведения врагов*
- *Легковес (Flyweight) - вынесение общих характеристик врагов и/или для оптимизации*
- *Абстрактная Фабрика/Фабричный Метод (Abstract Factory/Factory Method) - создание врагов/вещей разного типа в runtime*
- *Прототип (Prototype) - создание врагов/вещей на основе "заготовок"*

## Выполнение работы.

- 1) Был создан интерфейс CageEntity с виртуальным деструктором. Также в класс Cage добавлен поле entity, в котором содержится информация о содержимом клетки на поле (игрок, враг, предмет). Все содержимые объединены общим интерфейсом - CageEntity.
- 2) Далее были разработаны классы MovableEntity и Iteraction\_elements, которые наследуются от CageEntity. MovableEntity является абстрактным классом, содержащий только чистые виртуальные функции: *void fight(MovableEntity\* character)* для описания логики сражения персонажей, *bool isAlive()* для проверки жив ли персонаж, функция *void plusHealth(int number)* для прибавления (или для снятия) здоровья персонажам. Iteraction\_elements также является абстрактным классом, содержащий только чистые виртуальные функции: *void setState(int number)* для установки состояния (например, мы можем поменять параметр наносимого урона у оружия), *int getState()* для взятия состояния у предметов. От него наследуется 3 класса: Armor (предмет “броня”), Health (предмет “здоровье”), Weapon (предмет “оружие”). Помимо перегруженных виртуальных функций от Iteraction\_elementst классы содержат конструктор и соответствующие поля.
- 3) От MovableEntity наследуется 2 класса: MainCharacter и Enemies. Enemies это абстрактный класс, который содержит только чистые виртуальные функции: *void setHealth(int number)* который устанавливает здоровье, *void setDamage(int number)* который устанавливает урон, *int getHealth()* и *int getDamage()*, которые позволяют получать значения здоровья и урона, соответственно; *void fight(MovableEntity\* character)* (override) который описывает логику

сражения у врагов, *bool isAlive()* (override), проверяет жив ли враг, *void plusHealth(int number)* (override) для прибавления (или для снятия) здоровья врагам. От *Enemies* наследуется ещё 3 класса врагов: *Zombie*, *Ghost*, *Monster*, которые обладают разными наборами характеристик. В этих классах реализованы перегруженные методы класса *Enemies* и конструкторы для каждого класса. Класс *MainCharacter* это класс игрока, который обладает собственными характеристиками (у него есть поля *health*, *armor* и *power*), которые могут изменяться в ходе игры с помощью функций *void setArmor(int number)*, *void setHealth(int number)*, *void setPower(int number)* и получать характеристики с помощью функций *int getArmor() const*, *int getHealth() const*, *int getPower() const*. Логика подбора вещей реализована в функции *void TakeItem(CageEntity\* item)*, а логика сражения с помощью перегруженной функции *void fight(MovableEntity\* character)* (final) и *void plusHealth(int number)* (final), проверка жив ли персонаж реализована в функции *bool isAlive()* (final). Также функция *TakeItem* принимает указатель на базовый класс, а в теле проверяет при помощи *typeid* к какому классу они принадлежат (*Weapon*, *Armor*, *Health*).

- 4) В класс *Field* появилось поле указатель на героя, массив указателей на клетки с врагами, а также счётчик врагов. Указатели были добавлены для того, чтобы не обходить весь массив клеток каждый раз, когда мы хотим обратиться к герою или врагу.
- 5) Также были добавлены новые функции в классе *Field*, которые вызываются в конструкторе, а именно: *void create\_hero()*, которая позволяет создать игрока на поле, *void create\_enemies()*, которая позволяет создать врагов на поле (количество врагов определяется по

формуле:  $(\text{всего\_клеток} / 20) + 1$ ), *void create\_items() const*, которая позволяет создать предметы на поле (количество предметов:  $(\text{всего\_клеток} / 15) + 1$ ), *void moveEntity(const Cage& cell, char dir, int numberEnemy)*, которая позволяет менять содержимое клеток на поле, т.е. передвигать врагов, *bool HeroWin() const* позволяет определить дошел ли игрок до точки выхода. Места появления врагов и предметов определяются случайно. Герой всегда появляется на точке входа.

- 6) Функция *moveEntity* принимает в качестве аргументов клетку, содержимое которой надо переместить, направление движения, а также номер врага, которого мы хотим переместить, при необходимости. В теле функции сначала определяется возможно ли вообще наступить на клетку, на которую пытается совершить движение объект, а если возможно, то запоминаются её координаты. Далее, при помощи *typeid()* функция определяет кто на ней находится, *MainCharacter* или один из *Enemy*, и идёт по одной из веток. Перемещении на пустую клетку происходит по следующему алгоритму: мы кладём в её поле *entity*, содержимое нашей клетки, после чего делаем *entity* нашей клетки *nullptr*, также меняем указатель на клетку у полей *hero* или *enemies[i]*.

Если на клетке, куда хочет походить герой, находится враг, то вызывается метод *fight*, происходит это при помощи *dynamic\_cast*. *fight* вызывается у обоих участников боя, он отнимает у них соответствующие значения характеристик. Если враг убит, память выделенная под него очищается, а сам он исчезает с поля.

Если же. на клетке, куда хочет наступить герой, лежит предмет, то у героя вызывается метод *TakeItem*, характеристики героя меняются, а

предмет исчезает с поля. Если это делает враг, то предмет просто исчезает с поля.

Враги не могут драться с друг другом.

- 7) Для клетки произошли изменения и в классе `CellView`, где теперь для вывода, сначала надо определить тип содержимого, это делается при помощи `typeid`.
- 8) Сама игра запускается при помощи метода нового класса `Game` - *`StartGame()`*. Помимо конструктора и деструктора в нём реализованы два метода: *`void EndGame()`* который проверяет, должна ли закончиться игра, например, когда игрок умер и *`void StartGame()`*, в котором реализована вся логика игры: создаётся поле, вводятся его характеристики, запускается цикл игры. Тут же реализованы паттерны передвижения врагов, они ходят циклично, и направление их движения определяется с помощью остатка по модулю итератора.

Разработанный программный код см. в приложении А.

Диаграмму с зависимостями классов см. в приложении Б.

## Тестирование.

Результаты тестирования представлены в табл. 1.

