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

ОТЧЕТ

по лабораторной работе №4
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:

1. Создать класс считывающий ввод пользователя и преобразующий ввод пользователь в объект команды.
2. Создать класс отрисовки игры. Данный класс определяет то, как должно отображаться игра.
3. Создать шаблонный класс управления игрой. В качестве параметра шаблона должен передаваться класс, отвечающий за считывание и преобразование ввода. У себя он создает объект класса из параметра шаблона и получает от него команды, а далее вызывает нужное действие у классов игры. Данный класс не должен создавать объект класса игры. Реализация должна быть такой, что можно масштабировать программу, например, реализовать получение команд через интернет без использования реализации интерфейса, и просто подставить новый класс в качестве параметра шаблона.
4. Создать шаблонный класс визуализации игры. . В качестве параметра шаблона должен передаваться класс, отвечающий за способ отрисовки игры. Данный класс создает объект класса отрисовки игры, и реагирует на изменения в игре, и вызывает команду отрисовку. Реализация должна быть такой, что можно масштабировать программу, например, реализовать отрисовку в виде веб-страницы без использования реализации интерфейса, и просто подставить новый класс в качестве параметра шаблона.

На 8/4/1.5 баллов:

5. Добавить возможность настраивать управление игрой через файл (то, на какие клавиши должна выполняться та или иная команда). Если команды некорректные: отсутствует информация для какой-то команды, на одну клавишу две разные команды назначены, для одной команды назначены две разные клавиши, то в таком случае управление должно устанавливаться по умолчанию.

На 10/5/2 баллов:

6. Добавить систему логирования событий в игре. Система должна реагировать на игровые события, и записывать об этом событии (то и кому сколько урона было нанесено, на какие координаты перешел игрок, получение заклинания, и.т.д.). Игровые сущности не должны напрямую вызывать систему логировать, а только лишь информировать о событии. Запись может идти как в файл, так и в терминал, способ логирования определяется пользователем через параметры запуска программы.

Примечания:

- После считывания клавиши, считанный символ должен сразу обрабатываться, и далее работа должна проводить с сущностью, которая представляет команду

- Для представления команды можно разработать системы классов
- Хорошей практикой является создание “прослойки” между считыванием/обработкой команды и классом игры, которая сопоставляет команду и вызываемым методом игры. Существуют альтернативные решения без явной “прослойки”

Выполнение работы.

Выбрано задание на 8 баллов.

В ходе выполнения лабораторной работы было расширено архитектурное устройство игрового приложения на языке C++. Основной задачей являлось упрощение замены способов ввода команд и визуализации игры без изменения основной игровой логики.

Для этого были реализованы следующие изменения.

Во-первых, был добавлен тип Command, который описывает действие игрока в удобном для игры виде. Также реализован класс ConsoleCommandReader, который считывает пользовательский ввод с клавиатуры, учитывает текущие назначения клавиш управления и преобразует ввод в объекты Command. Благодаря этому игровая логика больше не работает напрямую с вводом пользователя, а получает уже готовые команды.

Во-вторых, был реализован класс ConsoleRenderer, отвечающий за вывод состояния игры в консоль. Для обеспечения возможности замены способа отображения был создан шаблонный класс GameVisualizer<RenderPolicy>, который делегирует отрисовку переданному классу-рендереру. Это позволяет в дальнейшем заменить консольный вывод, например, на графический интерфейс, не изменяя код игры.

В-третьих, был реализован шаблонный класс GameManager<InputProvider>, который управляет игровым процессом. Он получает команды от переданного считывателя ввода, передаёт их в игру через метод applyCommand, а затем вызывает обработку последствий хода. При этом объект Game не создаётся внутри менеджера, а передаётся извне, что делает архитектуру более гибкой.

Кроме того, класс Game был доработан для поддержки новой архитектуры. В него были добавлены методы applyCommand, handlePostPlayerAction и render. Также метод processPlayerTurn был переработан так, чтобы использовать новую цепочку обработки команд через ConsoleCommandReader и объекты Command.

Настройка управления через конфигурационный файл сохранена. В случае некорректного файла управления (дубликаты клавиш или отсутствующие команды) используются настройки по умолчанию, что обеспечивает устойчивость программы к ошибкам конфигурации.

В результате выполненных изменений игровая логика была отделена от ввода и визуализации, а архитектура стала более расширяемой и удобной для дальнейшего развития.

Архитектура.

Архитектура программы была изменена и расширена таким образом, чтобы разделить игровую логику, ввод пользователя и визуализацию. Основная идея заключалась в том, чтобы убрать жёсткие зависимости между этими частями и сделать код более понятным и удобным для дальнейшего расширения.

В основе новой архитектуры лежит представление действий игрока в виде отдельного объекта команды. Для этого был добавлен класс Command, который хранит информацию о действии игрока, например перемещение, применение заклинаний и выбор цели. Теперь игровой класс Game не работает напрямую с нажатиями клавиш, а получает уже готовые команды. Преобразованием пользовательского ввода в команды занимается класс ConsoleCommandReader, который учитывает текущие назначения клавиш управления, загруженные из конфигурационного файла.

Для управления игровым процессом был реализован шаблонный класс GameManager<InputProvider>. Он получает команды от переданного источника ввода, передаёт их в объект Game через метод applyCommand и затем вызывает обработку последствий хода. При этом GameManager не создаёт объект игры самостоятельно, а работает с уже существующим экземпляром, что упрощает изменение структуры программы и тестирование.

Визуализация игры вынесена в отдельный слой. Для этого был создан шаблонный класс GameVisualizer<RenderPolicy>, который использует переданный класс-рендерер для отображения состояния игры. В текущей реализации используется ConsoleRenderer, но при необходимости можно добавить другой способ отображения, не изменяя код класса Game.

Система настройки управления через конфигурационный файл сохранена. Если файл содержит ошибки, такие как дублирование клавиш или отсутствие обязательных команд, управление автоматически сбрасывается на значения по умолчанию. Это позволяет избежать некорректной работы программы из-за неправильных пользовательских настроек.

Класс Game был доработан для взаимодействия с новой архитектурой. В него добавлены методы applyCommand, handlePostPlayerAction и render, которые используются управляющими и визуализирующими компонентами. Благодаря этому игровая логика стала более изолированной и не зависит от конкретного способа ввода или вывода.

В результате выполненных изменений архитектура программы стала более модульной и гибкой. Добавление новых способов ввода или визуализации возможно без переписывания существующей логики, что делает проект удобным для дальнейшего развития и доработки.

