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

ОТЧЕТ

по лабораторной работе №4

по дисциплине «ООП»

Тема: Разработка игровой системы с тактическими боями.

Студент гр. 4384 _____ Стукалкин М. М.

Преподаватель _____ Жангиров Т.Р.

Санкт-Петербург

2025

Цель.

Разработать игровую систему, реализующую механику пошаговых тактических боев с управлением персонажем, врагами и специальными сооружениями. Построить архитектуру с учетом принципов ООП и расширяемости

Задание.

На 6/3/1 баллов:

Создать класс, считывающий ввод пользователя и преобразующий ввод пользователь в объект команды.

Создать класс отрисовки игры. Данный класс определяет то, как должно отображаться игра.

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.

Создать шаблонный класс визуализации игры. . В качестве параметра шаблона должен передаваться класс, отвечающий за способ отрисовки игры. Данный класс создает объект класса отрисовки игры, и реагирует на изменения в игре, и вызывает команду отрисовку. Реализация должна быть такой, что можно масштабировать программу, например, реализовать отрисовку в виде веб-страницы без использования реализации интерфейса, и просто подставить новый класс в качестве параметра шаблона.

На 8/4/1.5 баллов:

Добавить возможность настраивать управление игрой через файл (то, на какие клавиши должна выполняться та или иная команда). Если команды некорректные: отсутствует информация для какой-то команды, на одну клавишу

две разные команды назначены, для одной команды назначены две разные клавиши, то в таком случае управление должно устанавливаться по умолчанию.

Архитектурные решения и обоснование.

1. **Класс Command.**

Назначение: инкапсулировать действие игрока (движение, атака, сохранение) в виде объекта, чтобы отделить логику ввода от логики игры.

Методы:

- Абстрактный класс с виртуальным методом void execute(Game& game).

1.2. Классы конкретных команд: MoveCommand (хранит направление движения std::string direction). SpellCommand (инициирует режим использования заклинаний). SaveCommand (вызывает сохранение текущего состояния). ExitCommand (переключает флаг game_running). EmptyCommand (заглушка для обработки не назначенных клавиш).

Обоснование: благодаря такой архитектуре класс ввода не знает, как реализовано сохранение или движение, он просто создает соответствующий объект команды. Это упрощает добавление новых действий и смену управления.

2. **Класс InputReader.**

Назначение: отвечает за получение ввода от пользователя и преобразование его в соответствующую команду на основе настроек управления.

Атрибуты:

1. std::map<char, std::function<Command*()>> key_map - словарь "символ - команда".

Методы:

1. InputReader(const std::string& config_file) - конструктор, инициирующий загрузку конфига.
2. void load_config(const std::string& filename) - загружает управление из файла. Реализует валидацию данных: проверяет наличие дубликатов клавиш и дубликатов команд. В случае ошибок сбрасывает управление

на настройки по умолчанию. Использует идиому RAII через класс FileHandler для безопасного открытия файла.

3. void set_defaults() - устанавливает стандартную раскладку (WASD).
4. Command* get_command() - считывает символ с клавиатуры (используя функцию _getch() для мгновенного ввода без нажатия Enter) и возвращает объект соответствующей команды.

Обоснование: класс реализует требование мгновенной обработки ввода и настройки управления. Он не имеет прямой связи с классом игры и реализует команды. Использован класс FileHandler для поддержки идиомы RAII при работе с файлом настроек.

3. Класс ConsoleRenderer.

Назначение: отвечает за непосредственную отрисовку игрового поля в консоли.

Методы:

1. void render(Game& game) - получает данные поля через метод to_lines() класса GameArea и выводит их на экран. Также отображает HUD (здоровье, опыт, список заклинаний).

Обоснование: выделение рендеринга в отдельный класс позволяет легко заменить способ отображения (например, на графический интерфейс или веб-страницу), просто создав новый класс рендерера и передав его в шаблон визуализатора.

4. Интерфейс IGameView и класс ConsoleView

Назначение: Обеспечить взаимодействие игры с пользователем (вывод сообщений, меню диалогов) без прямой зависимости класса Game от потоков ввода-вывода (iostream).

Методы интерфейса IGameView:

1. void show_message(const std::string& msg) - вывод информационного сообщения.
2. void on_level_init(int level) - событие начала уровня.
3. void on_game_over(bool win) - событие окончания игры.

4. int ask_upgrade_choice() - запрос выбора улучшения персонажа.
5. int ask_game_over_action() - запрос действия после проигрыша.

Реализация ConsoleView:

Реализует методы интерфейса, используя std::cout и std::cin.

Обоснование: класс Game теперь зависит от абстракции, а не от конкретной реализации консоли. Это делает код более чистым и тестируемым.

