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

ОТЧЕТ

по лабораторной работе №4

по дисциплине «Объектно-ориентированное программирование»

Студент гр. 4384 Кочкин Д. М.

Преподаватель Жангиров Т. Р.

Санкт-Петербург

2025

Цель работы.

Изучение принципов объектно-ориентированного проектирования и реализация игровой логики на языке C++. В ходе работы создаются классы для представления и отрисовки сущностей игрового поля (игрока, врагов, зданий, клеток и самого поля) и реализуются их взаимодействия.

Задание.

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.

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.

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

Описание архитектуры программы.

Программа построена по принципам объектно-ориентированного программирования с чётким разделением ответственности между классами. Центральным элементом является класс Field, отвечающий за состояние игрового поля, размещение и обновление сущностей.

Система ввода.

Цель: Принимать пользовательский ввод (клавиши), преобразовывать его в команды и передавать контроллеру игры.

Компоненты: InputHandlerInterface (интерфейс), ConsoleInputHandler (реализация), GameController (обработчик команд).

Как работает: ConsoleInputHandler хранит отображение символ → CommandType в keyBindings_. При вводе символа (std::cin) метод getCommand() возвращает Command. GameController получает Command и вызывает соответствующие методы GameManager (движение, атака, сохранение и т.п.).

Ошибки/валидация: При загрузке конфигурации проверяются конфликты клавиш и обязательные команды; в случае ошибок применяется набор дефолтных привязок.

Файлы: input_handler.h/.cpp, game_controller.h.

Система отрисовки.

Цель: Отображать текущее состояние игры пользователю (поле, информация о игроке, меню).

Компоненты: RendererInterface (абстракция), ConsoleRenderer (консольная реализация), GameVisualizer (шаблонный класс, который использует конкретный Renderer).

Как работает: GameVisualizer получает ссылку на Field и Player и вызывает renderField, renderPlayerInfo, renderMessage. Field::print() тоже

реализует низкоуровневую печать, но GameVisualizer унифицирует интерфейс отображения.

Расширяемость: Для GUI достаточно реализовать новый класс, удовлетворяющий RendererInterface, и подставить в GameVisualizer.

Файлы: renderer.h, game_visualizer.h, field.cpp.

Система визуализации.

Цель: Управление жизненным циклом игры: меню, создание и загрузка уровней, игровой цикл и обработка ходов.

Класс: GameManager — главный контроллер приложения. Включает методы run(), startNewGame(), playLevel(), handlePlayerTurn() и т.д.

Взаимодействие: GameManager использует Field для состояния мира, GameVisualizer/Renderer для вывода, GameController/InputHandler для ввода, SaveSystem для сохранения.

Логика хода: В handlePlayerTurn() происходит чтение команды, определение действия (перемещение, атака, каст) и обновление Field. После действий вызывается updateEntities() для остальных сущностей.

Файлы: game_manager.h/.cpp, game.h/.cpp, game_visualizer.h.

Система логирования.

Цель: Централизованное логирование игровых событий (движение, урон, получение/каст заклинаний, спавн/смерть сущностей, сохранение/загрузка и пр.) с возможностью вывода в терминал и/или файл.

Компоненты: EventSystem (синглтон), GameEvent (структура), LoggerInterface (интерфейс логгера), ConsoleLogger, FileLogger.

Как работает: Подсистемы (например, Field, CombatSystem, Hand, GameManager) не пишут напрямую в файл/терминал. Они формируют событие или вызывают соответствующий метод EventSystem (например, logPlayerMoved, logPlayerTakeDamage, logSpellCasted) — EventSystem рассылает событие всем зарегистрированным логгерам.

Конфигурация при запуске: В main.cpp регистрируются логгеры по параметру запуска --log= (пример: --log=console,file:game.log). Если не указано — по умолчанию включается ConsoleLogger.

Плюсы: Разделение логики и вывода; легко сменить формат вывода (plain → JSON), добавить уровни логирования или хранить логи в файл.

Файлы: event_system.h, event_system.cpp.

Система редактирования используемых клавиш.

Цель: Позволять пользователю настраивать клавиши управления без перекомпиляции.

Формат файла: Текстовый файл, каждая строка — COMMAND_NAME KEY (примеры: MoveUp w). Комментарии начинаются с #.

