**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: Реализация консольной игры с использованием ООП.**

Студент гр. 4384 Мазеев В.А.

Преподаватель _____ Жангиров Т.Р.

_____

Санкт-Петербург
2025

**Цель работы.**

Целью работы является разработка консольной игры с использованием принципов объектно-ориентированного программирования, демонстрирующей взаимодействие классов, инкапсуляцию, наследование.

**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.**

В результате работы разработан интерфейс карточки заклинания, класс руки для заклинаний, интерфейсы заклинания прямого урона и урона по площади. Были добавлены следующие классы:

Программа состоит из семи основных классов:

1. Класс *Spell*.

Назначение: абстрактный базовый класс для всех типов заклинаний. Определяет единый интерфейс для работы с магией в игре.

Основные методы:

*cast()* – активирует эффект заклинания.

*getName()* – возвращает название заклинания.

*getDescription()* – возвращает описание закл*инания.*

*clone()* – создает копию заклинания (полиморфное клонирование).

*getRadius()* – возвращает радиус действия заклинания.

*getDamage()* – возвращает значен*ие урона заклинания.*

*isInRange()* – проверяет, находится ли цель в радиусе действия.

2. Класс DirectDamageSpell

Назначение: заклинание прямого урона. Наносит повреждения одному врагу в пределах радиуса действия.

Основные методы:

*cast(Player* caster, int targetX, int targetY, Field& field)* – применяет заклинание от имени кастера в указанные координаты. Наносит урон одному врагу в радиусе действия. Возвращает false если цель вне радиуса, позиция невалидна или враг не найден.

*getName()* – возвращает название заклинания "Прямой урон".

*getDescription()* – возвращает описание "Наносит урон одному врагу в радиусе действия".

*clone()* – создает и возвращает уникальный указатель на копию заклинания.

*isInRange(Player* caster, int targetX, int targetY)* – проверяет, находится ли цель в радиусе действия от позиции кастера.

Наследует методы *getDamage() и getRadius()* от базового класса *Spell.*

3. Класс *AreaDamageSpell*.

*cast(Player* caster, int targetX, int targetY, Field& field)* – применяет заклинание в указанные координаты. Наносит урон всем врагам в области 2x2 клетки. Всегда возвращает true при валидной цели, даже если врагов в области нет.

*getName()* – возвращает название заклинания "Урон по области".

*getDescription()* – возвращает описание "Наносит урон по области 2x2 клетки".

*clone()* – создает и возвращает уникальный указатель на копию заклинания.

*isInRange(Player* caster, int targetX, int targetY)* – проверяет, находится ли центр области в радиусе действия от позиции кастера.

Обрабатывает область 2x2 клетки, начиная с указанных координат как верхнего левого угла.

Наследует методы *getDamage()* и *getRadius()* от базового класса Spell.

4. Класс Hand.

Назначение: управляет коллекцией заклинаний игрока. Ограничивает количество карт, которые игрок может носить с собой.

Основные методы:

*Hand(int maxSize)* – конструктор, инициализирует руку с указанным максимальным размером. Автоматически добавляет одно случайное заклинание при создании.

*addSpell(std::unique_ptr<Spell> spell)* – добавляет заклинание в руку. Возвращает false если рука заполнена или передан невалидный указатель.

*getSpell(int index)* – возвращает указатель на заклинание по индексу. Возвращает nullptr при невали*дном индексе.*

*getSpellCount()* – возвращает *текущее количество заклинаний в руке.*

*getMaxSize()* – возвращает максимальный размер руки.

*isFull()* – проверяет, заполнена ли рука (достигнут максимальный размер).

*isEmpty()* – проверяет, пуста ли рука.

*createRandomSpell()* – создает случайное заклинание (DirectDamageSpell или AreaDamageSpell) со стандартными параметрами.

Конструктор копирования и оператор присваивания обеспечивают правильное полиморфное копирование через метод *clone()*.

5.Интеграция с Player.

Назначение: расширяет функциональность игрока системой заклинаний, позволяя использовать магию в дополнение к обычным атакам.

Основные методы:

*getHand()* – предоставляет доступ к коллекции заклинаний игрока (две перегрузки для константного и неконстантного доступа).

*castSpell(int spellIndex, int targetX, int targetY, Field& field)* – применяет заклинание из указанной ячейки руки по заданным координатам. Возвращает false при неверном индексе заклинания.

Конструкторы и операторы присваивания обеспечивают корректное копирование и перемещение объекта *Hand.*

Игрок автоматически инициализируется рукой заданного размера через параметр *handSize* в конструкторе.

6. Интеграция с Game.

Назначение: добавляет систему магии в основной игровой цикл и управление взаимодействием с заклинаниями.

Основные методы:

*run()* – основной игровой цикл с обработкой команды C для применения заклинаний.

*showHand()* – отображает полную информацию о коллекции заклинаний игрока (название, описание, урон, радиус).

*tryGiveNewSpell()* – награждает игрока случайным заклинанием за победу над врагом, если есть свободное место в руке.

