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»

Студентка гр. 4384 _____ Зайченко Е.Э.

Преподаватель _____ Жангиров Т.Р.

Санкт-Петербург

2025

Цель работы.

Ознакомиться с основами объектно-ориентированного программирования и применить их на практике, разработав на языке C++ прототип пошаговой игры, включающей перемещение игрового персонажа по карте и сражение с врагом.

Задание.

На 6/3/1 баллов:

Создать интерфейс карточки заклинания. Заклинание должно применяться игроком. На использование заклинания игрок тратит один ход.

Создать класс “руки” игрока, которая содержит все карточки заклинаний, которые игрок может применить в свой ход. Изначально рука игрока содержит только одно случайное заклинание. Реализовать возможность получать новые заклинание игроком, например, тратить очки на покупку или после уничтожения определенного кол-ва врагов. Размер “руки” должен быть ограничен и задается через конструктор.

Реализовать интерфейс заклинанием прямого урона. Это заклинание при использовании должно наносить урон врагу или вражескому зданию, если они находятся в достижимом радиусе. Если в качестве цели не выбран враг или вражеское здание, то заклинание не используется.

Реализовать интерфейс заклинания урона по площади. Это заклинание при использовании в допустимом радиусе наносит урон по области 2 на 2 клетки. Заклинание используется, даже если там нет никого.

На 8/4/1.5 баллов:

Реализовать интерфейс заклинания ловушки. Заклинание размещает на поле ловушку, если враг наступает на клетку с ловушкой, то ему наносится урон, и ловушка пропадает.

Создать класс вражеской башни. Вражеская башня размещается на поле, и если в радиусе ее атаки появляется игрок, то применяет ослабленную версию заклинания прямого урона. Не может применять заклинание несколько ходов подряд.

На 10/5/2 баллов:

Реализовать интерфейс заклинания призыва. Заклинание создает союзника рядом с игроком, который перемещается самостоятельно.

Реализовать интерфейс заклинание улучшения. Заклинание улучшает следующее используемое заклинание:

Заклинание прямого урона - увеличивает радиус применения

Заклинание урона по площади - увеличивает площадь

Заклинание ловушки - увеличивает урон

Заклинание призыва - призывает больше союзников

Заклинание улучшение - накапливает усиление, то есть при применении следующего заклинания отличного от улучшения, все улучшения применяются сразу

Примечания:

Интерфейс заклинания должен быть унифицирован, чтобы их можно было единообразно использовать через интерфейс. Не должно быть методов в интерфейсе, которые не используются каким-то классом наследником.

Избегайте явных проверок на тип данных.

Выполнение работы.

Была реализована программа, содержащая все указанные в условие лабораторной работы классы и их поля и методы, а именно:

- Класс игрока с характеристиками здоровья, урона, очков и возможностью перемещения по игровому полю;
- Класс врага с характеристиками здоровья и урона;
- Класс игрового поля, на котором размещаются игрок и враги;
- Проверка завершения игры при потере здоровья игроком.

Архитектура программы.

В программе реализована иерархия классов, соответствующая принципам ООП.

Основные классы:

- Player – класс игрока с характеристиками здоровья, урона, очков и возможностью перемещения по полю;
- Enemy – класс врага с характеристиками здоровья и урона, управляемый компьютером;
- GameField – класс игрового поля, на котором размещаются игрок и враги;
- Cell – класс отдельной клетки игрового поля.
- Hand – класс «руки» игрока, контейнер для хранения доступных заклинаний.
- Spell – виртуальный класс, задающий единый интерфейс для всех заклинаний.
- DirectDamageSpell – класс заклинания прямого урона.
- AreaDamageSpell – класс заклинания урона по площади.

Дополнительные enum классы:

- Cell::EntityType – класс типов содержимого клетки (kEmpty, kPlayer, kEnemy).

Описание классов.

Основные классы:

- Класс Player

  Класс содержит характеристики и методы игрового персонажа.

