

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

**ОТЧЕТ**  
**по лабораторной работе №3**  
**по дисциплине «Объектно-ориентированное программирование»**

Студентка гр. 4384

Калинина А.В.

Преподаватель

Жангиров Т.Р.

Санкт-Петербург

### **Цель работы.**

Изучить принципы объектно-ориентированного программирования. Разработать объектно-ориентированную программу на C++, реализующую пошаговую игру, с системой управления игровым процессом с полным циклом игры. Реализовать систему сохранения и загрузки игрового состояния с обработкой исключительных ситуаций. Добавить главное меню с выбором новой игры или загрузки сохранения. Обеспечить возможность сохранения в процессе игры и корректную обработку завершения игрового сеанса.

### **Задание.**

На 6/3/1 баллов:

1. Создать класс(ы) игры, который реализует основной цикл игры, и которому передаются команды от пользователя. Игровой цикл состоит из следующих шагов:

- a. Начало игры
- b. Запуск уровня
- c. Ход игрока. Ход, атака или применение заклинания.
- d. Ход союзников - если имеются
- e. Ход врагов
- f. Ход вражеской базы и башни - если имеются

Условие прохождения уровня студент определяет самостоятельно. Если игрок проигрывает, то игроку должно предлагаться начать заново игру, либо выйти из программы.

Все взаимодействие должно происходить через классы игры.

2. Реализовать систему сохранения и загрузки игры. Пользователь должен иметь возможность сохранить игру в любой момент. Пользователь должен иметь возможность загружаться при запуске программы (или выбрать новую), либо во время игры. Сохранения должны оставаться в консистентном состоянии между запусками игры.

3. Добавить обработку исключительных ситуаций для загрузки/сохранения, например, невозможность записать в файл, нельзя загрузиться так как файл не существует или в нем некорректные данные.

### **Выполнение работы.**

В ходе выполнения лабораторной работы №3 была реализована система сохранения и загрузки игрового состояния с полной интеграцией в существующую архитектуру игры.

### **Архитектура программы.**

В программе реализована иерархия классов, соответствующая принципам ООП.

#### **Описание классов**

##### **• Класс GameManager**

**Назначение:** класс GameManager является главным управляющим классом, который координирует весь игровой процесс. Он отвечает за запуск игры, отображение главного меню, управление игровым циклом и обработку завершения игры.

#### **Поля класса:**

- ❖ gameState (GameState) - состояние игры
- ❖ gameRules (GameRules) - правила игры
- ❖ turnManager (TurnManager) - менеджер очередности ходов
- ❖ nProcessor (ActionProcessor) - обработчик действий
- ❖ shopSystem (ShopSystem) - система магазина
- ❖ rewardSystem (RewardSystem) - система наград
- ❖ gameLogic (GameLogic) - игровая логика
- ❖ inputHandler (InputHandler) - обработчик ввода
- ❖ renderSystem (RenderSystem) - система отображения
- ❖ saveSystem (GameSaveSystem) - система сохранения игры
- ❖ gameRunning (bool) - флаг работы игры

## **Методы класса:**

### **Публичные методы:**

- GameManager() - конструктор, инициализирует все системы
- run() - запуск менеджера и главного цикла программы

### **Приватные методы:**

- showMainMenu() - отображение главного меню
- handleSaveLoad() - обработка загрузки сохранения
- runGameLoop() - запуск основного игрового цикла
- handleGameOver() - обработка завершения игры (поражение)
- handleGameEnd() - обработка обычного завершения игры

## **Вход в игру:**

При запуске программы создается объект класса GameManager

Вызывается метод run(), который отображает главное меню

Пользователь выбирает одну из трех опций:

- инициализируется новая игра через GameInitializer
- загружается сохранение через GameSaveSystem
- завершение программы

После выбора "New Game" или "Load Game" запускается игровой цикл runGameLoop()

### **• Класс GameSaveSystem**

**Назначение:** класс GameSaveSystem предназначен для управления операциями сохранения и загрузки игрового состояния. Он инкапсулирует всю логику работы с файловой системой, обеспечивает целостность данных и обрабатывает исключительные ситуации, возникающие при работе с файлами сохранения.

### **Поля класса:**

**Методы класса:** save.dat" - файл сохранения по умолчанию

Сохранение игры:

- saveGame(gameState) - сохранить в файл по умолчанию

- saveGame(gameState, filename) - сохранить в указанный файл

Загрузка игры:

- loadGame(gameState) - загрузить из файла по умолчанию
- loadGame(gameState, filename) - загрузить из указанного файла

Проверка сохранений:

- saveExists() - проверить существование файла по умолчанию
- saveExists(filename) - проверить существование указанного файла

Исключения:

- SaveLoadException - базовая ошибка сохранения
- FileNotFoundException - файл не найден
- FileWriteException - ошибка записи
- CorruptedSaveException - поврежденный файл

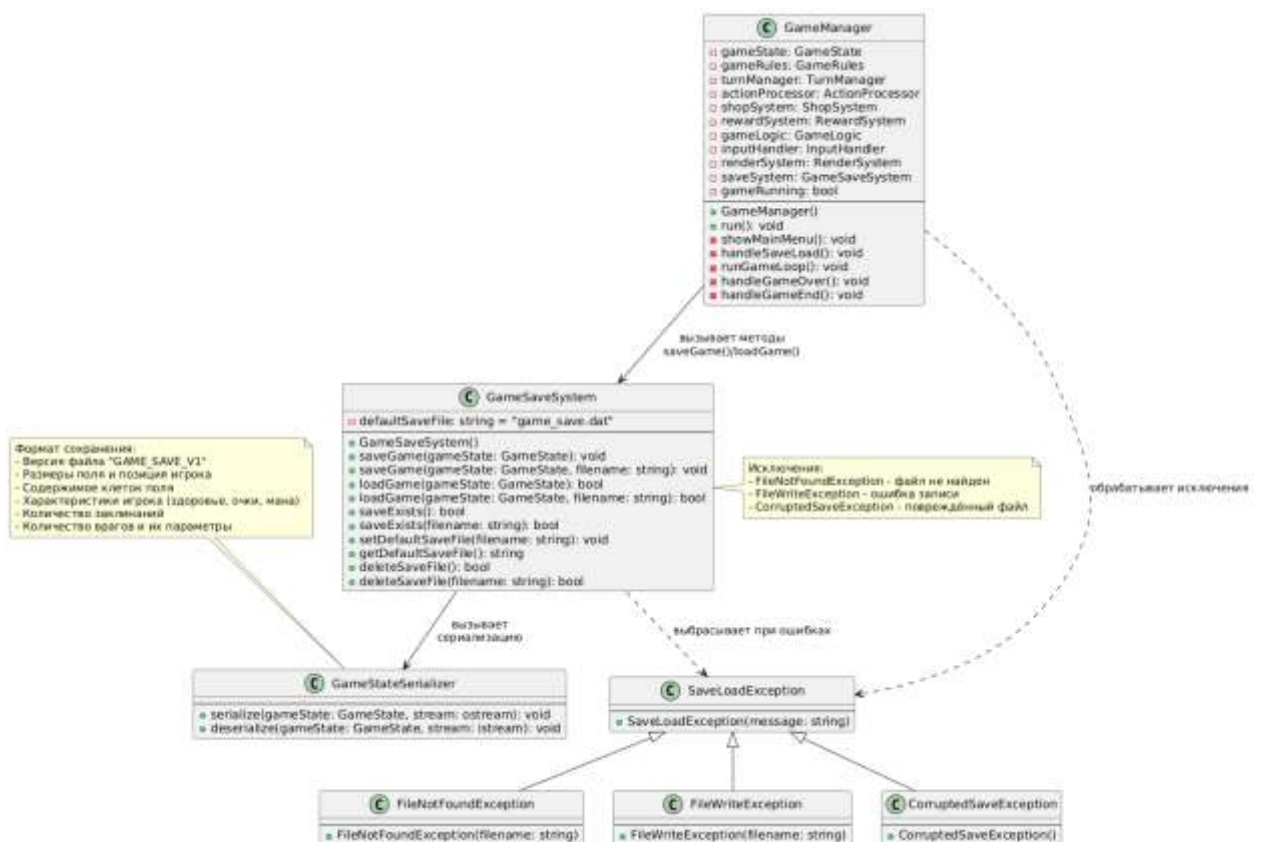


Рисунок 1 - Диаграмма UML-классов

Разработанный программный код см. в приложении А.

