МИНОБРНАУКИ РОССИИ

САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ «ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)

Кафедра МОЭВМ

ОТЧЕТ

по лабораторной работе №4

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 гибкую архитектуру, разделяющую ответственность между компонентами ввода, отрисовки, управления и визуализации, с использованием шаблонного программирования для обеспечения масштабируемости и возможности замены реализаций без модификации ядра игры.

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.

Выполнение работы.

В ходе выполнения лабораторной работы была реализована объектно-ориентированная пошаговая игра с использованием шаблонного программирования для обеспечения гибкости и масштабируемости архитектуры.

Архитектура программы.

В программе реализована иерархия классов, соответствующая принципам ООП.

Описание классов

Назначение: Интерфейс для получения команд от пользователя.

Методы:

- v
- virtual CommandType getCommandType() = 0 - определить тип команды
- virtual std::unique_ptr<Command> createCommand(CommandType type) = 0 - создать команду
- virtual std::pair<int, int> getSpellTarget() = 0 - получить цель заклинания
- virtual int getSpellChoice(int maxSpells) = 0 - выбрать заклинание
- virtual int getShopChoice(int maxSpells) = 0 - выбрать товар в магазине
- virtual void setShopSystem(ShopSystem& shopSystem) = 0 - установить магазин
- virtual void setSaveSystem(GameSaveSystem& saveSystem) = 0 - установить систему сохранения

n

d

Назначение: Обработчик ввода из консоли. Преобразует нажатия клавиш в объекты команд.

r

d

e

Поля:

указатель на систему магазина

Методы: указатель на систему сохранения

- C
- 6 C
- std::unique_ptr<Command> createCommand(CommandType type) - создаёт команду по типу
- m
- h
- d
- p
- void setSaveSystem(GameSaveSystem& saveSystem) - устанавливает систему сохранения
- p

- **Command (абстрактный класс команды)**

Назначение: Базовый класс для всех команд.

Методы:

- virtual bool execute(GameState& gameState) = 0 - выполнить команду
- virtual std::string getName() const = 0 - получить имя команды

Назначение: Команда перемещения игрока.

Поля: конструктор

Методы: смещение по X и Y

- MoveCommand(int deltaX, int deltaY) - конструктор

читает символ с клавиатуры

определяет тип команды по символу

- bool execute(GameState& gameState) override - выполнить перемещение
- std::string getName() const override - вернуть имя("MoveCommand")

Назначение: Команда применения заклинания.

Поля:

Методы: ссылка на обработчик ввода

- bool конструктор execute(GameState& gameState) override - выполнить применение заклинания
- std::string getName() const override - вернуть имя ("CastSpellCommand")

Назначение: Команда открытия магазина.

Поля:

ссылка на систему магазина

Методы: ссылка на обработчик ввода

- ShopCommand(ShopSystem& shop, InputHandler& handler) - конструктор
- bool execute(GameState& gameState) override - открыть магазин
- std::string getName() const override - вернуть имя ("ShopCommand")

Назначение: Команда сохранения игры.

Поля:

ссылка на систему сохранения

Методы:

- конструктор
- bool execute(GameState& gameState) override - выполнить сохранение
- std::string getName() const override - вернуть имя ("SaveCommand")

Назначение: Команда выхода из игры.

Методы:

- bool execute(GameState& gameState) override - выполнить выход
- std::string getName() const override - вернуть имя ("QuitCommand")

Назначение: Перечисление всех возможных команд.

Значения:

движение вверх (W)

движение вниз (S)

движение влево (A)
право (D)

применение заклинания (C)

открытие магазина (M)

сохранение игры (P)

выход из игры (Q)

неверная команда

Назначение: Интерфейс для отображения игрового состояния.

Методы:

виртуальный деструктор

- ➢ virtual void render(const GameState& gameState) const = 0 - отрисовать состояние игры
- ➢ virtual void showMessage(const std::string& message) const = 0 - показать сообщение

Назначение: Реализация для консольного вывода.

Методы:

конструктор

- ➢ void render(const GameState& gameState) const override - отрисовать игру в консоли
- ➢ void showMessage(const std::string& message) const override - вывести сообщение
- ➢ void updateFieldContent(GameField& field, const GameState& gameState) const - обновить содержимое поля (приватный)
- ➢ std::string getGameStateString(const GameState& gameState) const - получить строку состояния (приватный)

- **GameController<InputHandlerType> (шаблонный класс)**

Назначение: Посредник между вводом и игровой логикой. Получает команды и передаёт их на выполнение.

Поля:

ссылка на состояние игры

ссылка на обработчик действий

ссылка на систему магазина

ссылка на систему сохранения

- ❖ std::unique_ptr<InputHandlerType> inputHandler - обработчик ввода

флаг работы игры

Методы:

- GameController(GameState& state, ActionProcessor& processor, ShopSystem& shop, GameSaveSystem& saveSys) - конструктор

обработать ввод (получить команду и выполнить)

проверка, работает ли игра

остановить игру

- **GameVisualizer<RendererType> (шаблонный класс)**

Назначение: Посредник между игрой и отрисовкой. Реагирует на изменения в игре и вызывает отрисовку.

Поля:

- std::unique_ptr<RendererType> renderer- объект рендерера

Методы:

- конструктор
- void render(const GameState& gameState) const - отрисовать состояние
- void showMessage(const std::string& message) const - показать сообщение