  Поля класса:

  - health – текущее здоровье
  - damage – базовый урон
  - score – очки игрока
  - row_, col_ - координаты игрока на игровом поле

  Методы класса:

  - health() – возвращает текущее здоровье
  - damage() – возвращает значение урона
  - score() – возвращает очки игрока

- take_damage(int damage) – уменьшает здоровье на указанное количество урона
- add_score(int points) – увеличивает очки игрока на указанное количество
- is_alive() – проверяет, жив ли игрок (true, если здоровье > 0)
- set_position(int row, int col) – устанавливает координаты игрока
- row() – возвращает текущую строку позиции игрока
- col() – возвращает текущий столбец позиции игрока

- Класс Enemy

  Класс содержит характеристики и методы врага.

  Поля класса:

  - health – текущее здоровье
  - damage – базовый урон
  - row_, col_ - координаты игрока на игровом поле

  Методы класса:

  - health() – возвращает текущее здоровье
  - damage() – возвращает значение урона
  - set_position(int row, int col) – устанавливает координаты врага
  - row() – возвращает текущую строку позиции врага
  - col() – возвращает текущий столбец позиции врага

- Класс Cell

  Класс содержит характеристики и методы клеток игрового поля.

  Поля класса:

  - type_ – тип содержимого клетки (EntityType)

  Методы класса:

  - type() – возвращает текущий тип клетки
  - set_type(EntityType type) – устанавливает тип клетки

- Класс GameField

Класс содержит характеристики и методы игрового поля, а также характеристики и методы взаимодействия персонажей с игровым полем.

Поля класса:

- rows_ – количество строк игрового поля
- cols_ – количество столбцов игрового поля
- grid_ – двумерный вектор клеток (Cell)

Методы класса:

- is_valid_position(int row, int col) – проверяет, что координаты находятся внутри поля
- is_empty(int row, int col) – проверяет, пуста ли клетка
- place_player(const Player& player) – размещает игрока на поле
- place_enemy(const Enemy& enemy) – размещает врага на поле
- clear_cell(int row, int col) – очищает клетку
- move_player(int new_row, int new_col, Player& player) – перемещает игрока на указанную клетку, если она пуста
- move_enemy(int new_row, int new_col, Enemy& enemy, Player& player) – перемещает врага; если враг попадает на игрока, наносит ему урон
- is_game_over(const Player& player) – проверяет завершение игры (если здоровье игрока ≤ 0)
- Конструкторы копирования и перемещения, операторы присваивания с копированием и перемещением

- Класс Hand

Класс изначально содержит одно случайное заклинание, поддерживает добавление новых заклинаний и их использование.

Поля класса:

- spells_ – вектор указателей на объекты заклинаний.
- max_size – максимальное количество заклинаний в руке.

Методы класса:

- Hand – конструктор, инициализирует руку заданного размера и добавляет первое случайное заклинание.
- add_random_spell – добавляет случайное заклинание (прямого урона или урона по области), если есть свободное место.
- use_spell – применяет заклинание по указанному индексу к заданной клетке поля.
- show_spells – выводит список заклинаний в руке с их индексами

- Класс Spell

  Класс обеспечивает полиморфное использование различных типов заклинаний через общий интерфейс.

  Методы класса:

  - virtual ~Spell() — виртуальный деструктор для корректного удаления наследников.
  - virtual void use– виртуальный метод применения заклинания
  - virtual const char* name() – виртуальный метод, возвращающий название заклинания

- Класс DirectDamageSpell

  Класс наносит урон цели (врагу или вражескому зданию), находящейся в пределах заданного радиуса от игрока. Если цель вне радиуса или поля — заклинание не применяется.

  Поля класса:

  - damage_ – величина наносимого урона (int).
  - radius_ – максимальное манхэттенское расстояние до цели (int).