### **Выводы.**

В ходе выполнения лабораторной работы №3 была успешно реализована и интегрирована в существующую архитектуру игры система сохранения и загрузки игрового состояния. Была объявлена иерархия исключений для обработки различных сценариев ошибок при работе с файлами. Модифицированы существующие классы игры для поддержки сериализации и десериализации данных. Реализовано главное меню с возможностью выбора между новой игрой и загрузкой существующего сохранения.

## ПРИЛОЖЕНИЕ А

### ИСХОДНЫЙ КОД ПРОГРАММЫ

#### **Cell.cpp**

```
#include "Cell.h"

Cell::Cell() : content('.') {
}

char Cell::getContent() const {
    return content;
}

void Cell::setContent(char newContent) {
    content = newContent;
}

bool Cell::isEmpty() const {
    return content == '.';
}

void Cell::clear() {
    content = '.';
}
```

#### **Cell.h**

```
#ifndef CELL_H
#define CELL_H

class Cell {
private:
    char content;

public:
    Cell();
    char getContent() const;
    void setContent(char newContent);
    bool isEmpty() const;
    void clear();
};

#endif
```

#### **Enemy.cpp**

```
#include "Enemy.h"

Enemy::Enemy(int startX, int startY)
    : Entity(30, 5), x(startX), y(startY) {
}

int Enemy::getX() const {
    return x;
}

int Enemy::getY() const {
    return y;
}

void Enemy::move(int newX, int newY) {
```

```

        x = newX;
        y = newY;
    }

    // НОВЫЙ МЕТОД для сериализации
    void Enemy::setHealth(int newHealth) {
        health = newHealth;
        if (health < 0) health = 0;
    }
Enemy.h
#ifndef ENEMY_H
#define ENEMY_H

#include "Entity.h"

class Enemy : public Entity {
private:
    int x, y;

public:
    Enemy(int startX, int startY);
    int getX() const;
    int getY() const;
    void move(int newX, int newY);

    // НОВЫЙ МЕТОД для сериализации
    void setHealth(int newHealth);
};

#endif

Entity.cpp
#include "Entity.h"

Entity::Entity(int startHealth, int startDamage)
    : health(startHealth), damage(startDamage) {
}

int Entity::getHealth() const {
    return health;
}

int Entity::getDamage() const {
    return damage;
}

void Entity::takeDamage(int amount) {
    health -= amount;
    if (health < 0) {
        health = 0;
    }
}

bool Entity::isAlive() const {
    return health > 0;
}

```

## **Entity.h**

```
#ifndef ENTITY_H
#define ENTITY_H

class Entity {
protected:
    int health;
    int damage;

public:
    Entity(int startHealth, int startDamage);
    virtual ~Entity() = default;

    int getHealth() const;
    int getDamage() const;
    void takeDamage(int amount);
    bool isAlive() const;
};

#endif
```

## **GameField.cpp**

```
#include "GameField.h"
#include <stdexcept>
#include <utility>

GameField::GameField(int w, int h) : width(w), height(h), playerX(0),
playerY(0) {
    if (w < 10 || w > 25 || h < 10 || h > 25) {
        throw std::invalid_argument("Field size must be between 10x10
and 25x25");
    }
    grid.resize(height);
    for (int i = 0; i < height; i++) {
        grid[i].resize(width);
    }
}

GameField::~GameField() {}

GameField::GameField(const GameField& other)
    : width(other.width), height(other.height),
    playerX(other.playerX), playerY(other.playerY), grid(other.grid) {}

GameField::GameField(GameField&& other) noexcept
    : width(other.width), height(other.height),
    playerX(other.playerX), playerY(other.playerY),
    grid(std::move(other.grid)) {
    other.width = 0;
    other.height = 0;
    other.playerX = 0;
    other.playerY = 0;
}

GameField& GameField::operator=(const GameField& other) {
    if (this != &other) {
```

```

        GameField temp(other);
        swap(*this, temp);
    }
    return *this;
}

GameField& GameField::operator=(GameField&& other) noexcept {
    if (this != &other) {
        width = other.width;
        height = other.height;
        playerX = other.playerX;
        playerY = other.playerY;
        grid = std::move(other.grid);
        other.width = 0;
        other.height = 0;
        other.playerX = 0;
        other.playerY = 0;
    }
    return *this;
}

void swap(GameField& first, GameField& second) noexcept {
    using std::swap;
    swap(first.width, second.width);
    swap(first.height, second.height);
    swap(first.playerX, second.playerX);
    swap(first.playerY, second.playerY);
    swap(first.grid, second.grid);
}

int GameField::getWidth() const {
    return width;
}

int GameField::getHeight() const {
    return height;
}

void GameField::setPlayerPosition(int x, int y) {
    if (isValidPosition(x, y)) {
        playerX = x;
        playerY = y;
    }
}

std::pair<int, int> GameField::getPlayerPosition() const {
    return { playerX, playerY };
}

bool GameField::movePlayer(int newX, int newY) {
    if (!isValidPosition(newX, newY)) {
        return false;
    }
    if (!isCellEmpty(newX, newY)) {
        return false;
    }
    playerX = newX;
    playerY = newY;
}

```

```

        return true;
    }

    char GameField::getCellContent(int x, int y) const {
        if (!isValidPosition(x, y)) return '.';
        return grid[y][x].getContent();
    }

    void GameField::setCellContent(int x, int y, char content) {
        if (isValidPosition(x, y)) {
            grid[y][x].setContent(content);
        }
    }

    bool GameField::isCellEmpty(int x, int y) const {
        if (!isValidPosition(x, y)) return true;
        return grid[y][x].isEmpty();
    }

    void GameField::clearCell(int x, int y) {
        if (isValidPosition(x, y)) {
            grid[y][x].clear();
        }
    }

    void GameField::clearAllCells() {
        for (int y = 0; y < height; y++) {
            for (int x = 0; x < width; x++) {
                grid[y][x].clear();
            }
        }
    }

    bool GameField::isValidPosition(int x, int y) const {
        return x >= 0 && x < width && y >= 0 && y < height;
    }

```

#### **GameField.h**

```

#ifndef GAMEFIELD_H
#define GAMEFIELD_H

#include "Cell.h"
#include <vector>
#include <utility>

class GameField {
private:
    int width;
    int height;
    std::vector<std::vector<Cell>> grid;
    int playerX, playerY;

public:
    GameField(int w, int h);
    ~GameField();
    GameField(const GameField& other);
    GameField(GameField&& other) noexcept;
    GameField& operator=(const GameField& other);

```

```

    GameField& operator=(GameField&& other) noexcept;
    friend void swap(GameField& first, GameField& second) noexcept;

    int getWidth() const;
    int getHeight() const;
    void setPlayerPosition(int x, int y);
    std::pair<int, int> getPlayerPosition() const;
    bool movePlayer(int newX, int newY);
    char getCellContent(int x, int y) const;
    void setCellContent(int x, int y, char content);
    bool isCellEmpty(int x, int y) const;
    void clearCell(int x, int y);
    void clearAllCells();
    bool isValidPosition(int x, int y) const;
};

#endif

GameLogic.cpp
#include "GameLogic.h"
#include "GameConstants.h"
#include "GameInterface.h"
#include <iostream>

GameLogic::GameLogic(GameState& state, GameRules& rules, TurnManager&
turnMgr,
    ActionProcessor& actionProc, ShopSystem& shop)
    : gameState(state), gameRules(rules), turnManager(turnMgr),
    actionProcessor(actionProc), shopSystem(shop) {
}

bool GameLogic::processPlayerAction(char input, InputHandler&
inputHandler) {
    if (!turnManager.isPlayerTurn()) {
        return false;
    }

    bool actionSuccess = actionProcessor.processPlayerAction(gameState,
input, inputHandler, shopSystem);

    if (actionSuccess) {
        turnManager.endPlayerTurn();
    }

    return actionSuccess;
}

void GameLogic::processEnemyTurn() {
    if (turnManager.isPlayerTurn()) {
        return;
    }

    actionProcessor.processEnemyMoves(gameState);
    turnManager.endEnemyTurn();
}

bool GameLogic::isPlayerAlive() const {
    return gameRules.isPlayerAlive(gameState);
}

