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

ОТЧЕТ

по лабораторной работе №1

по дисциплине «ООП»

Тема: Разработка игровой системы с тактическими боями.

Студент гр. 4384 _____ Стукалкин М. М.

Преподаватель _____ Жангиров Т.Р.

Санкт-Петербург

2025

Цель работы.

Разработать игровую систему, реализующую механику пошаговых тактических боев с управлением персонажем, врагами и специальными сооружениями. Построить архитектуру с учетом принципов ООП и расширяемости.

Задание.

Создать класс игрока, который должен хранить информацию об игроке (его жизни, урон, очки, и т.д. - студент сам определяет необходимые для работы характеристики). Объект класса игрока должен перемещаться по карте. Если у игрока кончаются жизни, то происходит конец игры.

Создать класс врага, который хранит параметры жизней и урона. Объектами класса врага управляет компьютер. При перемещении, если враг пытается перейти на клетку с игроком, то перемещение не происходит, и игроку наносится урон.

Создать класс квадратного/прямоугольного игрового поля, по которому перемещаются игрок и враги. Игровое поле не должно быть меньше 10 на 10 клеток, и не больше 25 на 25 клеток. Размеры поля задаются через конструктор. Рекомендуется для хранения информации об отдельных клетках поля создать отдельный класс.

Реализовать конструкторы перемещения и копирования для поля, а также соответствующие операторы присваивания с копированием и перемещением (должна происходить глубокая копия).

Реализовать непроходимые клетки на поле. При попытке врагов или игрока перейти на такую клетку, перемещение не происходит. Заполнения поля непроходимыми клетками происходит в момент создания поля.

Добавить возможность для игрока переключаться на ближний или дальний бой с изменением значения наносимого урона. Такое переключение требует один ход.

Добавить класс вражеского здания. Такое здание размещается на карте, и раз в несколько ходов создает нового врага возле себя. Количество ходов до создания нового врага задается в конструкторе.

Реализовать замедляющие клетки на поле. Если игрок переходит на такую клетку, то он не может двигаться на следующий ход.

Архитектурные решения и обоснование.

1. Базовый класс Human. Назначение: Абстрактный базовый класс для всех персонажей в игре.

Атрибуты класса: все поля атрибуты protected, чтобы можно было их унаследовать.

1. Поля double health и double damage хранят количество здоровья и урона соответственно.
2. Поле Coords coords хранит координаты объекта на поле. В целом, все объекты, что будут появляться на поле, хранят свои координаты внутри себя.
3. Поле bool stunned необходимо для реализации функционала замедляющей клетки поля.

Методы класса:

1. Использование абстрактного метода attack() обусловлено тем, что у игрока и врагов есть свои особенности атаки (игрок должен получать опыт при победе, при победе врага игра должна завершиться).
2. Геттеры: get_coords, get_health, get_damage нужны для получения значений приватных полей.
3. Сеттеры: set_coords, в процессе игры персонажи перемещаются, поэтому приходится изменять их координаты. set_stunned, если персонаж попал в замедляющую клетку, необходимо поменять поле stunned или наоборот, если персонаж делает первый ход из замедляющей клетки, то нужно убрать «стан» и разрешить перемещаться.

4. bool moving – метод, отвечающий за перемещение персонажа. На вход он принимает строку, которая определяет направление перемещение персонажа, также на вход поступает ссылка на объект игрового поля. Так как класс Human не имеет прямого доступа к полю и его клеткам, данный метод активно взаимодействует с методами класса GameArea. Метод возвращает true, если перемещение удалось и false в обратном случае. Внутри метода moving вызывается метод attack, в случае, если персонаж попал на клетку с другим персонажем.
5. void take_damage – метод, для «принятия» урона персонажем.
6. double damage_calculation – рассчитывает урон. Есть базовый урон, в добавок к этому урон случайно варьируется при помощи надбавки и убавки урона случайным значением.
7. bool stunned_or_not – метод, который возвращает значение поля stunned.

2. Класс Player – наследник класса Human. Назначение: класс игрового персонажа, которым будет управлять пользователь.

Атрибуты класса: все атрибуты приватные, так как они могут характеризовать только класс Player.