*processEnemiesTurn()* – обрабатывает ход врагов только после использования хода игроком.

**Архитектура.**

Выбранная архитектура программы расширена системой заклинаний, основанной на принципах полиморфизма и инкапсуляции. Базовый абстрактный класс Spell определяет единый интерфейс для всех типов магии, а конкретные реализации (DirectDamageSpell, AreaDamageSpell) инкапсулируют специфическую логику применения заклинаний. Это позволяет обрабатывать разные типы магии через общий интерфейс и легко расширять систему новыми видами заклинаний.

Класс Hand, управляет коллекцией заклинаний как единым целым. Использование полиморфного метода clone() обеспечивает корректное копирование объектов заклинаний при работе с контейнером, сохраняя конкретные типы объектов.

Интеграция системы заклинаний с существующей архитектурой выполнена через композицию - класс Player содержит объект Hand, что позволяет игроку использовать магию в дополнение к базовой механике боя. Класс Game координирует взаимодействие системы заклинаний с игровым процессом, обрабатывая команды каста, отображение состояния руки и награждение новыми заклинаниями.

Принцип единственной ответственности соблюден: логика заклинаний инкапсулирована в отдельных классах, игрок управляет коллекцией, а игра обрабатывает взаимодействие. Это обеспечивает слабую связанность компонентов и простоту дальнейшего расширения системы.

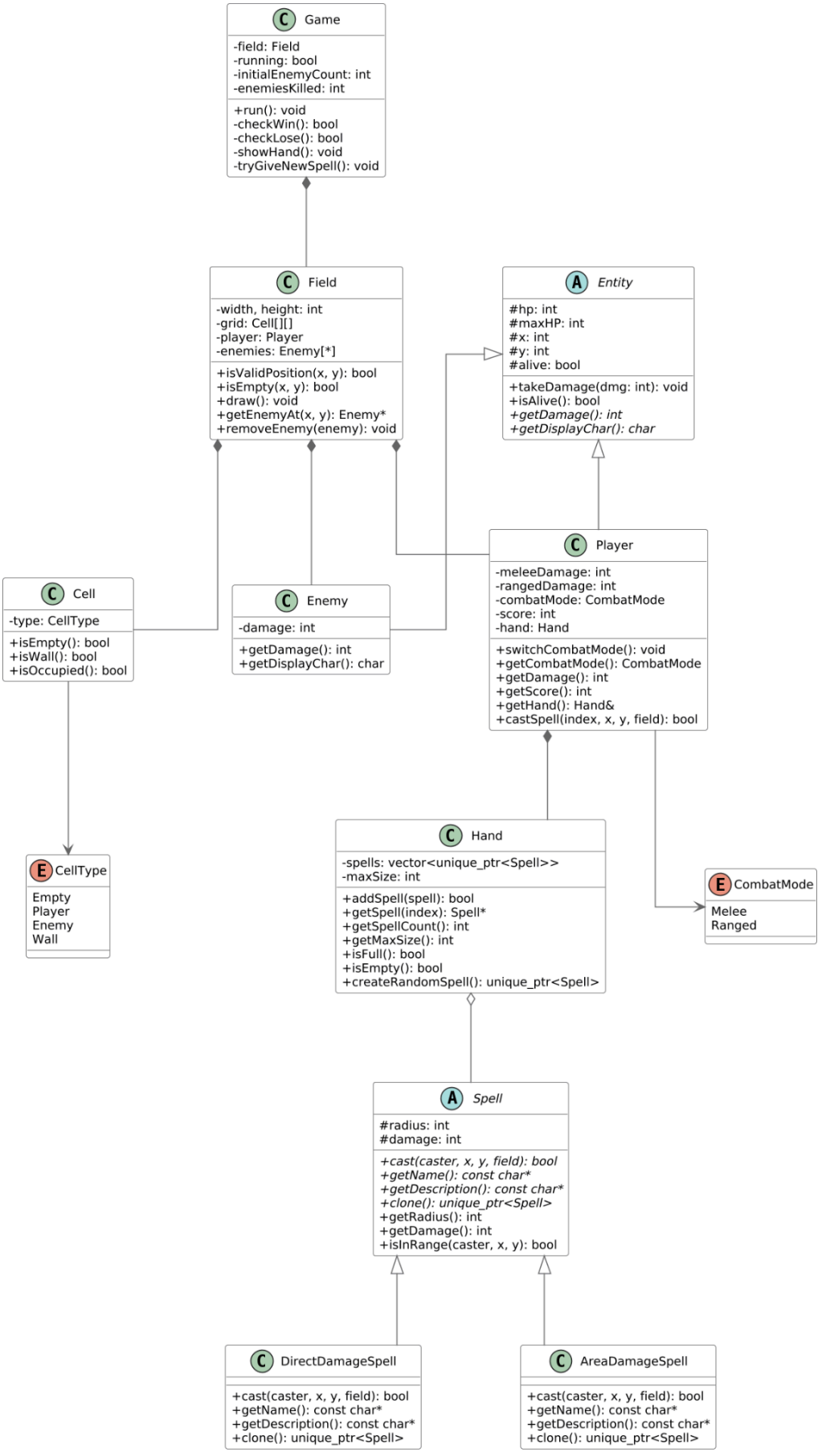


Рис. 1 UML-диаграмма классов.

