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: Шаблонны классы

Студент гр. 3341 Первалов П.И.

Преподаватель Жангиров Т.Р.

Санкт-Петербург

2024

Цель работы

Реализовать шаблонные классы для управления процессом игры.

Задание

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.

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.

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.

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

В процессе продолжения разработки игры "Морской бой" были реализованы классы для организации управления игровым процессом: GameController, GameDisplay, TerminalInputProcessor, TerminalRenderer.

Класс GameController

Класс предназначен для управления игровым процессом через обработку команд пользователя. Игрок вводит команды (например, атака, сохранение, загрузка или использование способности), которые обрабатываются игровым контроллером, а затем выполняются соответствующие действия в объекте игры.

Краткая реализация:

- Шаблонный класс GameController используется для связывания объекта игры и процессора ввода, что позволяет гибко задавать способ получения команд.
- Конструктор класса принимает указатель на объект игры и объект процессора ввода, инициализируя внутренние члены класса.
- Метод process_command:
  1. Получает команду от процессора ввода.
  2. Выполняет соответствующее действие в объекте игры через вызов метода, например attack(), save() и т. д.
  3. Если команда неизвестна, выводится сообщение об ошибке.

Класс GameDisplay

Класс GameDisplay отвечает за визуализацию состояния игры. Он связывает объект игры с рендерером, который отвечает за отрисовку. Это позволяет отделить игровую логику от процесса отображения, обеспечивая гибкость и модульность программы.

Краткая реализация:

- Класс шаблонный, параметризованный типом Renderer, что позволяет использовать разные способы визуализации.
- Конструктор принимает указатель на объект игры (Game) и экземпляр рендерера, инициализируя соответствующие члены.

- Метод render вызывает метод render у объекта рендерера, передавая ему указатель на игру для отображения текущего состояния.

Класс TerminalInputProcessor

Класс TerminalInputProcessor используется для обработки ввода с клавиатуры в игровом приложении. Он позволяет связывать нажатия клавиш с командами, загружая настройки из файла, либо используя предустановленные команды. Это обеспечивает гибкость и удобство настройки управления.

Краткая реализация:

- Класс содержит приватное поле command_map, которое хранит сопоставление клавиш с командами.
- Метод load_commands_from_file загружает настройки управления из указанного файла. Если файл недоступен, применяется стандартный набор команд (attack, load, save, use_ability).
- Конструктор вызывает метод загрузки, инициализируя управление с учётом переданного файла настроек.
- Метод get_command считывает символ с консоли, который интерпретируется как команда пользователя.
- В случае ошибок загрузки настроек из файла выводится предупреждение, и управление переключается на стандартные настройки.

Класс TerminalRenderer

Класс TerminalRenderer отвечает за отображение состояния игры в терминале. Он преобразует данные игры в визуальное представление для игрока, выводя информацию о текущем игровом процессе в консоль. Этот класс используется для реализации рендеринга в текстовом интерфейсе.

Краткая реализация:

- Класс реализует метод render, который принимает указатель на объект игры.
- В методе render выводится сообщение о рендеринге игрового поля. Затем вызывается метод display_playing_fields у объекта игры,

который отображает текущее игровое поле. Это позволяет пользователю видеть состояние игры в терминале.

- Рендеринг ограничивается выводом информации в текстовом формате, соответствующем терминальному интерфейсу.

Класс ориентирован на простое текстовое отображение данных игры, обеспечивая вывод информации о текущем игровом процессе в консоль.

Архитектурные решения

Разделение ответственности (Separation of Concerns):

В проекте чётко разделены различные компоненты, каждый из которых отвечает за свою часть функционала:

GameController — управляет логикой взаимодействия с игрой, обрабатывая команды от пользователя.

GameDisplay — отображает состояние игры, разделяя логику игры и визуализацию, что способствует расширяемости.

TerminalInputProcessor — обрабатывает пользовательский ввод, предоставляя гибкую настройку управления через файл или с помощью стандартных команд.

TerminalRenderer — отвечает за отображение состояния игры в текстовом виде, позволяя выводить данные на экран.

Использование шаблонов:

Класс GameController является шаблонным и может работать с любым типом процессора ввода, что позволяет легко интегрировать различные способы получения команд. Это решение повышает гибкость и переиспользуемость кода.