1. Унаследованы все поля класса Human.
2. Поле double experience хранит текущий опыт персонажа.
3. Поле double experience_for_new_level обозначает необходимое количество опыта для повышения уровня персонажа.
4. Поле int level хранит текущий уровень персонажа.
5. Поле bool melee является флагом, если поле true, то персонаж сражается в ближнем бою, если false, то персонаж переключен на дальний бой.

Методы класса:

1. Унаследованы все методы класса Human.
2. Геттеры: get_exp() – возвращает текущий опыт персонажа. Так как поле приватное, использован геттер.

3. Сеттеры: up_level, change_range, up_exp – методы, что меняют значения приватных полей. up_level – повышает уровень персонажа, увеличивая его урон и здоровье (поля damage и health) на 5%, так же осуществляется перенос опыта. change_range – изменяет поле melee и уменьшает урон на 20%, если переключение идет на дальнюю атаку, и возвращает 20%, если персонаж переключается обратно на ближний бой. up_exp – повышает количество текущего опыта.
4. bool attack – метод, реализующий атаку, перегружается виртуальный метод родительского класса. Возвращает true, если персонаж победил и начисляет опыт за победу, повышая уровень, в случае достижения необходимого количества опыта. Возвращает false, если враг не был побежден.

3. Класс Enemy – наследник класса Human. Назначение: класс врага, которым будет управлять компьютер.

Атрибуты класса: все атрибуты приватные, так как они могут характеризовать только класс Enemy.

1. Унаследованы все поля класса Human.
2. Поле double coast_exp – отвечает за то, сколько за победу над этим врагом игровому персонажу будет начислено опыта. coast_exp = heatlh / 2.

Методы класса:

1. Унаследованы все методы класса Human.
2. bool attack – метод, реализующий атаку, перегружается виртуальный метод родительского класса. В случае победы над игровым персонажем выбрасывает кастомная ошибка PlaeyrDyeException, которая означает конец игры. В случае, если победить не удалось, метод возвращает false.
3. double get_coast_exp – геттер, который возвращает значения поля coast_exp.

4. Класс Cell – класс клетки поля. Назначение: клетка поля, что хранит в себе информацию о типе клетки и указатель на игровой объект, если он там находится.

Атрибуты класса: все атрибуты приватные, так же определен enum CellType, в котором хранятся возможные типы клеток.

1. Поле CellType type – хранит в себе тип клетки.
2. Поле pair<Human*, Building*> object – это поле хранит в себе указатель на объект, который находится внутри клетки. Если клетка пустая, то оба указателя nullptr. Хранятся указатели на Родительские классы объектов, чтобы было удобней заполнять поле.
3. Поле bool is_passable – флаг, который true, если клетка проходимая и false, если через клетку нельзя пройти, то есть она BLOCKED.

Методы класса:

1. Геттеры: get_type, get_human, get_building, get_type_string. get_type – возвращает тип клетки. get_human – возвращает указатель на объект человека, что хранится в поле object. get_building – возвращает указатель на объект строения, что находится в поле object. get_type_string – поле, что превращает объект enum CellType в строчку, для удобной работы.
2. Сеттеры: set_human, set_building. Устанавливают указатель на объект, что пришел в клетку. set_type – меняет тип клетки.
3. string is_empty – возвращает строку “Empty”, если клетка пустая, “Human”, если внутри клетки объект класса Human, “Building”, если внутри объект класса Building.
4. void clear – очищает указатели в поле object.
5. bool player_or_enemy – возвращает true, если внутри поля object находится объект типа Player и false, если Enemy. Необходим, так как поле object хранит указатель типа Human.

5. Класс GameArea – класс игрового поля. Назначение: игровое поле, которое хранит в себе клетки и управляет ходом игры.

Атрибуты класса: все атрибуты приватные.

1. int height – хранит в себе высоту поля.
2. Int width – хранит в себе ширину поля.
3. vector<vector<Cell>> area – двумерный вектор, который содержит в себе клетки.