**Выводы.**

В ходе лабораторной работы была разработана консольная игра на языке C++, реализованная с применением объектно-ориентированного программирования. В процессе выполнения были созданы и связаны между собой классы Game, Field, Entity, Player, Enemy и Cell, каждый из которых выполняет строго определенную роль в архитектуре программы.

Разработанная структура демонстрирует принципы инкапсуляции, наследования и слабой связанности компонентов, что облегчает поддержку и масштабирование кода. Игровой процесс: игрок перемещается по полю, враги совершают случайные ходы и атакуют при столкновении, поле обновляется в реальном времени.

# ПРИЛОЖЕНИЕ А

# ТЕСТИРОВАНИЕ

**tests.cpp**

```
#include <iostream>
#include <cassert>
#include <memory>
#include <cstdlib>
#include <windows.h>

#include "Cell.h"
#include "Entity.h"
#include "Player.h"
#include "Enemy.h"
#include "Field.h"
#include "Hand.h"
#include "DirectDamageSpell.h"
#include "AreaDamageSpell.h"

namespace {
    void clearField(Field& field) {
        for (int y = 0; y < field.getHeight(); ++y) {
            for (int x = 0; x < field.getWidth(); ++x) {
                field.clearCell(x, y);
            }
        }
    }
}

// ===== Testing Cell =====
void testCell() {
    std::cout << "Testing Cell..." << std::endl;

    Cell cell;

    // Check initial state
    assert(cell.isEmpty() == true);
    assert(cell.isWall() == false);

    // Check type setting
    cell.setType(CellType::Wall);
    assert(cell.isWall() == true);
    assert(cell.isEmpty() == false);

    cell.setType(CellType::Player);
    assert(cell.getType() == CellType::Player);
    assert(cell.isOccupied() == true);

    std::cout << "Cell tests passed!" << std::endl;
}

// ===== Testing Entity =====
void testEntity() {
    std::cout << "Testing Entity..." << std::endl;

    // Create test class for abstract Entity
    class TestEntity : public Entity {
```

```
    public:
        TestEntity(int x, int y, int hp) : Entity(x, y, hp) {}
        int getDamage() const override { return 10; }
        char getDisplayChar() const override { return 'T'; }
    };

    TestEntity entity(5, 5, 100);

    // Check position and health
    assert(entity.getX() == 5);
    assert(entity.getY() == 5);
    assert(entity.getHP() == 100);
    assert(entity.isAlive() == true);

    // Check damage taking
    entity.takeDamage(30);
    assert(entity.getHP() == 70);

    // Check death
    entity.takeDamage(100);
    assert(entity.getHP() == 0);
    assert(entity.isAlive() == false);

    std::cout << "Entity tests passed!" << std::endl;
}

// ===== Testing Player =====
void testPlayer() {
    std::cout << "Testing Player..." << std::endl;

    Player player(2, 2, 100, 15, 5);

    // Check initial state
    assert(player.getHP() == 100);
    assert(player.getCombatMode() == CombatMode::Melee);
    assert(player.getDamage() == 15); // Melee mode

    // Check combat mode switching
    player.switchCombatMode();
    assert(player.getCombatMode() == CombatMode::Ranged);
    assert(player.getDamage() == 5); // Ranged mode

    // Check score system
    assert(player.getScore() == 0);
    player.addScore(10);
    assert(player.getScore() == 10);

    // Check display character
    assert(player.getDisplayChar() == 'P');

    std::cout << "Player tests passed!" << std::endl;
}

// ===== Testing Hand =====
void testHand() {
    std::cout << "Testing Hand..." << std::endl;

    std::srand(0); // deterministic first spell
```

```
    Hand hand(3);
    assert(hand.getMaxSize() == 3);
    assert(hand.getSpellCount() == 1);
    assert(!hand.isEmpty());
    assert(!hand.isFull());
    assert(hand.getSpell(0) != nullptr);