Таблица 1 – Результаты тестирования

п/п	Тест	Результат	Комментарии
1.	Есть возможность запустить игру, введя характеристики поля, и они будут соответствовать тому, что выводят.	ОК	Поле сгенерировано, если числа слишком большие, то они автоматически уменьшаются до верхней границы.
2.	Есть возможность перемещаться по полю, также перемещаются и враги.	ОК	Враги не могут наступать друг на друга или нападать на героя, а также ходят по одному паттерну.
3.	Есть возможность поднять предмет, при этом характеристики героя действительно изменяться.	ОК	Поднятие предметов происходит успешно
4.	Есть возможность	ОК	-



	поддаться с врагом, при этом герой получает урон, а враг рано или поздно умирает.		
5.	Есть возможность пройти игру, если наступить на клетку финиша.	ОК	Игра завершается корректно.
6.	Герой может умереть и игра закончится.	ОК	При битве с врагом герой потерял слишком много здоровья, игра закончилась и игра закончилась корректно.

### **Выводы.**

В ходе работы были изучены способы реализации наследования, переопределения функций, а также способы приведения объектов к определенному типу.

Разработана программа по типу мини игры, получающая от пользователя параметры поля и генерирующая его с определённым количеством непроходимых клеток и стартом/финишем. Также на ней

появляются враги и предметы, с которыми пользователь может взаимодействовать, перемещая своего персонажа по полю.

## Приложение А.

### Исходный код программы.

Название файла: main.cpp

```
#include "Tools/Game.h"

int main(){
    Game modelGame(10, 10);
    modelGame.StartGame();

    //    Cage model;
    //    model.setTypeCage();
    //    model.setTypeObj();
    //    int a = model.getTypeCage();
    //    int b = model.getTypeObj();
    //    std::cout << a << " " << b << std::endl;
    //    CageView view(model);
    //    view.printCage();
    //    view.printObj();

    //    int w = 7;
    //    int h = 7;
    //    Field model(w, h);
    //    FieldView modelView(model);

    //    modelView.printField(table);
    //    std::cout << "\n";
    //    modelView.printFieldObj(table);
    //    std::cout << "\n";

    //    Weapon modelWeapon(10);
    //    modelWeapon.setState(20);    // установить урон у оружие

- 20

    //    table[1][1].entity = &modelWeapon;
    //    table[1][1].setTypeObj(TypeObj::INTERACTION_ELEMENTS);
    //    table[1][1].setTypeCage(TypeCage::PASSABLE);
    //                                auto&                test                =
dynamic_cast<Weapon>(&(*table[1][1].entity));
    //    std::cout << typeid(test).name() << " vs " <<
typeid(Weapon).name() << "\n";
```



```

//                break;
//            case 'A':
//                model.moveEntity(*model.hero, 'A');
//                break;
//            case 'D':
//                model.moveEntity(*model.hero, 'D');
//                break;
//            case 'S':
//                model.moveEntity(*model.hero, 'S');
//                break;
//            case 'I':
//
//                player =
dynamic_cast<MainCharacter*>(*model.hero->entity);
//                std::cout << "Info hero: \nPower: " <<
player.getPower() << std::endl;
//                std::cout << "Health: " <<
player.getHealth() << std::endl;
//                std::cout << "Armor: " << player.getArmor()
<< std::endl;
//                break;
//            default:
//                break;
//        }
//        modelView.printField();
//        std::cout << "\n";
//        modelView.printFieldObj();
//        std::cout << "\n";
//    }

//    std::cout << "\n";
//    Field model2(3, 3);
//    model2 = std::move(model);
//    FieldView model2View(model2);
//    Cage** table2 = model2.getField();
//    model2View.printField(table2);
//    std::cout << "\n";
//
//
//    Field copyfield = model;
//    copyfield = model2;
//    FieldView copyfieldView(copyfield);
//    Cage** table3 = copyfield.getField();
//    copyfieldView.printField(table3);
//    std::cout << "\n";

    return 0;
}

```

Название файла: Tools/Game.h

#pragma once

```

#include "Field.h"
#include "View/FieldView.h"

class Game{
private:
    int height;
    int width;
    bool QUIT; // флаг для выхода из игры
protected:
    void EndGame();
public:
    Game();
    Game(int h, int w);
    ~Game();
    void StartGame();
    // void MoveAllEnemies();
};

```

**Название файла: Tools/Game.cpp**

```

#include "Game.h"

Game::Game() {
    this->height = 10;
    this->width = 10;
    this->QUIT = false;
};

Game::Game(int h, int w) {
    this->height = h;
    this->width = w;
    this->QUIT = false;
}

Game::~~Game() = default;

void Game::StartGame() {

    // auto* modelZombie = new Zombie; // ?
    // field.board[4][4].setEntity(modelZombie);
    // auto * modelGhost = new Ghost;
    // field.board[5][5].setEntity(modelGhost);
    // auto* modelMonster = new Monster;
    // field.board[2][3].setEntity(modelMonster);
    // fieldView.printField();
    // std::cout << "\n";
    Field field(this->width, this->height);
    // if (field.board[1][1].entity == nullptr){
    //     Weapon modelWeapon(20);
    //     field.board[1][1].setEntity(&modelWeapon);
    // }
}

```

```

//      if (field.board[2][2].entity == nullptr){
//          Health modelHealth(20);
//          field.board[2][2].setEntity(&modelHealth);
//      }
//      if (field.board[3][3].entity == nullptr){
//          Armor shield(10);
//          field.board[3][3].setEntity(&shield);
//      }

    FieldView fieldView(field);
    fieldView.printField();
    std::cout << "\n";
    fieldView.printFieldObj();
    std::cout << "\n";

    MainCharacter player =
dynamic_cast<MainCharacter*>(*(field.hero->getEntity()));
    int iter = 0;
    while (!this->QUIT){

        player =
dynamic_cast<MainCharacter*>(*(field.hero->getEntity()));
        char command;
        std::cout << "Enter command: ";
        std::cin >> command;
        std::cout << std::endl;
//      std::cout << "Y: " << player->getY() << std::endl;
//      std::cout << "X: " << player->getX() << std::endl;

        switch (command) {
            case 'Q':
                EndGame();
                break;
            case 'W':
                field.moveEntity(*field.hero, 'W', 0);
                if
(!dynamic_cast<MainCharacter*>(*(field.hero->getEntity()))>.isAlive
() or field.HeroWin())
                    EndGame();
                break;
            case 'A':
                field.moveEntity(*field.hero, 'A', 0);
                if
(!dynamic_cast<MainCharacter*>(*(field.hero->getEntity()))>.isAlive
() or field.HeroWin())
                    EndGame();
                break;
            case 'D':
                field.moveEntity(*field.hero, 'D', 0);
                if
(!dynamic_cast<MainCharacter*>(*(field.hero->getEntity()))>.isAlive
() or field.HeroWin())
                    EndGame();
                break;
        }
    }

