

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

ОТЧЕТ
по лабораторной работе №4
по дисциплине «Объектно-ориентированное программирование»

Студентка гр. 4384

Мулюкина Е.В.

Преподаватель

Жангиров Т.Р.

Санкт-Петербург

2025

Цель работы.

Изучить принципы объектно-ориентированного программирования.
Написать программу на языке C++, которая будет прототипом пошаговой игры с перемещением персонажа и сражением с врагами.

Задание.

1. Создать класс считающий 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.

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

- Отдельный класс для чтения ввода с консоли
- Отдельный класс для отрисовки текущего состояния игры
- Шаблонный класс управления игрой
- Шаблонный класс отрисовки игры, где параметрами могут быть любые классы отрисовки

За счет таких изменений «разгрузился» класс с основной логикой игры.

Также были изменены некоторые предыдущие классы для взаимодействия с новыми классами.

Архитектура программы.

В программе реализована иерархия классов, соответствующая принципам ООП без «божественных» классов.

Внесены изменения в класс с общей логикой игры – GameController, отвечающего за главное меню, запуск и переход на новые уровни и ход врагов.

После добавления новых классов класс GameController претерпел значительные изменения и стал гораздо чище. Раньше он одновременно хранил состояние игры, управлял основным циклом, читал весь ввод с клавиатуры через std::cin, выводил всё на экран через сотни строк std::cout, а также обрабатывал команды прямо внутри себя. Из-за этого класс был практически немасштабируемым.

Теперь же GameController отвечает исключительно за игровую логику и состояние игры. Из него полностью удалены все операции ввода-вывода: весь std::cin перенесён в ConsoleInputReader, весь std::cout — в ConsoleRenderer, основной цикл игры вместе с обработкой команд — в шаблонный GameLoop, а вызовы отрисовки — в шаблонный GameView.

Класс чтения ввода с консоли - ConsoleInputReader:

- Чтение базовых команд движения и действий (w/a/s/d — перемещение, c — каст заклинания, b — открытие магазина, e — сохранение, l — загрузка, q — выход).
- Запрос дополнительных данных: выбор заклинания из руки, выбор цели для заклинания (координаты x y), выбор предмета в магазине.
- Обработка подтверждения выхода с возможностью предварительного сохранения.
- Очистка буфера ввода после операций std::cin для предотвращения ошибок.

Класс получает ссылку на GameController в конструкторе, чтобы иметь возможность вызывать методы сохранения и загрузки напрямую. Все методы реализованы в отдельном .cpp-файле, а заголовок содержит только объявления.

Масштабируемость: в будущем можно создать аналогичный класс NetworkInputReader, который будет получать команды по сети, и использовать его вместо консольного без каких-либо изменений в остальном коде.

Класс отрисовки игрового интерфейса – ConsoleRenderer:

Класс ConsoleRenderer отвечает исключительно за отрисовку всего игрового интерфейса в консоль. Он содержит все операции вывода (std::cout), которые ранее были разбросаны по GameController.

- Отрисовка текущего состояния игры: игровое поле (символы '.', 'P', 'E'), статистика (здоровье игрока, очки, уровень, количество живых врагов).

- Вывод главного меню, экрана Game Over, сообщения о прохождении уровня.
- Отрисовка магазина заклинаний с подробной информацией о цене, описании и возможности покупки.
- Вывод списка доступных заклинаний в руке игрока с указанием статуса.
- Отображение помощи по выбору цели для заклинаний (прямого урона и площадного), включая пометку валидных целей символами '*' или '#'.

Класс получает доступ к состоянию игры только через константные геттеры GameController, не изменяя ничего.