Гибкость и конфигурируемость:

Класс TerminalInputProcessor позволяет загружать настройки управления из внешнего файла. Это даёт возможность изменять поведение игры без необходимости менять исходный код, а также использовать стандартные или пользовательские настройки управления. Подход с файлом конфигурации

способствует лёгкой настройке и адаптации игры под разные предпочтения пользователей.

Модульность и расширяемость:

Каждый из классов выполняет отдельную задачу, и изменения в одном компоненте (например, добавление новых команд или изменение способа рендеринга) не влияют на остальные части программы. Это облегчает добавление новых функциональностей, например, новых команд или способов отображения.

Обработка ошибок:

В классе TerminalInputProcessor предусмотрена обработка ошибок при открытии файла команд, что повышает надёжность приложения. В случае ошибки загружаются дефолтные команды, что гарантирует непрерывную работу программы.

Текстовый интерфейс:

Все классы, работающие с визуализацией и вводом, ориентированы на текстовый интерфейс, что упрощает работу с терминалом и делает систему доступной для разработчиков с минимальными требованиями к графике.

Интерфейсы и взаимодействие:

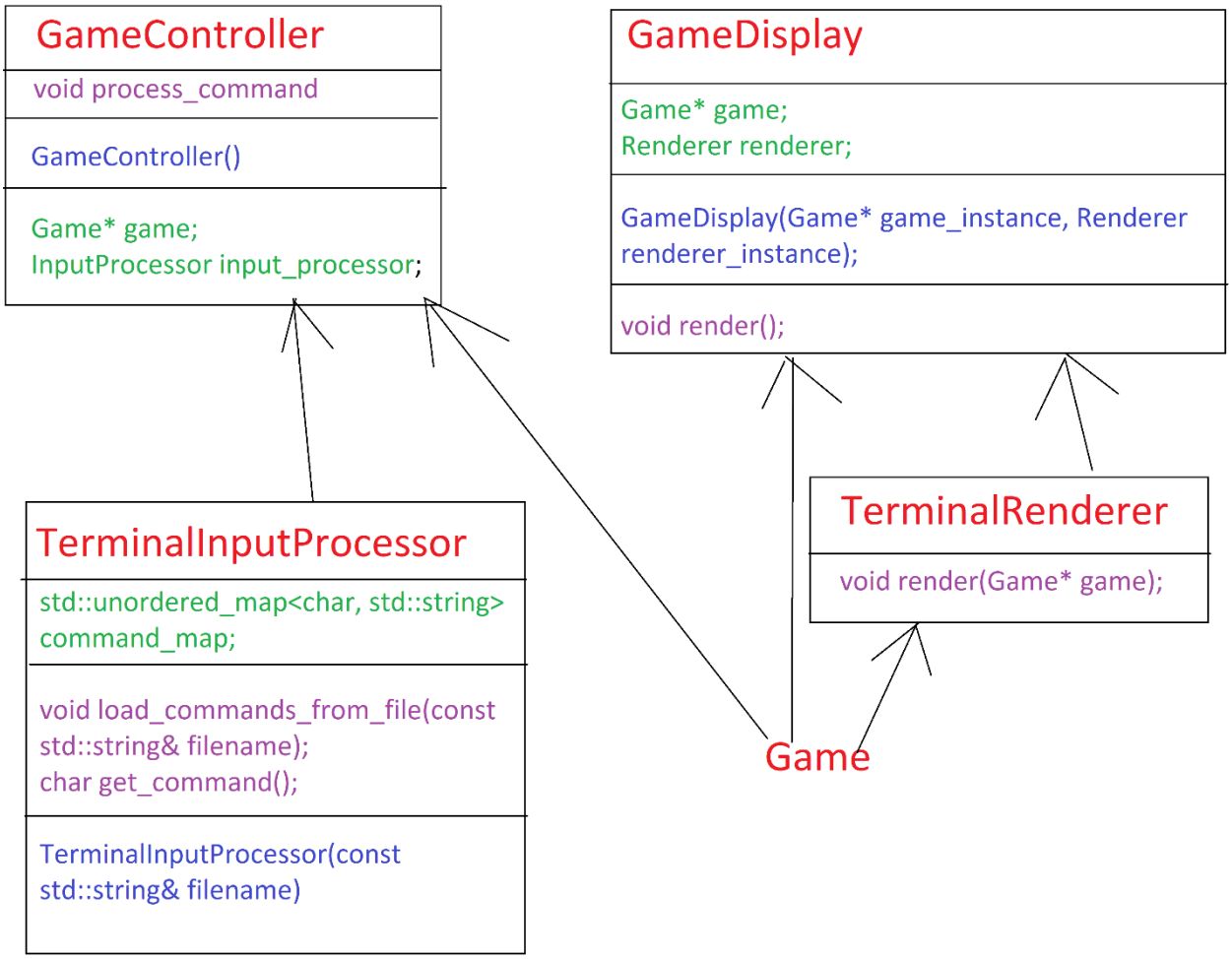
Классы взаимодействуют между собой через чётко определённые интерфейсы. Например, GameController использует TerminalInputProcessor для получения команд и GameDisplay для отображения игры, что упрощает понимание архитектуры и взаимодействия между компонентами.

```
norton@LAPTOP-563HCLJQ:/mnt/c/Users/79969/desktop/workspace/sem3/oop/sea_battle/src$ make
g++ -c main.cpp
g++ -c ship.cpp
g++ -c field_cell.cpp
g++ -c ship_manager.cpp
g++ -c sea_battle_playground.cpp
g++ -c exceptions.cpp
g++ -c ability.cpp
g++ -c double_damage.cpp
g++ -c random_bombardment.cpp
g++ -c scanner.cpp
g++ -c ability_manager.cpp
g++ -c game_state.cpp
g++ -c game.cpp
g++ -c terminal_renderer.cpp
g++ main.o ship.o field_cell.o ship_manager.o sea_battle_playground.o exceptions.o ability.o double_damage.o random_bombardment.o
scanner.o ability_manager.o game_state.o game.o terminal_renderer.o -o game
rm *.o
```

```cpp
#include <iostream>
#include <exception>
#include <vector>
#include <ctime>

#include "sea_battle_playground.h"
#include "ship_manager.h"