```

```

        break;
    case 'S':
        field.moveEntity(*field.hero, 'S', 0);
        if
(!dynamic_cast<MainCharacter>>(*field.hero->getEntity())).isAlive
() or field.HeroWin())
            EndGame();
        break;
    case 'I':
        // player =
dynamic_cast<MainCharacter>(*field.hero->entity());
        std::cout << "Info hero: \nPower: " <<
player.getPower() << std::endl;
        std::cout << "Health: " << player.getHealth()
<< std::endl;
        std::cout << "Armor: " << player.getArmor()
<< std::endl;
        if
(!dynamic_cast<MainCharacter>(*field.hero->getEntity())).isAlive
()) // умерает не сразу
            EndGame();
        break;
    default:
        break;
}

// field.MoveAllEnemies(); // вызывает ошибку
char ZDir, GDir, MDir;
if (iter % 12 == 0) {
    ZDir = 'W';
    GDir = 'A';
    MDir = 'D';
}
if (iter % 12 == 1) {
    ZDir = 'A';
    GDir = 'A';
    MDir = 'D';
}
if (iter % 12 == 2) {
    ZDir = 'S';
    GDir = 'W';
    MDir = 'D';
}
if (iter % 12 == 3) {
    ZDir = 'D';
    GDir = 'W';
    MDir = 'W';
}
if (iter % 12 == 4) {
    ZDir = 'W';
    GDir = 'D';
}

```

```

        MDir = 'W';
    }
    if (iter % 12 == 5) {
        ZDir = 'A';
        GDir = 'D';
        MDir = 'W';
    }
    if (iter % 12 == 6) {
        ZDir = 'S';
        GDir = 'S';
        MDir = 'A';
    }
    if (iter % 12 == 7) {
        ZDir = 'D';
        GDir = 'S';
        MDir = 'A';
    }
    if (iter % 12 == 8) {
        ZDir = 'W';
        GDir = 'A';
        MDir = 'A';
    }
    if (iter % 12 == 9) {
        ZDir = 'A';
        GDir = 'A';
        MDir = 'S';
    }
    if (iter % 12 == 10) {
        ZDir = 'S';
        GDir = 'S';
        MDir = 'S';
    }
    if (iter % 12 == 11) {
        ZDir = 'D';
        GDir = 'S';
        MDir = 'S';
    }
    for (int i = 0; i < field.getCountEnemies(); i++) {
        if (field.arr_enemies[i]->getEntity() != nullptr)
        {
                                                    if
        (typeid(*field.arr_enemies[i]->entity).name() !=
        typeid(MainCharacter).name()) {
            if (i % 3 == 0) {

field.moveEntity(*field.arr_enemies[i], ZDir, i);
            }
            if (i % 3 == 1) {

field.moveEntity(*field.arr_enemies[i], GDir, i);
            }

```



```

        if (i % 3 == 2) {
field.moveEntity(*field.arr_enemies[i], MDir, i);
        }
    }
    }
    iter++;
    fieldView.printField();
    std::cout << "\n";
    fieldView.printFieldObj();
    std::cout << "\n";
}
}

```

```

void Game::EndGame() {
    this->QUIT = true;
}

```

**Название файла: Tools/Field.h**

```

#pragma once
#include "Cage.h"
#include "View/CageView.h"
#include "../Entity/MovableCharacters/MainCharacter.h"

class Field{
private:    // Добавить Cell* hero, чтобы перемещать героя(но
там придется кучу добавлений делать)
    int width;
    int height;
    Cage start;    // Cage&?
    Cage finish;    // Cage&?
    Cage* arr_enemies[32]{};
    int countEnemies;
    Cage* hero;    // временно в public!!!
    Cage** board;    // временно в public!!!
public:
    //    Cage** board;    // временно в public!!!
    //    Cage* hero;    // временно в public!!!
        Field();    //    Field():width(0),    height(0),
board(nullptr){};
        Field(int w, int h);
        Field(const Field& obj);    // конструктор копирования
            Field& operator=(const Field& obj);    // оператор
присваивания с копированием
        Field(Field&& obj) noexcept;    // конструктор перемещения
            Field& operator=( Field&& obj) noexcept;    // оператор
перемещения
        ~Field();
        friend class FieldView;
        friend class Game;

```

```

        //      Cage **array_generator(unsigned int dim1, unsigned int
dim2);
        //      Cage **fill_array(Cage** arr);
        //      void array_destroyer(Cage **arr, unsigned int dim2);
        //      [[nodiscard]] Cage **getField() const;
        //      [[nodiscard]] bool isCorrectDistStartFinish(Cage start,
Cage finish) const;
        //      [[nodiscard]] Cage generateBorderPoint() const;
        void entry_exit_create();
        void create_hero();
        void moveEntity(const Cage& cell, char dir, int
numberEnemy);
        void create_enemies();
        //      void MoveAllEnemies();
        //      [[nodiscard]] int getCountEnemies() const;
        void create_items() const;
        //      [[nodiscard]] bool HeroWin() const;
};

```

**Название файла: Tools/Field.cpp**

```

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

Field::Field() {
    this->width = 0;
    this->height = 0;
    this->board = nullptr;
    this->hero = nullptr;
    this->countEnemies = 0;
}

Field::Field(int w, int h){
    this->hero = nullptr;
    this->countEnemies = 0;
    if (w < 2 and h < 2){
        w = 2;
        h = 2;
    }
    this->width = w;
    this->height = h;
    //      board = array_generator(w, h);
    //      board = fill_array(board);
    //      board = entry_exit_creat(board);
    board = new Cage * [height];
    for (int i = 0; i < height; i++) {
        board[i] = new Cage[width];
    }
    for (int i = 0; i < height; ++i) {
        for (int j = 0; j < width; j++) {

