

**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»**  
**Тема: «Шаблонные классы»**

Студент гр. 3343

Силяев Р.А.

Преподаватель

Жангиров Т. Р.

Санкт-Петербург

2024

## **Цель работы**

Создать модульную и расширяемую систему управления игрой, разделенную на управление, отображение и логику, с возможностью сохранения и загрузки игры.

## **Задание**

- a) Создать шаблонный класс управления игрой. Данный класс должен содержать ссылку на игру. В качестве параметра шаблона должен указываться класс, который определяет способ ввода команда, и переводящий введенную информацию в команду. Класс управления игрой, должен получать команду для выполнения, и вызывать соответствующий метод класса игры.
- b) Создать шаблонный класс отображения игры. Данный класс реагирует на изменения в игре, и производит отрисовку игры. То, как происходит отрисовка игры определяется классом переданном в качестве параметра шаблона.
- c) Реализовать класс считывающий ввод пользователя из терминала и преобразующий ввод в команду. Соответствие команды введенному символу должно задаваться из файла. Если невозможно считать из файла, то управление задается по умолчанию.
- d) Реализовать класс, отвечающий за отрисовку поля.

## **Примечание:**

- Класс отслеживания и класс отрисовки рекомендуется делать отдельными сущностями. Таким образом, класс отслеживания инициализирует отрисовку, и при необходимости можно заменить отрисовку (например, на GUI) без изменения самого отслеживания
- 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 или использовать перечисление enum.
- 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”
- При считывания управления необходимо делать проверку, что на все команды назначена клавиша, что на одну клавишу не назначено две

команды, что на одну команду не назначено две клавиши.

## Выполнение работы

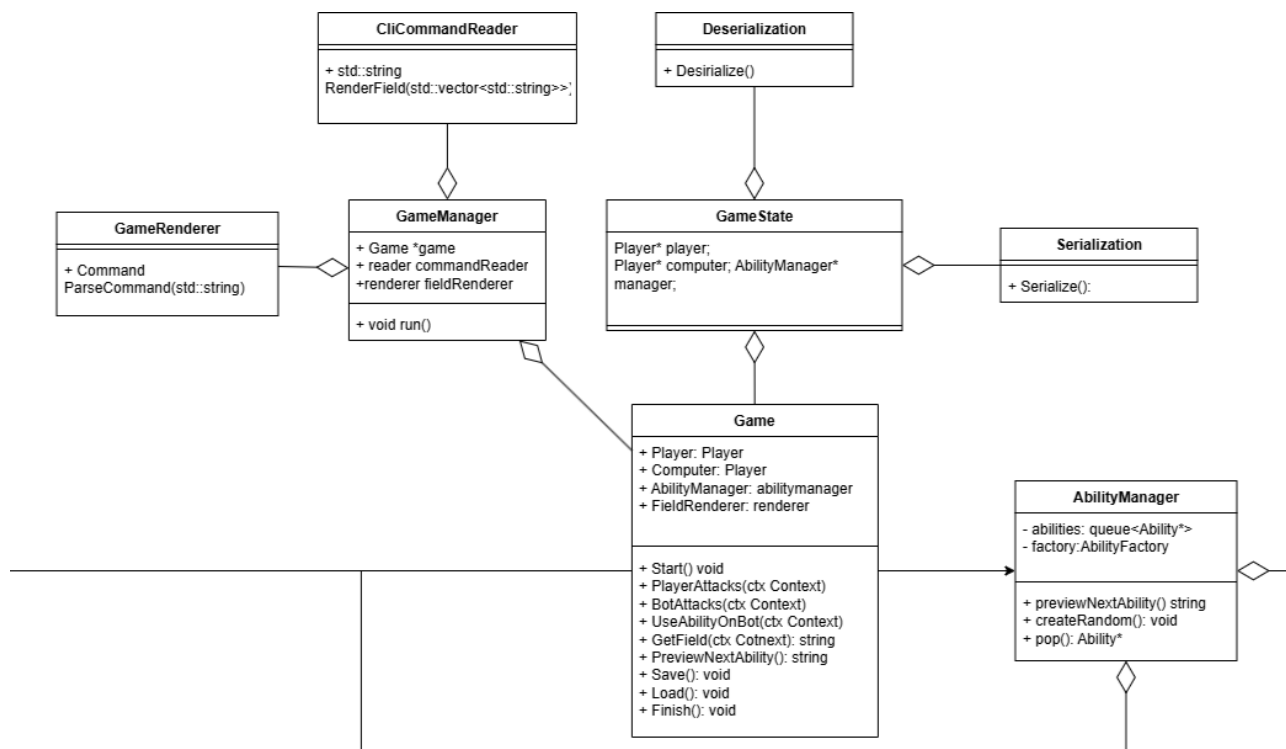


Рисунок 1 – UML-диаграмма классов

Код программы содержит реализацию классов: *GameManager*, *CliCommandReeader*, *GameRenderer*.