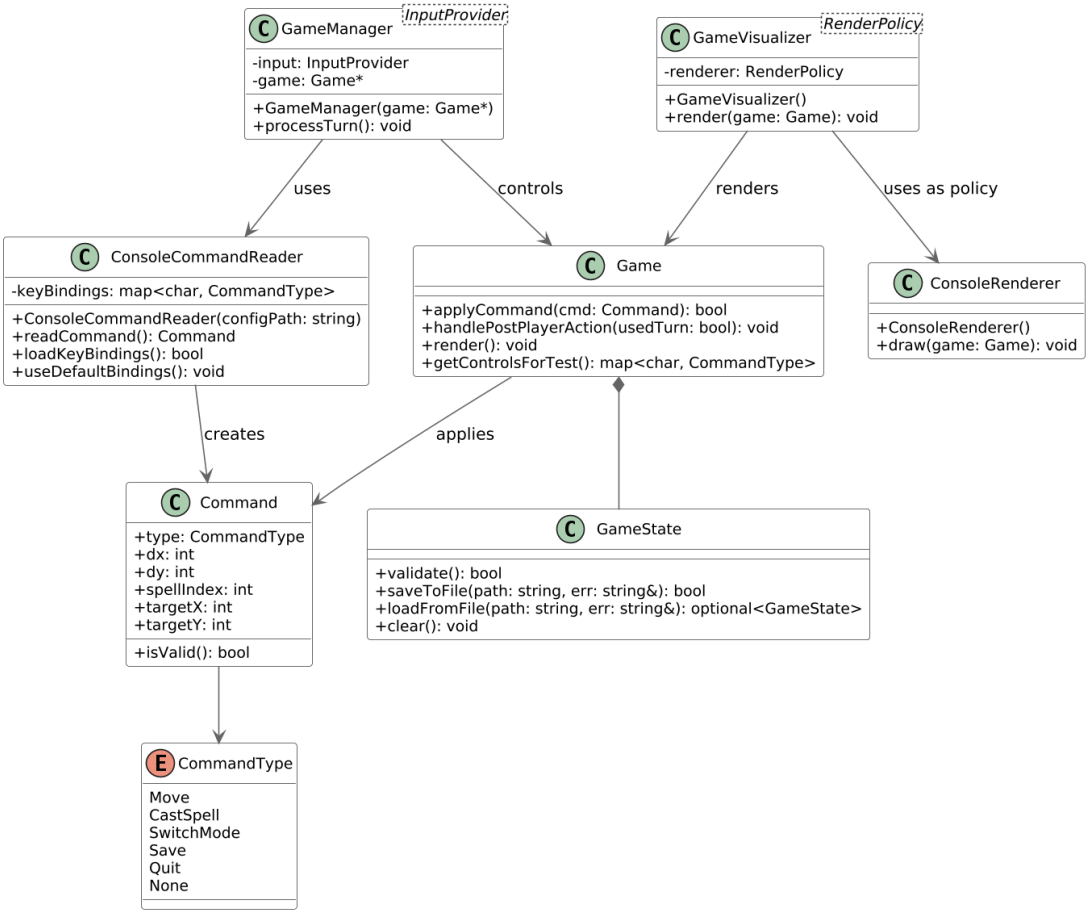

Рис. 1 UML-диаграмма добавленных классов.

Выводы.

В результате выполненной работы архитектура программы стала более простой и удобной для развития. Ввод пользователя, визуализация и игровая логика были разделены на отдельные части, которые слабо зависят друг от друга. Это позволяет изменять способ управления или отображения игры без необходимости переписывать основной код.

Использование объектов Command упростило передачу действий игрока в игровую логику и сделало обработку ввода более понятной. Применение шаблонных классов для управления игрой и визуализации позволило сделать архитектуру более гибкой и расширяемой.

В целом, полученная архитектура упрощает поддержку и доработку проекта. Добавление новых способов ввода или визуализации возможно без изменения существующей игровой логики, что делает программу более устойчивой к дальнейшим изменениям и удобной для развития.

ПРИЛОЖЕНИЕ А

ТЕСТИРОВАНИЕ

tests.cpp

```
#include <iostream>
#include <cassert>
#include <memory>
#include <cstdlib>
#include <windows.h>
#include <fstream>
#include <string>
#include <unordered_map>
#include <optional>

#include "Cell.h"
#include "Entity.h"
#include "Player.h"
#include "Enemy.h"
#include "Field.h"
#include "Hand.h"
#include "DirectDamageSpell.h"
#include "AreaDamageSpell.h"
#include "Game.h"
#include "GameState.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);
            }
        }
    }

    void writeControlsFile(const std::string& content) {
        std::ofstream out("controls.cfg", std::ios::trunc);
        out << content;
    }

       void writeFile(const std::string& path, const std::string&
content) {
        std::ofstream out(path, std::ios::trunc | std::ios::binary);
        out << content;
    }

    std::unordered_map<std::string, char> defaultControls() {
        return {
            {"MOVE_UP", 'w'},
            {"MOVE_DOWN", 's'},
            {"MOVE_LEFT", 'a'},
            {"MOVE_RIGHT", 'd'},
            {"SWITCH_MODE", 'm'},
            {"CAST_SPELL", 'c'},
            {"SAVE_GAME", 'v'},
            {"QUIT", 'q'}
        };
    }
```

```
}

// ===== 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);

    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;
}

void testControlsConfig() {
    std::cout << "Testing controls configuration..." << std::endl;

    // 1) Valid custom config should be applied
    writeControlsFile(
        "MOVE_UP=I\n"
        "MOVE_DOWN=K\n"
        "MOVE_LEFT=J\n"
        "MOVE_RIGHT=L\n"
        "SWITCH_MODE=U\n"
        "CAST_SPELL=O\n"
        "SAVE_GAME=P\n"
        "QUIT=Z\n"
    );
    {
        Game game;
        auto controls = game.getControlsForTest();
        assert(controls["MOVE_UP"] == 'I');
        assert(controls["MOVE_DOWN"] == 'K');
        assert(controls["MOVE_LEFT"] == 'J');
        assert(controls["MOVE_RIGHT"] == 'L');
        assert(controls["SWITCH_MODE"] == 'U');
        assert(controls["CAST_SPELL"] == 'O');
        assert(controls["SAVE_GAME"] == 'P');
        assert(controls["QUIT"] == 'Z');
    }

    // 2) Duplicate key should revert to defaults
    writeControlsFile(
        "MOVE_UP=W\n"
        "MOVE_DOWN=W\n"
        "MOVE_LEFT=A\n"
        "MOVE_RIGHT=D\n"
        "SWITCH_MODE=M\n"
```

```
            "CAST_SPELL=C\n"
            "SAVE_GAME=V\n"
            "QUIT=Q\n"
        );
        {
            Game game;
            auto controls = game.getControlsForTest();
            auto defaults = defaultControls();
            for (const auto& kv : defaults) {
                assert(controls[kv.first] == kv.second);
            }
        }

        // 3) Missing command should revert to defaults
        writeControlsFile(
            "MOVE_UP=W\n"
            "MOVE_DOWN=S\n"
            "MOVE_LEFT=A\n"
            "MOVE_RIGHT=D\n"
            "SWITCH_MODE=M\n"
            "CAST_SPELL=C\n"
            "SAVE_GAME=V\n"
            // QUIT is missing
        );
        {
            Game game;
            auto controls = game.getControlsForTest();
            auto defaults = defaultControls();
            for (const auto& kv : defaults) {
                assert(controls[kv.first] == kv.second);
            }
        }

        // Restore default controls file for future runs
        writeControlsFile(
            "MOVE_UP=W\n"
            "MOVE_DOWN=S\n"
            "MOVE_LEFT=A\n"
            "MOVE_RIGHT=D\n"
            "SWITCH_MODE=M\n"
            "CAST_SPELL=C\n"
            "SAVE_GAME=V\n"
            "QUIT=Q\n"
        );

        std::cout << "Controls configuration tests passed!" << std::endl;
    }

    void testGameStateSerialization() {
        std::cout << "Testing GameState serialization..." << std::endl;

        const std::string path = "test_save.dat";

        // Build state
        GameState state;
        state.setPlayer(3, 4, 75, 100, 50, 12, 7, 1);
        state.setHandMaxSize(3);
        state.addSpell({0, 2, 10});
```

```
state.addSpell({1, 3, 15});
state.setFieldSize(8, 9);
state.addWall(1, 1);
state.addWall(2, 2);
state.addEnemy(5, 5, 30, 6);
state.addEnemy(6, 6, 25, 5);
state.setProgress(2, 4, 1);

std::string error;
bool saved = state.saveToFile(path, error);
assert(saved && error.empty());

auto loaded = GameState::loadFromFile(path, error);
assert(loaded.has_value());
const GameState& s = *loaded;

assert(s.getPlayerX() == 3);
assert(s.getPlayerY() == 4);
assert(s.getPlayerHP() == 75);
assert(s.getPlayerMaxHP() == 100);
assert(s.getPlayerScore() == 50);
assert(s.getMeleeDamage() == 12);
assert(s.getRangedDamage() == 7);
assert(s.getCombatMode() == 1);
assert(s.getHandMaxSize() == 3);
assert(s.getSpells().size() == 2);
assert(s.getSpells()[0].type == 0);
assert(s.getSpells()[1].type == 1);
assert(s.getFieldWidth() == 8);
assert(s.getFieldHeight() == 9);
assert(s.getWalls().size() == 2);
assert(s.getEnemies().size() == 2);
assert(s.getEnemyHP()[0] == 30);
assert(s.getEnemyDamage()[1] == 5);
assert(s.getCurrentLevel() == 2);
assert(s.getInitialEnemyCount() == 4);
assert(s.getEnemiesKilled() == 1);

// Invalid header should fail
writeFile(path, "BAD_HEADER\n");
error.clear();
auto badHeader = GameState::loadFromFile(path, error);
assert(!badHeader.has_value());

// Invalid enemy count should fail
writeFile(path,
    "GAME_SAVE_V1\n"
    "PLAYER\n"
    "0 0 1 1 0 1 1 0\n"
    "HAND\n"
    "1 0\n"
    "FIELD\n"
    "5 5\n"
    "0\n"
    "ENEMIES\n"
    "-1\n"
    "GAME\n"
    "1 0 0\n"
```

```
        "END\n"
    );
    auto badEnemies = GameState::loadFromFile(path, error);
    assert(!badEnemies.has_value());

    // Clean up
    std::remove(path.c_str());

        std::cout << "GameState serialization 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();
        testControlsConfig();
        testGameStateSerialization();

        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;
}
```