    bool added = hand.addSpell(std::make_unique<DirectDamageSpell>(3,
25));
    assert(added);
    assert(hand.getSpellCount() == 2);
    assert(hand.getSpell(1) != nullptr);

    added = hand.addSpell(std::make_unique<AreaDamageSpell>(3, 15));
    assert(added);
    assert(hand.getSpellCount() == 3);
    assert(hand.isFull());

        added  =  hand.addSpell(std::make_unique<DirectDamageSpell>(3,
10));
    assert(added == false);

    Hand copy = hand;
    assert(copy.getSpellCount() == hand.getSpellCount());
    assert(copy.isFull() == hand.isFull());
    assert(copy.getSpell(0) != nullptr);

    Hand assigned(2);
    assigned = hand;
    assert(assigned.getSpellCount() == hand.getSpellCount());
    assert(assigned.isFull() == hand.isFull());

    std::cout << "Hand tests passed!" << std::endl;
}

void testEnemy() {
    std::cout << "Testing Enemy..." << std::endl;

    Enemy enemy(3, 3, 50, 8);

    assert(enemy.getHP() == 50);
    assert(enemy.getDamage() == 8);
    assert(enemy.isAlive() == true);

    enemy.takeDamage(20);
    assert(enemy.getHP() == 30);
    assert(enemy.isAlive() == true);

    assert(enemy.getDisplayChar() == 'E');

    std::cout << "Enemy tests passed!" << std::endl;
}

void testField() {
    std::cout << "Testing Field..." << std::endl;

    Field field(10, 10);
```

```
    clearField(field);

    assert(field.getWidth() == 10);
    assert(field.getHeight() == 10);

    assert(field.isValidPosition(0, 0) == true);
    assert(field.isValidPosition(9, 9) == true);
    assert(field.isValidPosition(-1, 0) == false);
    assert(field.isValidPosition(10, 10) == false);

    auto player = std::make_unique<Player>(1, 1);
    field.setPlayer(std::move(player));
    assert(field.getPlayer() != nullptr);
    assert(field.hasPlayerAt(1, 1) == true);

    auto enemy = std::make_unique<Enemy>(2, 2);
    field.addEnemy(std::move(enemy));
    assert(field.getEnemies().size() == 1);
    assert(field.hasEnemyAt(2, 2) == true);

    Enemy* enemyPtr = field.getEnemyAt(2, 2);
    assert(enemyPtr != nullptr);
    field.removeEnemy(enemyPtr);
    assert(field.hasEnemyAt(2, 2) == false);

    std::cout << "Field tests passed!" << std::endl;
}

void testMovement() {
    std::cout << "Testing Movement..." << std::endl;

    Field field(10, 10);
    clearField(field);
    auto player = std::make_unique<Player>(5, 5);
    field.setPlayer(std::move(player));

    assert(field.getPlayer() != nullptr);
    assert(field.getPlayer()->getX() == 5);
    assert(field.getPlayer()->getY() == 5);

    assert(field.getPlayer()->canMove(1, 0, field) == true);
    assert(field.getPlayer()->canMove(-5, 0, field) == false);

    std::cout << "Movement tests passed!" << std::endl;
}

// ===== Testing Direct Damage Spell =====
void testDirectDamageSpell() {
    std::cout << "Testing DirectDamageSpell..." << std::endl;

    Field field(10, 10);
    clearField(field);

    auto player = std::make_unique<Player>(1, 1);
    Player* playerPtr = player.get();
    field.setPlayer(std::move(player));

    DirectDamageSpell spell(3, 20);
```

```
    field.clearCell(2, 1);
    auto enemy1 = std::make_unique<Enemy>(2, 1, 30, 5);
    field.addEnemy(std::move(enemy1));

    bool castSuccess = spell.cast(playerPtr, 2, 1, field);
    assert(castSuccess == true);
    Enemy* enemyAfter = field.getEnemyAt(2, 1);
    assert(enemyAfter != nullptr);
    assert(enemyAfter->getHP() == 10);
    assert(playerPtr->getScore() == 0);

    bool castNoTarget = spell.cast(playerPtr, 3, 1, field);
    assert(castNoTarget == false);

    bool castOutOfRange = spell.cast(playerPtr, 9, 9, field);
    assert(castOutOfRange == false);

    field.clearCell(3, 2);
    auto enemy2 = std::make_unique<Enemy>(3, 2, 10, 5);