Методы класса:

1. Сеттеры: void set_human_in_cell и void set_building_in_cell — методы для размещения объектов на поле в указанных координатах. Являются основой для наполнения поля объектами.
2. void fill_area — ключевой метод для заполнения поля в начале. Он выполняет несколько задач: Случайным образом генерирует непроходимые (BLOCKED) и замедляющие (SLOW) клетки на карте. Размещает на поле переданные объекты: игрока, врагов и сооружения, используя методы set_human_in_cell и set_building_in_cell.
3. void print_area() — необходим для визуализации игрового поля в консоли. Он проходит по всем клеткам и выводит их символьное представление, что позволяет игроку видеть текущее состояние игры.
4. void move_human — реализует логику перемещения персонажа. Метод обновляет состояние клеток: очищает исходную клетку и устанавливает персонажа в следующую. Также здесь проверяется и применяется эффект замедляющей клетки (SLOW).
5. bool player_or_enemy — метод, который определяет, находится ли в указанной клетке игрок или враг. Это необходимо для корректного разрешения взаимодействий (например, атаки).
6. bool attack — центральный метод для обработки атаки. Он: Извлекает цель из клетки. Наносит урон цели. Если здоровье цели опускается до нуля, обрабатывает победу: начисляет опыт игроку (если он атаковал) и перемещает атакующего на клетку побежденного врага. Возвращает true, если цель была побеждена.

7. string can_move_to — метод проверки возможности перемещения в указанные координаты. Возвращает строку ("Empty", "Human", "Building", "Edge"), который используется для принятия решения о дальнейших действиях (перемещение, атака, остановка).
8. CellType get_cell_type и double get_coast_exp_enemy — геттеры для получения информации о конкретной клетке. Второй метод использует приведение типа static_cast для доступа к специфичным для врага данным.
9. bool is_valid_coords(Coords& coord) — служебный метод для проверки, находятся ли координаты в пределах границ игрового поля. Широко используется другими методами класса для предотвращения ошибок выхода за границы массива.

6. Класс Building – класс здания. Назначение: базовый класс для всех строений на игровом поле.

Атрибуты класса: все поля protected для того, чтобы была возможность унаследовать.

1. Поле double health – хранит в себе здоровье здания.
2. Поле Coords coords – хранит координаты сооружения на игровом поле.

Методы класса:

1. Геттер Coords get_coords – возвращает текущие координаты сооружения. Необходим для взаимодействия с классом GameArea при размещении и проверке объектов на поле.

7. Класс EnemyBuilding – класс вражеского здания. Назначение: конкретный класс сооружения, ответственного за спавн врагов на игровом поле. Наследуется от базового класса Building и добавляет функционал периодического создания противников.

Атрибуты класса: Все поля приватные, так как управление процессом спавна должно быть полностью инкапсулировано внутри класса.

1. Поле int count_step_spawn определяет период спавна - количество ходов, через которое будет создаваться новый враг.

2. Поле int step служит счетчиком ходов, отслеживающим текущий прогресс до следующего спавна.

Методы класса:

1. void plus_step – основной метод, вызываемый каждый ход для обновления состояния сооружения. Увеличивает счетчик шагов и при достижении заданного периода вызывает метод spawn_enemy(), после чего сбрасывает счетчик.
2. Coords find_nearest_free_cell – вспомогательный метод для поиска подходящей клетки для спавна врага. Проверяет соседние клетки по 8 направлениям и возвращает первую свободную и проходимую клетку. Если свободных клеток нет, возвращает собственные координаты сооружения.
3. void spawn_enemy – метод, создающий нового врага. Использует find_nearest_free_cell() для определения позиции спавна и, если найдена подходящая клетка, создает объект Enemy и размещает его на игровом поле через gamearea.set_human_in_cell().

8. Структура Coords – структура координат. Назначение: отвечает за то, чтобы хранить положение объектов на игровом поле.

Атрибуты:

1. int x и int y координаты по осям.

Методы:

1. Перегружен метод сложения, чтобы координаты можно было складывать. Складываются координаты соответствующих осей.
2. Перегружен оператор сравнения. Сравниваются соответствующие координаты.