  Методы класса:

  - DirectDamageSpell – конструктор с параметрами урона и радиуса действия.
  - use – реализация применения заклинания с проверкой расстояния и корректности координат.

- name – возвращает строку "Прямой урон".

- Класс AreaDamageSpell

  Класс наносит урон всем целям в области 2×2 клеток с центром в указанной точке. Заклинание применяется даже если в указанной области нет целей.

  Поля класса:

  - damage_ – величина урона по области (int).
  - radius_ – радиус действия (в текущей реализации используется для расширения функционала, но не влияет на логику площади 2×2) (int).

  Методы класса:

  - AreaDamageSpell – конструктор с параметрами урона и радиуса.
  - use – реализация применения заклинания с проверкой корректности координат цели.
  - name – возвращает строку "Урон по области".

Дополнительные enum классы:

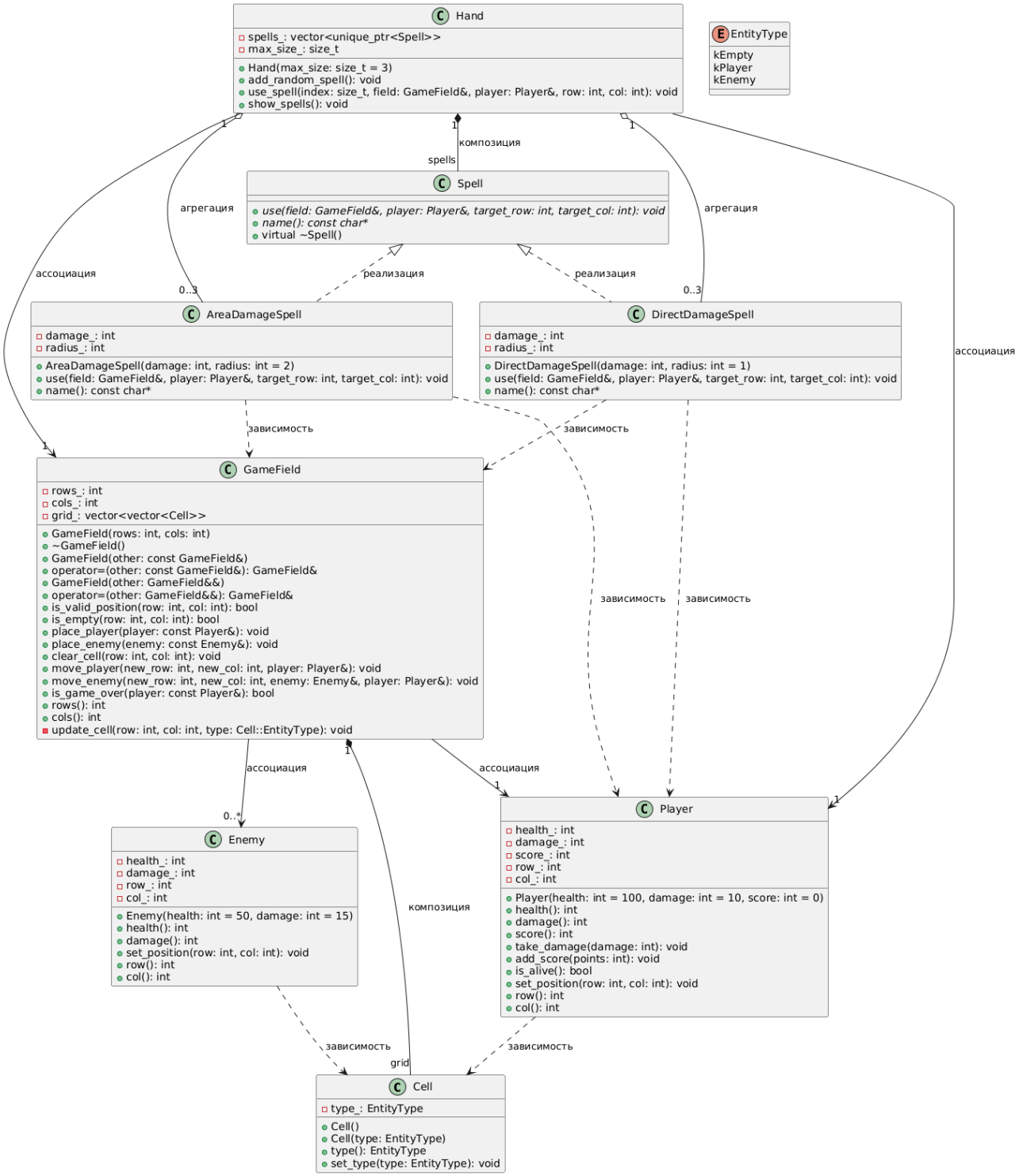
- Класс Cell::EntityType

  Значения:

  - kEmpty – пустая клетка
  - kPlayer – клетка с игроком
  - kEnemy – клетка с врагом

Разработанный программный код см. в приложении А.

UML-диаграмма



Вывод.

Была изучена парадигма объектно-ориентированного программирования. Была реализована программа на языке C++ включающая основные классы игры с необходимыми полями и методами.

ПРИЛОЖЕНИЕ А
ИСХОДНЫЙ КОД ПРОГРАММЫ

main.cpp

```
#define NOMINMAX
#include <Windows.h>
#include <iostream>
#include "cell.h"
#include "player.h"
#include "enemy.h"
#include "game_field.h"
#include "hand.h"

// === Тестирование базовых классов из ЛР1 ===
void test_basic_game_elements() {
    std::cout << "=== Тестирование базовых классов ===\n";

    Player player(100, 10, 0);
    Enemy enemy(50, 15);
    GameField field(10, 10);

    player.set_position(2, 2);
    enemy.set_position(2, 3);

    field.place_player(player);
    field.place_enemy(enemy);

    std::cout << "Игрок и враг размещены на поле.\n";
    std::cout << "Игрок: (" << player.row() << ", " << player.col()
<< "), здоровье: "
        << player.health() << "\n";
    std::cout << "Враг:  (" << enemy.row() << ", " << enemy.col()