```

```

}

bool GameLogic::checkPlayerVictory() const {
    return gameRules.checkPlayerVictory(gameState);
}

bool GameLogic::isPlayerTurn() const {
    return turnManager.isPlayerTurn();
}

int GameLogic::getPlayerScore() const {
    return gameState.getPlayer().getScore();
}

GameState& GameLogic::getGameState() {
    return gameState;
}

```

### **GameLogic.h**

```

#ifndef GAMELOGIC_H
#define GAMELOGIC_H

#include "GameState.h"
#include "GameRules.h"
#include "TurnAction.h"
#include "RewardShop.h"

class GameLogic {
private:
    GameState& gameState;
    GameRules& gameRules;
    TurnManager& turnManager;
    ActionProcessor& actionProcessor;
    ShopSystem& shopSystem;

public:
    GameLogic(GameState& state, GameRules& rules, TurnManager& turnMgr,
              ActionProcessor& actionProc, ShopSystem& shop);

    // Только методы координации
    bool processPlayerAction(char input, InputHandler& inputHandler);
    void processEnemyTurn();

    // Делегирование проверок
    bool isPlayerAlive() const;
    bool checkPlayerVictory() const;
    bool isPlayerTurn() const;
    int getPlayerScore() const;

    GameState& getGameState();
};

#endif

```

### **GameManager.cpp**

```

#include "GameManager.h"
#include "GameConstants.h"
#include "GameInitializer.h"

```

```

#include <iostream>
#include <exception>

GameManager::GameManager()
    : gameState(), gameRules(), turnManager(), actionProcessor(),
    shopSystem(),
    gameLogic(gameState, gameRules, turnManager, actionProcessor,
    shopSystem),
    gameRunning(false) {

    // Инициализация начального состояния
    GameInitializer initializer;
    initializer.initializeNewGame(gameState);
}

void GameManager::run() {
    gameRunning = true;

    while (gameRunning) {
        try {
            showMainMenu();
        }
        catch (const SaveLoadException& e) {
            std::cout << "Save/Load Error: " << e.what() << std::endl;
        }
        catch (const std::exception& e) {
            std::cout << "Unexpected error: " << e.what() << std::endl;
        }
    }
}

void GameManager::showMainMenu() {
    std::cout << "=== MAIN MENU ===" << std::endl;
    std::cout << "1. New Game" << std::endl;
    std::cout << "2. Load Game" << std::endl;
    std::cout << "3. Exit" << std::endl;
    std::cout << "Choose option: ";

    char choice;
    std::cin >> choice;

    switch (choice) {
    case '1':
    {
        GameInitializer initializer;
        initializer.initializeNewGame(gameState);
    }
    turnManager = TurnManager();
    runGameLoop();
    break;

    case '2':
        handleSaveLoad();
        break;

    case '3':
        gameRunning = false;
        std::cout << "Goodbye!" << std::endl;
    }
}

```

```

        break;

    default:
        std::cout << "Invalid option! Please try again." << std::endl;
    }
}

// Метод обработки загрузки сохранения
void GameManager::handleSaveLoad() {
    if (!saveSystem.saveExists()) {
        std::cout << "No save file found! Starting new game..." <<
std::endl;

        GameInitializer initializer;
        initializer.initializeNewGame(gameState);

        turnManager = TurnManager();
        runGameLoop();
        return;
    }

    try {
        if (saveSystem.loadGame(gameState)) {
            turnManager = TurnManager();
            std::cout << "Game loaded successfully! Starting game..."
<< std::endl;
            runGameLoop();
        }
    }
    catch (const SaveLoadException& e) {
        std::cout << "Failed to load game: " << e.what() << std::endl;
        std::cout << "Starting new game instead..." << std::endl;

        GameInitializer initializer;
        initializer.initializeNewGame(gameState);

        turnManager = TurnManager();
        runGameLoop();
    }
}

// Метод обработки проигрыша
void GameManager::handleGameOver() {
    std::cout << "=== GAME OVER ===" << std::endl;
    std::cout << "1. Restart Game" << std::endl;
    std::cout << "2. Exit" << std::endl;
    std::cout << "Choose option: ";

    char choice;
    std::cin >> choice;

    switch (choice) {
    case '1':
    {
        GameInitializer initializer;
        initializer.initializeNewGame(gameState);
    }
}

```

```

    turnManager = TurnManager();
    runGameLoop();
    break;

    case '2':
        gameRunning = false;
        break;

    default:
        std::cout << "Invalid choice! Returning to main menu..." <<
std::endl;
    }
}

void GameManager::runGameLoop() {
    std::cout << "=== GAME STARTED ===" << std::endl;
    std::cout << "Controls: WASD - move, C - cast spell, M - shop, P -
save, Q - quit" << std::endl;

    while (gameRunning && gameLogic.isPlayerAlive()) {
        try {
            renderSystem.displayGameState(gameLogic.getGameState());

            // Проверяем условие победы
            if (gameLogic.checkPlayerVictory()) {
                std::cout << "VICTORY! All enemies defeated!" <<
std::endl;
                std::cout << "Final score: " <<
gameLogic.getPlayerScore() << std::endl;
                handleGameEnd();
                return;
            }

            // Ход игрока
            if (gameLogic.isPlayerTurn()) {
                std::cout << "=== YOUR TURN ===" << std::endl;
                std::cout << "Choose action: ";
                char input = inputHandler.getGameInput();

                if (input == 'q' || input == 'Q') {
                    handleGameEnd();
                    return;
                }
                else if (input == 'p' || input == 'P') {
                    saveSystem.saveGame(gameLogic.getGameState());
                    std::cout << "Game saved! Continue playing..." <<
std::endl;
                    continue;
                }
                else {
                    if (!gameLogic.processPlayerAction(input,
inputHandler)) {
                        std::cout << "Action failed! Try again." <<
std::endl;
                    }
                }
            }
            // Ход врагов

```

```

        else {
            std::cout << "=== ENEMY TURN ===" << std::endl;
            gameLogic.processEnemyTurn();

            if (!gameLogic.isPlayerAlive()) {
                std::cout << "GAME OVER! You were defeated!" <<
std::endl;
                std::cout << "Your final score: " <<
gameLogic.getPlayerScore() << std::endl;
                handleGameOver();
                return;
            }
        }
    }
    catch (const std::exception& e) {
        std::cout << "Game error: " << e.what() << std::endl;
    }
}

```

```

void GameManager::handleGameEnd() {
    std::cout << "Game ended. Returning to main menu..." << std::endl;
}

```

### **GameManager.h**

```

#ifndef GAMEMANAGER_H
#define GAMEMANAGER_H

#include "GameState.h"
#include "GameRules.h"
#include "TurnManager.h"
#include "ActionProcessor.h"
#include "InputHandler.h"
#include "RenderSystem.h"
#include "GameSaveSystem.h"
#include "ShopSystem.h"
#include "RewardSystem.h"
#include "GameLogic.h"
#include "GameInitializer.h"

class GameManager {
private:
    GameState gameState;
    GameRules gameRules;
    TurnManager turnManager;
    ActionProcessor actionProcessor;
    ShopSystem shopSystem;
    RewardSystem rewardSystem;
    GameLogic gameLogic;
    InputHandler inputHandler;
    RenderSystem renderSystem;
    GameSaveSystem saveSystem;
    bool gameRunning;

    void handleGameEnd();
    void showMainMenu();
    void handleSaveLoad();
    void runGameLoop();
    void handleGameOver();

```

```
public:
    GameManager();
    void run();
};
```

```
#endif
```

### **InputHandler.cpp**

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

```
char InputHandler::getGameInput() {
    char input;
    std::cin >> input;
    return input;
}
```

```
std::pair<int, int> InputHandler::getSpellTarget() {
    int targetX, targetY;
    std::cout << "Enter target coordinates (x y): ";
    std::cin >> targetX >> targetY;
    return { targetX, targetY };
}
```

```
int InputHandler::getSpellChoice(int maxSpells) {
    int choice;
    std::cout << "Choose spell (1-" << maxSpells << "): ";
    std::cin >> choice;
    return choice;
}
```

```
int InputHandler::getShopChoice(int maxSpells) {
    int choice;
    std::cout << "Choose spell to buy (1-" << maxSpells << ") or 0 to
cancel: ";
    std::cin >> choice;
    return choice;
}
```