    field.addEnemy(std::move(enemy2));
    Enemy* enemyToDie = field.getEnemyAt(3, 2);
    assert(enemyToDie != nullptr);
    int previousScore = playerPtr->getScore();

    bool castKill = spell.cast(playerPtr, 3, 2, field);
    assert(castKill == true);
    assert(field.getEnemyAt(3, 2) == nullptr);
    assert(playerPtr->getScore() == previousScore + 10);

    std::cout << "DirectDamageSpell tests passed!" << std::endl;
}

// ===== Testing Area Damage Spell =====
void testAreaDamageSpell() {
    std::cout << "Testing AreaDamageSpell..." << std::endl;

    Field field(10, 10);
    clearField(field);

    auto player = std::make_unique<Player>(2, 2);
    Player* playerPtr = player.get();
    field.setPlayer(std::move(player));

    AreaDamageSpell spell(4, 15);

    field.clearCell(3, 3);
    field.clearCell(4, 3);
    field.clearCell(3, 4);
    field.clearCell(4, 4);

    auto enemy1 = std::make_unique<Enemy>(3, 3, 25, 5);
    auto enemy2 = std::make_unique<Enemy>(4, 4, 25, 5);
    field.addEnemy(std::move(enemy1));
    field.addEnemy(std::move(enemy2));

    bool castSuccess = spell.cast(playerPtr, 3, 3, field);
    assert(castSuccess == true);
```

```
    Enemy* enemyA = field.getEnemyAt(3, 3);
    Enemy* enemyB = field.getEnemyAt(4, 4);
    assert(enemyA != nullptr);
    assert(enemyB != nullptr);
    assert(enemyA->getHP() == 10);
    assert(enemyB->getHP() == 10);

    bool castEmptyArea = spell.cast(playerPtr, 7, 7, field);
    assert(castEmptyArea == true);

    std::cout << "AreaDamageSpell tests passed!" << std::endl;
}

// ===== Testing Player Casting via Hand =====
void testPlayerCasting() {
    std::cout << "Testing Player casting spells..." << std::endl;

    Field field(10, 10);
    clearField(field);

    auto player = std::make_unique<Player>(1, 1);
    Player* playerPtr = player.get();
    field.setPlayer(std::move(player));

    Hand& hand = playerPtr->getHand();
    int initialCount = hand.getSpellCount();

    bool added = hand.addSpell(std::make_unique<DirectDamageSpell>(3,
35));
    assert(added == true);
    int spellIndex = hand.getSpellCount() - 1;

    field.clearCell(2, 1);
    auto enemy = std::make_unique<Enemy>(2, 1, 25, 5);
    field.addEnemy(std::move(enemy));

    bool castSuccess = playerPtr->castSpell(spellIndex, 2, 1, field);
    assert(castSuccess == true);
    assert(field.getEnemyAt(2, 1) == nullptr);
    assert(playerPtr->getScore() >= 10);

    bool invalidIndex = playerPtr->castSpell(999, 2, 1, field);
    assert(invalidIndex == false);

    bool emptyTarget = playerPtr->castSpell(spellIndex, 3, 1, field);
    assert(emptyTarget == false);

    // Ensure original random spells remain accessible
    assert(hand.getSpellCount() >= initialCount);

    std::cout << "Player casting tests passed!" << std::endl;
}

int main() {
    SetConsoleOutputCP(CP_UTF8);
    SetConsoleCP(CP_UTF8);
    std::cout << "=== Starting Simple Game Tests ===" << std::endl;
```

```
        std::cout << "Running basic functionality tests...\n" <<
std::endl;

    try {
        testCell();
        testEntity();
        testPlayer();
        testEnemy();
        testHand();
        testField();
        testMovement();
        testDirectDamageSpell();
        testAreaDamageSpell();
        testPlayerCasting();

        std::cout << "\n=== ALL TESTS PASSED! ===" << std::endl;
        std::cout << "Basic game functionality is working correctly!"
<< std::endl;

    } catch (const std::exception& e) {
        std::cerr << "\nTest failed: " << e.what() << std::endl;
        return 1;
    } catch (...) {
        std::cerr << "\nUnknown test failure" << std::endl;
        return 1;
    }

    return 0;
}
```

