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»

Студент гр. 4384 Шакуров А.И.

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.

- 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.

На 10/5/2 баллов:

- 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 логирования, а только лишь информировать о событии. Запись может идти как в файл, так и в терминал, способ логирования определяется пользователем через параметры запуска программы.
- Примечания:

  После считывания клавиши, считанный символ должен сразу обрабатываться, и далее работа должна проводить с сущностью, которая представляет команду

  Для представления команды можно разработать системы классов

  Хорошей практикой является создание “прослойки” между считыванием/обработкой команды и классом игры, которая сопоставляет команду и вызываемым методом игры. Существуют альтернативные решения без явной “прослойки”

Архитектура программы.

В рамках лабораторной работы в проект добавлены подсистемы, дополняющие уровень игровой логики (Game/GameController) и обеспечивающие расширяемость по вводу/выводу и логированию.

Добавленные уровни/подсистемы:

Уровень ввода и команд: ConsoleCommandReader → Command.

Уровень визуализации: GameRenderer, GameVisualizer<Renderer>.

Уровень управления ходом: GameManager<InputReader>.

Уровень конфигурации управления: ConfigLoader (биндинги из файла + валидация).

Уровень логирования событий: ILogger + ConsoleLogger/FileLogger + EventBus.

Ключевое архитектурное решение: ввод и отрисовка отделены от логики игры и подключаются через шаблонные обёртки, что позволяет заменять способ ввода (например, сеть вместо консоли) и способ отображения (например, веб вместо консоли) без переписывания игровых классов.

1. Подсистема команд (Command)

Назначение: класс Command представляет пользовательское действие в виде объекта данных, с которым дальше работает игровая логика (а не с символами клавиатуры).

Ключевые решения:

Тип действия задаётся через CommandType . Параметры команды хранятся внутри объекта, чтобы после чтения клавиши дальнейшая обработка была полностью объектной. Такой подход упрощает масштабирование: источник команд может измениться, обработчик получает Command.

2. **ConsoleCommandReader - ввод пользователя → Command**

Назначение: ConsoleCommandReader изолирует std::cin и преобразует нажатую клавишу в объект Command.

Ключевые решения:

Класс хранит биндинги bindings: unordered_map<char, string> (клавиша → действие) и умеет работать с дефолтными биндингами через defaultBindings().

В методе readCommand(Player, Hand, currentLevel, requiredPoints) клавиша считывается, нормализуется (в верхний регистр) и сразу преобразуется в Command.

Если для команды нужны дополнительные параметры, reader запрашивает их отдельно: askSpellIndex(), askEnemyIndex(), askFilename().

Почему так сделано: это отделяет слой ввода от игры: при необходимости можно реализовать новый reader и подставить его в GameManager<InputReader> без изменений в Game/GameController.

3. Отрисовка: GameRenderer и GameVisualizer<Renderer>

3.1 GameRenderer

Назначение: GameRenderer определяет, как показывать состояние игры в консоли: поле (через GameField::display() из контроллера) и статус игрока/уровня.

Ключевые решения:

Отрисовка вынесена из игровой логики: render(const Game&, const GameController&) читает состояние и отображает его, не изменяя игру.

3.2 GameVisualizer<Renderer> — шаблонная обёртка

Назначение: GameVisualizer<Renderer> хранит конкретный renderer и вызывает его render(), позволяя заменить способ отображения подстановкой другого класса-рендера.

4. GameManager<InputReader> управление ходом

Назначение: GameManager<InputReader> отвечает за обработку одного хода: получить Command от reader, вызвать нужное действие в GameController/Game, инициировать отрисовку и опубликовать событие.

Ключевые решения:

GameManager не создаёт Game, а работает с переданными ссылками (Game&, GameController&, Visualizer&), что соответствует требованию “не владеть игрой”.

Внутри менеджер создаёт объект reader’а как поле InputReader reader, т.е. источник команд задаётся типом шаблона.

Главный метод processTurn(...) реализует “прослойку” между вводом и игровой логикой: сопоставляет CommandType и вызываемый метод/действие.

После выполнения действия менеджер вызывает visualizer.render(game, controller), обновляя экран централизованно.