### **InputHandler.h**

```
#ifndef INPUTHANDLER_H
#define INPUTHANDLER_H
```

```
#include <utility>
```

```
class InputHandler {
public:
    char getGameInput();
    std::pair<int, int> getSpellTarget();
    int getSpellChoice(int maxSpells);
    int getShopChoice(int maxSpells);
};
```

```
#endif
```

### **Main.cpp**

```
#include "GameManager.h"

int main() {
    GameManager game;
    game.run();
    return 0;
}
```

### **Player.cpp**

```
#include "Player.h"
#include "SpellSystem.h"
#include "GameField.h"
#include "Enemy.h"
#include "GameConstants.h"
#include <iostream>
#include <memory>
#include <cstdlib>
#include <ctime>

Player::Player()
    : Entity(100, 10), score(0), mana(50), maxMana(50), spellHand(3) {
    std::srand(std::time(nullptr));
    auto randomSpell = createRandomStarterSpell();
    spellHand.addSpell(std::move(randomSpell));
}

std::unique_ptr<Spell> Player::createRandomStarterSpell() const {
    int randomType = std::rand() % 2;
    if (randomType == 0) {
        return std::make_unique<DirectDamageSpell>("Starter Fireball",
8, 2, 15);
    }
    else {
        return std::make_unique<AreaDamageSpell>("Starter Fire Storm",
12, 1, 8);
    }
}

int Player::getScore() const {
    return score;
}

int Player::getMana() const {
    return mana;
}

int Player::getMaxMana() const {
    return maxMana;
}

SpellHand& Player::getSpellHand() {
    return spellHand;
}

const SpellHand& Player::getSpellHand() const {
    return spellHand;
}
```

```

void Player::addScore(int points) {
    score += points;
}

void Player::addMana(int amount) {
    mana += amount;
    if (mana > maxMana) {
        mana = maxMana;
    }
}

bool Player::useMana(int amount) {
    if (mana >= amount) {
        mana -= amount;
        return true;
    }
    return false;
}

void Player::restoreMana() {
    mana = maxMana;
}

bool Player::castSpell(int spellIndex, int targetX, int targetY,
    GameField& field, std::vector<Enemy>& enemies) {
    std::string spellName = spellHand.getSpellName(spellIndex);
    if (spellName.empty()) {
        std::cout << "Invalid spell index!" << std::endl;
        return false;
    }
    int manaCost = spellHand.getSpellManaCost(spellIndex);
    if (!useMana(manaCost)) {
        std::cout << "Not enough mana! Required: " << manaCost
            << ", available: " << mana << std::endl;
        return false;
    }
    std::cout << "Casting " << spellName << "..." << std::endl;
    auto playerPos = field.getPlayerPosition();
    SpellTarget target(targetX, targetY, playerPos.first,
        playerPos.second);
    return spellHand.castSpell(spellIndex, target, field, enemies,
        *this);
}

// НОВЫЕ МЕТОДЫ для сериализации
void Player::setHealth(int newHealth) {
    health = newHealth;
    if (health < 0) health = 0;
}

void Player::setMana(int newMana) {
    mana = newMana;
    if (mana > maxMana) mana = maxMana;
    if (mana < 0) mana = 0;
}

void Player::setMaxMana(int newMaxMana) {

```

```

        maxMana = newMaxMana;
        if (mana > maxMana) mana = maxMana;
    }

```

```

void Player::setScore(int newScore) {
    score = newScore;
    if (score < 0) score = 0;
}

```

```

void Player::clearSpellHand() {
    spellHand.clearHand();
}

```

# **Player.h**

```

#ifndef PLAYER_H
#define PLAYER_H

```

```

#include "Entity.h"
#include "SpellHand.h"
#include <memory>

```

```

class GameField;
class Enemy;

```

```

class Player : public Entity {
private:
    int score;
    int mana;
    int maxMana;
    SpellHand spellHand;
    std::unique_ptr<Spell> createRandomStarterSpell() const;

```

```

public:
    Player();
    int getScore() const;
    int getMana() const;
    int getMaxMana() const;
    SpellHand& getSpellHand();
    const SpellHand& getSpellHand() const;
    void addScore(int points);
    void addMana(int amount);
    bool useMana(int amount);
    void restoreMana();
    bool castSpell(int spellIndex, int targetX, int targetY,
        GameField& field, std::vector<Enemy>& enemies);

```

```

    // НОВЫЕ МЕТОДЫ для сериализации
    void setHealth(int newHealth);
    void setMana(int newMana);
    void setMaxMana(int newMaxMana);
    void setScore(int newScore);
    void clearSpellHand();
};

```

```

#endif

```

### **GameState.cpp**

```
#include "GameState.h"
#include <iostream>
#include <fstream>
#include <stdexcept>

GameState::GameState()
    : field(15, 15), player() {
}

// Простые геттеры без логики
GameField& GameState::getField() {
    return field;
}

Player& GameState::getPlayer() {
    return player;
}

std::vector<Enemy>& GameState::getEnemies() {
    return enemies;
}

const GameField& GameState::getField() const {
    return field;
}

const Player& GameState::getPlayer() const {
    return player;
}

const std::vector<Enemy>& GameState::getEnemies() const {
    return enemies;
}
```

### **GameState.h**

```
#ifndef GAMESTATE_H
#define GAMESTATE_H

#include "GameField.h"
#include "Player.h"
#include "Enemy.h"
#include <vector>
#include <fstream>
#include <stdexcept>

class GameState {
private:
    GameField field;
    Player player;
    std::vector<Enemy> enemies;

public:
    GameState();

    // Только геттеры и сеттеры - никакой бизнес-логики
    GameField& getField();
    Player& getPlayer();
```

```

        std::vector<Enemy>& getEnemies();

        const GameField& getField() const;
        const Player& getPlayer() const;
        const std::vector<Enemy>& getEnemies() const;

};

#endif

```

### RenderSystem.cpp

```

#include "RenderSystem.h"
#include "GameState.h"
#include "GameField.h"
#include "Player.h"
#include "SpellHand.h"
#include "SpellSystem.h"
#include "GameConstants.h"
#include <iostream>

void RenderSystem::displayGameState(const GameState& gameState) const {
    const auto& field = gameState.getField();
    const auto& player = gameState.getPlayer();
    const auto& enemies = gameState.getEnemies();

    GameField& nonConstField = const_cast<GameField&>(field);
    nonConstField.clearAllCells();

    for (const auto& enemy : enemies) {
        if (enemy.isAlive()) {
            nonConstField.setCellContent(enemy.getX(), enemy.getY(),
CellContent::ENEMY);
        }
    }

    auto playerPos = field.getPlayerPosition();
    nonConstField.setCellContent(playerPos.first, playerPos.second,
CellContent::PLAYER);

    displayGameInfo(player, playerPos.first, playerPos.second);
    displayField(field);
    displaySpells(player.getSpellHand());
}

void RenderSystem::displayGameInfo(const Player& player, int playerX,
int playerY) const {
    std::cout << "Health: " << player.getHealth();
    std::cout << " | Mana: " << player.getMana() << "/" <<
player.getMaxMana();
    std::cout << " | Score: " << player.getScore();
    std::cout << " | Position: (" << playerX << ", " << playerY << ")"
<< std::endl;
}

void RenderSystem::displayField(const GameField& field) const {
    for (int y = 0; y < field.getHeight(); y++) {
        for (int x = 0; x < field.getWidth(); x++) {

```

```

        std::cout << field.getCellContent(x, y) << " ";
    }
    std::cout << std::endl;
}

void RenderSystem::displaySpells(const SpellHand& hand) const {
    hand.displaySpells();
}

void RenderSystem::displayShop(const
std::vector<std::unique_ptr<Spell>>& availableSpells,
    const Player& player) const {
    std::cout << "=== SPELL SHOP ===" << std::endl;
    std::cout << "Your score: " << player.getScore() << " points" <<
std::endl;
    std::cout << "Free spell slots: "
        << (player.getSpellHand().getMaxSize() -
player.getSpellHand().getSpellCount())
        << std::endl;
    std::cout << "Available spells:" << std::endl;

    for (int i = 0; i < availableSpells.size(); i++) {
        int cost = availableSpells[i]->getManaCost();
        std::cout << i + 1 << ". " <<
availableSpells[i]->getDescription()
            << " - Cost: " << cost << " points" << std::endl;
    }
}