```

```

        board[i][j] = Cage(i, j); // Cage(i, j)
        board[i][j].setTypeCage(TypeCage::EMPTY);
        board[i][j].setTypeObj(TypeObj::NOTHING);
    }
}
entry_exit_create();
create_hero(); // инициализация hero нах-ся в этой ф-ии
create_enemies();
create_items();
}

Field::~Field() {
    //array_destroyer(board, height);
    for (int i = 0; i < height; i++) {
        for (int j = 0; j < width; ++j) {
            delete board[i][j].entity; //
board[j][i].entity;
        }
        delete [] board[i];
    }
    delete [] board;
    // удаление врагов
    // for (auto & arr_enemy : arr_enemies) {
    //     delete [] arr_enemy;
    // }
}

Field::Field(const Field& obj):width(obj.width),
height(obj.height){
    // исправить костыли! (исправлено)
    //TypeCage typesCage[5] = {TypeCage::START,
TypeCage::END, TypeCage::PASSABLE, TypeCage::IMPASSABLE,
TypeCage::EMPTY};
    //TypeObj typesObj[4] = {TypeObj::NOTHING,
TypeObj::PLAYER, TypeObj::ENEMY, TypeObj::INTERACTION_ELEMENTS};
    hero = obj.hero;
    start = obj.start;
    finish = obj.finish;
    board = new Cage * [height];
    for (int i = 0; i < height; ++i) {
        board[i] = new Cage[width];
    }
    for (int i = 0; i < height; ++i) {
        for (int j = 0; j < width; ++j) {
            board[i][j] = Cage(i, j);
            // prev:
board[i][j].setTypeCage(typesCage[obj.board[i][j].getTypeCage()]);
            // prev:
board[i][j].setTypeObj(typesObj[obj.board[i][j].getTypeObj()]);
board[i][j].setTypeCage(obj.board[i][j].getTypeCage());

```

```

board[i][j].setTypeObj(obj.board[i][j].getTypeObj());
    }
}

Field& Field::operator=(const Field& obj){
    // исправить костыли (исправлено)
    //TypeCage    typesCage[5]    =    {TypeCage::START,
TypeCage::END,        TypeCage::PASSABLE,        TypeCage::IMPASSABLE,
TypeCage::EMPTY};
    //TypeObj    typesObj[4]    =    {TypeObj::NOTHING,
TypeObj::PLAYER, TypeObj::ENEMY, TypeObj::INTERACTION_ELEMENTS};

    if (this != &obj){
        for (int i = 0; i < height; ++i) {
            delete[] board[i];
        }
        delete[] board;
        width = obj.width;
        height = obj.height;
        start = obj.start;
        finish = obj.finish;
        hero = obj.hero;
        board = new Cage * [height];
        for (int i = 0; i < height; ++i) {
            board[i] = new Cage[width];
        }
        for (int i = 0; i < height; ++i) {
            for (int j = 0; j < width; ++j) {
                board[i][j] = Cage(j, i); // Cage(j, i)
            }
        }

board[i][j].setTypeCage(obj.board[i][j].getTypeCage());

board[i][j].setTypeObj(obj.board[i][j].getTypeObj());
    //
board[i][j].setTypeCage(typesCage[obj.board[i][j].getTypeCage()]);
    //
board[i][j].setTypeObj(typesObj[obj.board[i][j].getTypeObj()]);
    }
}

return *this;
}

Field::Field(Field&& obj) noexcept{
    std::swap(width, obj.width);
    std::swap(height, obj.height);
    std::swap(start, obj.start);
    std::swap(finish, obj.finish);
}

```

```

        std::swap(board, obj.board);
        std::swap(hero, obj.hero);
    }

Field& Field::operator=( Field&& obj) noexcept {
    if (this != &obj){
        std::swap(width, obj.width);
        std::swap(height, obj.height);
        std::swap(start, obj.start);
        std::swap(finish, obj.finish);
        std::swap(board, obj.board);
        std::swap(hero, obj.hero);
    }
    return *this;
}

//Cage** Field::getField() const{
//    return this->board;
//}

bool Field::isCorrectDistStartFinish(Cage start, Cage finish)
const{
    int distStartFinish = std::max(((width + height) / 2),
2);
    return abs(start.getX() - finish.getX()) +
        abs(start.getY() - finish.getY()) >=
distStartFinish;
}

Cage Field::generateBorderPoint() const {
    switch (rand() % 4) {
        case 0:
            return {0,
                rand() % height};
        case 1:
            return {this->width - 1,
                rand() % height};
        case 2:
            return {rand() % width,
                0};
        case 3:
            return {rand() % width,
                this->height - 1};
    }
    return {0, 0};
}

void Field::entry_exit_create() {
    srand(time(nullptr));
    while (!isCorrectDistStartFinish(this->start,
this->finish)) {

```

```

        this->start = generateBorderPoint();
        this->finish = generateBorderPoint();
    }

    this->board[this->start.getX()][this->start.getY()].setTypeCage(TypeCage::START);

    this->board[this->finish.getX()][this->finish.getY()].setTypeCage(TypeCage::END);
}

    void Field::create_hero() { // стоит ли менять деконструктор?
        auto* Hero = new MainCharacter;
        // лишние this?

    this->board[this->start.getX()][this->start.getY()].entity = Hero;
    // prev: ... .setEntity(Hero), но появляется баг
                                this->hero =
    &board[this->start.getX()][this->start.getY()]; // который
    в таблице Объектов ставит start - pass
    }

    void Field::moveEntity(const Cage& cell, char dir, int
numberEnemy = 0) {
        int x, y;
        bool flag = false;
        switch (dir) {
            case 'W':
                if (cell.getX() != 0){
                    x = cell.getX() - 1;
                    y = cell.getY();
                    flag = true;
                }
                break;
            case 'S':
                if (cell.getX() != this->width - 1){
                    x = cell.getX() + 1;
                    y = cell.getY();
                    flag = true;
                }
                break;
            case 'A':
                if (cell.getY() != 0){
                    x = cell.getX();
                    y = cell.getY() - 1;
                    flag = true;
                }
                break;
            case 'D':
                if (cell.getY() != this->height - 1){