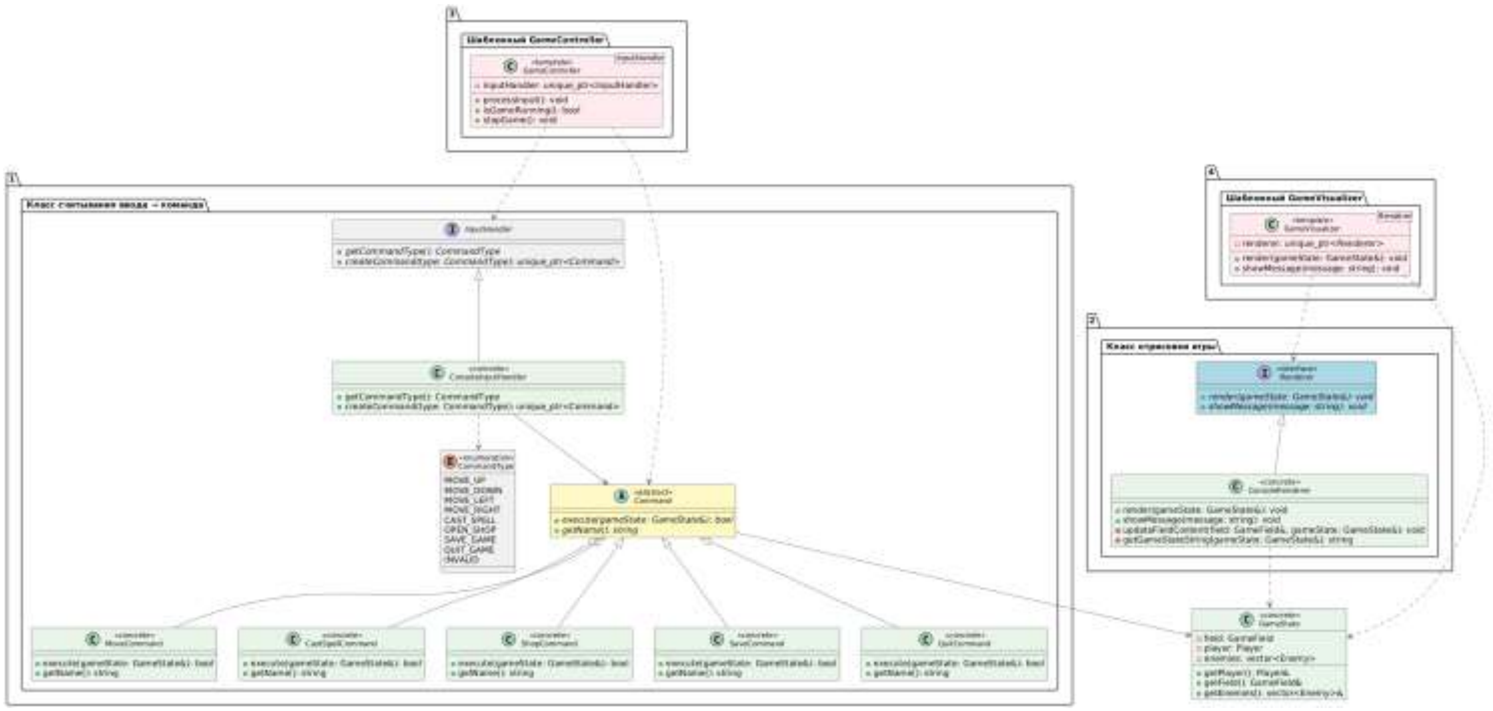

3
Шаблонный GameController
«template» GameController
InputHandler
- inputHandler: unique_ptr<InputHandler>
+ processInput(): void
+ isGameRunning(): bool
+ stopGame(): void
1
Класс считывания ввода → команда
InputHandler
+ getCommandType(): CommandType
+ createCommand(type: CommandType): unique_ptr<Command>
«concrete» ConsoleInputHandler
+ getCommandType(): CommandType
+ createCommand(type: CommandType): unique_ptr<Command>
«enumeration» CommandType
MOVE_UP
MOVE_DOWN
MOVE_LEFT
MOVE_RIGHT
CAST_SPELL
OPEN_SHOP
SAVE_GAME
QUIT_GAME
INVALID
«abstract» Command
+ execute(gameState: GameState&): bool
+ getName(): string
«concrete» MoveCommand
«concrete» CastSpellCommand
«concrete» ShopCommand
«concrete» SaveCommand
«concrete» QuitCommand
+ execute(gameState: GameState&): bool
+ getName(): string
4
Шаблонный GameVisualizer
«template» GameVisualizer
Renderer
- renderer: unique_ptr<Renderer>
+ render(gameState: GameState&): void
+ showMessage(message: string): void
2
Класс отрисовки игры
«interface» Renderer
+ render(gameState: GameState&): void
+ showMessage(message: string): void
«concrete» ConsoleRenderer
+ render(gameState: GameState&): void
+ showMessage(message: string): void
- updateFieldContent(field: GameField&, gameState: GameState&): void
- getGameStateString(gameState: GameState&): string
«concrete» GameState
- field: GameField
- player: Player
- enemies: vector<Enemy>
+ getPlayer(): Player&
+ getField(): GameField&
+ getEnemies(): vector<Enemy>&

Рисунок 1 - Диаграмма UML-классов

Разработанный программный код см. в приложении А.

Выводы.

В ходе выполнения лабораторной работы была успешно разработана объектно-ориентированная игра на C++, реализующая пошаговый игровой процесс. Была создана гибкая архитектура, разделяющая ответственность между компонентами ввода, отрисовки, управления и игровой логики. Применено шаблонное программирование для классов GameController и полным циклом: от главного меню до обработки победы или поражения. Класс GameManager координирует работу всех систем, обеспечивая целостность и стабильность игрового процесса.

ПРИЛОЖЕНИЕ А
ИСХОДНЫЙ КОД ПРОГРАММЫ

```
ActionProcessor.cpp
#include "ActionProcessor.h"
#include <iostream>

void ActionProcessor::processEnemyMoves(GameState& state) {
    physicalWorldSystem.processEnemyMoves(state);
}

bool ActionProcessor::processPlayerMove(GameState& state, int newX, int
newY) {
    return physicalWorldSystem.processPlayerMove(state, newX, newY);
}

ActionProcessor.h
#ifndef ACTIONPROCESSOR_H
#define ACTIONPROCESSOR_H

#include "GameState.h"
#include "InputHandler.h"
#include "ShopSystem.h"
#include "PhysicalWorldSystem.h"

class ActionProcessor {
private:
    PhysicalWorldSystem physicalWorldSystem;

public:

    void processEnemyMoves(GameState& state);

    bool processPlayerMove(GameState& state, int newX, int newY);
};

#endif


AreaDamageSpell.cpp
#include "AreaDamageSpell.h"
#include "GameField.h"
#include "SpellTarget.h"
#include "Enemy.h"
#include "Player.h"
#include <iostream>

AreaDamageSpell::AreaDamageSpell(const   std::string&   spellName,   int
cost, int spellRange, int spellDamage)
    : SpellBase(spellName, cost, spellRange, spellDamage) {
}

bool AreaDamageSpell::cast(const SpellTarget& target, GameField& field,
    std::vector<Enemy>& enemies, Player& player) {
    if (!validateTarget(target, field)) {
        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);
}

AreaDamageSpell.h
#ifndef AREADAMAGESPELL_H
#define AREADAMAGESPELL_H

#include "SpellBase.h"

class AreaDamageSpell : public SpellBase {
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;
};

#endif
```

CastSpellCommand.cpp

```
#include "CastSpellCommand.h"
#include "GameState.h"
#include "Player.h"
#include "GameField.h"
#include "Enemy.h"
#include "InputHandler.h"
#include <iostream>

CastSpellCommand::CastSpellCommand(InputHandler& handler)
    : inputHandler(handler) {
}

bool CastSpellCommand::execute(GameState& gameState) {
    auto& player = gameState.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()) {
        return false;
    }

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

std::string CastSpellCommand::getName() const {
    return "CastSpellCommand";
}
```

CastSpellCommand.h

```
#ifndef CASTSPELLCOMMAND_H
#define CASTSPELLCOMMAND_H

#include "Command.h"

class InputHandler;
class GameState;

class CastSpellCommand : public Command {
```

```
private:
    InputHandler& inputHandler;

public:
    CastSpellCommand(InputHandler& handler);
    bool execute(GameState& gameState) override;
    std::string getName() const override;
};

#endif
```

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
```

CellContent.h

```
#ifndef CELLCONTENT_H
#define CELLCONTENT_H

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

#endif
```

Command.h

```
#ifndef COMMAND_H
#define COMMAND_H

#include <string>
#include <memory>

class GameState;

class Command {
public:
    virtual ~Command() = default;
    virtual bool execute(GameState& gameState) = 0;
    virtual std::string getName() const = 0;
};

#endif
```

ConsoleInputHandler.cpp

```
#include "ConsoleInputHandler.h"
#include "ShopSystem.h"
#include "GameSaveSystem.h"
#include "Command.h"
#include "MoveCommand.h"
#include "CastSpellCommand.h"
#include "ShopCommand.h"
#include "SaveCommand.h"
#include "QuitCommand.h"
#include "CommandType.h"
#include <iostream>
#include <memory>
#include <cctype>

ConsoleInputHandler::ConsoleInputHandler()
    : shopSystem(nullptr), saveSystem(nullptr) {
}

CommandType ConsoleInputHandler::getCommandType() {
    char input = getRawInput();

    switch (std::tolower(static_cast<unsigned char>(input))) {
    case 'w': return CommandType::MOVE_UP;
    case 's': return CommandType::MOVE_DOWN;
    case 'a': return CommandType::MOVE_LEFT;
    case 'd': return CommandType::MOVE_RIGHT;
    case 'c': return CommandType::CAST_SPELL;
    case 'm': return CommandType::OPEN_SHOP;
    case 'p': return CommandType::SAVE_GAME;
    case 'q': return CommandType::QUIT_GAME;
    default:  return CommandType::INVALID;
    }
}

std::unique_ptr<Command> ConsoleInputHandler::createCommand(CommandType
type) {
    switch (type) {
    case CommandType::MOVE_UP:
        return std::make_unique<MoveCommand>(0, -1);
    case CommandType::MOVE_DOWN:
        return std::make_unique<MoveCommand>(0, 1);
    case CommandType::MOVE_LEFT:
        return std::make_unique<MoveCommand>(-1, 0);
    case CommandType::MOVE_RIGHT:
        return std::make_unique<MoveCommand>(1, 0);
    case CommandType::CAST_SPELL:
        return std::make_unique<CastSpellCommand>(*this);
    case CommandType::OPEN_SHOP:
        if (shopSystem)
            return std::make_unique<ShopCommand>(*shopSystem, *this);
        break;
    case CommandType::SAVE_GAME:
        if (saveSystem)
            return std::make_unique<SaveCommand>(*saveSystem);
        break;
    case CommandType::QUIT_GAME:
        return std::make_unique<QuitCommand>();
    case CommandType::INVALID:
        break;
    }
    return nullptr;
}

char ConsoleInputHandler::getRawInput() {
    char input;
    std::cout << "Enter command (WASD=move, C=spells, M=shop, P=save,
Q=quit): ";

    if (!(std::cin >> input)) {
        std::cin.clear();
        std::cin.ignore(std::numeric_limits<std::streamsize>::max(),
'\n');
        std::cout << "Input error!" << std::endl;
        return 0;
    }

    std::cin.ignore(std::numeric_limits<std::streamsize>::max(), '\n');
    return input;
}

std::pair<int, int> ConsoleInputHandler::getSpellTarget() {
    int targetX, targetY;
    std::cout << "Enter target coordinates (x y): ";
    std::cin >> targetX >> targetY;
    std::cin.ignore(std::numeric_limits<std::streamsize>::max(), '\n');
    return { targetX, targetY };
}
```