Класс *GameManager* является шаблонным классом, который управляет игровым процессом, взаимодействуя с пользователем и обрабатывая команды. Он использует два параметра шаблона: *renderer* и *reader*, которые отвечают за отображение игрового поля и чтение команд соответственно.

Поля класса:

- *bool gameisrunning*: Флаг, указывающий, запущена ли игра. Если *true*, игра продолжается; если *false*, игра завершена.
- *reader commandReader*: Объект, отвечающий за чтение и парсинг команд, введенных пользователем.
- *renderer fieldRenderer*: Объект, отвечающий за отображение игрового поля.
- *Game\* game*: Указатель на объект класса *Game*, который управляет логикой игры.

Методы класса:

- *GameManager()*: Конструктор класса, инициализирует объект *GameManager*.

- *void run()*: Запускает игровой цикл, который продолжает работать, пока игра активна. Внутри цикла происходит рендеринг полей и обработка ходов.

- *void processTurn()*: Обрабатывает ввод пользователя, считывает команды и выполняет соответствующие действия в игре. В зависимости от типа команды (атака, использование способности, загрузка, сохранение и т.д.) вызываются соответствующие методы игры. Если команда некорректна, выводится сообщение об ошибке.

- *void renderFields()*: Получает поля игрока и противника из объекта игры и отображает их с помощью *fieldRenderer*.

- *void waitForEnter()*: Ожидает нажатия клавиши Enter от пользователя, чтобы продолжить выполнение программы. Также очищает экран.

Основные функции:

- Управление игровым процессом, включая ввод команд и отображение состояния игры.

- Обработка различных команд, таких как атака, использование способностей, загрузка и сохранение игры.

- Обеспечение взаимодействия между пользователем и логикой игры через рендеринг полей и обработку ввода.

Класс *GameManager* является центральным элементом, который связывает пользовательский интерфейс с логикой игры, обеспечивая плавный игровой процесс.

Класс *CLICommandReader* предназначен для чтения и обработки команд из командной строки (CLI) на основе конфигурационного файла. Он загружает команды из указанного конфигурационного файла и предоставляет методы для парсинга и выполнения этих команд.

Поля класса:

- Контейнер *commandMap* (например, *std::map*), который сопоставляет строковые представления команд с их типами (*CommandType*).

Методы класса:

- Конструктор класса *CLICommandReader(const std::string& configPath)* принимает путь к конфигурационному файлу (*configPath*) и загружает команды из этого файла. Если загрузка не удалась, устанавливает команды по умолчанию.

- *bool loadConfig(const std::string& configPath)* загружает команды из конфигурационного файла. Принимает путь к конфигурационному файлу (*configPath*) и возвращает *true*, если загрузка прошла успешно, и *false* в противном случае. Открывает файл, читает команды и заполняет *commandMap*.

- *void trim(std::string &str)* удаляет пробелы в начале и в конце строки. Принимает строку для обработки (*str*) и удаляет все пробелы в начале и в конце строки.

- *Command parseCommand(const std::string& input)* парсит входную строку и возвращает соответствующую команду. Принимает входную строку команды (*input*) и возвращает объект типа *Command*. Разбивает входную строку на команду и аргументы, ищет команду в *commandMap* и вызывает соответствующий метод парсинга.

- *Command parseAttack(const std::vector<std::string> & args)* парсит команду атаки. Принимает вектор аргументов команды (*args*) и возвращает объект типа *Command*. Проверяет количество аргументов и преобразует их в координаты для команды атаки.

- *Command parseAbility(const std::vector<std::string> & args)* парсит команду использования способности. Принимает вектор аргументов команды (*args*) и возвращает объект типа *Command*. Проверяет количество аргументов и преобразует их в координаты для команды использования способности.