```

### **RenderSystem.h**

```

#ifndef RENDERSYSTEM_H
#define RENDERSYSTEM_H

#include <vector>
#include <memory>

class GameState;
class Player;
class SpellHand;
class Spell;
class GameField;

class RenderSystem {
public:
    void displayGameState(const GameState& gameState) const;
    void displayGameInfo(const Player& player, int playerX, int playerY)
const;
    void displayField(const GameField& field) const;
    void displaySpells(const SpellHand& hand) const;
    void displayShop(const std::vector<std::unique_ptr<Spell>>&
availableSpells,
        const Player& player) const;
};

#endif

```

### **RewardSystem.cpp**

```
#include "RewardSystem.h"
#include "Player.h"
#include <iostream>

RewardSystem::RewardSystem() : enemiesKilled(0) {
}

void RewardSystem::onEnemyKilled(Player& player) {
    enemiesKilled++;
    std::cout << "Enemies killed: " << enemiesKilled << std::endl;
}

void RewardSystem::giveVictoryReward(Player& player) const {
    std::cout << "Congratulations! You won!" << std::endl;
}

void RewardSystem::resetKillCounter() {
    enemiesKilled = 0;
}

int RewardSystem::getEnemiesKilled() const {
    return enemiesKilled;
}
```

### **RewardSystem.h**

```
#ifndef REWARDSYSTEM_H
#define REWARDSYSTEM_H

class Player;

class RewardSystem {
private:
    int enemiesKilled;

public:
    RewardSystem();
    void onEnemyKilled(Player& player);
    void giveVictoryReward(Player& player) const;
    void resetKillCounter();
    int getEnemiesKilled() const;
};

#endif
```

### ShopSystem.cpp

```
#include "ShopSystem.h"
#include "GameConstants.h"
#include <iostream>
#include <cstdlib>
#include <ctime>

ShopSystem::ShopSystem() {
    availableSpells.push_back(std::make_unique<DirectDamageSpell>("Ice Arrow", 40, 4, 20));
    availableSpells.push_back(std::make_unique<AreaDamageSpell>("Ice Storm", 90, 3, 12));
}

std::unique_ptr<Spell> ShopSystem::createSpellCopy(Spell* baseSpell) const {
    return baseSpell->clone();
}

int ShopSystem::calculateCost(Spell* spell) const {
    return spell->getManaCost();
}

bool ShopSystem::buySpell(Player& player, int spellIndex) const {
    if (spellIndex < 0 || spellIndex >= availableSpells.size()) {
        std::cout << "Invalid spell index!" << std::endl;
        return false;
    }

    Spell* baseSpell = availableSpells[spellIndex].get();
    int cost = calculateCost(baseSpell);

    if (player.getScore() < cost) {
        std::cout << "Not enough points! Need " << cost << ", but you have " << player.getScore() << std::endl;
        return false;
    }

    if (player.getSpellHand().isFull()) {
        std::cout << "Your hand is full! Cannot buy more spells." << std::endl;
        return false;
    }

    auto newSpell = createSpellCopy(baseSpell);
    if (newSpell) {
        player.addScore(-cost);
        player.getSpellHand().addSpell(std::move(newSpell));
        std::cout << "Bought " << baseSpell->getName() << " for " << cost << " points!" << std::endl;
        return true;
    }

    return false;
}

const std::vector<std::unique_ptr<Spell>>& ShopSystem::getAvailableSpells() const {
```

```

        return availableSpells;
    }

bool ShopSystem::interactWithPlayer(Player& player, InputHandler&
inputHandler) const {
    if (player.getSpellHand().isFull()) {
        std::cout << "Your hand is full! Cannot buy more spells." <<
std::endl;
        return false;
    }

    if (availableSpells.empty()) {
        std::cout << "No spells available in the shop!" << std::endl;
        return false;
    }

    std::cout << "=== SPELL SHOP ===" << std::endl;
    std::cout << "Your score: " << player.getScore() << " points" <<
std::endl;
    std::cout << "Free spell slots: "
        << (player.getSpellHand().getMaxSize() -
player.getSpellHand().getSpellCount())
        << "/" << player.getSpellHand().getMaxSize() << std::endl;
    std::cout << "Available spells:" << std::endl;

    for (int i = 0; i < availableSpells.size(); i++) {
        int cost = calculateCost(availableSpells[i].get());
        std::cout << i + 1 << ". " <<
availableSpells[i]->getDescription()
        << " - Cost: " << cost << " points" << std::endl;
    }

    int choice = inputHandler.getShopChoice(availableSpells.size());
    if (choice == 0) {
        std::cout << "Shop interaction cancelled." << std::endl;
        return true;
    }

    return processPlayerChoice(player, choice - 1);
}

bool ShopSystem::processPlayerChoice(Player& player, int spellIndex)
const {
    return buySpell(player, spellIndex);
}

```

### ShopSystem.h

```

#ifndef SHOPSYSTEM_H
#define SHOPSYSTEM_H

#include "Player.h"
#include "SpellSystem.h"
#include "InputHandler.h"
#include <vector>
#include <memory>

class ShopSystem {

```

```

private:
    std::vector<std::unique_ptr<Spell>> availableSpells;
    std::unique_ptr<Spell> createSpellCopy(Spell* baseSpell) const;
    bool processPlayerChoice(Player& player, int choice) const;

public:
    ShopSystem();
    bool buySpell(Player& player, int spellIndex) const;
    const std::vector<std::unique_ptr<Spell>>& getAvailableSpells()
const;
    int calculateCost(Spell* spell) const;
    bool interactWithPlayer(Player& player, InputHandler& inputHandler)
const;
};

#endif

```

### **SpellHand.cpp**

```

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

SpellHand::SpellHand(int size) : maxSize(size) {}

bool SpellHand::addSpell(std::unique_ptr<Spell> spell) {
    if (spells.size() >= maxSize) {
        std::cout << "Hand is full! Cannot add new spell." << std::endl;
        return false;
    }
    spells.push_back(std::move(spell));
    std::cout << "Added spell: " << spells.back()->getName() <<
std::endl;
    return true;
}

bool SpellHand::removeSpell(int index) {
    if (index < 0 || index >= spells.size()) {
        return false;
    }
    spells.erase(spells.begin() + index);
    return true;
}

void SpellHand::clearHand() {
    spells.clear();
}

bool SpellHand::castSpell(int index, const SpellTarget& target,
    GameField& field, std::vector<Enemy>& enemies, Player& player) const
{
    if (index < 0 || index >= spells.size()) {
        return false;
    }
    return spells[index]->cast(target, field, enemies, player);
}

std::string SpellHand::getSpellName(int index) const {

```

```

        if (index < 0 || index >= spells.size()) {
            return "";
        }
        return spells[index]->getName();
    }

    std::string SpellHand::getSpellDescription(int index) const {
        if (index < 0 || index >= spells.size()) {
            return "";
        }
        return spells[index]->getDescription();
    }

    int SpellHand::getSpellManaCost(int index) const {
        if (index < 0 || index >= spells.size()) {
            return 0;
        }
        return spells[index]->getManaCost();
    }

    int SpellHand::getSpellCount() const {
        return spells.size();
    }

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

    bool SpellHand::isFull() const {
        return spells.size() >= maxSize;
    }

    void SpellHand::displaySpells() const {
        std::cout << "=== Spells in hand ===" << std::endl;
        for (int i = 0; i < spells.size(); i++) {
            std::cout << i + 1 << ". " << spells[i]->getDescription() <<
std::endl;
        }
        std::cout << "Free slots: " << (maxSize - spells.size()) << "/" <<
maxSize << std::endl;
    }
}

```

### **SpellHand.h**

```

#ifndef SPELLHAND_H
#define SPELLHAND_H

#include "SpellSystem.h"
#include <vector>
#include <memory>

class SpellHand {
private:
    std::vector<std::unique_ptr<Spell>> spells;
    int maxSize;

public:
    SpellHand(int size);
}