Класс управления игрой – GameLoop:

Шаблонный класс GameLoop (по умолчанию с параметром InputReader = ConsoleInputReader) реализует основную логику управления игрой

- Принимает ссылку на существующий объект GameController (не создаёт его сам).
- Создаёт внутри себя объект класса ввода.
- Управляет главным меню (новая игра / загрузка / выход).
- Организует основной игровой цикл: отрисовка - получение команды от InputReader - выполнение соответствующего действия через методы GameController - обновление состояния игры (updateGame()).
- Обрабатывает особые ситуации в игре: каст заклинаний (спрашивает цель у игрока и проверяет, можно ли её выбрать); открытие магазина (показывает список заклинаний и обрабатывает покупку); подтверждение выхода (спрашивает, хочет ли игрок сохранить изменения).

нить игру перед выходом); экран конца игры с выбором: начать заново, загрузить сохранение или выйти.

Благодаря шаблонному параметру класс полностью масштабируем: для перехода на сетевую игру достаточно написать новый класс ввода и указать его как параметр шаблона (GameLoop).

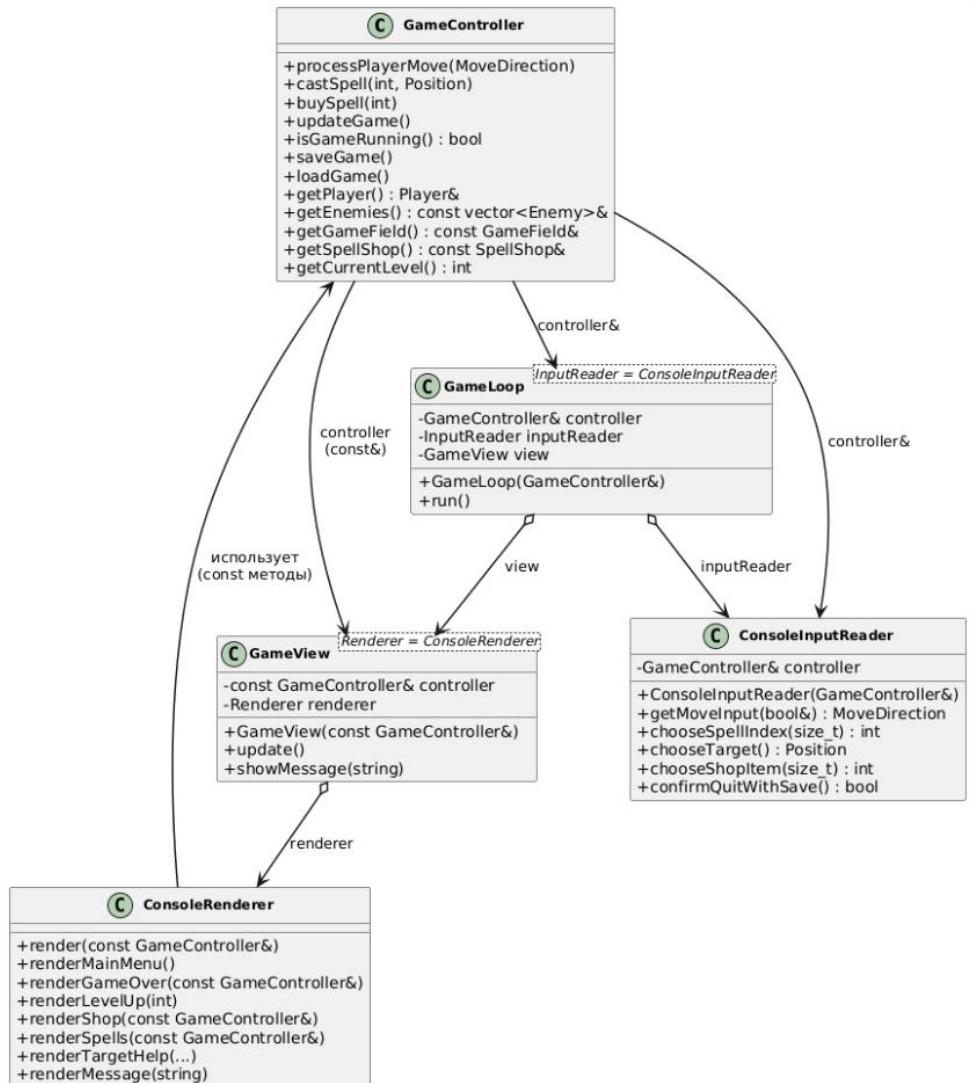
Класс визуализации изменений игры – GameView:

- Хранит константную ссылку на GameController.
- Создаёт внутри себя объект класса отрисовки.
- Предоставляет метод update(), который вызывается после каждого значимого изменения состояния игры и делегирует полную отрисовку рендереру.

Класс выступает в роли «наблюдателя» он реагирует на изменения модели (GameController) и обеспечивает актуальное отображение. Как и GameLoop, он использует шаблонный параметр для полной заменяемости способа отрисовки без изменения кода модели.

Реализация также полностью в заголовочном файле по причине шаблонности.

UML-диаграммы классов.



Выводы.

В процессе выполнения четвертой лабораторной работы я взяла за основу код из третьей лабораторной работы и разделила основную логику игры на несколько классов. Благодаря этому получилось более четкое и понятное разделения для будущего возможного масштабирования. В будущем можно масштабировать класс ввода с консоли в класс, который будет получать команды по сети, и использовать его вместо консольного без каких-либо изменений в остальном коде.

ПРИЛОЖЕНИЕ А

ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: binary_file.h

```
#ifndef BINARY_FILE_H
#define BINARY_FILE_H

#include <fstream>
#include "game_exceptions.h"

class BinaryFile {
    std::fstream file;
    std::string filename;
public:
    BinaryFile(const std::string& name, std::ios::openmode mode)
        : filename(name) {
        file.open(name, std::ios::binary | mode);
        if (!file.is_open()) {
            throw FileOpenException(name, "Check permissions or
disk space");
        }
    }
    ~BinaryFile() { if (file.is_open()) file.close(); }

    std::fstream& stream() { return file; }
    const std::string& getFilename() const { return filename; }
};

#endif
```

Название файла: game_exceptions.h

```
#ifndef GAME_EXCEPTIONS_H
#define GAME_EXCEPTIONS_H

#include <stdexcept>
#include <string>
```

```

class GameException : public std::runtime_error {
public:
    using std::runtime_error::runtime_error;
};

class SaveLoadException : public GameException {
protected:
    std::string filename;
public:
    SaveLoadException(const std::string& msg, const std::string&
file = "") :
        : GameException(msg), filename(file) {
    }
    const std::string& getFilename() const { return filename; }
};

class FileOpenException : public SaveLoadException {
public:
    FileOpenException(const std::string& file, const std::string&
reason = "") :
        : SaveLoadException("Cannot open file '" + file + "'",
                           reason.empty() ? "" : ": " + reason), file) {
    }
};

class FileWriteException : public SaveLoadException {
public:
    FileWriteException(const std::string& file) :
        : SaveLoadException("Write failed to '" + file + "'"),
file) {
    }
};

class FileReadException : public SaveLoadException {
public:
    FileReadException(const std::string& file) :
        : SaveLoadException("Read failed from '" + file + "'"),
file) {
    }
};

```

```

};

class CorruptedDataException : public SaveLoadException {
public:
    CorruptedDataException(const std::string& file, const
std::string& detail)
        : SaveLoadException("Corrupted save '" + file + "': " +
detail, file) {
    }
};

class VersionMismatchException : public SaveLoadException {
public:
    VersionMismatchException(const std::string& file, int
expected, int got)
        : SaveLoadException("Version mismatch in '" + file + "' :
expected " +
                std::to_string(expected) + ", got " +
                std::to_string(got), file) {
    }
};

class TooManyEnemiesException : public GameException {
public:
    TooManyEnemiesException(int count, int max = 5)
        : GameException("Too many enemies: " +
std::to_string(count) +
                ". Maximum allowed: " + std::to_string(max)) {
    }
};

#endif