- *Command parseLoad(const std::vector<std::string> & args)* парсит команду загрузки. Принимает вектор аргументов команды (*args*) и возвращает объект типа *Command*. Проверяет, что команда не имеет аргументов.

- *Command parseSave(const std::vector<std::string>& args)* парсит команду сохранения. Принимает вектор аргументов команды (*args*) и возвращает объект типа *Command*. Проверяет, что команда не имеет аргументов.

- *Command parsePreviewAbility(const std::vector<std::string>& args)* парсит команду предварительного просмотра способности. Принимает вектор аргументов команды (*args*) и возвращает объект типа *Command*. Проверяет, что команда не имеет аргументов.

Исключения:

Исключение *InvalidCommandException* выбрасывается при неверной команде или аргументах.

Класс *GameRenderer* отвечает за визуализацию игрового поля в текстовом формате. Он предоставляет метод *RenderField*, который принимает вектор векторов строк, представляющий игровое поле, и флаг, указывающий, принадлежит ли поле противнику. Метод форматирует поле в строку, добавляя заголовки, разделители и координаты, и возвращает эту строку для вывода в терминал или другое текстовое представление.

Методы класса

- *std::string RenderField(std::vector<std::vector<std::string>> field, bool isEnemyField)*: метод *RenderField* класса *GameRenderer* предназначен для преобразования двумерного вектора строк, представляющего игровое поле, в отформатированную текстовую строку, пригодную для вывода. Он принимает в качестве аргументов *field* (двумерный вектор строк, представляющий поле) и *isEnemyField* (флаг, указывающий, принадлежит ли поле противнику). Метод формирует строку, которая включает заголовок (“ENEMY FIELD” или “YOUR FIELD” в зависимости от флага), строку с номерами столбцов, разделительные линии, строки поля с их номерами, и нижнюю границу. Для лучшей визуализации используются ANSI-коды для выделения заголовков цветом. Результатом работы метода является текстовая строка, представляющая игровое поле в читаемом формате. Использование константы *INDENTATION* позволяет выравнивать игровое поле.



## Тестирование:

Программа работает без ошибок.

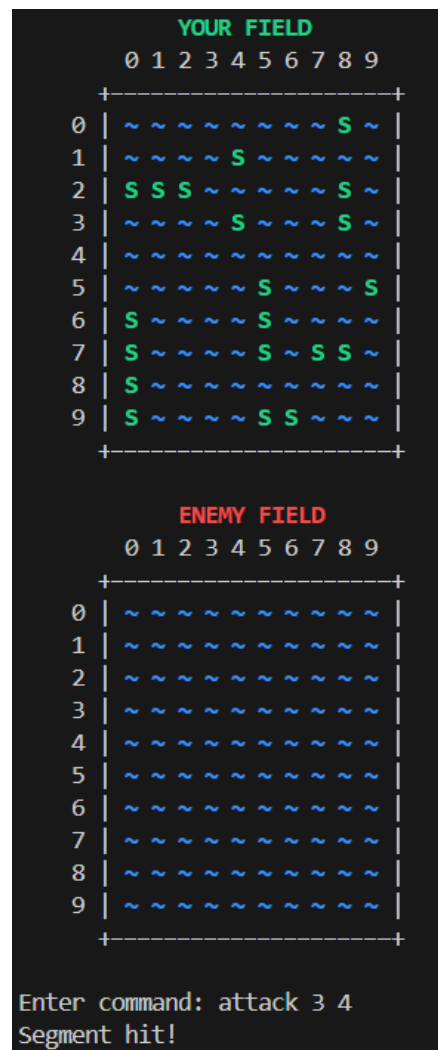


Рисунок 2 — Работа программы

## **Выводы**

Для достижения цели создания модульной и расширяемой системы управления игрой, разделенной на управление, отображение и логику, с возможностью сохранения и загрузки, был разработан набор классов.