```

```

        bool addSpell(std::unique_ptr<Spell> spell);
        bool removeSpell(int index);
        void clearHand();
        bool castSpell(int index, const SpellTarget& target,
            GameField& field, std::vector<Enemy>& enemies, Player& player)
const;
        std::string getSpellName(int index) const;
        std::string getSpellDescription(int index) const;
        int getSpellManaCost(int index) const;
        int getSpellCount() const;
        int getMaxSize() const;
        bool isFull() const;
        void displaySpells() const;
};

#endif

```

### **TurnAction.cpp**

```

#include "TurnAction.h"
#include "GameConstants.h"
#include <iostream>
#include <cstdlib>

// TurnManager implementation
TurnManager::TurnManager() : playerTurn(true) {}

bool TurnManager::isPlayerTurn() const {
    return playerTurn;
}

void TurnManager::endPlayerTurn() {
    playerTurn = false;
}

void TurnManager::endEnemyTurn() {
    playerTurn = true;
}

void TurnManager::reset() {
    playerTurn = true;
}

// ActionProcessor implementation
bool ActionProcessor::processPlayerAction(GameState& state, char input,
    InputHandler& inputHandler, ShopSystem& shopSystem) {
    bool actionSuccess = false;

    if (input == 'c' || input == 'C') {
        actionSuccess = processSpellCast(state, inputHandler);
    }
    else if (input == 'm' || input == 'M') {
        actionSuccess = processShopInteraction(state, shopSystem,
            inputHandler);
    }
    else {
        auto playerPos = state.getField().getPlayerPosition();
        int newX = playerPos.first;
    }
}

```

```

        int newY = playerPos.second;

        switch (input) {
        case 'w': case 'W': newY--; break;
        case 's': case 'S': newY++; break;
        case 'a': case 'A': newX--; break;
        case 'd': case 'D': newX++; break;
        default: return false;
        }
        actionSuccess = processPlayerMove(state, newX, newY);
    }

    return actionSuccess;
}

bool ActionProcessor::processPlayerMove(GameState& state, int newX, int
newY) {
    auto& field = state.getField();
    auto& player = state.getPlayer();
    auto& enemies = state.getEnemies();

    if (!field.isValidPosition(newX, newY)) {
        std::cout << "Cannot move there - out of bounds!" << std::endl;
        return false;
    }

    // Проверка атаки врага
    for (auto& enemy : enemies) {
        if (enemy.isAlive() && enemy.getX() == newX && enemy.getY() ==
newY) {
            enemy.takeDamage(player.getDamage());
            std::cout << "You attacked enemy! Enemy health: " <<
enemy.getHealth() << std::endl;
            if (!enemy.isAlive()) {
                player.addScore(10);
                std::cout << "Enemy defeated! +10 points" << std::endl;
            }
            return true;
        }
    }

    // Обычное движение
    if (field.movePlayer(newX, newY)) {
        player.addScore(1);
        player.addMana(5);
        return true;
    }

    return false;
}

bool ActionProcessor::processSpellCast(GameState& state, InputHandler&
inputHandler) {
    auto& player = state.getPlayer();

    if (player.getSpellHand().getSpellCount() == 0) {
        std::cout << "You have no spells in hand!" << std::endl;
        return false;
    }

```

```

    }

    int spellChoice =
inputHandler.getSpellChoice(player.getSpellHand().getSpellCount());
    if (spellChoice < 1 || spellChoice >
player.getSpellHand().getSpellCount()) {
        std::cout << "Invalid spell choice!" << std::endl;
        return false;
    }

    std::pair<int, int> target = inputHandler.getSpellTarget();
    return player.castSpell(spellChoice - 1, target.first,
target.second,
        state.getField(), state.getEnemies());
}

bool ActionProcessor::processShopInteraction(GameState& state,
ShopSystem& shopSystem, InputHandler& inputHandler) {
    return shopSystem.interactWithPlayer(state.getPlayer(),
inputHandler);
}

void ActionProcessor::processEnemyMoves(GameState& state) {
    auto& field = state.getField();
    auto& player = state.getPlayer();
    auto& enemies = state.getEnemies();
    auto playerPos = field.getPlayerPosition();
    int playerX = playerPos.first;
    int playerY = playerPos.second;

    for (auto& enemy : enemies) {
        if (!enemy.isAlive()) continue;

        int randomNumber = std::rand() % 4;
        int oldX = enemy.getX();
        int oldY = enemy.getY();
        int newX = oldX;
        int newY = oldY;

        switch (randomNumber) {
            case 0: newX++; break; // EAST
            case 1: newX--; break; // WEST
            case 2: newY++; break; // SOUTH
            case 3: newY--; break; // NORTH
        }

        if (field.isValidPosition(newX, newY)) {
            bool canMove = true;
            if (newX == playerX && newY == playerY) {
                player.takeDamage(enemy.getDamage());
                std::cout << "Enemy attacked you! Lost " <<
enemy.getDamage() << " health!" << std::endl;
                canMove = false;
            }

            for (auto& otherEnemy : enemies) {
                if (&enemy != &otherEnemy && otherEnemy.isAlive() &&

```

```

        otherEnemy.getX() == newX && otherEnemy.getY() ==
newY) {
            canMove = false;
            break;
        }
    }

    if (canMove && field.isCellEmpty(newX, newY)) {
        enemy.move(newX, newY);
    }
}
}
}

```

### **TurnAction.h**

```

#ifndef TURNACTION_H
#define TURNACTION_H

#include "GameState.h"
#include "GameInterface.h"
#include "RewardShop.h"

class TurnManager {
private:
    bool playerTurn;

public:
    TurnManager();
    bool isPlayerTurn() const;
    void endPlayerTurn();
    void endEnemyTurn();
    void reset();
};

class ActionProcessor {
private:
    bool processPlayerMove(GameState& state, int newX, int newY);
    bool processSpellCast(GameState& state, InputHandler&
inputHandler);
    bool processShopInteraction(GameState& state, ShopSystem&
shopSystem, InputHandler& inputHandler);

public:
    bool processPlayerAction(GameState& state, char input, InputHandler&
inputHandler, ShopSystem& shopSystem);
    void processEnemyMoves(GameState& state);
};

#endif

```

### **GameConstants.h**

```

#ifndef GAMECONSTANTS_H
#define GAMECONSTANTS_H

enum class SpellType {
    DIRECT_DAMAGE = 0,
    AREA_DAMAGE = 1
};

```

```

enum class Direction {
    EAST = 0,
    WEST = 1,
    SOUTH = 2,
    NORTH = 3
};

namespace CellContent {
    constexpr char EMPTY = '.';
    constexpr char PLAYER = 'P';
    constexpr char ENEMY = 'E';
}

#endif

```

### **GameSaveSystem.cpp**

```

#include "GameSaveSystem.h"
#include "GameStateSerializer.h" // Новый класс для сериализации
#include <iostream>

GameSaveSystem::GameSaveSystem(const std::string& saveFileName)
    : defaultSaveFile(saveFileName) {
}

void GameSaveSystem::saveGame(const GameState& gameState) const {
    saveGame(gameState, defaultSaveFile);
}

void GameSaveSystem::saveGame(const GameState& gameState, const
std::string& filename) const {
    std::ofstream file(filename, std::ios::binary);
    if (!file.is_open()) {
        throw FileWriteException(filename);
    }

    try {
        GameStateSerializer serializer;
        serializer.serialize(gameState, file);

        if (file.fail()) {
            throw FileWriteException(filename);
        }

        std::cout << "Game saved successfully to: " << filename <<
std::endl;
    }
    catch (const std::exception& e) {
        throw FileWriteException(filename + " - " + e.what());
    }
}

bool GameSaveSystem::loadGame(GameState& gameState) const {
    return loadGame(gameState, defaultSaveFile);
}

bool GameSaveSystem::loadGame(GameState& gameState, const std::string&
filename) const {