<< "), урон: "
        << enemy.damage() << "\n";

    std::cout << "Враг атакует игрока...\n";
    field.move_enemy(player.row(), player.col(), enemy, player);
```

```
        std::cout << "Здоровье игрока после атаки: " << player.health()
<< "\n";

        if (field.is_game_over(player))
            std::cout << "Игра окончена — игрок погиб.\n";
        else
            std::cout << "Игрок выжил!\n";

        std::cout << "=== Тестирование базовых классов завершено
===\n\n";
    }

    // === Тестирование заклинаний из ЛР2 ===
    void test_spells() {
        std::cout << "=== Тестирование системы заклинаний (ЛР2) ===\n";

        GameField field(10, 10);
        Player player(100, 10, 0);
        player.set_position(4, 4);

        Hand hand(3); // максимум 3 заклинания в руке
        std::cout << "\nНачальные заклинания:\n";
        hand.show_spells();

        std::cout << "\nИгрок использует заклинание №0 на клетку (4,
5)...\n";
        hand.use_spell(0, field, player, 4, 5);

        std::cout << "\nИгрок получает новое заклинание...\n";
        hand.add_random_spell();

        std::cout << "\nОбновлённая рука игрока:\n";
        hand.show_spells();

        std::cout << "\nИгрок использует заклинание №1 на клетку (3,
3)...\n";
        hand.use_spell(1, field, player, 3, 3);
```

```
        std::cout << "\n=== Тестирование системы заклинаний завершено
===\n\n";
    }

    int main() {
        // Настройка кодировки консоли
        SetConsoleOutputCP(1251);
        SetConsoleCP(1251);
        setlocale(LC_ALL, "");

        try {
            std::cout << "=== Лабораторная работа №2 ===\n";
            std::cout << "Тема: Интерфейс заклинаний и рука игрока\n\n";

            test_basic_game_elements(); // тесты из ЛР1
            test_spells();              // тесты из ЛР2

            std::cout << "=== Все тесты успешно завершены ===\n";
        }
        catch (const std::exception& e) {
            std::cerr << "Ошибка: " << e.what() << "\n";
            return 1;
        }

        return 0;
    }
```