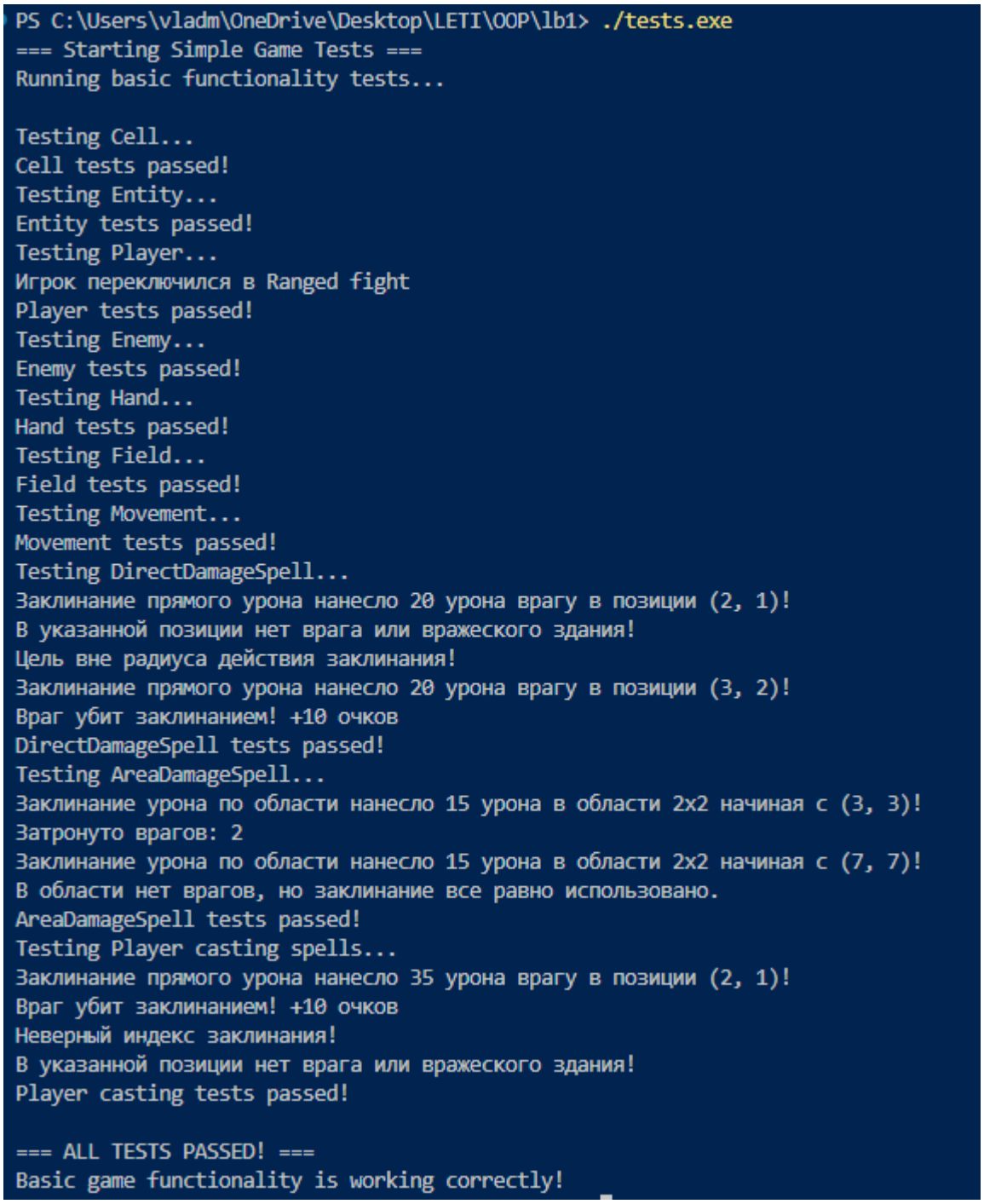

Рис. 2 Результаты тестирования.