```
int ConsoleInputHandler::getSpellChoice(int maxSpells) {
    int choice;
    std::cout << "Choose spell (1-" << maxSpells << "): ";
    std::cin >> choice;
    std::cin.ignore(std::numeric_limits<std::streamsize>::max(), '\n');
    return choice;
}

int ConsoleInputHandler::getShopChoice(int maxSpells) {
    int choice;
    std::cout << "Choose spell to buy (1-" << maxSpells << ") or 0 to
cancel: ";
    std::cin >> choice;
    std::cin.ignore(std::numeric_limits<std::streamsize>::max(), '\n');
    return choice;
}

void ConsoleInputHandler::setShopSystem(ShopSystem& shopSystem) {
    this->shopSystem = &shopSystem;
}

void ConsoleInputHandler::setSaveSystem(GameSaveSystem& saveSystem) {
    this->saveSystem = &saveSystem;
}
```

ConsoleInputHandler.h

```
#ifndef CONSOLEINPUTHANDLER_H
#define CONSOLEINPUTHANDLER_H

#include "InputHandler.h"
#include <memory>
#include <limits>

class ConsoleInputHandler : public InputHandler {
private:
    ShopSystem* shopSystem;
    GameSaveSystem* saveSystem;

    char getRawInput();

public:
    ConsoleInputHandler();

    // НОВЫЕ МЕТОДЫ:
    CommandType getCommandType() override;
    std::unique_ptr<Command> createCommand(CommandType type) override;

    // УДАЛИТЬ: char getGameInput() override; // Этого метода больше нет
в InputHandler

    // Старые методы:
    std::pair<int, int> getSpellTarget() override;
    int getSpellChoice(int maxSpells) override;
    int getShopChoice(int maxSpells) override;
    void setShopSystem(ShopSystem& shopSystem) override;
    void setSaveSystem(GameSaveSystem& saveSystem) override;
};

#endif // CONSOLEINPUTHANDLER_H
```

ConsoleRenderer.cpp

```
#include "ConsoleRenderer.h"
#include "GameState.h"
#include "CellContent.h"
#include <iostream>

void ConsoleRenderer::updateFieldContent(GameField& field, const
GameState& gameState) const {
    field.clearAllCells();

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

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

std::string ConsoleRenderer::getGameStateString(const GameState&
gameState) const {
    std::stringstream ss;

    const auto& player = gameState.getPlayer();
    auto playerPos = gameState.getField().getPlayerPosition();

    ss << "Health: " << player.getHealth();
    ss << " | Mana: " << player.getMana() << "/" << player.getMaxMana();
    ss << " | Score: " << player.getScore();
    ss << " | Position: (" << playerPos.first << "," << playerPos.second
<< ")\n\n";

    auto& field = const_cast<GameField&>(gameState.getField());
    updateFieldContent(field, gameState);

    for (int y = 0; y < field.getHeight(); y++) {
        for (int x = 0; x < field.getWidth(); x++) {
            ss << field.getCellContent(x, y) << " ";
        }
        ss << "\n";
    }

    ss << "\n=== Spells ===\n";
    const auto& spellHand = player.getSpellHand();
    if (spellHand.getSpellCount() > 0) {
        for (int i = 0; i < spellHand.getSpellCount(); i++) {
            ss << i + 1 << ". " << spellHand.getSpellDescription(i) <<
"\n";
        }
    }
    else {
        ss << "No spells\n";
    }

    ss << "\n=== Enemies ===\n";
    const auto& enemies = gameState.getEnemies();
    int aliveEnemies = 0;
    for (const auto& enemy : enemies) {
        if (enemy.isAlive()) {
            aliveEnemies++;
            ss << "Enemy at (" << enemy.getX() << "," << enemy.getY()
                << ") HP: " << enemy.getHealth() << "\n";
        }
    }

    if (aliveEnemies == 0) {
        ss << "No enemies\n";
    }

    return ss.str();
}

void ConsoleRenderer::render(const GameState& gameState) const {
    std::string gameStateStr = getGameStateString(gameState);
    std::cout << gameStateStr << std::endl;
}

void ConsoleRenderer::showMessage(const std::string& message) const {
    std::cout << message << std::endl;
}
```

ConsoleRenderer.h

```
#ifndef CONSOLERENDERER_H
#define CONSOLERENDERER_H

#include "Renderer.h"
#include "GameState.h"
#include "CellContent.h"
#include <sstream>
#include <string>

class ConsoleRenderer : public Renderer {
private:
    void   updateFieldContent(GameField&   field,   const   GameState&
gameState) const;
    std::string getGameStateString(const GameState& gameState) const;

public:
    ConsoleRenderer() = default;
    void render(const GameState& gameState) const override;
    void showMessage(const std::string& message) const override;
};

#endif
```

DirectDamageSpell.cpp

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

DirectDamageSpell::DirectDamageSpell(const std::string& spellName, int
cost, int spellRange, int spellDamage)
    : SpellBase(spellName, cost, spellRange, spellDamage) {
}

bool DirectDamageSpell::cast(const SpellTarget& target, GameField&
field,
    std::vector<Enemy>& enemies, Player& player) {
    if (!validateTarget(target, field)) {
        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);
}
```

DirectDamageSpell.h

```
#ifndef DIRECTDAMAGESPELL_H
#define DIRECTDAMAGESPELL_H

#include "SpellBase.h"

class DirectDamageSpell : public SpellBase {
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;
};

#endif
```

Direction.h

```
#ifndef DIRECTION_H
#define DIRECTION_H

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

#endif
```

Enemy.cpp

```
#include "Enemy.h"
#include "GameConstants.h"

Enemy::Enemy(int startX, int startY)
    :                         Entity(GameConstants::ENEMY_START_HEALTH,
GameConstants::ENEMY_DAMAGE),
    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;
}

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);

    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);
    ~Entity() = default;

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

#endif


FileNotFoundException.h
#ifndef FILENOTFOUNDEXCEPTION_H
#define FILENOTFOUNDEXCEPTION_H

#include "SaveLoadException.h"

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

#endif


FileWriteException.h
#ifndef FILEWRITEEXCEPTION_H
#define FILEWRITEEXCEPTION_H

#include "SaveLoadException.h"

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

#endif
```

GameConstants.h

```
#ifndef GAMECONSTANTS_H
#define GAMECONSTANTS_H

namespace GameConstants {
    const int PLAYER_START_HEALTH = 100;
    const int PLAYER_START_MANA = 50;
    const int PLAYER_MAX_MANA = 100;
    const int PLAYER_DAMAGE = 10;
    const int PLAYER_START_SCORE = 0;

    const int ENEMY_START_HEALTH = 30;
    const int ENEMY_DAMAGE = 5;

    const int SCORE_FOR_MOVE = 1;
    const int SCORE_FOR_KILL = 10;
    const int MANA_FOR_MOVE = 5;

    const int FIELD_WIDTH = 15;
    const int FIELD_HEIGHT = 15;

    const int STARTER_FIREBALL_COST = 8;
    const int STARTER_FIREBALL_RANGE = 2;
    const int STARTER_FIREBALL_DAMAGE = 15;

    const int STARTER_FIRESTORM_COST = 12;
    const int STARTER_FIRESTORM_RANGE = 1;
    const int STARTER_FIRESTORM_DAMAGE = 8;

    const int ICE_ARROW_COST = 40;
    const int ICE_ARROW_RANGE = 4;
    const int ICE_ARROW_DAMAGE = 20;

    const int ICE_STORM_COST = 90;
    const int ICE_STORM_RANGE = 3;
    const int ICE_STORM_DAMAGE = 12;
}

#endif
```