player.cpp

```
#include "player.h"

Player::Player(int health, int damage, int score)
    : health_(health), damage_(damage), score_(score) {
}

void Player::take_damage(int damage) {
    if (damage > 0) {
        health_ -= damage;
        if (health_ < 0) health_ = 0;
    }
}

void Player::add_score(int points) {
    if (points > 0) {
        score_ += points;
    }
}

bool Player::is_alive() const {
    return health_ > 0;
}

void Player::set_position(int row, int col) {
    row_ = row;
    col_ = col;
}
```

player.h

```
#ifndef GAME_PLAYER_H_
#define GAME_PLAYER_H_

class Player {
public:
    explicit Player(int health = 100, int damage = 10, int score =
0);

    int health() const { return health_; }
    int damage() const { return damage_; }
    int score() const { return score_; }

    void take_damage(int damage);
    void add_score(int points);
    bool is_alive() const;

    void set_position(int row, int col);
    int row() const { return row_; }
    int col() const { return col_; }

private:
    int health_;
    const int damage_;
    int score_;
    int row_ = -1;
    int col_ = -1;
};

#endif
```

enemy.cpp

```
#include "enemy.h"

Enemy::Enemy(int health, int damage)
    : health_(health), damage_(damage) {
}

void Enemy::set_position(int row, int col) {
    row_ = row;
    col_ = col;
}
```

enemy.h

```
#ifndef GAME_ENEMY_H_
#define GAME_ENEMY_H_

class Enemy {
public:
    explicit Enemy(int health = 50, int damage = 15);

    int health() const { return health_; }
    int damage() const { return damage_; }

    void set_position(int row, int col);
    int row() const { return row_; }
    int col() const { return col_; }

private:
    int health_;
    const int damage_;
    int row_ = -1;
    int col_ = -1;
};

#endif // GAME_ENEMY_H_
```

game_field.cpp

```
#include "game_field.h"

GameField::GameField(int rows, int cols)
    : rows_(rows), cols_(cols), grid_(rows, std::vector<Cell>(cols))
{
    if (rows < 10 || rows > 25 || cols < 10 || cols > 25) {
        throw std::invalid_argument("Field size must be between
10x10 and 25x25.");
    }
}

// Copy constructor
GameField::GameField(const GameField& other)
    : rows_(other.rows_), cols_(other.cols_), grid_(other.grid_) {
}

// Copy assignment
GameField& GameField::operator=(const GameField& other) {
    if (this != &other) {
        rows_ = other.rows_;
        cols_ = other.cols_;
        grid_ = other.grid_;
    }
    return *this;
}

// Move constructor
GameField::GameField(GameField&& other) noexcept
    :           rows_(other.rows_),           cols_(other.cols_),
grid_(std::move(other.grid_)) {
}

// Move assignment
GameField& GameField::operator=(GameField&& other) noexcept {
    if (this != &other) {
        rows_ = other.rows_;
        cols_ = other.cols_;
```

```
            grid_ = std::move(other.grid_);
        }
        return *this;
    }

    bool GameField::is_valid_position(int row, int col) const {
        return row >= 0 && row < rows_ && col >= 0 && col < cols_;
    }

    bool GameField::is_empty(int row, int col) const {
        return is_valid_position(row, col) &&
            grid_[row][col].type() == Cell::EntityType::kEmpty;
    }

    void GameField::update_cell(int row, int col, Cell::EntityType type)
{
        if (is_valid_position(row, col)) {
            grid_[row][col].set_type(type);
        }
    }

    void GameField::place_player(const Player& player) {
        update_cell(player.row(),                          player.col(),
Cell::EntityType::kPlayer);
    }

    void GameField::place_enemy(const Enemy& enemy) {
        update_cell(enemy.row(), enemy.col(), Cell::EntityType::kEnemy);
    }

    void GameField::clear_cell(int row, int col) {
        update_cell(row, col, Cell::EntityType::kEmpty);
    }

    void  GameField::move_player(int  new_row,  int  new_col,  Player&
player) {
        if (!is_valid_position(new_row, new_col)) return;
        if (!is_empty(new_row, new_col)) return;
```

```
        clear_cell(player.row(), player.col());
        player.set_position(new_row, new_col);
        place_player(player);
    }

    void GameField::move_enemy(int new_row, int new_col, Enemy& enemy,
Player& player) {
        if (!is_valid_position(new_row, new_col)) return;

        if (new_row == player.row() && new_col == player.col()) {
            player.take_damage(enemy.damage());
            return;
        }

        if (!is_empty(new_row, new_col)) return;

        clear_cell(enemy.row(), enemy.col());
        enemy.set_position(new_row, new_col);
        place_enemy(enemy);
    }

    bool GameField::is_game_over(const Player& player) const {
        return !player.is_alive();
    }
```