```

```

        x = cell.getX();
        y = cell.getY() + 1;
        flag = true;
    }
    break;
default:
    x = cell.getX();
    y = cell.getY();
    flag = false;
    break;
}
if (flag){
    if (typeid(*cell.entity).name() ==
typeid(MainCharacter).name()){
        if (this->board[x][y].getEntity() != nullptr){
            // нужна проверка на то, что это не враг
            auto tmp =
typeid(*board[x][y].entity).name();
            // std::cout << tmp << "\n";
            auto* enemy =
dynamic_cast<Enemies*>(this->board[x][y].getEntity()); // auto? ,
возможно стоит занести в else
            auto* player =
dynamic_cast<MainCharacter*>(cell.getEntity());
            // поменять условие
            if (tmp != typeid(Zombie).name() and tmp !=
typeid(Ghost).name() and tmp != typeid(Monster).name()){
                if (tmp == typeid(Weapon).name() or tmp
== typeid(Health).name() or tmp == typeid(Armor).name()){

                    (dynamic_cast<MainCharacter*>(*hero->entity).TakeItem(board[x][y].
getEntity())); // ? // pr &board[x][y]
                }
                // подумать как избежать дублирование
кода
                this->board[x][y].entity =
cell.getEntity();

                this->board[x][y].setTypeObj(TypeObj::PLAYER);

                this->board[cell.getX()][cell.getY()].entity = nullptr;

                this->board[cell.getX()][cell.getY()].setTypeObj(TypeObj::NOTHING)
;

                hero = &board[x][y];
            } else{
                player->fight(enemy); //
dynamic_cast<MainCharacter*>(cell.entity)->fight(enemy)
                enemy->fight(player); //
dynamic_cast<Enemies*>(this->board[x][y].entity)->fight(player);;
                if (!enemy->isAlive()){

```

```

this->board[x][y].entity =
cell.getEntity();

this->board[x][y].setTypeObj (TypeObj::PLAYER);

this->board[cell.getX()][cell.getY()].entity = nullptr;

this->board[cell.getX()][cell.getY()].setTypeObj (TypeObj::NOTHING)
;
        hero = &board[x][y];
    }
}

} else{
    this->board[x][y].entity = cell.getEntity();

this->board[x][y].setTypeObj (TypeObj::PLAYER);
    this->board[cell.getX()][cell.getY()].entity
= nullptr;

this->board[cell.getX()][cell.getY()].setTypeObj (TypeObj::NOTHING)
;
        hero = &board[x][y];
    }
} else{ // прописать, чтобы враги тоже наносили
урон, когда идут на игрока
    if (board[x][y].entity != nullptr){
        auto tmp =
typeid(*board[x][y].entity).name();
        if (tmp != typeid(MainCharacter).name() and
tmp != typeid(Zombie).name() and tmp != typeid(Ghost).name()
and tmp != typeid(Monster).name()){
            board[x][y].entity = cell.getEntity();
            board[cell.getX()][cell.getY()].entity =
nullptr;
            arr_enemies[numberEnemy] = &board[x][y];
        }
    } else{
        board[x][y].entity = cell.getEntity();
        board[cell.getX()][cell.getY()].entity =
nullptr;
        arr_enemies[numberEnemy] = &board[x][y];
    }
}
// }
}

}

void Field::create_enemies() { // подумать как сделать лучше

```





```

        modelWeapon->setState(40);
        this->board[x][y].setEntity(modelWeapon);
        created = true;
    } else if ((i % 3) == 1){
        auto* modelHealth = new Health;
        modelHealth->setState(100);
        this->board[x][y].setEntity(modelHealth);
        created = true;
    } else if ((i % 3) == 2){
        auto* modelArmor = new Armor;
        modelArmor->setState(30);
        this->board[x][y].setEntity(modelArmor);
        created = true;
    }
    }
}

bool Field::HeroWin() const {
    if (this->board[hero->getX()][hero->getY()].getTypeCage()
== TypeCage::END){
        return true;
    } else{
        return false;
    }
}

```

Название файла: Tools/Cage.h

```

#pragma once
#include <iostream>
#include "../Entity/CageEntity.h"
#include "../Entity/Items/Weapon.h"
#include "../Entity/Items/Armor.h"
#include "../Entity/Items/Health.h"
#include "../Entity/MovableCharacters/Enemies/Zombie.h"
#include "../Entity/MovableCharacters/Enemies/Ghost.h"
#include "../Entity/MovableCharacters/Enemies/Monster.h"
#include "../Entity/MovableCharacters/MainCharacter.h"

enum TypeObj{    // исправил на верхний регистр, т.к. это
константы
    NOTHING,
    PLAYER,
    ENEMY,
    INTERACTION_ELEMENTS    // нигде не используется???
    подумать как использовать!
};

enum TypeCage{
    START,

```

```

        END,
        PASSABLE,
        IMPASSABLE,
        EMPTY
};

enum TypeItems{    // добавлено (типы предметов)
    WEAPON,
    HEALTH,
    ARMOR
};

enum TypeEnemy{
    Zombies,
    Ghosts,
    Monsters
};

class Cage{
private:
    int typeCage;
    int typeObj;
    int x, y;
public:
    CageEntity* entity;
    friend class CageView;
    Cage();                // prev: Cage(): x(0), y(0) {}
    Cage(int x, int y);    // prev: Cage(int x, int y): x(x),
y(y) {}

    ~Cage();
    void setTypeCage(TypeCage typeCage = TypeCage::EMPTY);
    void setTypeObj(TypeObj typeObj = TypeObj::NOTHING);
    void setEntity(CageEntity* value);
    CageEntity* getEntity() const;
    [[nodiscard]] TypeCage getTypeCage() const;
    [[nodiscard]] TypeObj getTypeObj() const;
    [[nodiscard]] int getX() const;
    [[nodiscard]] int getY() const;
    // перегрузить оператор +
    // перегрузить оператор -
};

```

**Название файла: Tools/Cage.cpp**

```

#include "Cage.h"
#include <iostream>

Cage::Cage() {
    this->x = 0;
    this->y = 0;
    this->typeObj = TypeObj::NOTHING;
}