GameController.h

```
#ifndef GAMECONTROLLER_H
#define GAMECONTROLLER_H

#include "GameState.h"
#include "InputHandler.h"
#include "ActionProcessor.h"
#include "Command.h"
#include "ShopSystem.h"
#include "GameSaveSystem.h"
#include <memory>
#include <iostream>

template<typename InputHandlerType>
class GameController {
```

```
private:
    GameState& gameState;
    ActionProcessor& actionProcessor;
    ShopSystem& shopSystem;
    GameSaveSystem& saveSystem;
    std::unique_ptr<InputHandlerType> inputHandler;
    bool isRunning;

public:
    explicit    GameController(GameState&    state,    ActionProcessor&
processor,
        ShopSystem& shop, GameSaveSystem& saveSys)
        : gameState(state),
        actionProcessor(processor),
        shopSystem(shop),
        saveSystem(saveSys),
        inputHandler(std::make_unique<InputHandlerType>()),
        isRunning(true) {
        inputHandler->setShopSystem(shopSystem);
        inputHandler->setSaveSystem(saveSystem);
    }

    void processInput() {
        auto command = inputHandler->getCommand();

        if (!command) {
            std::cout << "Invalid command!" << std::endl;
            return;
        }

        bool success = command->execute(gameState);

        if (success) {
            std::cout << "Command executed successfully." << std::endl;
        }
        else {
            std::cout << "Command failed!" << std::endl;
        }

        if (command->getName() == "QuitCommand") {
            stopGame();
        }
    }

    bool isGameRunning() const {
        return isRunning;
    }

    void stopGame() {
        isRunning = false;
    }
};

#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 <iostream>
```

```
#include "InputHandler.h"
#include "Command.h"

GameLogic::GameLogic(GameState&  state,  ActionProcessor&  processor,
ShopSystem& shop)
    : gameState(state), actionProcessor(processor), shopSystem(shop),
    gameRunning(false), playerTurn(true) {
}

void GameLogic::startGame() {
    gameRunning = true;
    playerTurn = true;
    std::cout << "Game started!" << std::endl;
}

void GameLogic::stopGame() {
    gameRunning = false;
    std::cout << "Game stopped." << std::endl;
}

bool GameLogic::isGameRunning() const {
    return gameRunning;
}

bool    GameLogic::processPlayerAction(char    input,    InputHandler&
inputHandler) {
    if (!playerTurn) {
        std::cout << "Not your turn!" << std::endl;
        return false;
    }

    // Получаем команду от InputHandler
    auto command = inputHandler.getCommand();

    if (!command) {
        std::cout << "Invalid command!" << std::endl;
        return false;
    }

    // Выполняем команду
    bool actionSuccess = command->execute(gameState);

    if (actionSuccess) {
        playerTurn = false;
        std::cout << "Player turn ended." << std::endl;

        // Если это команда выхода, останавливаем игру
        if (command->getName() == "QuitCommand") {
            gameRunning = false;
        }
    }

    return actionSuccess;
}

void GameLogic::processEnemyTurn() {
    if (playerTurn) {
        return;
    }

    std::cout << "=== ENEMY TURN ===" << std::endl;
    actionProcessor.processEnemyMoves(gameState);

    if (!isPlayerAlive()) {
        std::cout << "Player was defeated by enemies!" << std::endl;
        gameRunning = false;
    }
    else {
        playerTurn = true;
        std::cout << "Enemy turn ended. Player's turn." << std::endl;
    }
}

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

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

bool GameLogic::checkPlayerVictory() const {
    const auto& enemies = gameState.getEnemies();
    for (const auto& enemy : enemies) {
        if (enemy.isAlive()) {
            return false;
        }
    }
    return true;
}

bool GameLogic::isGameOver() const {
    return !isPlayerAlive() || checkPlayerVictory();
}

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

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

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

int GameLogic::getPlayerHealth() const {
    return gameState.getPlayer().getHealth();
}

int GameLogic::getPlayerMana() const {
    return gameState.getPlayer().getMana();
}

void GameLogic::reset() {
    playerTurn = true;
    gameRunning = false;
}

GameLogic.h
#ifndef GAMELOGIC_H
#define GAMELOGIC_H

#include "GameState.h"
#include "ActionProcessor.h"
#include "ShopSystem.h"
#include "InputHandler.h"
#include <memory>
#include <iostream>

class InputHandler;
class Command;


class GameLogic {
private:
    GameState& gameState;
    ActionProcessor& actionProcessor;
    ShopSystem& shopSystem;
    bool gameRunning;
    bool playerTurn;

public:
    GameLogic(GameState& state, ActionProcessor& processor, ShopSystem&
shop);

    void startGame();
    void stopGame();
    bool isGameRunning() const;

    bool processPlayerAction(char input, InputHandler& inputHandler);
    void processEnemyTurn();

    bool isPlayerTurn() const;
    bool isPlayerAlive() const;
    bool checkPlayerVictory() const;
    bool isGameOver() const;

    GameState& getGameState();
    const GameState& getGameState() const;
    int getPlayerScore() const;
    int getPlayerHealth() const;
    int getPlayerMana() const;

    void reset();
};

#endif



GameManager.cpp
#include "GameManager.h"
```

```cpp
#include <iostream>

GameManager::GameManager()
    : gameLogic(gameState, actionProcessor, shopSystem),
    controller(gameState, actionProcessor, shopSystem, saveSystem),
    visualizer() {
}

void GameManager::run() {
    showMainMenu();
}

void GameManager::showMainMenu() {
    int choice;
    do {
        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: ";
        std::cin >> choice;

        switch (choice) {
        case 1:
            startNewGame();
            break;
        case 2:
            loadGame();
            break;
        case 3:
            std::cout << "Goodbye!" << std::endl;
            return;
        default:
            std::cout << "Invalid option!" << std::endl;
        }
    } while (choice != 3);
}

void GameManager::startNewGame() {
    gameState.initializeNewGame();
    gameLogic.reset();
    gameLogic.startGame();
    std::cout << "New game started!" << std::endl;
    runGameLoop();
}

void GameManager::loadGame() {
    try {
        if (saveSystem.loadGame(gameState)) {
            gameLogic.reset();
            gameLogic.startGame();
            std::cout << "Game loaded successfully!" << std::endl;
            runGameLoop();
        }
    }
    catch (const SaveLoadException& e) {
        std::cout << "ERROR: " << e.what() << std::endl;
        std::cout << "Starting new game instead..." << std::endl;
        startNewGame();
    }
}

void GameManager::runGameLoop() {
    visualizer.showMessage("Game Started! Use WASD to move, C to cast
spells, M for shop, P to save, Q to quit");
    visualizer.render(gameState);

    while (controller.isGameRunning() && gameLogic.isGameRunning()) {
        try {
            if (gameLogic.checkPlayerVictory()) {
                handleVictory();
                break;
            }

            visualizer.showMessage("=== YOUR TURN ===");
            controller.processInput();

            visualizer.showMessage("=== ENEMY TURN ===");
            actionProcessor.processEnemyMoves(gameState);

            if (!gameLogic.isPlayerAlive()) {
                handleGameOver();
                break;
            }

            visualizer.render(gameState);
        }
        catch (const std::exception& e) {
            std::cout << "Error in game loop: " << e.what() << std::endl;
        }
    }
}

void GameManager::handleGameOver() {
    visualizer.showMessage("GAME OVER! You were defeated!");
    visualizer.showMessage("Final          score:          "          +
std::to_string(gameLogic.getPlayerScore()));
    showMainMenu();
}

void GameManager::handleVictory() {
    visualizer.showMessage("VICTORY! All enemies defeated!");
    visualizer.showMessage("Final          score:          "          +
std::to_string(gameLogic.getPlayerScore()));
    showMainMenu();
}


GameManager.h
#ifndef GAMEMANAGER_H
#define GAMEMANAGER_H

#include "GameState.h"
#include "GameLogic.h"
#include "GameController.h"
#include "ConsoleRenderer.h"
```

```
#include "GameVisualizer.h"
#include "ConsoleInputHandler.h"
#include "ActionProcessor.h"
#include "ShopSystem.h"
#include "GameSaveSystem.h"
#include "SaveLoadException.h"
#include <memory>

class GameManager {
private:
    GameState gameState;
    ActionProcessor actionProcessor;
    ShopSystem shopSystem;
    GameSaveSystem saveSystem;
    GameLogic gameLogic;
    GameController<ConsoleInputHandler> controller;
    GameVisualizer<ConsoleRenderer> visualizer;

    void handleGameOver();
    void handleVictory();
    void showMainMenu();
    void startNewGame();
    void loadGame();
    void runGameLoop();

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

#endif // GAMEMANAGER_H
```