game_field.h

```
#ifndef GAME_GAME_FIELD_H_
#define GAME_GAME_FIELD_H_

#include <vector>
#include <stdexcept>
#include "cell.h"
#include "player.h"
#include "enemy.h"

class GameField {
public:
    GameField(int rows, int cols);
    ~GameField() = default;

    // Copy
    GameField(const GameField& other);
    GameField& operator=(const GameField& other);

    // Move
    GameField(GameField&& other) noexcept;
    GameField& operator=(GameField&& other) noexcept;

    bool is_valid_position(int row, int col) const;
    bool is_empty(int row, int col) const;

    void place_player(const Player& player);
    void place_enemy(const Enemy& enemy);
    void clear_cell(int row, int col);

    void move_player(int new_row, int new_col, Player& player);
    void move_enemy(int new_row, int new_col, Enemy& enemy, Player&
player);

    bool is_game_over(const Player& player) const;

    int rows() const { return rows_; }
    int cols() const { return cols_; }
```

```
private:
    void update_cell(int row, int col, Cell::EntityType type);

    int rows_;
    int cols_;
    std::vector<std::vector<Cell>> grid_;
};

#endif // GAME_GAME_FIELD_H_
```

cell.h

```
#ifndef GAME_CELL_H_
#define GAME_CELL_H_

class Cell {
public:
    enum class EntityType {
        kEmpty,
        kPlayer,
        kEnemy
    };

    Cell() = default;
    explicit Cell(EntityType type) : type_(type) {}

    EntityType type() const { return type_; }
    void set_type(EntityType type) { type_ = type; }

private:
    EntityType type_ = EntityType::kEmpty;
};

#endif // GAME_CELL_H_
```

hand.cpp

```
#include "hand.h"

Hand::Hand(size_t max_size) : max_size_(max_size) {
    std::srand(static_cast<unsigned>(std::time(nullptr)));
    add_random_spell();
}

void Hand::add_random_spell() {
    if (spells_.size() >= max_size_) {
        std::cout << "Рука заполнена, нельзя добавить новое заклинание.\n";
        return;
    }

    int choice = std::rand() % 2;
    if (choice == 0)
        spells_.push_back(std::make_unique<DirectDamageSpell>(20, 1));
    else
        spells_.push_back(std::make_unique<AreaDamageSpell>(10, 2));

    std::cout << "Игрок получил новое заклинание: " << spells_.back()->name() << "\n";
}

void Hand::use_spell(size_t index, GameField& field, Player& player, int row, int col) {
    if (index >= spells_.size()) {
        std::cout << "Некорректный выбор заклинания.\n";
        return;
    }
    spells_[index]->use(field, player, row, col);
}

void Hand::show_spells() const {
    std::cout << "Заклинания в руке:\n";
    for (size_t i = 0; i < spells_.size(); ++i)
```

```
            std::cout << "   [" << i << "] " << spells_[i]->name() <<
"\n";
    }
```