```

```

        this->typeCage = TypeCage::EMPTY;
        this->entity = nullptr;
    }

    Cage::Cage(int x, int y) { // 3й аргумент - Cage* entity =
    nullptr?; Cage::Cage(int x, int y)
        this->x = x; // !!
        this->y = y; // !!
        this->typeObj = TypeObj::NOTHING;
        this->typeCage = TypeCage::EMPTY;
        this->entity = nullptr;
    }

    Cage::~Cage() = default;

    void Cage::setTypeCage(TypeCage typeCage) {
        this->typeCage = typeCage;
    }

    void Cage::setTypeObj(TypeObj typeObj) {
        this->typeObj = typeObj;
    }

    void Cage::setEntity(CageEntity *value) {
        //this->entity = value;
        auto t = typeid(*value).name();
        if (t == typeid(Weapon).name() or t ==
    typeid(Health).name() or t == typeid(Armor).name()){
            this->typeObj = TypeObj::NOTHING; //
INTERACTION_ELEMENTS
            this->typeCage = TypeCage::PASSABLE;
        } else if (t == typeid(Zombie).name() or t ==
    typeid(Ghost).name() or t == typeid(Monster).name()){
            this->typeObj = TypeObj::NOTHING; // ENEMY!!!
            this->typeCage = TypeCage::PASSABLE;
        }

        this->entity = value;
    }

    TypeCage Cage::getTypeCage() const{ // prev: int
        return static_cast<TypeCage>(this->typeCage); // prev:
    this->typeCage
    }

    TypeObj Cage::getTypeObj() const { // prev: int
        return static_cast<TypeObj>(this->typeObj); // prev:
    this->typeObj
    }

    int Cage::getX() const {

```

```

        return this->x;
    }

    int Cage::getY() const {
        return this->y;
    }

    CageEntity *Cage::getEntity() const {
        return this->entity;
    }

```

### Название файла: Tools/View/CageView.h

```

#pragma once
#include "../Cage.h"

class CageView{
private:
    Cage& cell;
public:
    explicit CageView(Cage& obj);
    void printCage() const;
    void printObj() const;
};

```

### Название файла: Tools/View/CageView.cpp

```

#include "CageView.h"
#include <iostream>
#include <map>
#include <string>
/*
#include "Characters/CageEntity.h"
#include "Characters/Interaction_elements.h"
#include "Characters/Weapon.h"
#include "Characters/MainCharacter.h"
#include "Characters/Armor.h"
#include "Characters/Zombie.h"
*/

CageView::CageView(Cage &obj): cell(obj){}

void CageView::printObj() const {
    // нужно убрать switches
    //      switch (cell.getTypeObj()) {
    //          case NOTHING:
    //              std::cout << "[ ]";
    //              break;
    //          case PLAYER:
    //              std::cout << "[P]";
    //              break;
    //          case ENEMY:

```

```

//          std::cout << "[E]";
//          break;
//      case INTERACTION_ELEMENTS:
//          std::cout << "[I_E]";
//          break;
//      }

std::map<TypeCage, std::string> outCage;
std::map<TypeObj, std::string> out;    // можно ли
использовать контейнер string?
std::map<TypeItems, std::string> out_items;    // можно ли
использовать контейнер string?
std::map<TypeEnemy, std::string> out_enemy;    // можно ли
использовать контейнер string?
out[TypeObj::NOTHING] = "[ ]";
out[TypeObj::PLAYER] = "[P]";
out[TypeObj::ENEMY] = "[E]";
out[TypeObj::INTERACTION_ELEMENTS] = "[I_E]";
out_items[TypeItems::WEAPON] = "[W]";
out_items[TypeItems::ARMOR] = "[A]";
out_items[TypeItems::HEALTH] = "[H]";
out_items[TypeItems::ARMOR] = "[A]";
out_enemy[TypeEnemy::Zombies] = "[Z]";
out_enemy[TypeEnemy::Ghosts] = "[G]";
out_enemy[TypeEnemy::Monsters] = "[M]";
outCage[TypeCage::END] = "[F]";
outCage[TypeCage::START] = "[S]";
if (cell.entity){
    //          auto& test = dynamic_cast<Weapon>(&*cell.entity);
// возможно стоит сначала на Iter.-el., а потом на Weapon
//          if (typeid(test).name() == typeid(Weapon).name()){
//          std::cout << out_items[TypeItems::WEAPON];    //
потом поменять на норм!
//          }

if (typeid(*cell.entity).name() ==
typeid(Weapon).name()){
    std::cout << out_items[TypeItems::WEAPON];
    } else if (typeid(*cell.entity).name() ==
typeid(MainCharacter).name()){
    std::cout << out[TypeObj::PLAYER];
    } else if (typeid(*cell.entity).name() ==
typeid(Health).name()){
    std::cout << out_items[TypeItems::HEALTH];
    } else if (typeid(*cell.entity).name() ==
typeid(Armor).name()){
    std::cout << out_items[TypeItems::ARMOR];
    } else if (typeid(*cell.entity).name() ==
typeid(Zombie).name()){
    std::cout << out_enemy[TypeEnemy::Zombies];
    } else if (typeid(*cell.entity).name() ==
typeid(Ghost).name()){
    std::cout << out_enemy[TypeEnemy::Ghosts];

```

```

        } else if (typeid(*cell.entity).name() ==
typeid(Monster).name()) {
            std::cout << out_enemy[TypeEnemy::Monsters];
        }
    }
    else{
        if (cell.getTypeCage() == TypeCage::END){
            std::cout << outCage[TypeCage::END];
        } else if (cell.getTypeCage() == TypeCage::START){
            std::cout << outCage[TypeCage::START];
        }
        else{
            std::cout << out[cell.getTypeObj()];
        }
    }

    // auto& test = dynamic_cast<Weapon&>(*cell.entity);
    // if (typeid(test).name() == typeid(Weapon).name()){
    //     std::cout << out_items[TypeItems::WEAPON]; //
потом поменять на норм!
    // } else{
    //     std::cout << out[cell.getTypeObj()];
    // }
    // std::cout << out[cell.getTypeObj()];

}

void CageView::printCage() const {
    // нужно убрать switches
    // switch (cell.getTypeCage()) {
    //     case START:
    //         std::cout << "[START]";
    //         break;
    //     case END:
    //         std::cout << "[END]";
    //         break;
    //     case PASSABLE:
    //         std::cout << "[PASS]";
    //         break;
    //     case IMPASSABLE:
    //         std::cout << "[IPASS]";
    //         break;
    //     case EMPTY:
    //         std::cout << "[EMPTY]";
    // }

    std::map<TypeCage, std::string> out; // можно ли
использовать контейнер string?
    out[TypeCage::START] = "[START]";
    out[TypeCage::END] = "[END]";
    out[TypeCage::PASSABLE] = "[PASS]";
    out[TypeCage::EMPTY] = "[EMPTY]";
    std::cout << out[cell.getTypeCage()];