Логика парсинга: KeyBindingsConfig::loadConfigFile (в key_bindings_config.h) строит отображение char → CommandType, проверяет дубликаты и обязательный набор

команд. Если в конфиге ошибки — возвращается false и используется набор дефолтных привязок.

Применение: При старте Game читает keybindings.cfg через inputHandler_.loadFromFile(...); в меню GameManager есть reloadConfig("keybindings.cfg").

Файлы: key_bindings_config.h, keybindings.cfg.

Краткое резюме одного “хода”.

Пользователь нажимает клавишу → ConsoleInputHandler::getCommand() преобразует символ в Command → GameController/GameManager обрабатывает команду → Field изменяется (перемещение/атака/каст) → при ключевых событиях подсистемы вызывают EventSystem для логирования → GameVisualizer/Renderer отображают обновлённое состояние.

Сохранение/загрузка: GameManager вызывает SaveSystem::saveToFile / loadFromFile и через EventSystem логирует соответствующие события.

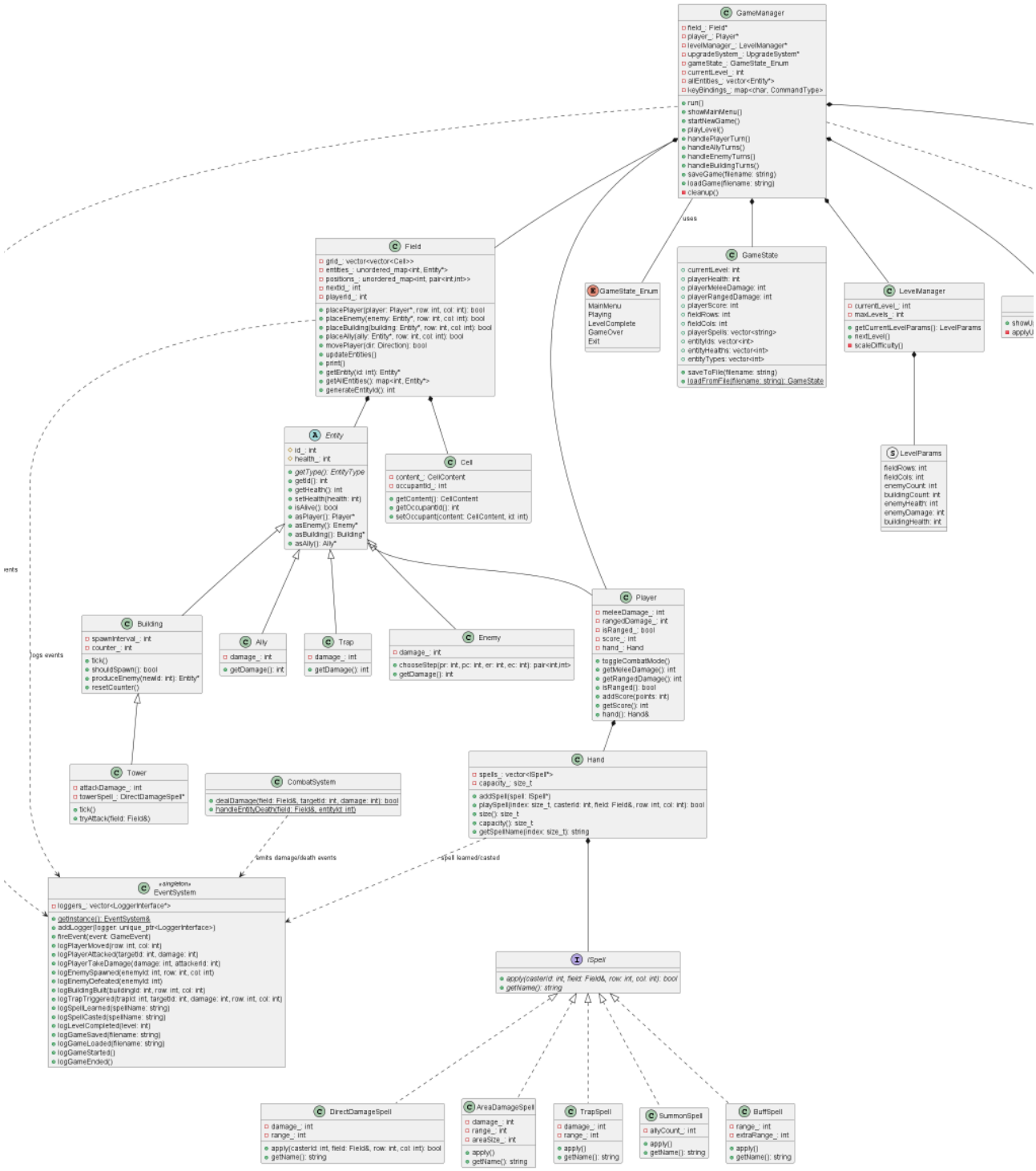

Рисунок 1 - UML-диаграмма

Проверка работы программы.

Была проведена проверка работы игры. Реализован графический интерфейс. Игрок может изменять клавиши для применения действий. Все действия во время игры фиксируются в консоль/в файл с расширением .log (по выбору пользователя).

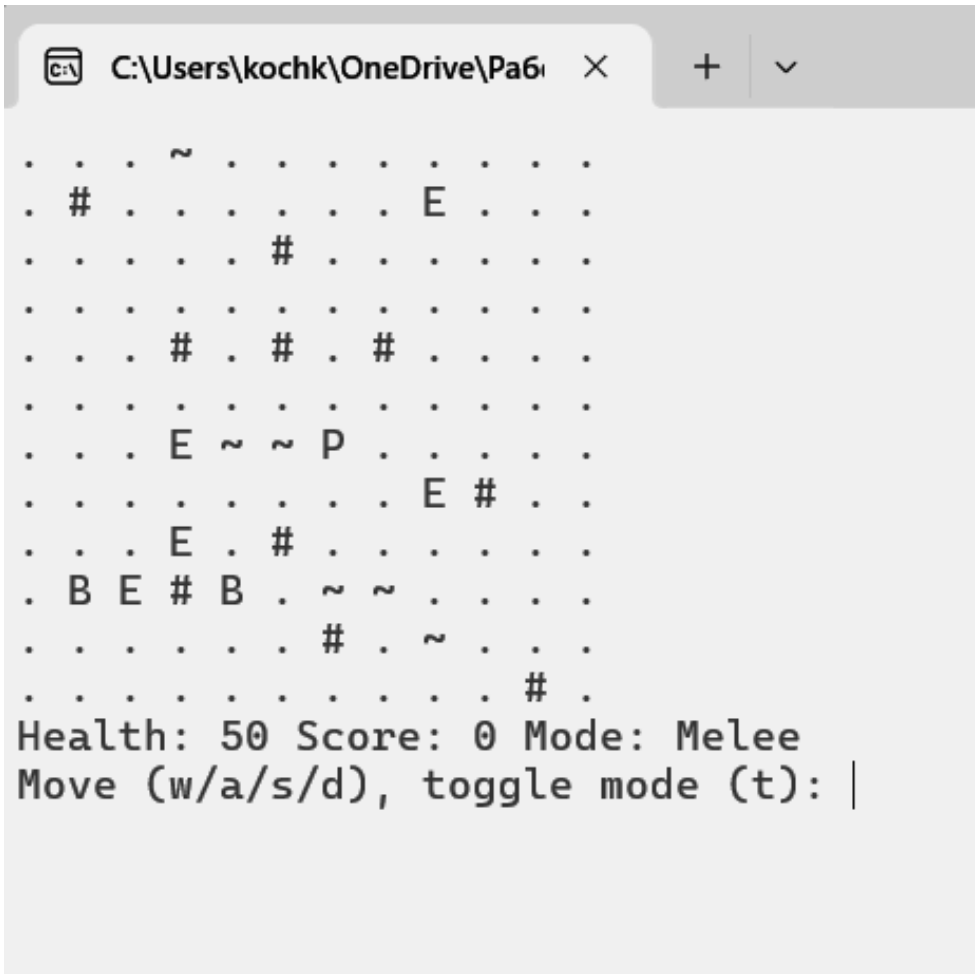

Рисунок 2 - Игровое поле

Выводы.

В результате выполнения лабораторной работы были изучены принципы ООП: наследование, инкапсуляция и полиморфизм. Разработана архитектура игры с несколькими типами сущностей и взаимодействиями

между ними. Программа успешно проходит тестирование и демонстрирует корректное поведение объектов на игровом поле.