```

```

        if (!saveExists(filename)) {
            throw FileNotFoundException(filename);
        }

        std::ifstream file(filename, std::ios::binary);
        if (!file.is_open()) {
            throw FileNotFoundException(filename);
        }

        try {
            GameStateSerializer serializer;
            serializer.deserialize(gameState, file);

            std::cout << "Game loaded successfully from: " << filename <<
std::endl;
            return true;
        }
        catch (const CorruptedSaveException&) {
            throw;
        }
        catch (const std::exception& e) {
            throw CorruptedSaveException();
        }
    }

bool GameSaveSystem::saveExists() const {
    return saveExists(defaultSaveFile);
}

bool GameSaveSystem::saveExists(const std::string& filename) const {
    std::ifstream file(filename);
    return file.good();
}

void GameSaveSystem::setDefaultSaveFile(const std::string& filename) {
    defaultSaveFile = filename;
}

std::string GameSaveSystem::getDefaultSaveFile() const {
    return defaultSaveFile;
}

bool GameSaveSystem::deleteSaveFile() const {
    return deleteSaveFile(defaultSaveFile);
}

bool GameSaveSystem::deleteSaveFile(const std::string& filename) const
{
    if (std::remove(filename.c_str()) == 0) {
        std::cout << "Save file deleted: " << filename << std::endl;
        return true;
    }
    return false;
}

GameSaveSystem.h
#ifndef GAMESAVESYSTEM_H
#define GAMESAVESYSTEM_H

```

```

#include "GameState.h"
#include <string>
#include <fstream>
#include <stdexcept>

// Иерархия исключений для обработки ошибок сохранения/загрузки
class SaveLoadException : public std::runtime_error {
public:
    explicit SaveLoadException(const std::string& message)
        : std::runtime_error("Save/Load Error: " + message) {}
};

class FileNotFoundException : public SaveLoadException {
public:
    explicit FileNotFoundException(const std::string& filename)
        : SaveLoadException("File not found: " + filename) {}
};

class FileWriteException : public SaveLoadException {
public:
    explicit FileWriteException(const std::string& filename)
        : SaveLoadException("Cannot write to file: " + filename) {}
};

class CorruptedSaveException : public SaveLoadException {
public:
    CorruptedSaveException()
        : SaveLoadException("Save file is corrupted or invalid format")
    {
    }
};

class GameSaveSystem {
private:
    std::string defaultSaveFile = "game_save.dat";

public:
    // Конструктор
    GameSaveSystem() = default;
    explicit GameSaveSystem(const std::string& saveFileName);

    // Основные методы сохранения/загрузки
    void saveGame(const GameState& gameState) const;
    void saveGame(const GameState& gameState, const std::string&
filename) const;

    bool loadGame(GameState& gameState) const;
    bool loadGame(GameState& gameState, const std::string& filename)
const;

    // Методы проверки существования сохранений
    bool saveExists() const;
    bool saveExists(const std::string& filename) const;

    // Методы для управления файлами сохранений
    void setDefaultSaveFile(const std::string& filename);

```

```

        std::string getDefaultSaveFile() const;
        bool deleteSaveFile() const;
        bool deleteSaveFile(const std::string& filename) const;
};

#endif

```

### **GameRules.cpp**

```

#include "GameRules.h"
#include "GameState.h"

bool GameRules::checkPlayerVictory(const GameState& state) const {
    for (const auto& enemy : state.getEnemies()) {
        if (enemy.isAlive()) {
            return false;
        }
    }
    return true;
}

bool GameRules::isPlayerAlive(const GameState& state) const {
    return state.getPlayer().isAlive();
}

bool GameRules::isValidMove(const GameState& state, int newX, int newY)
const {
    const auto& field = state.getField();

    if (!field.isValidPosition(newX, newY)) {
        return false;
    }

    // Проверяем, не занята ли клетка врагом (для атаки)
    for (const auto& enemy : state.getEnemies()) {
        if (enemy.isAlive() && enemy.getX() == newX && enemy.getY() ==
newY) {
            return true; // Можно атаковать
        }
    }

    // Проверяем, пуста ли клетка для движения
    return field.isCellEmpty(newX, newY);
}

bool GameRules::canPlayerCastSpell(const Player& player, int spellIndex)
const {
    if (spellIndex < 0 || spellIndex >=
player.getSpellHand().getSpellCount()) {
        return false;
    }

    int manaCost = player.getSpellHand().getSpellManaCost(spellIndex);
    return player.getMana() >= manaCost;
}

```

## **GameRules.h**

```
#ifndef GAMERULES_H
#define GAMERULES_H

#include "GameState.h"

class GameRules {
public:
    bool checkPlayerVictory(const GameState& state) const;
    bool isPlayerAlive(const GameState& state) const;
    bool isValidMove(const GameState& state, int newX, int newY) const;
    bool canPlayerCastSpell(const Player& player, int spellIndex) const;
};

#endif
```

## **GameInitializer.cpp**

```
#include "GameInitializer.h"

void GameInitializer::initializeNewGame(GameState& state) {
    // Сброс состояния
    state = GameState();

    // Начальная позиция игрока
    state.getField().setPlayerPosition(0, 0);

    // Создание врагов
    state.getEnemies().clear();
    state.getEnemies().push_back(Enemy(5, 5));
    state.getEnemies().push_back(Enemy(10, 10));
}

void GameInitializer::initializeTestGame(GameState& state) {
    initializeNewGame(state);
}
```

## **GameInitializer.h**

```
#ifndef GAMEINITIALIZER_H
#define GAMEINITIALIZER_H

#include "GameState.h"
#include "Enemy.h"

class GameInitializer {
public:
    void initializeNewGame(GameState& state);
    void initializeTestGame(GameState& state);
};

#endif
```

## **AreaDamageSpell.cpp**

```
#include "AreaDamageSpell.h"
```

```

#include "GameField.h"
#include "Enemy.h"
#include "Player.h"
#include <iostream>
#include <cmath>

AreaDamageSpell::AreaDamageSpell(const std::string& spellName, int
cost, int spellRange, int spellDamage)
    : name(spellName), manaCost(cost), range(spellRange),
damage(spellDamage) {
}

bool AreaDamageSpell::cast(const SpellTarget& target, GameField& field,
std::vector<Enemy>& enemies, Player& player) {
    int distance = std::abs(target.targetX - target.casterX) +
std::abs(target.targetY - target.casterY);
    if (distance > range) {
        std::cout << "Target is too far! Distance: " << distance << ",
max: " << range << std::endl;
        return false;
    }

    if (!field.isValidPosition(target.targetX, target.targetY)) {
        std::cout << "Invalid target position!" << std::endl;
        return false;
    }

    std::cout << name << " hits 2x2 area!" << std::endl;
    int enemiesHit = 0;

    for (int y = target.targetY; y <= target.targetY + 1; y++) {
        for (int x = target.targetX; x <= target.targetX + 1; x++) {
            if (field.isValidPosition(x, y)) {
                for (auto& enemy : enemies) {
                    if (enemy.isAlive() && enemy.getX() == x &&
enemy.getY() == y) {
                        enemy.takeDamage(damage);
                        std::cout << "Enemy at (" << x << ", " << y <<
") takes " << damage << " damage!" << std::endl;
                        if (!enemy.isAlive()) {
                            std::cout << "Enemy defeated!" << std::endl;
                            player.addScore(10);
                        }
                        enemiesHit++;
                    }
                }
            }
        }
    }

    if (enemiesHit == 0) {
        std::cout << "No enemies found in the affected area." <<
std::endl;
    }
    return true;
}

std::string AreaDamageSpell::getDescription() const {

```

```

        return name + " - area damage: " + std::to_string(damage) + " (2x2),
range: " + std::to_string(range);
    }

    std::unique_ptr<Spell> AreaDamageSpell::clone() const {
        return std::make_unique<AreaDamageSpell>(name, manaCost, range,
damage);
    }

    std::string AreaDamageSpell::getName() const {
        return name;
    }

    int AreaDamageSpell::getManaCost() const {
        return manaCost;
    }

    int AreaDamageSpell::getRange() const {
        return range;
    }

```

### **AreaDamageSpell.h**

```

#ifndef AREADAMAGESPELL_H
#define AREADAMAGESPELL_H

#include "Spell.h"
#include "SpellTarget.h"

class AreaDamageSpell : public Spell {
private:
    std::string name;
    int manaCost;
    int range;
    int damage;

public:
    AreaDamageSpell(const std::string& name, int cost, int spellRange,
int spellDamage);
    bool cast(const SpellTarget& target, GameField& field,
        std::vector<Enemy>& enemies, Player& player) override;
    std::string getDescription() const override;
    std::unique_ptr<Spell> clone() const override;
    std::string getName() const override;