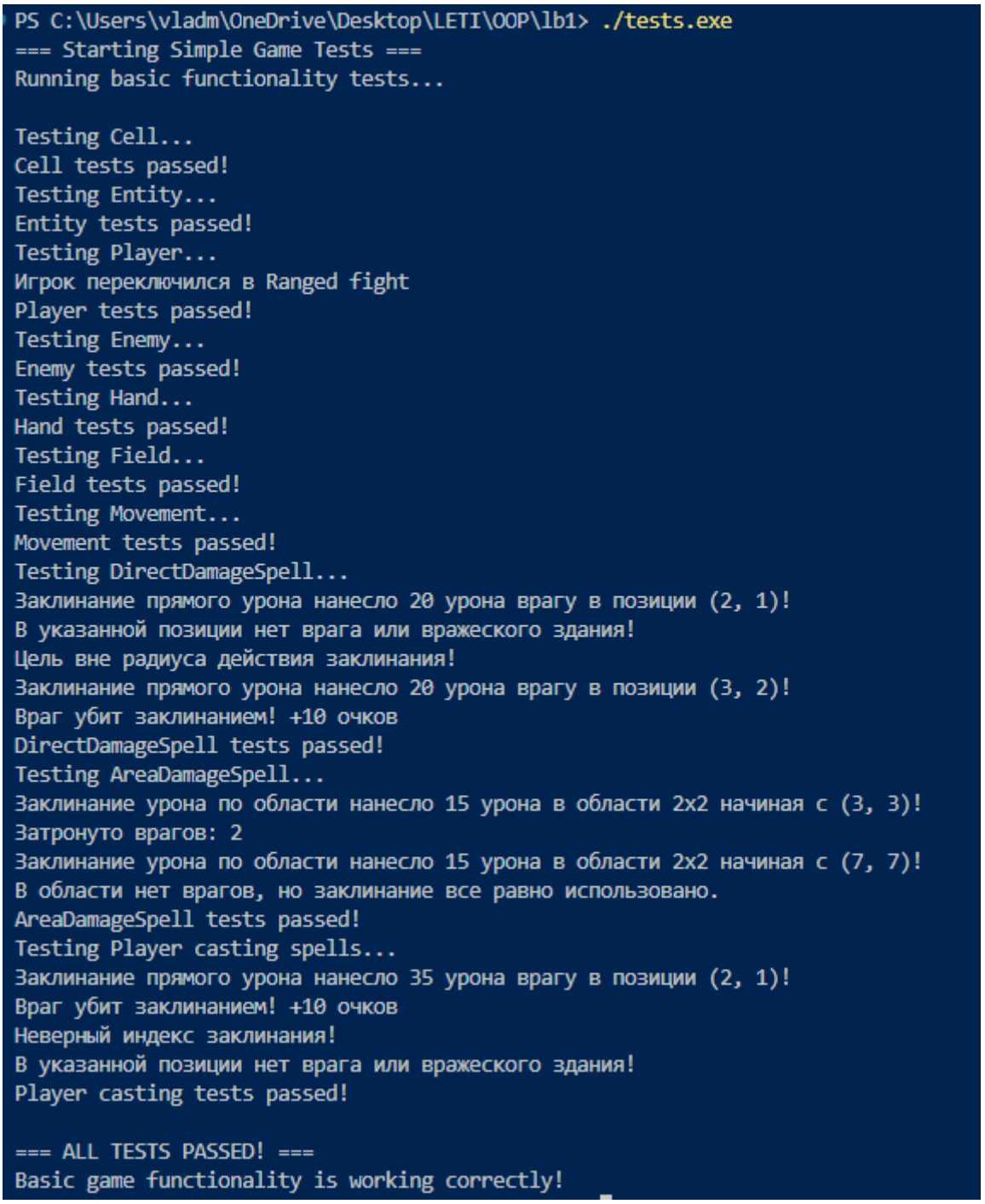


Рис. 2 Результаты тестирования.

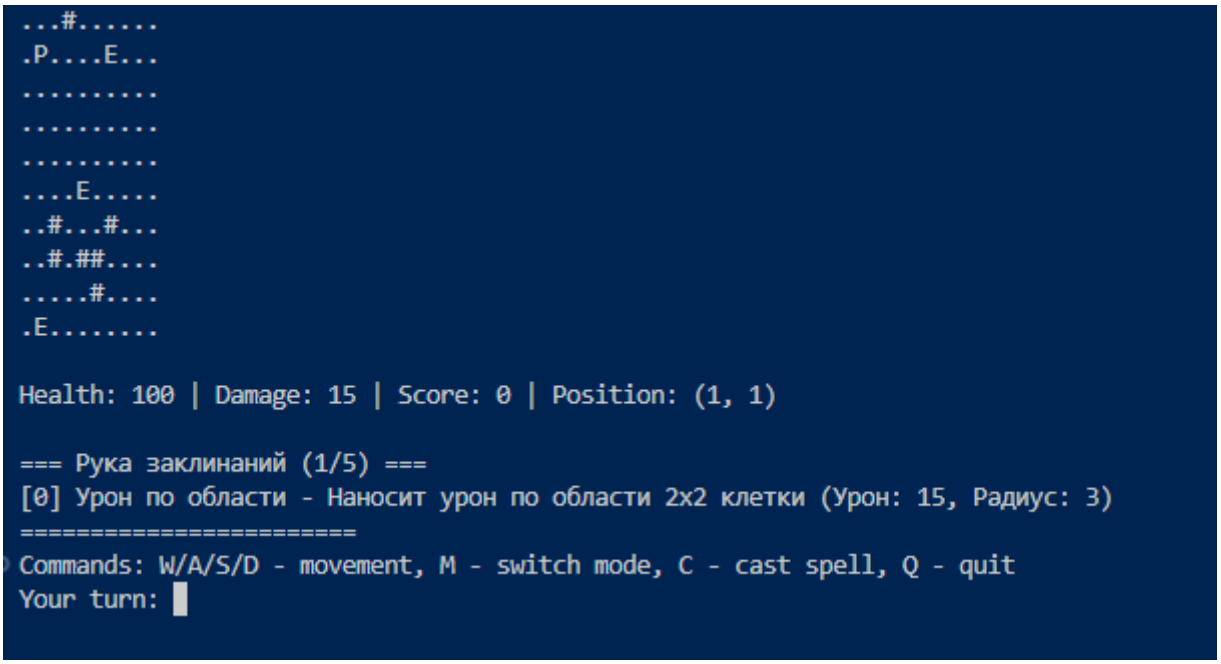


Рис. 3 Начало игры.