```

Название файла: player.h

```

#ifndef PLAYER_H
#define PLAYER_H

```

```

#include "game_object.h"
#include "game_constants.h"

class Player : public GameObject {
public:
    Player();
    void move(int deltaX, int deltaY);
};

#endif

```

Название файла: player.cpp

```

#include "player.h"

Player::Player()
: GameObject(Position(0, 0),
GameConstants::INITIAL_PLAYER_HEALTH,
GameConstants::INITIAL_PLAYER_DAMAGE) {
}

```

```

void Player::move(int deltaX, int deltaY) {
    currentPosition.setX(currentPosition.getX() + deltaX);
    currentPosition.setY(currentPosition.getY() + deltaY);
}

```

Название файла: enemy.h

```

#ifndef ENEMY_H
#define ENEMY_H

#include "game_object.h"
#include "game_constants.h"

class Enemy : public GameObject {
public:
    Enemy(const Position& position);
    bool canAttackPlayer(const Position& playerPosition) const;

private:
    static constexpr int ATTACK_RANGE = 1;
};

```

```
#endif
```

Название файла: enemy.cpp

```
#include "enemy.h"
#include <cmath>

Enemy::Enemy(const Position& position)
    : GameObject(position,
        GameConstants::INITIAL_ENEMY_HEALTH,
        GameConstants::INITIAL_ENEMY_DAMAGE) {
}

bool Enemy::canAttackPlayer(const Position& playerPosition) const
{
    return currentPosition == playerPosition;
}
```

Название файла: position.h

```
#ifndef POSITION_H
#define POSITION_H

class Position {
public:
    Position(int x = 0, int y = 0);

    int getX() const;
    int getY() const;
    void setX(int x);
    void setY(int y);
    void setPosition(int x, int y);

    bool operator==(const Position& other) const;
    bool operator!=(const Position& other) const;

private:
    int xCoordinate;
    int yCoordinate;
};
```

```

#endif

Название файла: position.cpp

#include "position.h"

Position::Position(int x, int y) : xCoordinate(x), yCoordinate(y)
{ }

int Position::getX() const {
    return xCoordinate;
}

int Position::getY() const {
    return yCoordinate;
}

void Position::setX(int x) {
    xCoordinate = x;
}

void Position::setY(int y) {
    yCoordinate = y;
}

void Position::setPosition(int x, int y) {
    xCoordinate = x;
    yCoordinate = y;
}

bool Position::operator==(const Position& other) const {
    return xCoordinate == other.xCoordinate && yCoordinate == other.yCoordinate;
}

bool Position::operator!=(const Position& other) const {
    return !(*this == other);
}

```

Название файла: cell.h

```
#ifndef CELL_H
```

```

#define CELL_H

enum class CellType {
    EMPTY,
    PLAYER,
    ENEMY
};

class Cell {
public:
    Cell();
    CellType getType() const;
    void setType(CellType type);
    bool isEmpty() const;

private:
    CellType cellType;
};

```

#endif

Название файла: cell.cpp

```

#include "cell.h"

Cell::Cell() : cellType(CellType::EMPTY) {}

CellType Cell::getType() const {
    return cellType;
}

void Cell::setType(CellType type) {
    cellType = type;
}

bool Cell::isEmpty() const {
    return cellType == CellType::EMPTY;
}

```

Название файла: main.cpp

```
#include "game_controller.h"
```

```

#include <iostream>

int main() {
    try {
        GameController gameController(15, 15, 3);
        gameController.runGame();

    }
    catch (const std::exception& exception) {
        std::cerr << "Error: " << exception.what() << std::endl;
        return 1;
    }

    return 0;
}