```

```

        int getManaCost() const override;
        int getRange() const override;
};

#endif

```

### **DirectDamageSpell.cpp**

```

#include "DirectDamageSpell.h"
#include "GameField.h"
#include "Enemy.h"
#include "Player.h"
#include <iostream>
#include <cmath>

DirectDamageSpell::DirectDamageSpell(const std::string& spellName, int
cost, int spellRange, int spellDamage)
    : name(spellName), manaCost(cost), range(spellRange),
damage(spellDamage) {
}

bool DirectDamageSpell::cast(const SpellTarget& target, GameField&
field,
    std::vector<Enemy>& enemies, Player& player) {
    int distance = std::abs(target.targetX - target.casterX) +
std::abs(target.targetY - target.casterY);
    if (distance > range) {
        std::cout << "Target is too far! Distance: " << distance << ",
max: " << range << std::endl;
        return false;
    }

    if (!field.isValidPosition(target.targetX, target.targetY)) {
        std::cout << "Invalid target position!" << std::endl;
        return false;
    }

    bool enemyFound = false;
    for (auto& enemy : enemies) {
        if (enemy.isAlive() && enemy.getX() == target.targetX &&
enemy.getY() == target.targetY) {
            enemy.takeDamage(damage);
            std::cout << name << " deals " << damage << " damage to
enemy!" << std::endl;
            if (!enemy.isAlive()) {
                std::cout << "Enemy defeated!" << std::endl;
                player.addScore(10);
            }
            enemyFound = true;
            break;
        }
    }

    if (!enemyFound) {
        std::cout << "No enemy on target cell! Spell failed." <<
std::endl;
        return false;
    }
}

```

```

        return true;
    }

    std::string DirectDamageSpell::getDescription() const {
        return name + " - damage: " + std::to_string(damage) + ", range: "
+ std::to_string(range);
    }

    std::unique_ptr<Spell> DirectDamageSpell::clone() const {
        return std::make_unique<DirectDamageSpell>(name, manaCost, range,
damage);
    }

    std::string DirectDamageSpell::getName() const {
        return name;
    }

    int DirectDamageSpell::getManaCost() const {
        return manaCost;
    }

    int DirectDamageSpell::getRange() const {
        return range;
    }

```

### **DirectDamageSpell.h**

```

#ifndef DIRECTDAMAGESPELL_H
#define DIRECTDAMAGESPELL_H

#include "Spell.h"
#include "SpellTarget.h"

class DirectDamageSpell : public Spell {
private:
    std::string name;
    int manaCost;
    int range;
    int damage;

public:
    DirectDamageSpell(const std::string& name, int cost, int spellRange,
int spellDamage);
    bool cast(const SpellTarget& target, GameField& field,
        std::vector<Enemy>& enemies, Player& player) override;
    std::string getDescription() const override;
    std::unique_ptr<Spell> clone() const override;
    std::string getName() const override;
    int getManaCost() const override;
    int getRange() const override;
};

#endif

```

### **SpellTarget.cpp**

```
#include "SpellTarget.h"
```

```
SpellTarget::SpellTarget(int tX, int tY, int cX, int cY)
    : targetX(tX), targetY(tY), casterX(cX), casterY(cY) {
}
```

### **SpellTarget.h**

```
#ifndef SPELLTARGET_H
#define SPELLTARGET_H
```

```
class SpellTarget {
public:
    int targetX;
    int targetY;
    int casterX;
    int casterY;
    SpellTarget(int tX, int tY, int cX, int cY);
};
```

```
#endif
```

### **Spell.h**

```
#ifndef SPELL_H
#define SPELL_H
```

```
#include <string>
#include <memory>
#include <vector>
```

```
class GameField;
class Enemy;
class Player;
class SpellTarget;
```

```
class Spell {
public:
    virtual ~Spell() = default;
    virtual bool cast(const SpellTarget& target, GameField& field,
        std::vector<Enemy>& enemies, Player& player) = 0;
    virtual std::string getDescription() const = 0;
    virtual std::unique_ptr<Spell> clone() const = 0;
    virtual std::string getName() const = 0;
    virtual int getManaCost() const = 0;
    virtual int getRange() const = 0;
};
```

```
#endif
```

### **ActionProcessor.h**

```
#ifndef ACTIONPROCESSOR_H
#define ACTIONPROCESSOR_H
```

```
#include "GameState.h"
#include "InputHandler.h"
```

```

#include "ShopSystem.h"

class ActionProcessor {
private:
    bool processPlayerMove(GameState& state, int newX, int newY);
    bool processSpellCast(GameState& state, InputHandler&
inputHandler);
    bool processShopInteraction(GameState& state, ShopSystem&
shopSystem, InputHandler& inputHandler);

public:
    bool processPlayerAction(GameState& state, char input, InputHandler&
inputHandler, ShopSystem& shopSystem);
    void processEnemyMoves(GameState& state);
};

#endif

```

### **GameStateSerializer.cpp**

```

#include "GameStateSerializer.h"
#include "GameState.h"

void GameStateSerializer::serialize(const GameState& gameState,
std::ostream& stream) const {
    stream << "GAME_SAVE_V1" << std::endl;

    const auto& field = gameState.getField();
    auto playerPos = field.getPlayerPosition();
    stream << field.getWidth() << " " << field.getHeight() << " "
        << playerPos.first << " " << playerPos.second << std::endl;

    for (int y = 0; y < field.getHeight(); y++) {
        for (int x = 0; x < field.getWidth(); x++) {
            stream << field.getCellContent(x, y) << " ";
        }
        stream << std::endl;
    }

    const auto& player = gameState.getPlayer();
    stream << player.getHealth() << " " << player.getScore() << " "
        << player.getMana() << " " << player.getMaxMana() << std::endl;

    stream << player.getSpellHand().getSpellCount() << std::endl;

    const auto& enemies = gameState.getEnemies();
    stream << enemies.size() << std::endl;
    for (const auto& enemy : enemies) {
        stream << enemy.getX() << " " << enemy.getY() << " "
            << enemy.getHealth() << " " << enemy.getDamage() <<
std::endl;
    }
}

void GameStateSerializer::deserialize(GameState& gameState,
std::istream& stream) const {
    std::string version;
    stream >> version;

```

```

    if (version != "GAME_SAVE_V1") {
        throw std::runtime_error("Invalid save file version");
    }

    int width, height, playerX, playerY;
    stream >> width >> height >> playerX >> playerY;

    auto& field = gameState.getField();
    field = GameField(width, height);
    field.setPlayerPosition(playerX, playerY);

    for (int y = 0; y < height; y++) {
        for (int x = 0; x < width; x++) {
            char content;
            stream >> content;
            field.setCellContent(x, y, content);
        }
    }

    int health, score, mana, maxMana;
    stream >> health >> score >> mana >> maxMana;

    auto& player = gameState.getPlayer();
    player.setHealth(health);
    player.setScore(score);
    player.setMana(mana);
    player.setMaxMana(maxMana);

    int spellCount;
    stream >> spellCount;
    player.clearSpellHand();

    int enemyCount;
    stream >> enemyCount;
    auto& enemies = gameState.getEnemies();
    enemies.clear();
    for (int i = 0; i < enemyCount; i++) {
        int x, y, enemyHealth, enemyDamage;
        stream >> x >> y >> enemyHealth >> enemyDamage;
        Enemy enemy(x, y);
        enemy.setHealth(enemyHealth);
        enemies.push_back(enemy);
    }
}

```

### **GameStateSerializer.h**

```

#ifndef GAMESTATESERIALIZER_H
#define GAMESTATESERIALIZER_H

#include "GameState.h"
#include <iostream>
#include <fstream>

class GameStateSerializer {
public:

```

```
        void serialize(const GameState& gameState, std::ostream& stream)
const;
        void deserialize(GameState& gameState, std::istream& stream) const;
};

#endif
```

## ПРИЛОЖЕНИЕ Б

### ТЕСТИРОВАНИЕ

[illegible]

Рисунок 2 – Запуск новой игры

```
File: src/01_27_5
=== YOUR TURN ===
Choose action: q
Game ended. Returning to main menu...
=== MAIN MENU ===
1. New Game
2. Load Game
3. Exit
Choose option: |
```

Рисунок 3 – Выход в меню