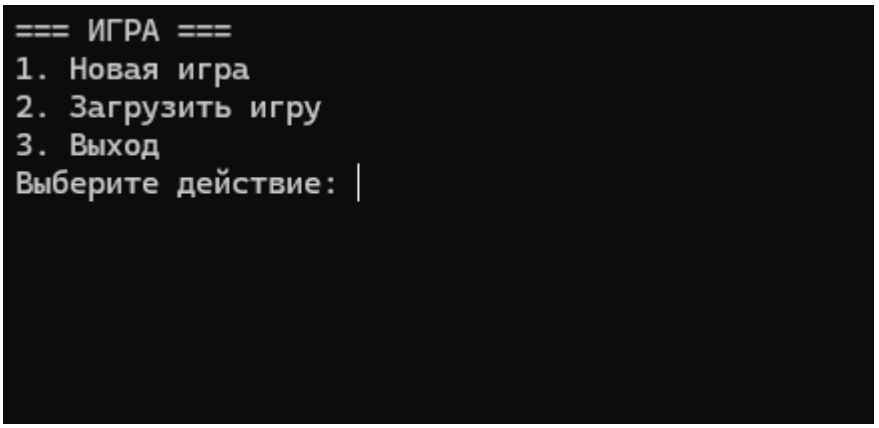

Рис. 3 Начало игры.

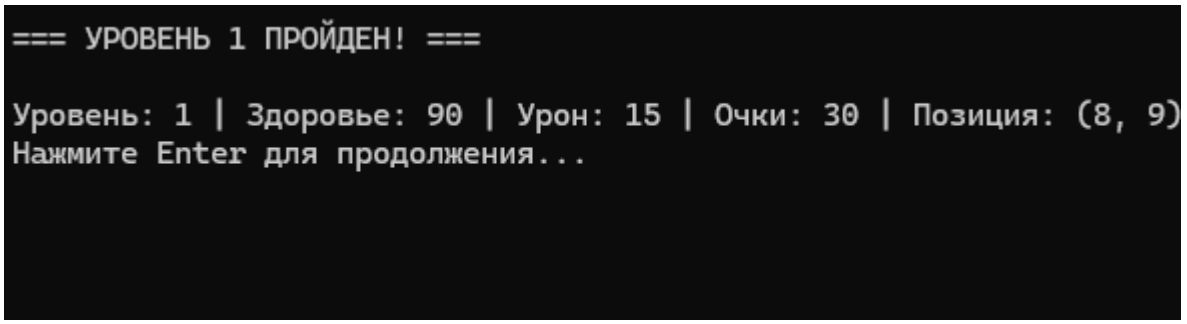

Рис. 5 Прохождение уровня.