```

```
}
```

Название файла: Tools/View/FieldView.h

```
#pragma once
```

```
#include "../Field.h"
```

```
class FieldView{
```

```
private:
```

```
    Field& grid;
```

```
public:
```

```
    explicit FieldView(Field& obj);
```

```
    void printField() const;
```

```
    void printFieldObj() const;
```

```
};
```

Название файла: Tools/View/FieldView.cpp

```
#include "FieldView.h"
```

```
FieldView::FieldView(Field &obj): grid(obj){
```

```
//    int width = this->grid.width;
```

```
//    int height = this->grid.height;
```

```
//    boardView = new CageView * [height];
```

```
//    for (int i = 0; i < height; i++) {
```

```
//        boardView[i] = new CageView[width];
```

```
//    }
```

```
//    for (int i = 0; i < height; ++i) {
```

```
//        for (int j = 0; j < width; j++) {
```

```
//            boardView[i][j] = &grid.board[i][j];    //
```

```
Cage(i, j)
```

```
//        }
```

```
//    }
```

```
}
```

```
void FieldView::printField() const{
```

```
    for (int i = 0; i < grid.height; i++) {
```

```
        for (int j = 0; j < grid.width; j++) {
```

```
            CageView tmpCage(this->grid.board[i][j]);
```

```
            tmpCage.printCage();
```

```
        }
```

```
        std::cout << "\n";
```

```
    }
```

```
}
```

```
void FieldView::printFieldObj() const {
```

```
    for (int i = 0; i < grid.height; i++) {
```

```
        for (int j = 0; j < grid.width; j++) {
```

```
            CageView tmpCage(this->grid.board[i][j]);
```

```
            tmpCage.printObj();
```

```
        }
```

```
        std::cout << "\n";
```

```
    }
```



```
}
```

**Название файла: Entity/CageEntity.h**

```
#pragma once
```

```
// интерфейс
```

```
class CageEntity{
```

```
public:
```

```
    virtual ~CageEntity() = default;
```

```
};
```

**Название файла: Entity/Items/Interaction\_elements.h**

```
#pragma once
```

```
#include "../CageEntity.h"
```

```
class Interaction_elements: public CageEntity{    // нужны  
деконструкторы к дочерним классам
```

```
public:
```

```
    virtual void setState(int number) = 0;
```

```
    virtual int getState() = 0;
```

```
};
```

**Название файла: Entity/Items/Armor.h**

```
#pragma once
```

```
#include "Interaction_elements.h"
```

```
class Armor: public Interaction_elements{
```

```
private:
```

```
    int armor;
```

```
public:
```

```
    Armor();
```

```
    explicit Armor(int number);
```

```
    void setState(int number) final;
```

```
    int getState() final;
```

```
};
```

**Название файла: Entity/Items/Armor.cpp**

```
#include "Armor.h"
```

```
Armor::Armor() {
```

```
    this->armor = 0;
```

```
}
```

```
Armor::Armor(int number) {
```

```
    this->armor = number;
```

```
}
```

```
void Armor::setState(int number) {
```

```
    this->armor = number;
```

```
}
```

```
int Armor::getState() {
```

```
        return this->armor;
    }
```

### Название файла: Entity/Items/Health.h

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

class Health: public Interaction_elements{
private:
    int health;
public:
    Health();
    explicit Health(int number);
    void setState(int number) final;
    int getState() final;
};
```

### Название файла: Entity/Items/Health.cpp

```
#include "Health.h"

Health::Health() {
    this->health = 50;
}

Health::Health(int number) {
    this->health = number;
}

void Health::setState(int number) {
    this->health = number;
}

int Health::getState() {
    return this->health;
}
```

### Название файла: Entity/Items/Weapon.h

```
#pragma once
#include "Interaction_elements.h"
// Должен прибавлять главному герою +20 урона, +40 урона,
+100 урона
// патронов бесконечное кол-во

class Weapon: public Interaction_elements{
private:
    int damage;
public:
    Weapon();
    explicit Weapon(int number);
```

```

        void setState(int number) final;
        int getState() final;
/*
    bool take() final;
*/
};

```

### Название файла: Entity/Items/Weapon.cpp

```

#include "Weapon.h"

Weapon::Weapon() {
    this->damage = 5;
}

Weapon::Weapon(int number) {
    this->damage = number;
}

void Weapon::setState(int number) {
    this->damage = number;
}

int Weapon::getState() {
    return this->damage;
}

```

### Название файла: Entity/MovableCharacters/MovableEntity.h

```

#pragma once
#include "../CageEntity.h"

class MovableEntity: public CageEntity{
public:
    virtual void fight(MovableEntity* character) = 0;    // бой
    virtual bool isAlive() = 0;
    virtual void plusHealth(int number) = 0;
//    virtual int getState() = 0;
//    virtual void setState(int number) = 0;
};

```

### Название файла: Entity/MovableCharacters/MainCharacters.h

```

#pragma once
#include "MovableEntity.h"
#include "../../Tools/Cage.h"
#include "../Items/Weapon.h"
#include "../Items/Health.h"
#include "../Items/Armor.h"

class MainCharacter: public MovableEntity{
private:
    int health;

```

```

        int armor;
        int power;
public:
    MainCharacter();
    ~MainCharacter() override; // ?
    void TakeItem(CageEntity* item);
    void setArmor(int number);
    [[nodiscard]] int getArmor() const;
    void setHealth(int number);
    [[nodiscard]] int getHealth() const;
    void setPower(int number);
    [[nodiscard]] int getPower() const;
    void fight(MovableEntity* character) final;
    void plusHealth(int number) final;
    bool isAlive() final;

};

```

**Название файла:** Entity/MovableCharacters/MainCharacters.cpp

```

#include "MainCharacter.h"

MainCharacter::MainCharacter() {
    this->health = 100;
    this->armor = 0;
    this->power = 10;
}