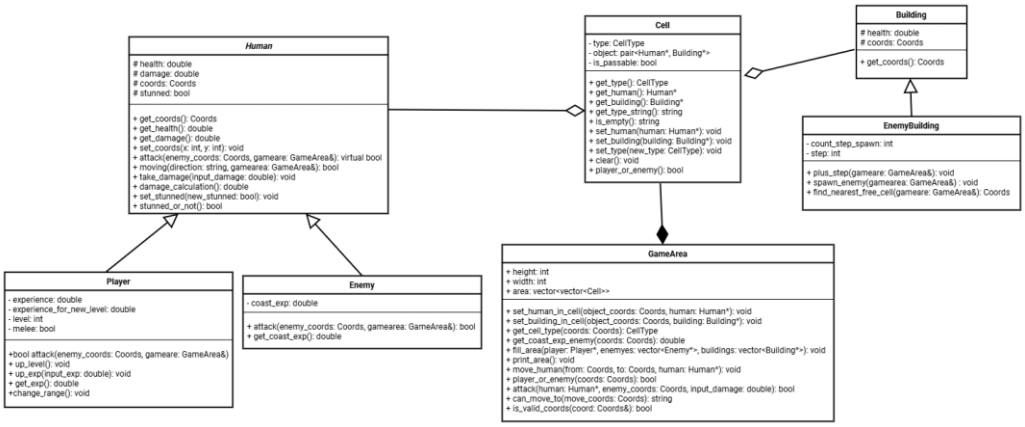


Рисунок 1 - UML-диаграмма классов

Выводы.

Разработана игровая система, реализующая механику пошаговых тактических боев с управлением персонажем, врагами и специальными сооружениями. Архитектура построена с учетом принципов ООП и расширяемости.

ПРИЛОЖЕНИЕ А
ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: human.h

```
class Human {
protected:
double health;
double damage;
Coords coords;
bool stunned = false;
public:
Human(double h, double d, Coords coords) : health(h), damage(d),
coords(coords) {}
Coords get_coords();
double get_health();
double get_damage();
void set_coords(int x, int y);
virtual bool attack(Coords enemy_coords, GameArea& gamearea) = 0;
bool moving(std::string direction, GameArea& gamearea);
void take_damage(double input_damage);
double damage_calculation();
void set_stunned(bool new_stunned);
bool stunned_or_not();
};
```

Название файла: human.cpp

```
#include "Human.h"
#include "GameArea.h"
#include <cstdlib>
#include <random>


Coords Human::get_coords()
{
     return coords;
}

void Human::set_coords(int x, int y)
{
     coords.x = x;
     coords.y = y;
}

bool Human::moving(std::string direction, GameArea& gamearea) {
    Coords new_coords = coords;

    if (direction == "top") new_coords.y--;
    else if (direction == "down") new_coords.y++;
    else if (direction == "right") new_coords.x++;
    else if (direction == "left") new_coords.x--;
    else return false;

    std::string obj_in_new_cell = gamearea.can_move_to(new_coords);
```

```
    if (obj_in_new_cell == "Edge" || obj_in_new_cell == "Tower")
        return false;

    if (stunned) {
        gamearea.move_human(coords, new_coords, this);
        return false;
    }

    if          (obj_in_new_cell          ==          "Empty"          &&
gamearea.get_cell_type(new_coords) != CellType::BLOCKED)
        gamearea.move_human(coords, new_coords, this);
    else       if       (obj_in_new_cell       ==       "Empty"       &&
gamearea.get_cell_type(new_coords) == CellType::BLOCKED) {
        std::cout << "Cell is blocked. You cant move here.";
        return false;
    }
    else if (obj_in_new_cell == "Human") {
        try {
            bool win = attack(new_coords, gamearea);
            if (!win) {
                return false;
            }
        }
        catch (const PlayerDyeException& e) {
            std::cout << e.what() << std::endl;
        }

    }
    set_coords(new_coords.x, new_coords.y);
    return true;
}

double Human::get_health()
{
    return health;
}

double Human::get_damage()
{
    return damage;
}

double Human::damage_calculation()
{
    std::random_device rd;
    std::mt19937 gen(rd());
    std::uniform_int_distribution<> distrib(0, 1);
    double out_damage = damage;
    bool random_plus_or_minus = distrib(gen);
    if (random_plus_or_minus)
    {
        out_damage += rand() % ((int)damage - 5);
    }
    else
    {
        out_damage -= rand() % ((int)damage - 5);
    }
    return out_damage;
```

```
}

void Human::set_stunned(bool new_stunned)
{
    stunned = new_stunned;
}

bool Human::stunned_or_not()
{
    return stunned;
}

void Human::take_damage(double input_damage)
{
    health -= input_damage;
}
```

Название файла: enemy.h

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



class Enemy : public Human {
private:
     double coast_exp;
public:
     Enemy(double h, double d, Coords coords) : Human(h, d, coords),
coast_exp(h / 2) {}
     bool attack(Coords enemy_coords, GameArea& gamearea);
     double get_coast_exp();
};
```

Название файла: enemy.cpp