GameSaveSystem.cpp

```
#include "GameSaveSystem.h"
#include "GameStateSerializer.h"
#include "FileNotFoundException.h"
#include "FileWriteException.h"
#include <iostream>
#include <fstream>
#include <sstream>
#include <memory>

GameSaveSystem::GameSaveSystem()
    : serializer(std::make_unique<GameStateSerializer>()) {
}

GameSaveSystem::GameSaveSystem(const std::string& saveFileName)
    : defaultSaveFile(saveFileName),
    serializer(std::make_unique<GameStateSerializer>()) {
}

GameSaveSystem::~GameSaveSystem() = default;

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);
    if (!file.is_open()) {
        throw FileWriteException(filename);
    }

    try {
        std::stringstream buffer;
        serializer->serialize(gameState, buffer);

        std::string data = buffer.str();
        int checksum = calculateChecksum(data); // Вычисляем контрольную
сумму

        file << data;
        file << "CHECKSUM:" << checksum << std::endl;

        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);
    if (!file.is_open()) {
        throw FileNotFoundException(filename);
    }

    try {
        std::stringstream buffer;
        buffer << file.rdbuf();
        std::string data = buffer.str();

        // Проверяем контрольную сумму на целостность данных
        if (!validateChecksum(data)) {
            std::cout << "ERROR: Save file has been modified! Loading
blocked." << std::endl;
            throw  SaveLoadException("Save  file  corrupted  or  invalid
checksum");
        }
        buffer.seekg(0);
        serializer->deserialize(gameState, buffer);

        std::cout << "Game loaded successfully from: " << filename <<
std::endl;
        return true;
    }
    catch (const SaveLoadException&) {
        throw;
    }
    catch (const std::exception& e) {
        throw   SaveLoadException("Failed   to   load   game:   "   +
std::string(e.what()));
    }
}

// Метод для расчета контрольной суммы строки данных
int GameSaveSystem::calculateChecksum(const std::string& data) const {
    int sum = 0;
    for (char c : data) {
        sum = (sum * 31 + static_cast<int>(c)) % 100000;
    }
    return sum;
}

// Метод для проверки контрольной суммы данных
bool GameSaveSystem::validateChecksum(const std::string& data) const {
    size_t checksumPos = data.find("CHECKSUM:");
    if (checksumPos == std::string::npos) {
        return false;
    }

    std::string gameData = data.substr(0, checksumPos);
    std::string checksumStr = data.substr(checksumPos + 9);

    try {
        int savedChecksum = std::stoi(checksumStr);
        int calculatedChecksum = calculateChecksum(gameData);
        return savedChecksum == calculatedChecksum;
    }
    catch (...) {
        return false;
    }
}

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

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

    try {
        std::string firstLine;
        std::getline(file, firstLine);
        return firstLine.find("GAME_SAVE") != std::string::npos;
    }
    catch (...) {
        return false;
    }
}

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 <memory>

class GameStateSerializer;

class GameSaveSystem {
private:
    std::string defaultSaveFile = "game_save.txt";
    std::unique_ptr<GameStateSerializer> serializer;

    int calculateChecksum(const std::string& data) const;
    bool validateChecksum(const std::string& data) const;

public:
    GameSaveSystem();
    explicit GameSaveSystem(const std::string& saveFileName);
    ~GameSaveSystem();

    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;
```

```
    bool saveExists() const;
    bool saveExists(const std::string& filename) const;

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

#endif
```

GameState.cpp

```
#include "GameState.h"
#include "GameConstants.h"
#include <stdexcept>

GameState::GameState()
    : field(GameConstants::FIELD_WIDTH, GameConstants::FIELD_HEIGHT) {
}

void GameState::initializeNewGame() {
    enemies.clear();
    field.clearAllCells();
    field.setPlayerPosition(0, 0);

    player.setHealth(GameConstants::PLAYER_START_HEALTH);
    player.setMana(GameConstants::PLAYER_START_MANA);
    player.setMaxMana(GameConstants::PLAYER_MAX_MANA);
    player.setScore(GameConstants::PLAYER_START_SCORE);

    enemies.push_back(Enemy(5, 5));
    enemies.push_back(Enemy(10, 10));

    resetTurns();
}

void GameState::initializeTestGame() {
    initializeNewGame();
}

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

bool GameState::isValidMove(int newX, int newY) const {
    if (!field.isValidPosition(newX, newY)) {
        return false;
    }

    // Проверяем, не занята ли клетка врагом (для атаки)
    for (const auto& enemy : enemies) {
        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 GameState::canPlayerCastSpell(int spellIndex) const {
    if      (spellIndex      <      0      ||      spellIndex      >=
player.getSpellHand().getSpellCount()) {
        return false;
    }
    int manaCost = player.getSpellHand().getSpellManaCost(spellIndex);
    return player.getMana() >= manaCost;
}
```

GameState.h

```
#ifndef GAMESTATE_H
#define GAMESTATE_H

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

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

public:
    GameState();

    void initializeNewGame();
    void initializeTestGame();

    bool isPlayerTurn() const { return playerTurn; }
    void endPlayerTurn() { playerTurn = false; }
    void endEnemyTurn() { playerTurn = true; }
    void resetTurns() { playerTurn = true; }

    bool checkPlayerVictory() const;
    bool isPlayerAlive() const { return player.isAlive(); }
    bool isValidMove(int newX, int newY) const;
    bool canPlayerCastSpell(int spellIndex) const;

    GameField& getField() { return field; }
    Player& getPlayer() { return player; }
    std::vector<Enemy>& getEnemies() { return enemies; }

    const GameField& getField() const { return field; }
    const Player& getPlayer() const { return player; }
    const std::vector<Enemy>& getEnemies() const { return enemies; }
```

```
};

#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>

class GameStateSerializer {
public:
    void  serialize(const  GameState&  gameState,  std::ostream&  stream)
const;
    void deserialize(GameState& gameState, std::istream& stream) const;
};

#endif


GameVisualizer.h
#ifndef GAMEVISUALIZER_H
#define GAMEVISUALIZER_H

#include "Renderer.h"
#include <memory>

template<typename RendererType>
class GameVisualizer {
```

```cpp
private:
    std::unique_ptr<RendererType> renderer;

public:
    GameVisualizer();

    void render(const GameState& gameState) const;
    void showMessage(const std::string& message) const;
};

template<typename RendererType>
GameVisualizer<RendererType>::GameVisualizer()
    : renderer(std::make_unique<RendererType>()) {
}

template<typename RendererType>
void GameVisualizer<RendererType>::render(const GameState& gameState)
const {
    renderer->render(gameState);
}

template<typename RendererType>
void    GameVisualizer<RendererType>::showMessage(const    std::string&
message) const {
    renderer->showMessage(message);
}

#endif
```

main.cpp

```cpp
#include "GameManager.h"

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

InputHandler.h

```cpp
#ifndef INPUTHANDLER_H
#define INPUTHANDLER_H

#include "Command.h"
#include "CommandType.h"
#include <utility>
#include <memory>

class ShopSystem;
class GameSaveSystem;

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

    // НОВЫЕ МЕТОДЫ:
```

```
    virtual CommandType getCommandType() = 0;                    // <-
ВМЕСТО getCommand()
    virtual std::unique_ptr<Command> createCommand(CommandType type) =
0; // <- ВМЕСТО getCommand()

    // Старые методы остаются:
    virtual std::pair<int, int> getSpellTarget() = 0;
    virtual int getSpellChoice(int maxSpells) = 0;
    virtual int getShopChoice(int maxSpells) = 0;

    virtual void setShopSystem(ShopSystem& shopSystem) = 0;
    virtual void setSaveSystem(GameSaveSystem& saveSystem) = 0;
};

#endif // INPUTHANDLER_H
```

MoveCommand.cpp

```
#include "MoveCommand.h"
#include "GameState.h"
#include "GameField.h"
#include "Player.h"
#include "Enemy.h"
#include "GameConstants.h"
#include <iostream>