## ПРИЛОЖЕНИЕ А

### ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: GameManager.cpp

```
#ifndef GAMEMANAGER_HPP
#define GAMEMANAGER_HPP

#include "FieldRenderer.hpp"
#include "Game.hpp"
#include "Command.hpp"
#include "Context.hpp"
#include "CLICommandReader.hpp"

template <class renderer, class reader>
class GameManager {
public:
    GameManager(){};
    void run() {
        game = new Game;
        gameisrunning = true;

        while (gameisrunning) {
            renderFields();
            processTurn();
        }
    }

private:
    bool gameisrunning;
    reader commandReader;
    renderer fieldRenderer;
    Game* game;
    void processTurn() {
        std::string input;
        std::string commandResult;
        while (true) {
            std::cout << "Enter command: ";
            std::getline(std::cin, input);
            if (input == "exit") {
                gameisrunning = false;
                break;
            }
            try {
                Command command = commandReader.parseCommand(input);
                Context context;
                switch (command.type) {
                    case CommandType::Attack:
                        context.SetParam("x",
std::to_string(command.x));
                        context.SetParam("y",
std::to_string(command.y));
                        commandResult = game->MakeTurn(context);

                        std::cout << commandResult << std::endl;
                        waitEnter();
                }
            }
        }
    }
};
```

```

        break;
        case CommandType::Ability:
            context.SetParam("x",
std::to_string(command.x));
            context.SetParam("y",
std::to_string(command.y));
            commandResult =
game->UseAbilityOnComputer(context);

            std::cout << commandResult << std::endl;
            waitForEnter();
            break;
        case CommandType::Load:
            commandResult = game->Load(context);

            std::cout << commandResult << std::endl;
            waitForEnter();
            break;
        case CommandType::Save:
            commandResult = game->Save(context);

            std::cout << commandResult << std::endl;
            waitForEnter();
            break;
        case CommandType::PreviewAbility:
            commandResult =
game->PreviewNextAbility(context);

            std::cout << commandResult << std::endl;
            waitForEnter();
            break;
        case CommandType::Invalid:
            std::cout << "Invalid Command" << std::endl;
            break;
    }
    break;
} catch (const InvalidCommandException& e) {
    std::cerr << "Error: " << e.what() << std::endl;
}
}

void renderFields() {
    std::vector<std::vector<std::string>> playerField =
game->GetField(true);
    std::vector<std::vector<std::string>> enemyField =
game->GetField(false);
    std::cout << fieldRenderer.RenderField(playerField, false)
<< std::endl;
    std::cout << fieldRenderer.RenderField(enemyField, true) <<
std::endl;
}

void waitForEnter(){
    std::cout << "Press Enter to continue..." << std::endl;
    std::cin.get();
    system("clear");
}

```

```
    }  
};
```

```
#endif
```

## Название файла: CLICommandReader.cpp

```
#include "CLICommandReader.hpp"  
#include <iostream>  
#include <fstream>  
#include <sstream>  
#include <algorithm>  
#include <cctype>  
#include <stdexcept>
```

```
CLICommandReader::CLICommandReader(const std::string& configPath) {  
    if (!loadConfig(configPath)) {  
        // Если не удалось загрузить конфигурацию, установим значения  
по умолчанию  
        commandMap = {  
            {"attack", static_cast<CommandType>(0)},  
            {"ability", static_cast<CommandType>(1)},  
            {"load", static_cast<CommandType>(2)},  
            {"save", static_cast<CommandType>(3)},  
            {"preview", static_cast<CommandType>(4)}  
        };  
        std::cerr << "Failed to load config file: " << configPath <<  
". Using default commands." << std::endl;  
    }  
}
```

```
bool CLICommandReader::loadConfig(const std::string& configPath) {  
    std::ifstream configFile(configPath);  
    if (!configFile.is_open()) {  
        return false;  
    }  
  
    commandMap.clear();  
    std::string commandName;  
    int commandIndex = 0;  
    while (std::getline(configFile, commandName)) {  
        trim(commandName);  
        if (commandName.empty()) continue;  
        switch (commandIndex) {  
            case 0: commandMap[commandName] =  
static_cast<CommandType>(0);  
break;  
            case 1: commandMap[commandName] =  
static_cast<CommandType>(1);  
break;  
            case 2: commandMap[commandName] =  
static_cast<CommandType>(2);  
break;  
            case 3: commandMap[commandName] =  
static_cast<CommandType>(3);  
break;  
            case 4: commandMap[commandName] =  
static_cast<CommandType>(4);  
break;
```

```

        default:
            std::cerr << "Too many command config names" <<
std::endl;

            return false;
        }
        commandIndex++;
    }
    configFile.close();
    return commandIndex == 5;
}

void CLICommandReader::trim(std::string &str) {
    str.erase(str.begin(), std::find_if(str.begin(), str.end(),
[] (unsigned char ch) {
    return !std::isspace(ch);
}));
    str.erase(std::find_if(str.rbegin(), str.rend(), [] (unsigned
char ch) {
    return !std::isspace(ch);
}).base(), str.end());
}

Command CLICommandReader::parseCommand(const std::string& input) {
    std::string trimmedInput = input;
    trim(trimmedInput);
    std::istringstream iss(trimmedInput);
    std::string commandName;
    iss >> commandName;
    std::vector<std::string> args;
    std::string arg;
    while (iss >> arg) {
        args.push_back(arg);
    }

    auto it = commandMap.find(commandName);
    if (it == commandMap.end()) {
        throw InvalidCommandException("Invalid command: " +
commandName);
    }

    switch (it->second) {
        case static_cast<CommandType>(0):
            return parseAttack(args);
        case static_cast<CommandType>(1):
            return parseAbility(args);
        case static_cast<CommandType>(2):
            return parseLoad(args);
        case static_cast<CommandType>(3):
            return parseSave(args);
        case static_cast<CommandType>(4):
            return parsePreviewAbility(args);
        default:
            throw InvalidCommandException("Invalid command: " +
commandName);
    }
}