hand.h

```
#ifndef GAME_HAND_H_
#define GAME_HAND_H_

#include <vector>
#include <memory>
#include <iostream>
#include <cstdlib>
#include <ctime>
#include "spell.h"
#include "direct_damage_spell.h"
#include "area_damage_spell.h"

class Hand {
public:
    explicit Hand(size_t max_size = 3);

    void  add_random_spell();                // Добавить случайное
заклинание
    void use_spell(size_t index, GameField& field, Player& player,
int row, int col);
    void show_spells() const;

private:
    std::vector<std::unique_ptr<Spell>> spells_;
    size_t max_size_;
};

#endif // GAME_HAND_H_
```

spell.h

```
#ifndef GAME_SPELL_H_
#define GAME_SPELL_H_

#include "enemy.h"
#include "game_field.h"

class Spell {
public:
    virtual ~Spell() = default;

    // Применить заклинание (игрок тратит 1 ход)
    virtual void use(GameField& field, Player& player, int target_row, int target_col) = 0;

    // Название заклинания
    virtual const char* name() const = 0;
};

#endif // GAME_SPELL_H_
```

direct_damage_spell.cpp

```
#include "direct_damage_spell.h"
#include <cmath>
#include <iostream>

void DirectDamageSpell::use(GameField& field, Player& player, int target_row, int target_col) {
    if (!field.is_valid_position(target_row, target_col)) {
        std::cout << "Цель вне поля!\n";
        return;
    }

    int dist = std::abs(player.row() - target_row) + std::abs(player.col() - target_col);
    if (dist > radius_) {
        std::cout << "Цель вне радиуса заклинания!\n";
        return;
    }

    // В этой базовой версии считаем, что вражеское здание отсутствует, проверяем только врагов
    std::cout << "Заклинание '" << name() << "' нанесло " << damage_ << " урона в точку ("
        << target_row << ", " << target_col << ")\n";
}
```

direct_damage_spell.h

```
#ifndef GAME_DIRECT_DAMAGE_SPELL_H_
#define GAME_DIRECT_DAMAGE_SPELL_H_

#include "spell.h"

class DirectDamageSpell : public Spell {
public:
    explicit DirectDamageSpell(int damage, int radius = 1)
        : damage_(damage), radius_(radius) {
    }

    void use(GameField& field, Player& player, int target_row, int
target_col) override;
    const char* name() const override { return "Прямой урон"; }

private:
    int damage_;
    int radius_;
};

#endif // GAME_DIRECT_DAMAGE_SPELL_H_
```

area_damage_spell.cpp

```
#include "area_damage_spell.h"
#include <iostream>

void AreaDamageSpell::use(GameField& field, Player& player, int target_row, int target_col) {
    if (!field.is_valid_position(target_row, target_col)) {
        std::cout << "Цель вне поля!\n";
        return;
    }

    std::cout << "Заклинание '" << name() << "' нанесло " << damage_
        << " урона по области 2x2 с центром в (" << target_row << ", " << target_col << ")\n";
}
```

area_damage_spell.h

```
#ifndef GAME_AREA_DAMAGE_SPELL_H_
#define GAME_AREA_DAMAGE_SPELL_H_

#include "spell.h"

class AreaDamageSpell : public Spell {
public:
    explicit AreaDamageSpell(int damage, int radius = 2)
        : damage_(damage), radius_(radius) {
    }

    void use(GameField& field, Player& player, int target_row, int
target_col) override;
    const char* name() const override { return "Урон по области"; }

private:
    int damage_;
    int radius_;
};

#endif // GAME_AREA_DAMAGE_SPELL_H_
```