MoveCommand::MoveCommand(int deltaX, int deltaY)
    : dx(deltaX), dy(deltaY) {
}

bool MoveCommand::execute(GameState& gameState) {
    auto playerPos = gameState.getField().getPlayerPosition();
    int newX = playerPos.first + dx;
    int newY = playerPos.second + dy;

    auto& field = gameState.getField();
    if (!field.isValidPosition(newX, newY)) {
        return false;
    }

    auto& enemies = gameState.getEnemies();
    for (auto& enemy : enemies) {
        if (enemy.isAlive() && enemy.getX() == newX && enemy.getY() ==
newY) {
            enemy.takeDamage(gameState.getPlayer().getDamage());
            if (!enemy.isAlive()) {

gameState.getPlayer().addScore(GameConstants::SCORE_FOR_KILL);
            }
            return true;
        }
    }

    if (field.isCellEmpty(newX, newY)) {
        field.movePlayer(newX, newY);
        gameState.getPlayer().addScore(GameConstants::SCORE_FOR_MOVE);
        gameState.getPlayer().addMana(GameConstants::MANA_FOR_MOVE);
        return true;
    }
    return false;
}

std::string MoveCommand::getName() const {
    if (dx == 0 && dy == -1) return "MoveCommand(UP)";
    if (dx == 0 && dy == 1)  return "MoveCommand(DOWN)";
    if (dx == -1 && dy == 0) return "MoveCommand(LEFT)";
    if (dx == 1 && dy == 0)  return "MoveCommand(RIGHT)";
    return "MoveCommand(UNKNOWN)";
}
```

MoveCommand.h

```
#ifndef MOVECOMMAND_H
#define MOVECOMMAND_H

#include "Command.h"

class MoveCommand : public Command {
private:
    int dx, dy;
public:
    MoveCommand(int deltaX, int deltaY);
    bool execute(GameState& gameState) override;
    std::string getName() const override;
};

#endif
```

PhysicalWorldSystem.cpp

```
#include "PhysicalWorldSystem.h"
#include "CellContent.h"
bool PhysicalWorldSystem::processPlayerMove(GameState& state, int newX,
int newY) {
    auto& field = state.getField();
    auto& player = state.getPlayer();

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

    if (attackEnemyAt(state, newX, newY)) {
        return true;
    }

    if (movePlayerTo(state, newX, newY)) {
        player.addScore(GameConstants::SCORE_FOR_MOVE);
        player.addMana(GameConstants::MANA_FOR_MOVE);
        return true;
    }

    return false;
}
```

```
bool PhysicalWorldSystem::attackEnemyAt(GameState& state, int x, int y)
{
    auto& enemies = state.getEnemies();
    auto& player = state.getPlayer();

    for (auto& enemy : enemies) {
        if (enemy.isAlive() && enemy.getX() == x && enemy.getY() == y)
{
            enemy.takeDamage(player.getDamage());
            std::cout << "You attacked enemy! Enemy health: "
                << enemy.getHealth() << std::endl;

            if (!enemy.isAlive()) {
                player.addScore(GameConstants::SCORE_FOR_KILL);
                std::cout << "Enemy defeated! +10 points" << std::endl;
            }
            return true;
        }
    }
    return false;
}

bool PhysicalWorldSystem::movePlayerTo(GameState& state, int newX, int
newY) {
    return state.getField().movePlayer(newX, newY);
}

void PhysicalWorldSystem::processEnemyMoves(GameState& state) {
    auto& enemies = state.getEnemies();

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

        // Даже если враг рядом с игроком, он может попытаться
переместиться
        // Например, с 50% вероятностью атаковать, с 50% попытаться
двигаться
        if (enemyCanAttackPlayer(state, enemy)) {
            // Случайно решаем: атаковать или двигаться
            if (std::rand() % 2 == 0) {
                // Атаковать
                auto& player = state.getPlayer();
                player.takeDamage(enemy.getDamage());
                std::cout << "Enemy attacked you! Lost "
                    << enemy.getDamage() << " health!" << std::endl;
            }
            else {
                // Попытаться переместиться (может попасть на игрока и
нанести урон)
                moveEnemyRandomly(state, enemy);
            }
        }
        else {
            // Враг далеко - просто двигаемся
            moveEnemyRandomly(state, enemy);
        }
    }
}

void  PhysicalWorldSystem::moveEnemyRandomly(GameState&  state,  Enemy&
enemy) {
    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;  // Вправо
    case 1: newX--; break;  // Влево
    case 2: newY++; break;  // Вниз
    case 3: newY--; break;  // Вверх
    }

    auto& field = state.getField();

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

    char cellContent = field.getCellContent(newX, newY);

    // Если на клетке игрок
    if (cellContent == CellContent::PLAYER) {
        auto& player = state.getPlayer();
        player.takeDamage(enemy.getDamage());
        std::cout << "Enemy tried to move onto you and attacked! Lost "
            << enemy.getDamage() << " health!" << std::endl;
        return;  // Перемещение не происходит
    }

    // Если клетка пустая
    if (cellContent == CellContent::EMPTY) {
        enemy.move(newX, newY);
    }

    // Если клетка занята врагом (CellContent::ENEMY) - не двигаемся
}

bool  PhysicalWorldSystem::enemyCanAttackPlayer(const  GameState&  state,
const Enemy& enemy) {
    auto playerPos = state.getField().getPlayerPosition();
    int dx = std::abs(enemy.getX() - playerPos.first);
    int dy = std::abs(enemy.getY() - playerPos.second);
    return dx <= 1 && dy <= 1;
}

PhysicalWorldSystem.h
#ifndef PHYSICALWORLDSYSTEM_H
#define PHYSICALWORLDSYSTEM_H

#include "GameState.h"
#include "GameConstants.h"
#include <iostream>
#include <cstdlib>
```

```
class PhysicalWorldSystem {
public:
    bool processPlayerMove(GameState& state, int newX, int newY);

    void processEnemyMoves(GameState& state);

private:
    bool attackEnemyAt(GameState& state, int x, int y);
    bool movePlayerTo(GameState& state, int newX, int newY);
    void moveEnemyRandomly(GameState& state, Enemy& enemy);
    bool  enemyCanAttackPlayer(const  GameState&  state,  const  Enemy&
enemy);
};

#endif

Player.cpp
#include "Player.h"
#include "Spell.h"
#include "GameField.h"
#include "DirectDamageSpell.h"
#include "AreaDamageSpell.h"
#include "Enemy.h"
#include "SpellTarget.h"
#include "GameConstants.h"
#include <iostream>
#include <memory>
#include <cstdlib>
#include <ctime>

Player::Player()
    :                       Entity(GameConstants::PLAYER_START_HEALTH,
GameConstants::PLAYER_DAMAGE),
    score(GameConstants::PLAYER_START_SCORE),
    mana(GameConstants::PLAYER_START_MANA),
    maxMana(GameConstants::PLAYER_MAX_MANA),
    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",
            GameConstants::STARTER_FIREBALL_COST,
            GameConstants::STARTER_FIREBALL_RANGE,
            GameConstants::STARTER_FIREBALL_DAMAGE
        );
    }
    else {
        return std::make_unique<AreaDamageSpell>(
            "Starter Fire Storm",
            GameConstants::STARTER_FIRESTORM_COST,
            GameConstants::STARTER_FIRESTORM_RANGE,
            GameConstants::STARTER_FIRESTORM_DAMAGE
        );
    }
}

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);
}

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);

    void setHealth(int newHealth);
    void setMana(int newMana);
    void setMaxMana(int newMaxMana);
    void setScore(int newScore);
    void clearSpellHand();
};

#endif
```

QuitCommand.cpp

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

bool QuitCommand::execute(GameState& gameState) {
    std::cout << "Quitting game..." << std::endl;
    return true;
}

std::string QuitCommand::getName() const {
    return "QuitCommand";
}
```

QuitCommand.h

```
#ifndef QUITCOMMAND_H
#define QUITCOMMAND_H

#include "Command.h"

class QuitCommand : public Command {
public:
    bool execute(GameState& gameState) override;
    std::string getName() const override;
};

#endif
```