```
#include "Enemy.h"
#include "GameArea.h"


bool Enemy::attack(Coords enemy_coords, GameArea& gamearea)
{
    bool win = gamearea.attack(this, enemy_coords, damage_calculation());
    if (win) {
        throw PlayerDyeException();
    }
    return win;
}

double Enemy::get_coast_exp()
{
    return coast_exp;
}
```

Название файла: player.h

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


class Player : public Human {
```

```
private:
	double experience;
	double experience_for_new_level;
	int level;
	bool melee = true;
public:
	Player(double h, double d, Coords coords) : Human(h, d, coords),
experience(0), experience_for_new_level(50), level(1) {}
	bool attack(Coords enemy_coords, GameArea& gamearea);
	void up_level();
	void up_exp(double input_exp);
	double get_exp();
	void change_range();
};
```

Название файла: player.cpp

```
#include "Player.h"
#include "GameArea.h"


bool Player::attack(Coords enemy_coords, GameArea& gamearea)
{
	bool	win	=	gamearea.attack(this,	enemy_coords,
damage_calculation());
	if (win) {
		if (experience > experience_for_new_level) {
			up_level();
		}
	}
	return win;
}

void Player::up_level()
{
	double rest_exp = experience - experience_for_new_level;
	level++;
	experience = rest_exp;
	health += (health / 100) * 5;
	damage += (damage / 100) * 5;
}

void Player::up_exp(double input_exp)
{
	experience += input_exp;
}

double Player::get_exp()
{
	return experience;
}

void Player::change_range()
{
	if (melee) {
		melee = false;
		damage -= (damage / 100) * 20;
		return;
	}
```

```
      melee = true;
      damage += (damage / 100) * 20;
}
```

Название файла: cell.h

```
#pragma once
#include <iostream>
#include <string>
#include <utility>


class Human;
class Building;

enum class CellType {
      BASIC,       // Обычная клетка
      BLOCKED,     // Непроходимое препятствие
      SLOW        // Замедляющая клетка
};

class Cell {
private:
      CellType type = CellType::BASIC;
      std::pair<Human*, Building*> object;
      bool is_passable = true;
public:
      Cell() : object(nullptr, nullptr) {}
      CellType get_type();
      Human* get_human();
      Building* get_building();
      std::string get_type_string();
      std::string is_empty();
      void set_human(Human* human);
      void set_building(Building* building);
      void set_type(CellType new_type);
      void clear();
      bool player_or_enemy();
};
```

Название файла: cell.cpp

```
#include "Cell.h"
#include "Player.h"
#include "Enemy.h"
#include "Building.h"

CellType Cell::get_type()
{
    return type;
}

std::string Cell::is_empty()
{
    if (!(object.first) && !(object.second)) {
        return std::string("Empty");
    }
    else if (object.first && !(object.second)) {
        return std::string("Human");
    }
```

```
    else {
        return std::string("Building");
    }
}

void Cell::set_human(Human* human)
{
    if (this->is_empty() == "Empty") {
        object.first = human;
    }
}

void Cell::set_building(Building* tower)
{
    if(this->is_empty() == "Empty") {
        object.second = tower;
    }
}

void Cell::set_type(CellType new_type)
{
    type = new_type;
}

Human* Cell::get_human()
{
    return object.first;
}

Building* Cell::get_building()
{
    return object.second;
}

bool Cell::player_or_enemy() // Вернет true, если Player иначе Enemy
{
    Human* human = object.first;
    if (Player* player = dynamic_cast<Player*>(human)) {
        std::cout << "Это Player!\n";
        return true;
    }
    else if (Enemy* enemy = dynamic_cast<Enemy*>(human)) {
        std::cout << "Это Enemy!\n";
        return false;
    }
    else {
        return false; //Ошибку запихать
    }
}

void Cell::clear() {
    object.first = nullptr;
    object.second = nullptr;
}

std::string Cell::get_type_string() {
    switch (type) {
    case CellType::BASIC: return "Basic";
```

```
    case CellType::BLOCKED: return "Blocked";
    case CellType::SLOW: return "Slow";
    default: return "Unknown";
    }
}
```