```

```

    }

    Command CLICCommandReader::parseAttack(const
std::vector<std::string>& args) {
        if (args.size() != 2) {
            throw InvalidCommandException("Invalid number of arguments
for attack command. Expected 2, but got " + std::to_string(args.size()));
        }
        try {
            int x = std::stoi(args[0]);
            int y = std::stoi(args[1]);
            return Command(static_cast<CommandType>(0), x, y);
        } catch (const std::invalid_argument& e) {
            throw InvalidCommandException("Invalid arguments for attack
command. Coordinates must be integers");
        }
    }

    Command CLICCommandReader::parseAbility(const
std::vector<std::string>& args) {
        if (args.empty()) {
            return Command(static_cast<CommandType>(1));
        }
        if (args.size() != 2) {
            throw InvalidCommandException("Invalid number of arguments
for ability command. Expected 0 or 2, but got " +
std::to_string(args.size()));
        }
        try {
            int x = std::stoi(args[0]);
            int y = std::stoi(args[1]);
            return Command(static_cast<CommandType>(1), x, y);
        } catch (const std::invalid_argument& e) {
            throw InvalidCommandException("Invalid arguments for ability
command. Coordinates must be integers");
        }
    }

    Command CLICCommandReader::parseLoad(const std::vector<std::string>&
args) {
        if (!args.empty()) {
            throw InvalidCommandException("Invalid number of arguments
for load command. Expected 0, but got " + std::to_string(args.size()));
        }
        return Command(static_cast<CommandType>(2));
    }

    Command CLICCommandReader::parseSave(const std::vector<std::string>&
args) {
        if (!args.empty()) {
            throw InvalidCommandException("Invalid number of arguments
for save command. Expected 0, but got " + std::to_string(args.size()));
        }
        return Command(static_cast<CommandType>(3));
    }

    Command CLICCommandReader::parsePreviewAbility(const
std::vector<std::string>& args) {

```

```

        if (!args.empty()) {
            throw InvalidCommandException("Invalid number of arguments
for preview command. Expected 0, but got " + std::to_string(args.size()));
        }
        return Command(static_cast<CommandType>(4));
    }

```

### Название файла: CLICommandReader.hpp

```

#ifndef CLICOMMANDREADER_HPP
#define CLICOMMANDREADER_HPP

#include <iostream>
#include <string>
#include <vector>
#include <map>
#include "Exceptions.hpp"
#include "Command.hpp"

class CLICommandReader {
public:
    CLICommandReader(const std::string& configPath =
"commands.conf");
    Command parseCommand(const std::string& input);

private:
    Command parseAttack(const std::vector<std::string>& args);
    Command parseAbility(const std::vector<std::string>& args);
    Command parseLoad(const std::vector<std::string>& args);
    Command parseSave(const std::vector<std::string>& args);
    Command parsePreviewAbility(const std::vector<std::string>&
args);
    void trim(std::string &str);
    bool loadConfig(const std::string& configPath);
    std::map<std::string, CommandType> commandMap;
};

#endif

```