5. Шаблонный класс GameVisualizer<TRenderer> .

Назначение: класс визуализации, связывающий состояние игры с конкретным рендерером.

Параметры шаблона: TRenderer - тип класса, отвечающего за отрисовку (в этой реализации ConsoleRenderer).

Методы:

1. void draw(Game& game) - делегирует задачу отрисовки объекту renderer.

Обоснование: использование шаблонов позволяет внедрять зависимость на этапе компиляции, избегая накладных расходов на виртуальные функции там, где это не требуется.

6. Шаблонный класс GameManager<TInputReader, TVisualizer>.

Назначение: управляет основным игровым циклом.

Параметры шаблона:

1. TInputReader - класс для получения команд.
2. TVisualizer - класс для визуализации.

Методы:

1. void run(Game& game) - запускает игровой цикл. На каждой итерации вызывает отрисовку, запрашивает команду у InputReader, выполняет её и обновляет состояние мира игры.

Обоснование: класс не создает объект игры сам, а принимает его извне. Это обеспечивает гибкость конфигурации системы. Использование

шаблонов позволяет легко менять компоненты системы ввода и вывода без изменения кода менеджера.

7. Изменения в классе Game.

Класс был рефакторизован для соответствия новой архитектуре:

1. Удалена вся логика прямого ввода-вывода (std::cin, std::cout).
2. Добавлено поле IGameView* view для связи с интерфейсом пользователя.
3. Добавлены методы для команд (move_player, cast_spell_mode, save_game_command).
4. Добавлены геттеры (get_area, get_player) для доступа со стороны системы визуализации.

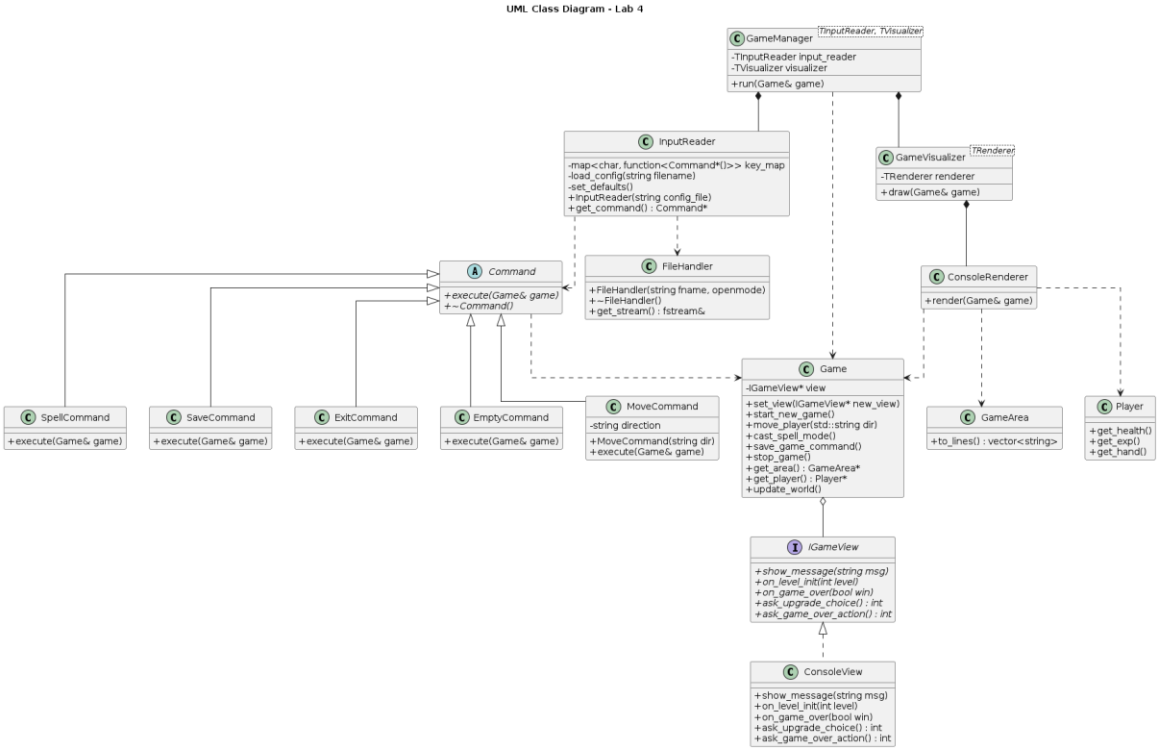

Рисунок 1 - UML-диаграмма классов

Выводы.

Разработана игровая система, реализующая механику пошаговых тактических боев с управлением персонажем, врагами и специальными сооружениями. Архитектура построена с учетом принципов ООП и расширяемости.