Название файла: EnemyBuilding.h

```
#pragma once
#include "Building.h"
#include "Enemy.h"


class EnemyBuilding : public Building {
private:
     int count_step_spawn;
     int step;
public:
     EnemyBuilding(int h, Coords crd, int c) : Building(h, crd),
count_step_spawn(c), step(0) {}
     void plus_step(GameArea& gamearea);
     void spawn_enemy(GameArea& gamearea);
     Coords find_nearest_free_cell(GameArea& gamearea);
};
```

Название файла: EnemyBuilding.cpp

```
#include "EnemyBuilding.h"
#include "GameArea.h"
#include <vector>

void EnemyBuilding::plus_step(GameArea& gamearea)
{
     step++;
    if (step == count_step_spawn) {
        spawn_enemy(gamearea);
        step = 0;
    }
}

Coords EnemyBuilding::find_nearest_free_cell(GameArea& gamearea) {

    const std::vector<Coords> directions = {
        Coords(1, 0),   // право
        Coords(0, 1),   // вверх
        Coords(-1, 0),  // лево
        Coords(0, -1),  // вниз
        Coords(1, 1),   // право-вверх
        Coords(-1, 1),  // лево-вверх
        Coords(1, -1),  // право-вниз
        Coords(-1, -1)  // лево-вниз
    };

    for (auto dir : directions) {
        Coords candidate = coords + dir;
        if     (gamearea.can_move_to(candidate)     ==     "Empty"     &&
gamearea.get_cell_type(candidate) != CellType::BLOCKED) {
            return candidate;
```

```
        }
    }

    return coords;
}

void EnemyBuilding::spawn_enemy(GameArea& gamearea) {
    Coords spawn_pos = find_nearest_free_cell(gamearea);
    if (spawn_pos != coords) {
        Enemy* new_enemy = new Enemy(10, 5, spawn_pos);
        gamearea.set_human_in_cell(spawn_pos, new_enemy);

    }
}
```

Название файла: GameArea.h

```
#pragma once
#include "Cell.h"
#include "Coords.h"
#include <vector>


class Player;
class Enemy;
class Human;

class GameArea {
private:
     int height;
     int width;
     std::vector<std::vector<Cell>> area;

public:
     GameArea(int h, int w) : height(h), width(w), area(h,
std::vector<Cell>(w)) {}
    // Конструктор копирования
    GameArea(const GameArea& other)
        : height(other.height), width(other.width), area(other.area) {
    }

    // Операторприсваивания с копированием
    GameArea& operator=(const GameArea& other) {
        if (this != &other) {
            height = other.height;
            width = other.width;
            area = other.area; // Глубокая копия через оператор= вектора
        }
        return *this;
    }

    // Конструктор перемещения
    GameArea(GameArea&& other) noexcept
        : height(std::exchange(other.height, 0)),
        width(std::exchange(other.width, 0)),
        area(std::move(other.area)) {
    }

    // Оператор присваивания с перемещением
```

```
    GameArea& operator=(GameArea&& other) noexcept {
        if (this != &other) {
            height = std::exchange(other.height, 0);
            width = std::exchange(other.width, 0);
            area = std::move(other.area);
        }
        return *this;
    }
     void set_human_in_cell(Coords object_coords, Human* human);
    void set_building_in_cell(Coords object_coords, Building* building);
     CellType get_cell_type(Coords coords);
     double get_coast_exp_enemy(Coords coords);
     void   fill_area(Player*   player,   std::vector<Enemy*>   enemyes,
std::vector<Building*> buildings);
     void print_area();
     void move_human(Coords from, Coords to, Human* human);
     bool player_or_enemy(Coords coords);
    bool attack(Human* human, Coords enemy_coords, double input_damage);
     std::string can_move_to(Coords move_coords);
     bool is_valid_coords(Coords& coord);

};
```