MainCharacter::~~MainCharacter() = default; // поменять потом

void MainCharacter::TakeItem(CageEntity* item) { // Cage
*item
    if (item){
        auto t = typeid(*item).name(); //
        typeid(*item).name()
        if (typeid(*item).name() == typeid(Weapon).name()){
            // auto& t =
            dynamic_cast<Weapon&>(*item->entity);
            // std::cout <<
            (dynamic_cast<Weapon&>(*item->entity)).getState() << "\n";
            this->power =
            (dynamic_cast<Weapon&>(*item)).getState();
        } else if (typeid(*item).name() ==
        typeid(Health).name()){
            this->health +=
            (dynamic_cast<Health&>(*item)).getState();
        } else if (typeid(*item).name() ==
        typeid(Armor).name()){

```

```

                                                                    this->armor    +=
(dynamic_cast<Armor*>(*item)).getState();
        }
    }

void MainCharacter::setArmor(int number) {
    this->armor = number;
}

int MainCharacter::getArmor() const {
    return this->armor;
}

void MainCharacter::setHealth(int number) {
    this->health = number;
}

int MainCharacter::getHealth() const {
    return this->health;
}

void MainCharacter::setPower(int number) {
    this->power = number;
}

int MainCharacter::getPower() const {
    return this->power;
}

bool MainCharacter::isAlive() {
    if (this->health > 0){
        return true;
    } else{
        return false;
    }
}

void MainCharacter::fight(MovableEntity *character) {
    character->plusHealth(-(this->getPower()));
}

void MainCharacter::plusHealth(int number) {
    //    this->health += number;
    int broken_shields = 0;
    if (this->armor > 0){
        broken_shields = this->armor + number;
        this->armor += number;
        if (this->armor < 0){
            this->armor = 0;
            this->health += broken_shields;
        }
    }
}

```

```

    }
    } else{
        this->health += number;
    }
}

```

**Название файла:** Entity/MovableCharacters/Enemies/Enemies.h

```

#pragma once
#include "../MovableEntity.h"

class Enemies: public MovableEntity{
public:
    virtual void setHealth(int number) = 0;
    virtual void setDamage(int number) = 0;
    virtual int getHealth() = 0;
    virtual int getDamage() = 0;
    void fight(MovableEntity* character) override = 0;    //
бой
    bool isAlive() override = 0;
    void plusHealth(int number) override = 0;
};

```

**Название файла:** Entity/MovableCharacters/Enemies/Ghost.h

```

#pragma once
#include "Enemies.h"

class Ghost: public Enemies{
private:
    int health;
    int damage;
public:
    Ghost();
    explicit Ghost(int h, int d);
    void setHealth(int number) final;
    void setDamage(int number) final;
    int getHealth() final;
    int getDamage() final;
    bool isAlive() final;
    void plusHealth(int number) final;
    void fight(MovableEntity* character) final;
    // PlusHealth
    // fight (чтобы отнимать здоровье)
};

```

**Название файла:** Entity/MovableCharacters/Enemies/Ghost.cpp

```

#include "Ghost.h"

Ghost::Ghost() {
    this->health = 50;
    this->damage = 10;
}

```

```

Ghost::Ghost(int h, int d) {
    this->health = h;
    this->damage = d;
}

void Ghost::setDamage(int number) {
    this->damage = number;
}

void Ghost::setHealth(int number) {
    this->health = number;
}

int Ghost::getDamage() {
    return this->damage;
}

int Ghost::getHealth() {
    return this->health;
}

bool Ghost::isAlive() {
    if (this->health > 0){
        return true;
    } else{
        return false;
    }
}

void Ghost::plusHealth(int number) {
    this->health += number;
}

void Ghost::fight(MovableEntity *character) {
    character->plusHealth(-(this->getDamage()));
}

```

**Название файла:** Entity/MovableCharacters/Enemies/Monster.h

```
#pragma once
```

```
#include "Enemies.h"
```

```

class Monster: public Enemies{
private:
    int health;
    int damage;
public:
    Monster();
    explicit Monster(int h, int d);
    void setHealth(int number) final;
    void setDamage(int number) final;
    int getHealth() final;
}

```

```

        int getDamage() final;
        bool isAlive() final;
        void plusHealth(int number) final;
        void fight(MovableEntity* character) final;

};

```

**Название файла:** Entity/MovableCharacters/Enemies/Monster.cpp

```
#include "Monster.h"
```

```

Monster::Monster() {
    this->health = 1000;
    this->damage = 50;
}

```

```

Monster::Monster(int h, int d) {
    this->health = h;
    this->damage = d;
}

```

```

void Monster::setDamage(int number) {
    this->damage = number;
}

```

```

void Monster::setHealth(int number) {
    this->health = number;
}

```

```

int Monster::getDamage() {
    return this->damage;
}

```

```

int Monster::getHealth() {
    return this->health;
}

```

```

bool Monster::isAlive() {
    if (this->health > 0){
        return true;
    } else{
        return false;
    }
}

```

```

void Monster::plusHealth(int number) {
    this->health += number;
}

```

```

void Monster::fight(MovableEntity *character) {
    character->plusHealth(-(this->getDamage()));
}

```



Название файла: Entity/MovableCharacters/Enemies/Zombie.h

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

class Zombie: public Enemies{
private:
    int health;
    int damage;
public:
    Zombie();
    explicit Zombie(int h, int d);
    void setHealth(int number) final;
    void setDamage(int number) final;
    int getHealth() final;
    int getDamage() final;
    bool isAlive() final;
    void plusHealth(int number) final;
    void fight(MovableEntity* character) final;
    // PlusHealth
    // fight (чтобы отнимать здоровье)
};
```

Название файла: Entity/MovableCharacters/Enemies/Zombie.cpp

```
#include "Zombie.h"

Zombie::Zombie() {
    this->health = 25;
    this->damage = 5;
}

Zombie::Zombie(int h, int d) {
    this->health = h;
    this->damage = d;
}

void Zombie::setHealth(int number) {
    this->health = number;
}

void Zombie::setDamage(int number) {
    this->damage = number;
}

int Zombie::getHealth() {
    return this->health;
}

int Zombie::getDamage() {
    return this->damage;
}
```

```
bool Zombie::isAlive() {  
    if (this->health > 0){  
        return true;  
    } else{  
        return false;  
    }  
}  
  
void Zombie::plusHealth(int number) {  
    this->health += number;  
}  
  
void Zombie::fight(MovableEntity *character) {  
    character->plusHealth(-(this->getDamage()));  
}
```

## **Приложение Б**

### **UML-диаграмма классов**