5. ConfigLoader — биндинги из файла с откатом на дефолт

Назначение: ConfigLoader загружает файл конфигурации управления (формат KEY=ACTION) и возвращает карту биндингов.

Ключевые решения:

Метод loadBindings(path) читает файл построчно, парсит KEY=ACTION, нормализует ключ и чистит пробелы.

Метод validate(bindings) проверяет корректность настроек: наличие полного набора обязательных действий и уникальность назначений.

При любой ошибке (не открылся файл / неверный формат / провал валидации) возвращается пустая карта, что приводит к использованию ConsoleCommandReader::defaultBindings().

6. Логирование событий: ILogger / ConsoleLogger / FileLogger / EventBus

Назначение: подсистема логирования фиксирует игровые события (движение, атаки, покупка, сохранение/загрузка и т.д.) без прямой зависимости игровых сущностей от логгера.

Компоненты и решения:

ILogger — интерфейс с методом log(message).

ConsoleLogger пишет сообщения в консоль, FileLogger пишет сообщения в файл (RAII: ofstream хранится внутри).

EventBus хранит shared_ptr<ILogger> и предоставляет publish(message): если логгер задан — сообщение логируется.

GameManager публикует события через EventBus при выполнении команд, тем самым реализуя “сущности не вызывают логгер напрямую, а только информируют о событии”.

Как работает логирование:

1. main создаёт логгер (или nullptr)
2. EventBus инициализируется с логгером
3. GameManager публикует события через bus.publish("...")
4. EventBus передаёт сообщение логгеру
5. Логгер выводит или сохраняет сообщение

Пример цикла обработки команды "движение вверх"

Этап 1: Считывание (ConsoleCommandReader::readCommand)

Пользователь нажимает 'W'. Reader:

1. Выводит статус игрока (HP, DMG, SCORE)
2. Выводит текущие биндинги
3. Читает символ 'W'
4. Нормализует в верхний регистр
5. Находит в карте: 'W' → "UP"
6. Создаёт Command{type: Move, direction: UP}

Этап 2: Обработка (GameManager::processTurn)

Менеджер получает Command:

1. Переключается по cmd.type == Move
2. Вызывает controller.turnPlayer(Direction::UP)
3. Публикует событие: bus->publish("Player moved to (4, 5)")

Этап 3: Выполнение (GameController::turnPlayer)

Контроллер:

1. Вызывает field->moveUnit(&player, UP)
2. Проверяет коллизии и перемещает игрока
3. Вызывает field->display() для вывода сетки
4. Очищает мёртвых врагов

Этап 4: Логирование (EventBus → ILogger)

Шина:

1. Проверяет наличие логгера
2. Если логгер установлен → вызывает logger->log("Player moved to (4, 5)")

Этап 5: Визуализация (GameVisualizer → GameRenderer)

Визуализатор:

1. Вызывает renderer.render(game, controller)
2. Renderer выводит поле (уже вывод в turnPlayer)
3. Выводит статус игрока (HP, DMG, SCORE)
4. Выводит информацию уровня

Результат: состояние игры обновлено, событие залогировано, экран переотрисован.

Пользователь нажимает 'W'

ConsoleCommandReader::readCommand()

возвращает Command{type: Move, direction: UP}

GameManager::processTurn()

```
├─ case CommandType::Move:
│   ├─ controller.turnPlayer(Direction::UP)
│   │   └─ field->moveUnit(&player, UP)
│   │       └─ field->display() [вывод сетки]
│   │
│   ├─ bus->publish("Player moved to (4, 5)")
│   │   └─ logger->log("Player moved to (4, 5)")
│   │       └─ std::cout или std::ofstream
│   │
│   └─ visualizer.render(game, controller)
│       └─ renderer.render(game, controller)
│           └─ std::cout << "HP: ..." [вывод статуса]
│
└─ return true
```

Выводы.

Была изучена парадигма объектно-ориентированного программирования. Была реализована программа на языке C++ содержащая основные классы игры с необходимыми полями и методами.

ПРИЛОЖЕНИЕ А
UML ДИАГРАММА