ПРИЛОЖЕНИЕ В
ИСХОДНЫЙ КОД

Game.cpp

```
#include "Game.h"
#include "DirectDamageSpell.h"
#include "AreaDamageSpell.h"
#include "Spell.h"
#include <cstdlib>
#include <ctime>
#include <iostream>
#include <limits>
#include <algorithm>

const std::string Game::SAVE_FILE = "game_save.dat";

Game::Game() : field(10, 10), running(true), currentLevel(1),
                               initialEnemyCount(0),  enemiesKilled(0),
currentPhase(GamePhase::Menu) {
    SetConsoleOutputCP(CP_UTF8);
    SetConsoleCP(CP_UTF8);
    std::srand(static_cast<unsigned int>(std::time(nullptr)));
}

void Game::run() {
    while (running && currentPhase != GamePhase::Exit) {
        switch (currentPhase) {
            case GamePhase::Menu:
                processMenu();
                break;
            case GamePhase::GameStart:
                processGameStart();
                break;
            case GamePhase::LevelStart:
                processLevelStart();
                break;
            case GamePhase::PlayerTurn:
                processPlayerTurn();
                break;
            case GamePhase::AlliesTurn:
                processAlliesTurn();
                break;
            case GamePhase::EnemiesTurn:
                processEnemiesTurn();
                break;
            case GamePhase::EnemyBaseTurn:
                processEnemyBaseTurn();
                break;
            case GamePhase::LevelComplete:
                processLevelComplete();
                break;
            case GamePhase::GameOver:
                processGameOver();
                break;
            default:
                running = false;
                break;
```

```
        }
    }
}

void Game::processMenu() {
    system("cls");
    std::cout << "=== ИГРА ===" << std::endl;
    std::cout << "1. Новая игра" << std::endl;
    std::cout << "2. Загрузить игру" << std::endl;
    std::cout << "3. Выход" << std::endl;
    std::cout << "Выберите действие: ";

    int choice;
    std::cin >> choice;
    std::cin.ignore();

    switch (choice) {
        case 1:
            currentPhase = GamePhase::GameStart;
            break;
        case 2:
            if (loadGame()) {
                currentPhase = GamePhase::LevelStart;
            } else {
                  pause("Не удалось загрузить игру. Нажмите Enter для
продолжения...");
            }
            break;
        case 3:
            currentPhase = GamePhase::Exit;
            break;
        default:
                        pause("Неверный выбор. Нажмите Enter для
продолжения...");
            break;
    }
}

void Game::processGameStart() {
    initializeNewGame();
    currentPhase = GamePhase::LevelStart;
}

void Game::processLevelStart() {
    system("cls");
      std::cout << "=== УРОВЕНЬ " << currentLevel << " ===" <<
std::endl;
    pause("Нажмите Enter, чтобы начать уровень...");
    currentPhase = GamePhase::PlayerTurn;
}

void Game::processPlayerTurn() {
    if (!field.getPlayer() || !field.getPlayer()->isAlive()) {
        currentPhase = GamePhase::GameOver;
        return;
    }

    if (checkLevelComplete()) {
```

```
        currentPhase = GamePhase::LevelComplete;
        return;
    }

    displayGameScreen();

    std::cout << "Команды: W/A/S/D - движение, M - переключить режим,
C - заклинание, V - сохранить, Q - выход\n";
    std::cout << "Ваш ход: ";

    char command;
    std::cin >> command;
    std::cin.ignore();

    int dx = 0, dy = 0;
    bool turnUsed = false;

    switch (std::tolower(command)) {
        case 'w': dy = -1; break;
        case 's': dy = 1;  break;
        case 'a': dx = -1; break;
        case 'd': dx = 1;  break;
        case 'm':
            if (field.getPlayer()) {
                field.getPlayer()->switchCombatMode();
            }
            return; // Don't use turn for mode switch
        case 'c': {
            if (!field.getPlayer()) break;

            int spellIndex, targetX, targetY;
                   std::cout << "Введите индекс заклинания (0-" <<
(field.getPlayer()->getHand().getSpellCount() - 1) << "): ";
            std::cin >> spellIndex;
            std::cout << "Введите координату X цели: ";
            std::cin >> targetX;
            std::cout << "Введите координату Y цели: ";
            std::cin >> targetY;
            std::cin.ignore();

                if (field.getPlayer()->castSpell(spellIndex, targetX,
targetY, field)) {
                turnUsed = true;
                pause();
            } else {
                    pause("Заклинание не применено. Нажмите Enter для
продолжения...");
            }
            break;
        }
        case 'v': {
            if (saveGame()) {
                        pause("Игра сохранена! Нажмите Enter для
продолжения...");
            } else {
                      pause("Ошибка сохранения! Нажмите Enter для
продолжения...");
            }
```

```
            return; // Don't use turn for saving
        }
        case 'q':
            currentPhase = GamePhase::Menu;
            return;
        default:
                        pause("Неверная команда! Нажмите Enter для
продолжения...");
            return;
    }

    if ((dx != 0 || dy != 0) && field.getPlayer() && !turnUsed) {
        field.getPlayer()->move(dx, dy, field);
        turnUsed = true;
    }

    // Check if enemy was killed and give new spell
                                int        currentEnemyCount        =
static_cast<int>(field.getEnemies().size());
        int  newKills  =  initialEnemyCount  -  currentEnemyCount  -
enemiesKilled;
    if (newKills > 0) {
        enemiesKilled += newKills;
        tryGiveNewSpell();
    }

    if (turnUsed) {
        currentPhase = GamePhase::AlliesTurn;
    }
}

void Game::processAlliesTurn() {
    // Placeholder for future allies implementation
    // For now, just move to enemies turn
    currentPhase = GamePhase::EnemiesTurn;
}

void Game::processEnemiesTurn() {
    for (const auto& enemy : field.getEnemies()) {
        int dx = (std::rand() % 3) - 1; // -1, 0, 1
        int dy = (std::rand() % 3) - 1;

        if (dx != 0 || dy != 0) {
            enemy->move(dx, dy, field);
        }

        if (!field.getPlayer() || !field.getPlayer()->isAlive()) {
            currentPhase = GamePhase::GameOver;
            return;
        }
    }

    currentPhase = GamePhase::EnemyBaseTurn;
}

void Game::processEnemyBaseTurn() {
    // Placeholder for future enemy bases/towers implementation
    // For now, just move back to player turn
```

```
    currentPhase = GamePhase::PlayerTurn;
}

void Game::processLevelComplete() {
    system("cls");
    field.draw();
     std::cout << "\n=== УРОВЕНЬ " << currentLevel << " ПРОЙДЕН! ==="
<< std::endl;
    showStats();

    currentLevel++;
    pause("Нажмите Enter для продолжения...");

    // Initialize next level
    initializeLevel(currentLevel);
    currentPhase = GamePhase::LevelStart;
}

void Game::processGameOver() {
    system("cls");
    field.draw();
    std::cout << "\n=== ИГРА ОКОНЧЕНА ===" << std::endl;
    showStats();

    std::cout << "\n1. Начать заново" << std::endl;
    std::cout << "2. Выход в меню" << std::endl;
    std::cout << "Выберите действие: ";

    int choice;
    std::cin >> choice;
    std::cin.ignore();

    if (choice == 1) {
        currentPhase = GamePhase::GameStart;
    } else {
        currentPhase = GamePhase::Menu;
    }
}

void Game::initializeNewGame() {
    currentLevel = 1;
    enemiesKilled = 0;
    initializeLevel(currentLevel);
}

void Game::initializeLevel(int level) {
    // Clear field
    for (int y = 0; y < field.getHeight(); ++y) {
        for (int x = 0; x < field.getWidth(); ++x) {
            field.clearCell(x, y);
        }
    }

    // Initialize walls
    for (int i = 0; i < field.getHeight(); i++) {
        for (int j = 0; j < field.getWidth(); j++) {
            if (std::rand() % 10 == 0) {
```

```
                // Walls are handled by Field's initializeWalls, but
we need to clear first
            }
        }
    }

    // Create new field with walls
    field = Field(field.getWidth(), field.getHeight());

    // Place player
    auto player = std::make_unique<Player>(1, 1);
    field.setPlayer(std::move(player));

    // Place enemies (more enemies per level)
    int enemyCount = 3 + (level - 1) * 2;
    initialEnemyCount = enemyCount;
    int width = field.getWidth();
    int height = field.getHeight();

    for (int i = 0; i < enemyCount; i++) {
        int x, y;
        int attempts = 0;
        do {
            x = std::rand() % width;
            y = std::rand() % height;
            attempts++;
        } while ((!field.isEmpty(x, y) ||
                          (x == field.getPlayer()->getX() && y ==
field.getPlayer()->getY())) &&
                 attempts < 100);

        if (attempts < 100) {
            field.addEnemy(std::make_unique<Enemy>(x, y));
        }
    }

    enemiesKilled = 0;
}

bool Game::checkLevelComplete() const {
    return field.getEnemies().empty();
}

bool Game::checkGameOver() const {
    return !field.getPlayer() || !field.getPlayer()->isAlive();
}

void Game::tryGiveNewSpell() {
    if (!field.getPlayer()) return;

    Hand& hand = field.getPlayer()->getHand();
    if (hand.isFull()) {
           std::cout << "Рука заклинаний полна! Новое заклинание не
получено.\n";
        return;
    }

    std::unique_ptr<Spell> newSpell;
```

```
    int spellType = std::rand() % 2;

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

    if (hand.addSpell(std::move(newSpell))) {
        std::cout << "Вы получили новое заклинание: "
                          << hand.getSpell(hand.getSpellCount() -
1)->getName() << "!\n";
        pause();
    }
}

void Game::showStats() const {
    if (field.getPlayer()) {
        std::cout << "\nУровень: " << currentLevel
                  << " | Здоровье: " << field.getPlayer()->getHP()
                  << " | Урон: " << field.getPlayer()->getDamage()
                  << " | Очки: " << field.getPlayer()->getScore()
                   << " | Позиция: (" << field.getPlayer()->getX() <<
", " << field.getPlayer()->getY() << ")\n";
    }
}

void Game::showHand() const {
    if (field.getPlayer()) {
        const Hand& hand = field.getPlayer()->getHand();
                  std::cout  <<  "\n===  Рука  заклинаний  ("  <<
hand.getSpellCount()
                  << "/" << hand.getMaxSize() << ") ===\n";
        for (int i = 0; i < hand.getSpellCount(); i++) {
            Spell* spell = hand.getSpell(i);
            if (spell) {
                std::cout << "[" << i << "] " << spell->getName()
                          << " - " << spell->getDescription()
                          << " (Урон: " << spell->getDamage()
                             << ", Радиус: " << spell->getRadius() <<
")\n";
            }
        }
        std::cout << "========================\n";
    }
}

void Game::displayGameScreen() const {
    system("cls");
    field.draw();
    showStats();
    showHand();
}

void Game::pause(const std::string& message) const {
    std::cout << message;
    std::cin.ignore();
    std::cin.get();
```

```
}

bool Game::saveGame() const {
    GameState state = captureGameState();
    return state.saveToFile(SAVE_FILE);
}

bool Game::loadGame() {
    GameState state;
    if (!state.loadFromFile(SAVE_FILE)) {
        return false;
    }
    restoreGameState(state);
    return true;
}

GameState Game::captureGameState() const {
    GameState state;

    if (field.getPlayer()) {
        Player* player = field.getPlayer();
        state.playerX = player->getX();
        state.playerY = player->getY();
        state.playerHP = player->getHP();
        state.playerMaxHP = player->getMaxHP();
        state.playerScore = player->getScore();
        state.meleeDamage = player->getMeleeDamage();
        state.rangedDamage = player->getRangedDamage();
                    state.combatMode   =   (player->getCombatMode()   ==
CombatMode::Melee) ? 0 : 1;

        const Hand& hand = player->getHand();
        state.handMaxSize = hand.getMaxSize();
        state.spells.clear();
        for (int i = 0; i < hand.getSpellCount(); ++i) {
            Spell* spell = hand.getSpell(i);
            if (spell) {
                GameState::SpellData spellData;
                // Determine spell type by name or use RTTI
                std::string name = spell->getName();
                if (name == "Прямой урон") {
                    spellData.type = 0;
                } else {
                    spellData.type = 1;
                }
                spellData.radius = spell->getRadius();
                spellData.damage = spell->getDamage();
                state.spells.push_back(spellData);
            }
        }
    }

    state.fieldWidth = field.getWidth();
    state.fieldHeight = field.getHeight();

    // Save walls
    state.walls.clear();
    for (int y = 0; y < field.getHeight(); ++y) {
```

```
        for (int x = 0; x < field.getWidth(); ++x) {
            if (field.isWall(x, y)) {
                state.walls.push_back({x, y});
            }
        }
    }

    // Save enemies
    state.enemies.clear();
    state.enemyHP.clear();
    state.enemyDamage.clear();
    for (const auto& enemy : field.getEnemies()) {
        state.enemies.push_back({enemy->getX(), enemy->getY()});
        state.enemyHP.push_back(enemy->getHP());
        state.enemyDamage.push_back(enemy->getDamage());
    }

    state.currentLevel = currentLevel;
    state.initialEnemyCount = initialEnemyCount;
    state.enemiesKilled = enemiesKilled;

    return state;
}

void Game::restoreGameState(const GameState& state) {
    currentLevel = state.currentLevel;
    initialEnemyCount = state.initialEnemyCount;
    enemiesKilled = state.enemiesKilled;

    // Recreate field without walls first
    field = Field(state.fieldWidth, state.fieldHeight);

    // Clear all cells first
    for (int y = 0; y < field.getHeight(); ++y) {
        for (int x = 0; x < field.getWidth(); ++x) {
            field.clearCell(x, y);
        }
    }

    // Restore walls
    for (const auto& wall : state.walls) {
        field.setWall(wall.first, wall.second);
    }

    // Restore player
           auto   player   =   std::make_unique<Player>(state.playerX,
state.playerY,
                                                        state.playerMaxHP,
state.meleeDamage, state.rangedDamage);
    player->setPosition(state.playerX, state.playerY);
    player->setHP(state.playerHP);
    player->setScore(state.playerScore);

        if  (state.combatMode  ==  1  &&  player->getCombatMode()  ==
CombatMode::Melee) {
        player->switchCombatMode();
    }
```

```
    // Restore hand
    Hand& hand = player->getHand();
    // Clear existing spells and add saved ones
    while (hand.getSpellCount() > 0) {
        hand.removeSpell(0);
    }

    for (const auto& spellData : state.spells) {
        std::unique_ptr<Spell> spell;
        if (spellData.type == 0) {
                                                                spell     =
std::make_unique<DirectDamageSpell>(spellData.radius,
spellData.damage);
        } else {
                                                                spell     =
std::make_unique<AreaDamageSpell>(spellData.radius,
spellData.damage);
        }
        hand.addSpell(std::move(spell));
    }

    field.setPlayer(std::move(player));

    // Restore enemies
    for (size_t i = 0; i < state.enemies.size(); ++i) {
         auto enemy = std::make_unique<Enemy>(state.enemies[i].first,
state.enemies[i].second,
                                                        state.enemyHP[i],
state.enemyDamage[i]);
        field.addEnemy(std::move(enemy));
    }
}
```