Renderer.h

```
#ifndef RENDERER_H
#define RENDERER_H
```

```
#include <string>

class GameState;

class Renderer {
public:
    virtual ~Renderer() = default;
    virtual  void  render(const  GameState&  gameState)  const  =  0;    //
ОСНОВНОЙ МЕТОД
    virtual void showMessage(const std::string& message) const = 0;
};

#endif


SaveCommand.cpp
#include "SaveCommand.h"
#include "GameSaveSystem.h"
#include <iostream>

SaveCommand::SaveCommand(GameSaveSystem& save)
    : saveSystem(save) {
}

bool SaveCommand::execute(GameState& gameState) {
    try {
        saveSystem.saveGame(gameState);
        std::cout << "Game saved successfully!" << std::endl;
        return true;
    }
    catch (const std::exception& e) {
        std::cout << "ERROR saving game: " << e.what() << std::endl;
        return false;
    }
}

std::string SaveCommand::getName() const {
    return "SaveCommand";
}


SaveCommand.h
#ifndef SAVECOMMAND_H
#define SAVECOMMAND_H

#include "Command.h"

class GameSaveSystem;
class GameState;

class SaveCommand : public Command {
private:
    GameSaveSystem& saveSystem;

public:
    SaveCommand(GameSaveSystem& save);
    bool execute(GameState& gameState) override;
    std::string getName() const override;
};

#endif
```

SaveLoadException.h

```
#ifndef SAVELOADEXCEPTION_H
#define SAVELOADEXCEPTION_H

#include <stdexcept>
#include <string>

class SaveLoadException : public std::runtime_error {
public:
    explicit SaveLoadException(const std::string& message)
        : std::runtime_error("Save/Load Error: " + message) {}
};

#endif
```

ShopCommand.cpp

```
#include "ShopCommand.h"
#include "GameState.h"
#include "ShopSystem.h"
#include "Player.h"

ShopCommand::ShopCommand(ShopSystem& shop, InputHandler& handler)
    : shopSystem(shop), inputHandler(handler) {
}

bool ShopCommand::execute(GameState& gameState) {
    return         shopSystem.interactWithPlayer(gameState.getPlayer(),
inputHandler);
}

std::string ShopCommand::getName() const {
    return "ShopCommand";
}
```

ShopCommand.h

```
#ifndef SHOPCOMMAND_H
#define SHOPCOMMAND_H

#include "Command.h"

class ShopSystem;
class InputHandler;
class GameState;

class ShopCommand : public Command {
private:
    ShopSystem& shopSystem;
    InputHandler& inputHandler;
public:
    ShopCommand(ShopSystem& shop, InputHandler& handler);
    bool execute(GameState& gameState) override;
    std::string getName() const override;
};

#endif
```

ShopSystem.cpp

```
#include "Player.h"
#include "Spell.h"
#include "SpellHand.h"
#include "ShopSystem.h"
#include "GameConstants.h"
#include "DirectDamageSpell.h"
#include "AreaDamageSpell.h"
#include <iostream>
#include <cstdlib>
#include <ctime>

ShopSystem::ShopSystem() {
    availableSpells.push_back(std::unique_ptr<Spell>(new
DirectDamageSpell(
        "Ice Arrow",
        GameConstants::ICE_ARROW_COST,
        GameConstants::ICE_ARROW_RANGE,
        GameConstants::ICE_ARROW_DAMAGE
    )));

    availableSpells.push_back(std::unique_ptr<Spell>(new
AreaDamageSpell(
        "Ice Storm",
        GameConstants::ICE_STORM_COST,
        GameConstants::ICE_STORM_RANGE,
        GameConstants::ICE_STORM_DAMAGE
    )));
}

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 "Spell.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
```

```
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
```

```
SpellBase.cpp
#include "SpellBase.h"
#include "SpellTarget.h"

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

bool SpellBase::validateTarget(const SpellTarget& target, const
GameField& field) const {
    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;
    }

    return true;
}

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

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

int SpellBase::getRange() const {
    return range;
}
```

SpellBase.h
```
#ifndef SPELLBASE_H
#define SPELLBASE_H

#include "Spell.h"
#include "GameField.h"
#include <iostream>
#include <cmath>

class SpellBase : public Spell {
protected:
    std::string name;
    int manaCost;
    int range;
    int damage;

    bool validateTarget(const SpellTarget& target, const GameField&
field) const;

public:
    SpellBase(const std::string& spellName, int cost, int spellRange,
int spellDamage);

    virtual bool cast(const SpellTarget& target, GameField& field,
        std::vector<Enemy>& enemies, Player& player) override = 0;

    std::string getDescription() const override = 0;
    std::unique_ptr<Spell> clone() const override = 0;

    std::string getName() const override;
    int getManaCost() const override;
    int getRange() const override;
};

#endif
```

SpellHand.cpp
```
#include "SpellHand.h"
#include "Spell.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 "Spell.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
```

SpellTarget.cpp

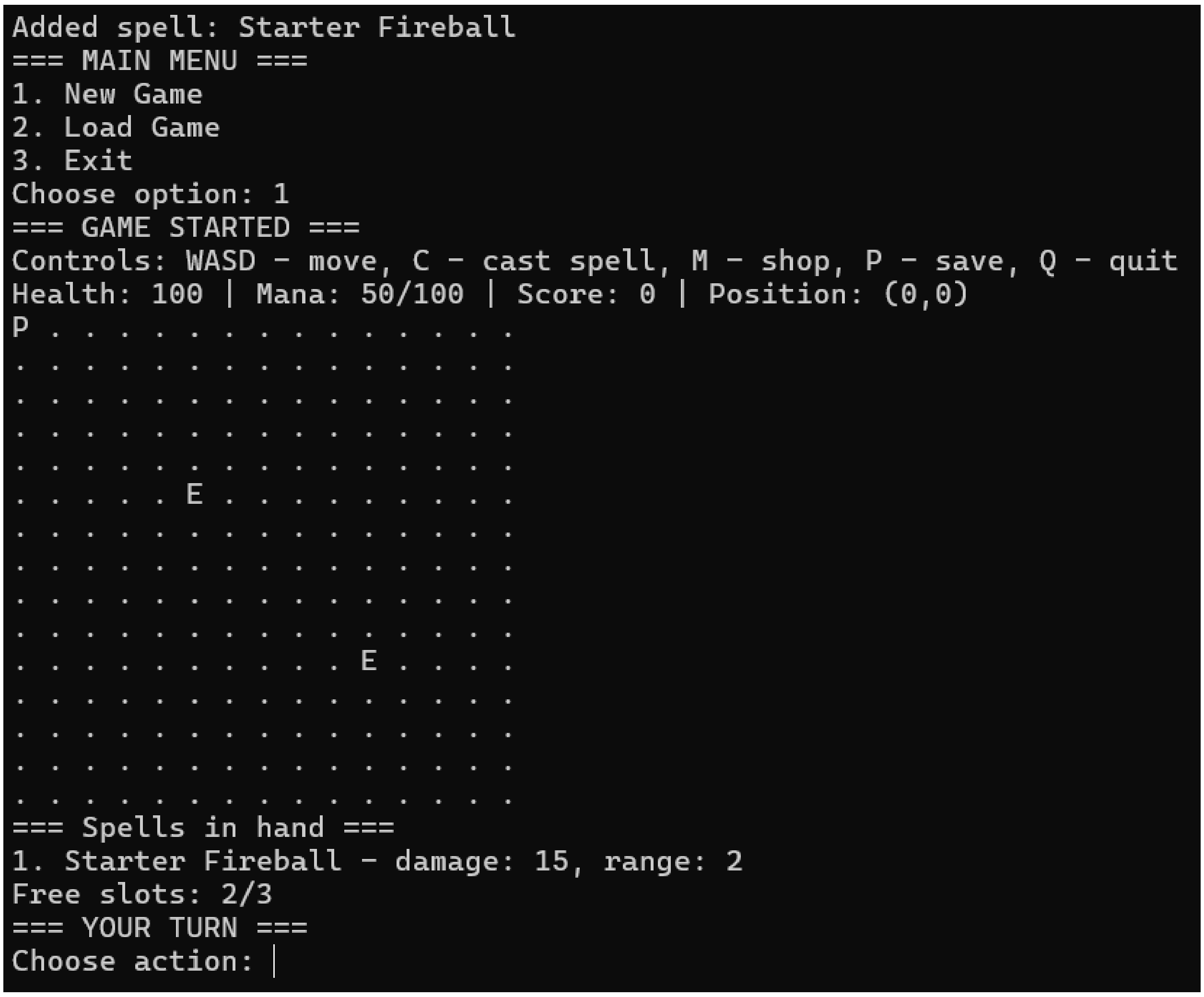
```
#include "SpellTarget.h"

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);
};
```