```

Название файла: game_field.cpp

```

#include "game_field.h"
#include <stdexcept>

GameField::GameField(int width, int height)
    : fieldWidth(width), fieldHeight(height) {

    if (width < GameConstants::MIN_FIELD_SIZE ||
        width > GameConstants::MAX_FIELD_SIZE ||
        height < GameConstants::MIN_FIELD_SIZE ||
        height > GameConstants::MAX_FIELD_SIZE) {
        throw std::invalid_argument("Invalid field dimensions");
    }

    initializeGrid();
}

GameField::GameField(const GameField& other) //конструктор
копирования
    : fieldWidth(other.fieldWidth),
    fieldHeight(other.fieldHeight) {
    copyFrom(other);
}

```

```

GameField::GameField(GameField&& other) noexcept { //конструктор
перемещения
    moveFrom(std::move(other));
}

GameField& GameField::operator=(const GameField& other)
{ //оператор присваивания с копированием
    if (this != &other) {
        fieldWidth = other.fieldWidth;
        fieldHeight = other.fieldHeight;
        copyFrom(other);
    }
    return *this;
}

GameField& GameField::operator=(GameField&& other) noexcept
{ //оператор присваивания с перемещением
    if (this != &other) {
        moveFrom(std::move(other));
    }
    return *this;
}

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

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

CellType GameField::getCellType(const Position& position) const {
    if (!isValidPosition(position)) {
        throw std::out_of_range("Position is out of field
bounds");
    }
    return grid[position.getY()][position.getX()].getType();
}

```

```

}

bool GameField::isValidPosition(const Position& position) const {
    return position.getX() >= 0 && position.getX() < fieldWidth &&
        position.getY() >= 0 && position.getY() < fieldHeight;
}

bool GameField::isPositionEmpty(const Position& position) const {
    return isValidPosition(position) &&
        grid[position.getY()][position.getX()].isEmpty();
}

void GameField::setCellType(const Position& position, CellType type) {
    if (!isValidPosition(position)) {
        throw std::out_of_range("Position is out of field bounds");
    }
    grid[position.getY()][position.getX()].setType(type);
}

void GameField::clearCell(const Position& position) {
    setCellType(position, CellType::EMPTY);
}

void GameField::initializeGrid() {
    grid.resize(fieldHeight);
    for (int y = 0; y < fieldHeight; y++) {
        grid[y].resize(fieldWidth);
        for (int x = 0; x < fieldWidth; x++) {
            grid[y][x] = Cell();
        }
    }
}

void GameField::copyFrom(const GameField& other) {
    grid = other.grid; // копирование и указателей и элементов
}

```

```

void GameField::moveFrom(GameField&& other) noexcept {
    fieldWidth = other.fieldWidth;
    fieldHeight = other.fieldHeight;
    grid = std::move(other.grid);

    other.fieldWidth = 0;
    other.fieldHeight = 0;
}

```

Название файла: game_field.h

```

#ifndef GAME_FIELD_H
#define GAME_FIELD_H

#include "cell.h"
#include "position.h"
#include "game_constants.h"
#include <vector>
#include <memory>

class GameField {
public:
    GameField(int width = GameConstants::DEFAULT_FIELD_SIZE,
              int height = GameConstants::DEFAULT_FIELD_SIZE);

    GameField(const GameField& other);
    GameField(GameField&& other) noexcept;

    GameField& operator=(const GameField& other);
    GameField& operator=(GameField&& other) noexcept;

    int getWidth() const;
    int getHeight() const;
    CellType getCellType(const Position& position) const;

    bool isValidPosition(const Position& position) const;
    bool isPositionEmpty(const Position& position) const;

    void setCellType(const Position& position, CellType type);
    void clearCell(const Position& position);
}

```

```
private:
    int fieldWidth;
    int fieldHeight;
    std::vector<std::vector<Cell>> grid;

    void initializeGrid();
    void copyFrom(const GameField& other);
    void moveFrom(GameField&& other) noexcept;
};

#endif
```