Название файла: GameArea.cpp

```
#include "GameArea.h"
#include "Player.h"
#include "Enemy.h"
#include "Building.h"

void GameArea::set_human_in_cell(Coords object_coords, Human* human)
{
     area[object_coords.y][object_coords.x].set_human(human);
}

void   GameArea::set_building_in_cell(Coords   object_coords,   Building*
building)
{
     area[object_coords.y][object_coords.x].set_building(building);
}

void  GameArea::fill_area(Player*  player,  std::vector<Enemy*>  enemyes,
std::vector<Building*> buildings)
{
    double count_another_cell = (((double)height * (double)width) / 100)
* 10 + 1;
     for (int i = 0; i < count_another_cell; i++) {
          int x_block = rand() % width;
          int y_block = rand() % height;
          area[x_block][y_block].set_type(CellType::BLOCKED);

          int x_slow = rand() % width;
          int y_slow = rand() % height;
          area[x_slow][y_slow].set_type(CellType::SLOW);
     }

     Coords player_coords = player->get_coords();
     set_human_in_cell(player_coords, player);
     for (auto enemy : enemyes) {
```

```
            Coords enemy_coords = enemy->get_coords();
            set_human_in_cell(enemy_coords, enemy);
      }

      for (auto building : buildings) {
            Coords building_coords = building->get_coords();
            set_building_in_cell(building_coords, building);
      }
}

void GameArea::print_area()
{
      for (int y = 0; y < height; y++) {
            for (int x = 0; x < width; x++) {
                  std::string obj = area[y][x].is_empty();
                  if (obj == "Empty") {
                        std::cout << area[y][x].get_type_string() << ' ';
                  }
                  else {
                        std::cout << obj << ' ';
                  }
            }
            std::cout << '\n';
      }
}

void GameArea::move_human(Coords from, Coords to, Human* human)
{

      if (human->stunned_or_not()) {
            human->set_stunned(false);
            std::cout << "You are stanned.\n";
            return;
      }

      area[from.y][from.x].clear();
      area[to.y][to.x].set_human(human);

      if (area[to.y][to.x].get_type() == CellType::SLOW) {
            human->set_stunned(true);
      }
}

bool GameArea::player_or_enemy(Coords coords)
{
      return area[coords.y][coords.x].player_or_enemy();
}

bool   GameArea::attack(Human*   human,   Coords   enemy_coords,   double
input_damage)
{
      Human* enemy = area[enemy_coords.y][enemy_coords.x].get_human();
      enemy->take_damage(input_damage);

      double enemy_health = enemy->get_health();
      if (enemy_health <= 0) {
            Coords old_coords = human->get_coords();
            if (area[old_coords.y][old_coords.x].player_or_enemy()) {
```

```
                Player* plaeyr_ptr = static_cast<Player*>(human);
                plaeyr_ptr->up_exp(get_coast_exp_enemy(enemy_coords));
                //std::cout << get_coast_exp_enemy(enemy_coords) << " up
expirence";
           }
           move_human(old_coords, enemy_coords, human);
           return true;
     }
     return false;
}

std::string GameArea::can_move_to(Coords move_coords)
{
     if (!is_valid_coords(move_coords) || !is_valid_coords(move_coords))
{
           return std::string("Edge");
     }
     else {
           return area[move_coords.y][move_coords.x].is_empty();
     }
}

CellType GameArea::get_cell_type(Coords coords)
{
     return area[coords.y][coords.x].get_type();
}

double GameArea::get_coast_exp_enemy(Coords coords)
{
     Human* enemy = area[coords.y][coords.x].get_human();
     Enemy* enemy_ptr = static_cast<Enemy*>(enemy);
     return enemy_ptr->get_coast_exp();
}

bool GameArea::is_valid_coords(Coords& coord) {
     return coord.x >= 0 && coord.x < width && coord.y >= 0 && coord.y <
height;
}
```

Название файла: building.h

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

class Building {
protected:
     double health;
     Coords coords;
public:
     Building(int h, Coords crd) : health(h), coords(crd) {}
     Coords get_coords();
};
```

Название файла: building.cpp

```
#pragma once
#include "Building.h"
Coords Building::get_coords() {
     return coords;
}
```

Название файла: coords.h