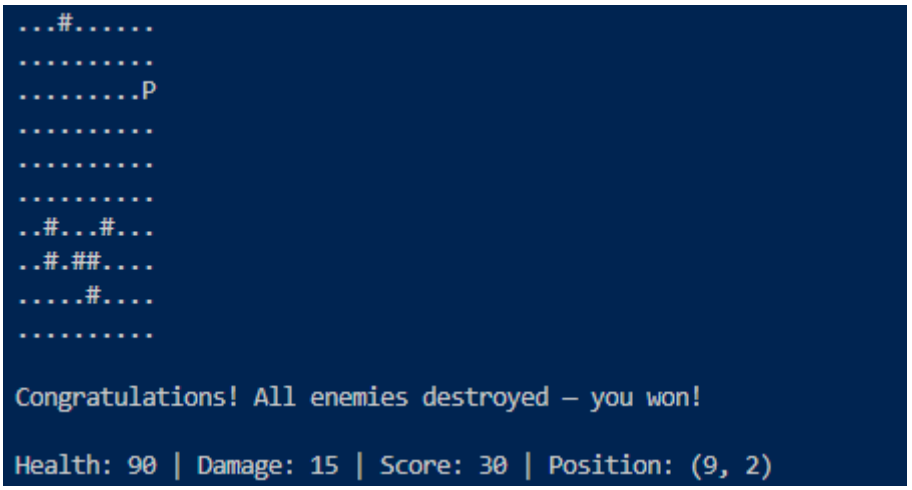


Рис. 4 Конец игры.

# ПРИЛОЖЕНИЕ В
# ИСХОДНЫЙ КОД

**hand.cpp**

```
#include "Hand.h"
#include "DirectDamageSpell.h"
#include "AreaDamageSpell.h"
#include <cstdlib>
#include <ctime>

Hand::Hand(int maxSize) : maxSize(maxSize) {
    if (maxSize < 1) {
        this->maxSize = 1;
    }

    // Initially, hand contains one random spell
    spells.push_back(createRandomSpell());
}

Hand::Hand(const Hand& other) : maxSize(other.maxSize) {
    spells.reserve(other.spells.size());
    for (const auto& spell : other.spells) {
        if (spell) {
            spells.push_back(spell->clone());
        }
    }
}

Hand& Hand::operator=(const Hand& other) {
    if (this == &other) {
        return *this;
    }

    spells.clear();
    maxSize = other.maxSize;

    spells.reserve(other.spells.size());
    for (const auto& spell : other.spells) {
        if (spell) {
            spells.push_back(spell->clone());
        }
    }

    return *this;
}

std::unique_ptr<Spell> Hand::createRandomSpell() const {
    // Randomly choose between DirectDamageSpell and AreaDamageSpell
    int spellType = std::rand() % 2;

    if (spellType == 0) {
        return std::make_unique<DirectDamageSpell>(3, 20);
    } else {
        return std::make_unique<AreaDamageSpell>(3, 15);
    }
}
```

```
bool Hand::addSpell(std::unique_ptr<Spell> spell) {
    if (!spell) return false;
    if (isFull()) return false;

    spells.push_back(std::move(spell));
    return true;
}

Spell* Hand::getSpell(int index) const {
    if (index < 0 || index >= static_cast<int>(spells.size())) {
        return nullptr;
    }
    return spells[index].get();
}

int Hand::getSpellCount() const {
    return static_cast<int>(spells.size());
}

int Hand::getMaxSize() const {
    return maxSize;
}

bool Hand::isFull() const {
    return spells.size() >= static_cast<size_t>(maxSize);
}

bool Hand::isEmpty() const {
    return spells.empty();
}
```

**hand.h**

```
#pragma once
#include "Spell.h"
#include <vector>
#include <memory>

// Forward declaration
class Spell;

// Hand class - manages player's spell cards
class Hand {
private:
    std::vector<std::unique_ptr<Spell>> spells;
    int maxSize;

    // Helper function to create a random spell
    std::unique_ptr<Spell> createRandomSpell() const;

public:
    Hand(int maxSize = 5);

    Hand(const Hand& other);
    Hand& operator=(const Hand& other);
    Hand(Hand&& other) noexcept = default;
    Hand& operator=(Hand&& other) noexcept = default;
```

```
    ~Hand() = default;

    // Add a spell to the hand (returns false if hand is full)
    bool addSpell(std::unique_ptr<Spell> spell);

    // Get spell at index (returns nullptr if index is invalid)
    Spell* getSpell(int index) const;

    // Get number of spells in hand
    int getSpellCount() const;

    // Get maximum hand size
    int getMaxSize() const;

    // Check if hand is full
    bool isFull() const;

    // Check if hand is empty
    bool isEmpty() const;
};
```