Game.h

```
#pragma once
#include <iostream>
#include <windows.h>
#include <cstdlib>
#include <ctime>
#include <memory>
#include <string>

#include "Field.h"
#include "Player.h"
#include "Enemy.h"
#include "GameState.h"

enum class GamePhase {
    Menu,
    GameStart,
    LevelStart,
    PlayerTurn,
    EnemiesTurn,
    AlliesTurn, //d
```

```
    EnemyBaseTurn, //d
    LevelComplete,
    GameOver,
    Exit
};

class Game {
private:
    Field field;
    bool running;
    int currentLevel;
    int initialEnemyCount;
    int enemiesKilled;
    GamePhase currentPhase;
    static const std::string SAVE_FILE;

    // Game loop phases
    void processMenu();
    void processGameStart();
    void processLevelStart();
    void processPlayerTurn();
    void processAlliesTurn(); // Placeholder for future allies
    void processEnemiesTurn();
        void processEnemyBaseTurn(); // Placeholder for future
bases/towers
    void processLevelComplete();
    void processGameOver();

    // Game state management
    void initializeNewGame();
    void initializeLevel(int level);
    bool checkLevelComplete() const;
    bool checkGameOver() const;
    void tryGiveNewSpell();

    // UI
    void showStats() const;
    void showHand() const;
    void displayGameScreen() const;

    // Save/Load
    bool saveGame() const;
    bool loadGame();
    GameState captureGameState() const;
    void restoreGameState(const GameState& state);

    // Helper methods
        void pause(const std::string& message = "Press Enter to
continue...") const;

public:
    Game();
    ~Game() = default;

    Game(const Game& other) = delete;
    Game& operator=(const Game& other) = delete;
    Game(Game&& other) = delete;
    Game& operator=(Game&& other) = delete;
```

```
    void run();
};
```