```
#pragma once
struct Coords {
	int x;
	int y;
	Coords(int x, int y) : x(x), y(y) {}
	Coords operator+(Coords& other) {
		return Coords(x + other.x, y + other.y);
	}
	bool operator!=(Coords& other) {
		return x != other.x || y != other.y;
	}
};
```

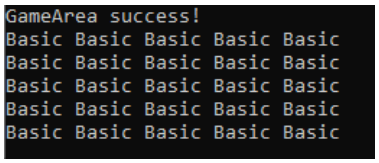
Название файла: human.h

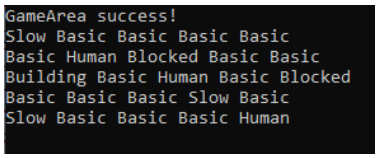
Название файла: human.h

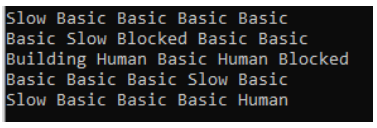
Название файла: human.h

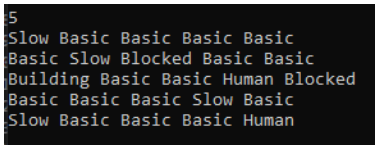
ПРИЛОЖЕНИЕ Б
ТЕСТИРОВАНИЕ

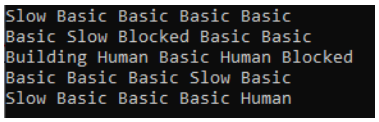
Таблица Б.2 - Примеры тестовых случаев

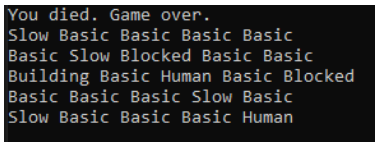
№ п/п	Входные данные	Выходные данные	Комментарии
1.	GameArea gamearea(5, 5); gamearea.print_area();		Поле создано.

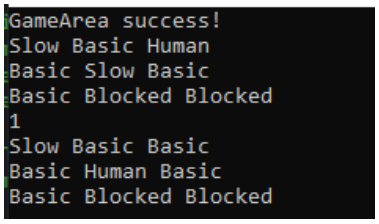
2.	Coords player_coords(1, 1);
	Player player(100, 10, player_coords);
	Enemy enemy1(10, 101, Coords(2, 2));
	Enemy enemy2(20, 2, Coords(4,4));
	std::vector<Enemy*> vct;
	vct.push_back(&enemy1);
	vct.push_back(&enemy2);
	EnemyBuilding building1(15, Coords(0,2), 3);
	std::vector<Building*> vct_b;
	vct_b.push_back(&building1);
	gamearea.fill_area(&player, vct, vct_b);
	gamearea.print_area();
	std::cout << '\n';
			Поле заполнено игровым персонажем, врагами и зданием. Так же появились специфичные клетки.

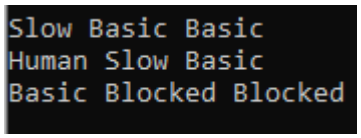
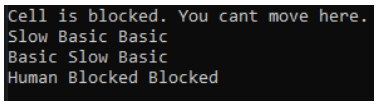
3.	player.moving(std::string("down"), gamearea);
	enemy1.moving(std::string("right"), gamearea);
	gamearea.print_area();
			Игрок переместился на клетку вниз, враг на клетку вправо, все работает.

4.	player.moving(std::string("right"), gamearea);
	player.moving(std::string("right"), gamearea);
	std::cout <<'\n' << player.get_exp() << '\n';
	gamearea.print_area();
			Атаковали enemy1, победили и получили опыт в размере 5-ти единиц. Переместился персонаж на клетку поверженного врага.

5.	building1.plus_step(gamearea);
	building1.plus_step(gamearea);
	building1.plus_step(gamearea);
	gamearea.print_area();
			Здание заспавнило нового юнита.

6.	enemy1.moving(std::string("left"), gamearea); gamearea.print_area();		Враг напал на игрового персонажа и убил его. Игра окончена.

7.	GameArea gamearea(3, 3);
	std::cout << "GameArea success!\n";
	Coords player_coords(2, 0);
	Player player(100, 10, player_coords);
	std::vector<Enemy*> vct;
	std::vector<Building*> vct_b;
	gamearea.fill_area(&player, vct, vct_b);
	gamearea.print_area();
	player.moving("left", gamearea);
	player.moving("down", gamearea);
			Инициализировано и заполнено поле 3 на 3. Попадая в замедляющую клетку, флаг класса Player принимает значение 1.

	std::cout << player.stunned_or_not() << '\n';		
8.	player.moving("left", gamearea); player.moving("left", gamearea);		Игроку приходится потратить два хода, чтобы пойти налево.
9.	player.moving("down", gamearea); player.moving("right", gamearea);		Игрок не может пройти через замедляющую клетку.