## AreaDamageSpell.cpp

```
#include "AreaDamageSpell.h"
#include "Player.h"
#include "Field.h"
#include "Enemy.h"
#include <iostream>

AreaDamageSpell::AreaDamageSpell(int radius, int damage)
    : Spell(radius, damage) {}

bool AreaDamageSpell::cast(Player* caster, int targetX, int targetY,
Field& field) {
    if (!caster) return false;

    // Check if target position is valid
    if (!field.isValidPosition(targetX, targetY)) {
        std::cout << "Неверная позиция цели!\n";
        return false;
    }

    // Area damage affects a 2x2 area starting from target position
     // We'll use targetX, targetY as the top-left corner of the 2x2
area
    int enemiesHit = 0;

    for (int dy = 0; dy < 2; dy++) {
        for (int dx = 0; dx < 2; dx++) {
            int checkX = targetX + dx;
            int checkY = targetY + dy;

            if (!field.isValidPosition(checkX, checkY)) {
                continue;
            }
```

```
            // Deal damage to any enemy in this cell
            Enemy* enemy = field.getEnemyAt(checkX, checkY);
            if (enemy) {
                enemy->takeDamage(damage);
                enemiesHit++;

                // Remove enemy if killed
                if (!enemy->isAlive()) {
                    caster->addScore(10);
                    field.removeEnemy(enemy);
                }
            }
        }
    }

    std::cout << "Заклинание урона по области нанесло " << damage
              << " урона в области 2x2 начиная с (" << targetX << ",
" << targetY << ")!\n";
    if (enemiesHit > 0) {
        std::cout << "Затронуто врагов: " << enemiesHit << "\n";
    } else {
         std::cout << "В области нет врагов, но заклинание все равно
использовано.\n";
    }

       // Spell is used even if no enemies are present (as per
requirements)
    return true;
}

const char* AreaDamageSpell::getName() const {
    return "Урон по области";
}

const char* AreaDamageSpell::getDescription() const {
    return "Наносит урон по области 2x2 клетки";
}

std::unique_ptr<Spell> AreaDamageSpell::clone() const {
    return std::make_unique<AreaDamageSpell>(radius, damage);
}
```

## AreaDamageSpell.h

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

// Area damage spell - deals damage to a 2x2 area
class AreaDamageSpell : public Spell {
public:
    AreaDamageSpell(int radius = 3, int damage = 15);

    AreaDamageSpell(const AreaDamageSpell& other) = default;
       AreaDamageSpell&  operator=(const  AreaDamageSpell&  other)  =
default;
```

```
    AreaDamageSpell(AreaDamageSpell&& other) noexcept = default;
      AreaDamageSpell& operator=(AreaDamageSpell&& other) noexcept =
default;

    bool cast(Player* caster, int targetX, int targetY, Field& field)
override;
    const char* getName() const override;
    const char* getDescription() const override;
    std::unique_ptr<Spell> clone() const override;
};
```

## DirectDamageSpell.cpp

```
#include "DirectDamageSpell.h"
#include "Player.h"
#include "Field.h"
#include "Enemy.h"
#include "Entity.h"
#include <iostream>

DirectDamageSpell::DirectDamageSpell(int radius, int damage)
    : Spell(radius, damage) {}

bool   DirectDamageSpell::cast(Player*   caster,   int   targetX,   int
targetY, Field& field) {
    if (!caster) return false;

    // Check if target is in range
    if (!isInRange(caster, targetX, targetY)) {
        std::cout << "Цель вне радиуса действия заклинания!\n";
        return false;
    }

    // Check if target position is valid
    if (!field.isValidPosition(targetX, targetY)) {
        std::cout << "Неверная позиция цели!\n";
        return false;
    }

    // Try to find an enemy at target position
    Enemy* enemy = field.getEnemyAt(targetX, targetY);

    // If no enemy found, spell is not used (as per requirements)
    if (!enemy) {
          std::cout << "В указанной позиции нет врага или вражеского
здания!\n";
        return false;
    }

    // Deal damage to the enemy
    enemy->takeDamage(damage);
    std::cout << "Заклинание прямого урона нанесло " << damage
                << " урона врагу в позиции (" << targetX << ", " <<
targetY << ")!\n";

    // Remove enemy if killed
    if (!enemy->isAlive()) {
        std::cout << "Враг убит заклинанием! +10 очков\n";
```

```
            caster->addScore(10);
            field.removeEnemy(enemy);
        }

        return true;
}

const char* DirectDamageSpell::getName() const {
    return "Прямой урон";
}

const char* DirectDamageSpell::getDescription() const {
    return "Наносит урон одному врагу в радиусе действия";
}

std::unique_ptr<Spell> DirectDamageSpell::clone() const {
    return std::make_unique<DirectDamageSpell>(radius, damage);
}
```

## DirectDamageSpell.h

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

// Direct damage spell - deals damage to a single target (enemy or
enemy building)
class DirectDamageSpell : public Spell {
public:
    DirectDamageSpell(int radius = 3, int damage = 20);

    DirectDamageSpell(const DirectDamageSpell& other) = default;
      DirectDamageSpell& operator=(const DirectDamageSpell& other) =
default;
    DirectDamageSpell(DirectDamageSpell&& other) noexcept = default;
     DirectDamageSpell& operator=(DirectDamageSpell&& other) noexcept
= default;

    bool cast(Player* caster, int targetX, int targetY, Field& field)
override;
    const char* getName() const override;
    const char* getDescription() const override;
    std::unique_ptr<Spell> clone() const override;
};
```