GameState.cpp

```
#include "GameState.h"
#include <fstream>
#include <sstream>

namespace {
constexpr const char* HEADER = "GAME_SAVE_V1";
}

GameState::GameState() {
    clear();
}

void GameState::setPlayer(int x, int y, int hp, int maxHp, int score,
int meleeDmg, int rangedDmg, int mode) {
    playerX = x;
    playerY = y;
    playerHP = hp;
    playerMaxHP = maxHp;
    playerScore = score;
    meleeDamage = meleeDmg;
    rangedDamage = rangedDmg;
    combatMode = mode;
}

void GameState::setHandMaxSize(int size) {
    handMaxSize = size;
}

void GameState::addSpell(const SpellData& spell) {
    spells.push_back(spell);
}

void GameState::setFieldSize(int width, int height) {
    fieldWidth = width;
    fieldHeight = height;
}

void GameState::addWall(int x, int y) {
    walls.emplace_back(x, y);
}

void GameState::addEnemy(int x, int y, int hp, int damage) {
    enemies.emplace_back(x, y);
    enemyHP.push_back(hp);
    enemyDamage.push_back(damage);
}

void  GameState::setProgress(int  level,  int  initialEnemies,  int
killed) {
```

```
    currentLevel = level;
    initialEnemyCount = initialEnemies;
    enemiesKilled = killed;
}

int GameState::getPlayerX() const { return playerX; }
int GameState::getPlayerY() const { return playerY; }
int GameState::getPlayerHP() const { return playerHP; }
int GameState::getPlayerMaxHP() const { return playerMaxHP; }
int GameState::getPlayerScore() const { return playerScore; }
int GameState::getMeleeDamage() const { return meleeDamage; }
int GameState::getRangedDamage() const { return rangedDamage; }
int GameState::getCombatMode() const { return combatMode; }
int GameState::getHandMaxSize() const { return handMaxSize; }
const std::vector<GameState::SpellData>& GameState::getSpells() const
{ return spells; }
int GameState::getFieldWidth() const { return fieldWidth; }
int GameState::getFieldHeight() const { return fieldHeight; }
const std::vector<std::pair<int, int>>& GameState::getWalls() const {
return walls; }
const std::vector<std::pair<int, int>>& GameState::getEnemies() const
{ return enemies; }
const  std::vector<int>&  GameState::getEnemyHP()  const  {  return
enemyHP; }
const  std::vector<int>&  GameState::getEnemyDamage()  const  {  return
enemyDamage; }
int GameState::getCurrentLevel() const { return currentLevel; }
int     GameState::getInitialEnemyCount()     const     {     return
initialEnemyCount; }
int GameState::getEnemiesKilled() const { return enemiesKilled; }

bool GameState::saveToFile(const std::string& filename, std::string&
error) const {
    std::ofstream file(filename, std::ios::binary);
    if (!file.is_open()) {
        error = "Не удалось открыть файл для записи: " + filename;
        return false;
    }

    file << HEADER << "\n";

    file << "PLAYER\n";
     file << playerX << " " << playerY << " " << playerHP << " " <<
playerMaxHP << " "
         << playerScore << " " << meleeDamage << " " << rangedDamage
<< " " << combatMode << "\n";

    file << "HAND\n";
    file << handMaxSize << " " << spells.size() << "\n";
    for (const auto& spell : spells) {
            file  <<  spell.type  <<  "  "  <<  spell.radius  <<  "  "  <<
spell.damage << "\n";
    }

    file << "FIELD\n";
    file << fieldWidth << " " << fieldHeight << "\n";
    file << walls.size() << "\n";
    for (const auto& wall : walls) {
```

```
        file << wall.first << " " << wall.second << "\n";
    }

    file << "ENEMIES\n";
    file << enemies.size() << "\n";
    for (size_t i = 0; i < enemies.size(); ++i) {
        file << enemies[i].first << " " << enemies[i].second << " "
             << enemyHP[i] << " " << enemyDamage[i] << "\n";
    }

    file << "GAME\n";
      file << currentLevel << " " << initialEnemyCount << " " <<
enemiesKilled << "\n";
    file << "END\n";

    if (!file || file.fail()) {
        error = "Ошибка записи в файл: " + filename;
        return false;
    }

    return true;
}

std::optional<GameState>  GameState::loadFromFile(const  std::string&
filename, std::string& error) {
    std::ifstream file(filename, std::ios::binary);
    if (!file.is_open()) {
        error = "Не удалось открыть файл для чтения: " + filename;
        return std::nullopt;
    }

    GameState state;

    std::string line;
    std::getline(file, line);
    if (line != HEADER) {
        error = "Неверный формат файла сохранения";
        return std::nullopt;
    }

    // PLAYER
    std::getline(file, line);
     if (line != "PLAYER") { error = "Ожидался раздел PLAYER"; return
std::nullopt; }
    int px, py, php, pmax, pscore, pmd, prd, mode;
     if (!(file >> px >> py >> php >> pmax >> pscore >> pmd >> prd >>
mode)) {
        error = "Ошибка чтения PLAYER";
        return std::nullopt;
    }
    file.ignore();
    state.setPlayer(px, py, php, pmax, pscore, pmd, prd, mode);

    // HAND
    std::getline(file, line);
      if (line != "HAND") { error = "Ожидался раздел HAND"; return
std::nullopt; }
    int handSize, spellCount;
```

```
     if (!(file >> handSize >> spellCount)) { error = "Ошибка чтения
HAND"; return std::nullopt; }
     if (spellCount < 0 || handSize <= 0) { error = "Некорректные
параметры HAND"; return std::nullopt; }
    file.ignore();
    state.setHandMaxSize(handSize);
    for (int i = 0; i < spellCount; ++i) {
        GameState::SpellData s;
        if (!(file >> s.type >> s.radius >> s.damage)) {
            error = "Ошибка чтения заклинаний";
            return std::nullopt;
        }
        file.ignore();
        state.addSpell(s);
    }

    // FIELD
    std::getline(file, line);
     if (line != "FIELD") { error = "Ожидался раздел FIELD"; return
std::nullopt; }
    int w, h;
     if (!(file >> w >> h)) { error = "Ошибка чтения FIELD"; return
std::nullopt; }
     if (w <= 0 || h <= 0) { error = "Некорректные размеры поля";
return std::nullopt; }
    state.setFieldSize(w, h);
    file.ignore();
    int wallCount;
    if (!(file >> wallCount)) { error = "Ошибка чтения WALLS"; return
std::nullopt; }
     if (wallCount < 0) { error = "Некорректное число стен"; return
std::nullopt; }
    file.ignore();
    for (int i = 0; i < wallCount; ++i) {
        int wx, wy;
           if (!(file >> wx >> wy)) { error = "Ошибка чтения стен";
return std::nullopt; }
        file.ignore();
        state.addWall(wx, wy);
    }

    // ENEMIES
    std::getline(file, line);
      if (line != "ENEMIES") { error = "Ожидался раздел ENEMIES";
return std::nullopt; }
    int enemyCount;
     if (!(file >> enemyCount)) { error = "Ошибка чтения ENEMIES";
return std::nullopt; }
    if (enemyCount < 0) { error = "Некорректное число врагов"; return
std::nullopt; }
    file.ignore();
    for (int i = 0; i < enemyCount; ++i) {
        int ex, ey, ehp, edmg;
          if (!(file >> ex >> ey >> ehp >> edmg)) { error = "Ошибка
чтения врагов"; return std::nullopt; }
        file.ignore();
        state.addEnemy(ex, ey, ehp, edmg);
    }
```

```
    // GAME
    std::getline(file, line);
      if (line != "GAME") { error = "Ожидался раздел GAME"; return
std::nullopt; }
    int lvl, initEnemies, killed;
     if (!(file >> lvl >> initEnemies >> killed)) { error = "Ошибка
чтения GAME"; return std::nullopt; }
    state.setProgress(lvl, initEnemies, killed);
    file.ignore();

    std::getline(file, line);
     if (line != "END") { error = "Файл сохранения поврежден"; return
std::nullopt; }

    if (!state.validate(error)) {
        return std::nullopt;
    }

    return state;
}

bool GameState::validate(std::string& error) const {
     if (fieldWidth <= 0 || fieldHeight <= 0) { error = "Размеры поля
некорректны"; return false; }
    if (handMaxSize <= 0) { error = "Размер руки некорректен"; return
false; }
     if (spells.size() > static_cast<size_t>(handMaxSize)) { error =
"Слишком много заклинаний"; return false; }
       if (enemies.size() != enemyHP.size() || enemies.size() !=
enemyDamage.size()) {
        error = "Несогласованные данные врагов"; return false; }
    return true;
}

void GameState::clear() {
    playerX = playerY = 0;
    playerHP = playerMaxHP = 100;
    playerScore = 0;
    meleeDamage = 15;
    rangedDamage = 5;
    combatMode = 0;
    handMaxSize = 5;
    spells.clear();
    fieldWidth = fieldHeight = 10;
    walls.clear();
    enemies.clear();
    enemyHP.clear();
    enemyDamage.clear();
    currentLevel = 1;
    initialEnemyCount = 0;
    enemiesKilled = 0;
}
```