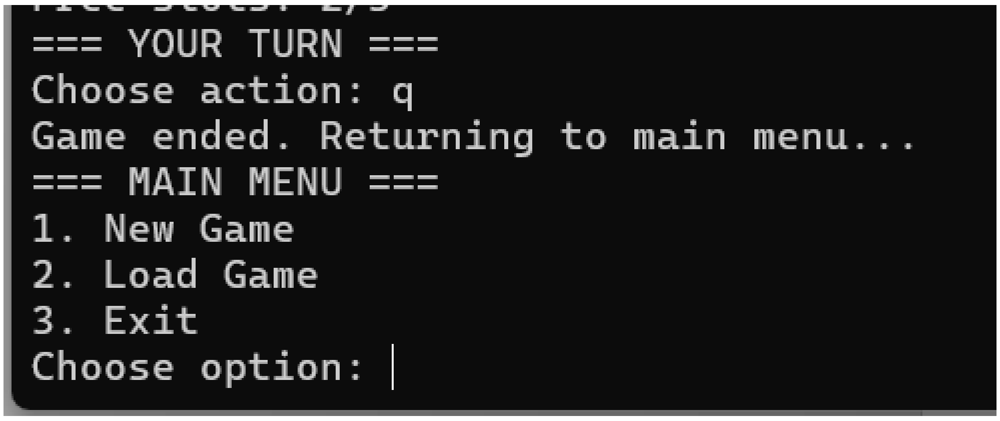
```
#endif


SpellType.h
#ifndef SPELLTYPE_H
#define SPELLTYPE_H

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

#endif


CommandType.h
#ifndef COMMANDTYPE_H
#define COMMANDTYPE_H

enum class CommandType {
    MOVE_UP,
    MOVE_DOWN,
    MOVE_LEFT,
    MOVE_RIGHT,
    CAST_SPELL,
    OPEN_SHOP,
    SAVE_GAME,
    QUIT_GAME,
    INVALID
};

#endif // COMMANDTYPE_H
```

ПРИЛОЖЕНИЕ Б
ТЕСТИРОВАНИЕ



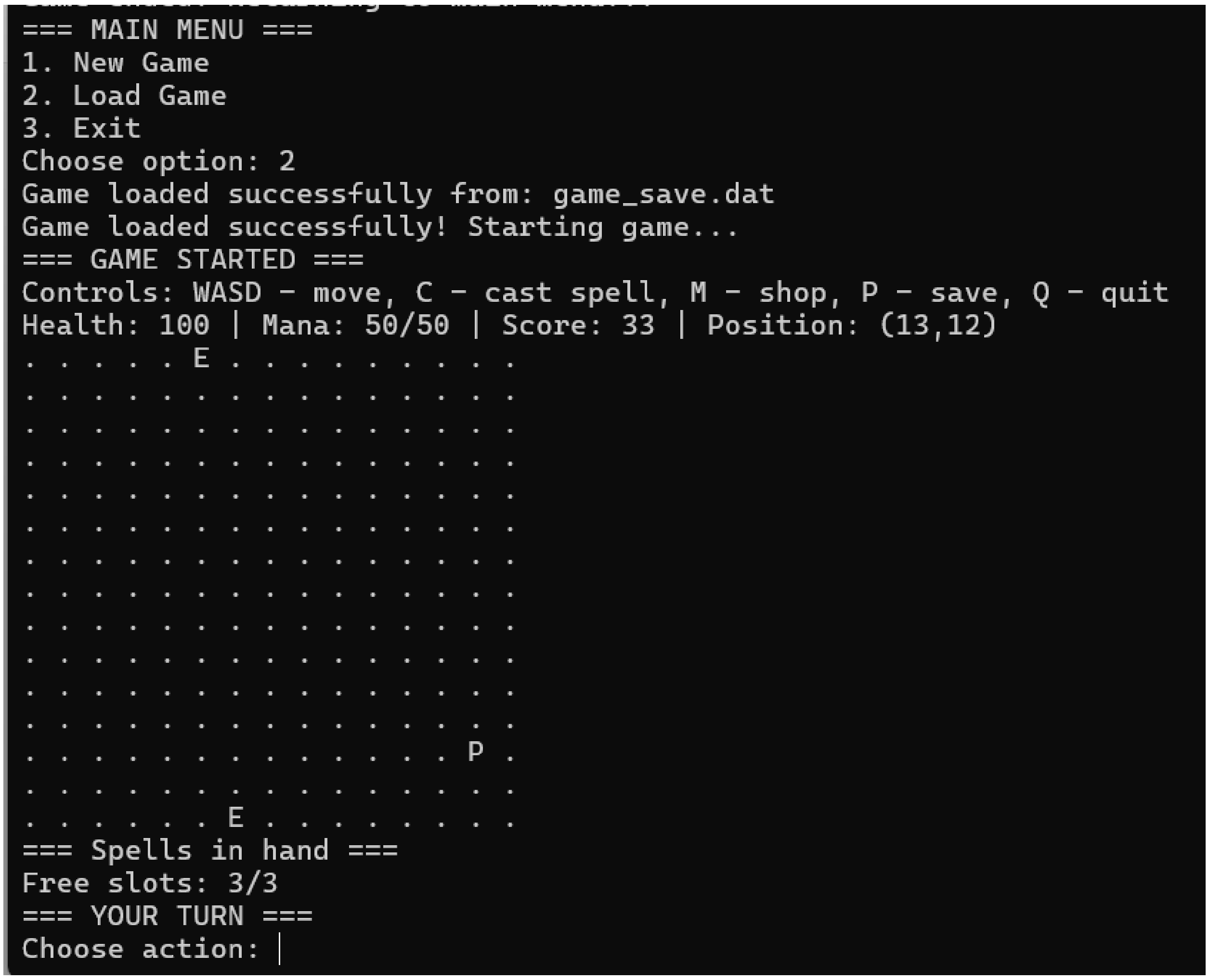
```
Added spell: Starter Fireball
=== MAIN MENU ===
1. New Game
2. Load Game
3. Exit
Choose option: 1
=== GAME STARTED ===
Controls: WASD - move, C - cast spell, M - shop, P - save, Q - quit
Health: 100 | Mana: 50/100 | Score: 0 | Position: (0,0)
P . . . . . . . . . . . . . .
. . . . . . . . . . . . . . .
. . . . . . . . . . . . . . .
. . . . . . . . . . . . . . .
. . . . . . . . . . . . . . .
. . . . . E . . . . . . . . .
. . . . . . . . . . . . . . .
. . . . . . . . . . . . . . .
. . . . . . . . . . . . . . .
. . . . . . . . . . . . . . .
. . . . . . . . . . E . . . .
. . . . . . . . . . . . . . .
. . . . . . . . . . . . . . .
. . . . . . . . . . . . . . .
. . . . . . . . . . . . . . .
=== Spells in hand ===
1. Starter Fireball - damage: 15, range: 2
Free slots: 2/3
=== YOUR TURN ===
Choose action:
```

Рисунок 2 – Запуск новой игры



```
=== 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



```
=== MAIN MENU ===
1. New Game
2. Load Game
3. Exit
Choose option: 2
Game loaded successfully from: game_save.dat
Game loaded successfully! Starting game...
=== GAME STARTED ===
Controls: WASD - move, C - cast spell, M - shop, P - save, Q - quit
Health: 100 | Mana: 50/50 | Score: 33 | Position: (13,12)
. . . . . E . . . . . . . . .
. . . . . . . . . . . . . . .
. . . . . . . . . . . . . . .
. . . . . . . . . . . . . . .
. . . . . . . . . . . . . . .
. . . . . . . . . . . . . . .
. . . . . . . . . . . . . . .
. . . . . . . . . . . . . . .
. . . . . . . . . . . . . . .
. . . . . . . . . . . . . . .
. . . . . . . . . . . . . . .
. . . . . . . . . . . . . . .
. . . . . . . . . . . . . P .
. . . . . . . . . . . . . . .
. . . . . . E . . . . . . . .
=== Spells in hand ===
Free slots: 3/3
=== YOUR TURN ===
Choose action: 
```

Рисунок 4 – Запуск сохранившейся игры