GameState.h

```
#pragma once
#include <iostream>
#include <windows.h>
#include <cstdlib>
#include <ctime>
#include <memory>
#include <string>
#include <unordered_map>

#include "Command.h"
#include "Field.h"
#include "Player.h"
#include "Enemy.h"
#include "GameState.h"

enum class GamePhase {
    Menu,
    GameStart,
    LevelStart,
    PlayerTurn,
    EnemiesTurn,
    AlliesTurn, //d
    EnemyBaseTurn, //d
    LevelComplete,
    GameOver,
    Exit
};

class Game {
private:
    struct ControlSettings {
        char moveUp{'w'};
        char moveDown{'s'};
        char moveLeft{'a'};
        char moveRight{'d'};
        char switchMode{'m'};
        char castSpell{'c'};
        char saveGame{'v'};
        char quitToMenu{'q'};
    };

    Field field;
    bool running;
    int currentLevel;
    int initialEnemyCount;
    int enemiesKilled;
    GamePhase currentPhase;
    static const std::string SAVE_FILE;
    static const std::string CONTROLS_FILE;
    ControlSettings controls;

    // Game loop phases
    void processMenu();
    void processGameStart();
```

```
    void processLevelStart();
    void processPlayerTurn();
    void processAlliesTurn(); // Placeholder for future allies
    void processEnemiesTurn();
         void  processEnemyBaseTurn();  //  Placeholder  for  future
bases/towers
    void processLevelComplete();
    void processGameOver();

    // Game state management
    void initializeNewGame();
    void initializeLevel(int level);
    bool checkLevelComplete() const;
    bool checkGameOver() const;
    void tryGiveNewSpell();

    // UI
    void showStats() const;
    void showHand() const;
    void displayGameScreen() const;

    // Save/Load
    bool saveGame() const;
    bool loadGame();
    GameState captureGameState() const;
    void restoreGameState(const GameState& state);

    // Helper methods
        void  pause(const  std::string&  message  =  "Press  Enter  to
continue...") const;
    void setDefaultControls();
    void loadControls();
    bool parseControlsFile(ControlSettings& outSettings, std::string&
error) const;
          bool  validateControls(const  ControlSettings&  settings,
std::string& error) const;

public:
    Game();
    ~Game() = default;

    Game(const Game& other) = delete;
    Game& operator=(const Game& other) = delete;
    Game(Game&& other) = delete;
    Game& operator=(Game&& other) = delete;

    void run();
    std::unordered_map<std::string, char> getControlsForTest() const;
    bool applyCommand(const Command& cmd);
    void handlePostPlayerAction(bool turnUsed);
    void render() const;
};
```

Command.h

```
#pragma once
#pragma once
```

```
enum class CommandType {
    None,
    Move,
    SwitchMode,
    CastSpell,
    SaveGame,
    QuitToMenu,
    Invalid
};

struct Command {
    CommandType type{CommandType::None};
    int dx{0};
    int dy{0};
    int spellIndex{-1};
    int targetX{0};
    int targetY{0};
};
```

InputReader.h

```
#pragma once

#include <iostream>
#include <cctype>
#include <unordered_map>
#include "Command.h"
#include "Game.h"

class ConsoleCommandReader {
public:
    Command readCommand(const Game& game) const {
        Command cmd;

        const auto controls = game.getControlsForTest();
        auto showKey = [](char key) {
                                                                return
static_cast<char>(std::toupper(static_cast<unsigned char>(key)));
        };

        std::cout << "Команды: "
                        << showKey(controls.at("MOVE_UP")) << "/" <<
showKey(controls.at("MOVE_LEFT")) << "/"
                      << showKey(controls.at("MOVE_DOWN")) << "/" <<
showKey(controls.at("MOVE_RIGHT"))
                                            << " - движение, " <<
showKey(controls.at("SWITCH_MODE")) << " - переключить режим, "
                        << showKey(controls.at("CAST_SPELL")) << " -
заклинание, "
                         << showKey(controls.at("SAVE_GAME")) << " -
сохранить, "
                  << showKey(controls.at("QUIT")) << " - выход\n";
        std::cout << "Ваш ход: ";

        char inputChar;
        std::cin >> inputChar;
        std::cin.ignore();
```

```
        auto matches = [&](const std::string& keyName) {
                            return  std::tolower(static_cast<unsigned
char>(inputChar)) ==
                                    std::tolower(static_cast<unsigned
char>(controls.at(keyName)));
        };

        if (matches("MOVE_UP")) {
            cmd.type = CommandType::Move;
            cmd.dy = -1;
        } else if (matches("MOVE_DOWN")) {
            cmd.type = CommandType::Move;
            cmd.dy = 1;
        } else if (matches("MOVE_LEFT")) {
            cmd.type = CommandType::Move;
            cmd.dx = -1;
        } else if (matches("MOVE_RIGHT")) {
            cmd.type = CommandType::Move;
            cmd.dx = 1;
        } else if (matches("SWITCH_MODE")) {
            cmd.type = CommandType::SwitchMode;
        } else if (matches("CAST_SPELL")) {
            cmd.type = CommandType::CastSpell;
            std::cout << "Введите индекс заклинания: ";
            std::cin >> cmd.spellIndex;
            std::cout << "Введите координату X цели: ";
            std::cin >> cmd.targetX;
            std::cout << "Введите координату Y цели: ";
            std::cin >> cmd.targetY;
            std::cin.ignore();
        } else if (matches("SAVE_GAME")) {
            cmd.type = CommandType::SaveGame;
        } else if (matches("QUIT")) {
            cmd.type = CommandType::QuitToMenu;
        } else {
            cmd.type = CommandType::Invalid;
        }

        return cmd;
    }
};
```

Visualizer.h

```
#pragma once

#include "Game.h"

template <typename RenderPolicy>
class GameVisualizer {
public:
    explicit GameVisualizer(Game& gameRef) : game(gameRef) {}

    void refresh() {
        renderer.draw(game);
    }
```

```
private:
    Game& game;
    RenderPolicy renderer;
};
```