#include "ability_manager.h"
#include "game_state.h"
#include "game.h"
#include "game_controller.h"
#include "game_display.h"
#include "terminal_input_processor.h"
#include "terminal_renderer.h"

int main() {
    std::srand(std::time(0));
    try {
        std::cout << "a - attack\nu - use ability\nl - load game save\ns - save game\n";

        int field_width = 10;
        int field_length = 10;
        std::vector<int> ship_sizes = {1, 2, 3, 4};

        SeaBattlePlayground user_field(field_width, field_length);
        SeaBattlePlayground enemy_field(field_width, field_length);
        ShipManager user_ships(ship_sizes.size(), ship_sizes);
        ShipManager enemy_ships(ship_sizes.size(), ship_sizes);
        AbilityManager user_abilities;
        GameState game_state(&user_field, &enemy_field, &user_abilities, &user_ships, &enemy_ships);
        Game sea_battle(&user_field, &enemy_field, &user_ships, &enemy_ships, &user_abilities, &game_state);

        TerminalInputProcessor input_processor("commands.txt");
        TerminalRenderer renderer;

        GameController<TerminalInputProcessor> controller(&sea_battle, input_processor);
        GameDisplay<TerminalRenderer> display(&sea_battle, renderer);

        while (true) {
            try {
                display.render();
                controller.process_command();
            }
            catch (const std::exception& e) {
                std::cout << "Exception: " << e.what() << std::endl;
            }
            catch (const char* e) {
                std::cerr << "Error: " << e << std::endl;
            }
        }
    }
    catch (const std::exception& e) {
        std::cout << "Exception: " << e.what() << std::endl;
    }
    catch (const char* e) {
        std::cerr << "Error: " << e << std::endl;
    }
    return 0;
}
```

```
norton@LAPTOP-563HCLJQ:/mnt/c/Users/79969/desktop/workspace/sem3/oop/sea_battle/src$ ./game
a - attack
u - use ability
l - load game save
s - save game
q - quit
Start of round 1...
Rendering game field...
Your field:
  0  1  2  3  4  5  6  7  8  9
0 ~  ~  ~  ~  ~  ~  ~  2  2  ~
1 ~  ~  ~  ~  ~  ~  ~  ~  ~  ~
2 ~  ~  ~  ~  ~  ~  ~  ~  ~  ~
3 ~  ~  ~  ~  ~  ~  ~  ~  ~  ~
4 ~  ~  ~  ~  ~  ~  2  ~  ~  ~
5 ~  ~  ~  ~  ~  ~  2  ~  ~  ~
6 ~  ~  ~  ~  ~  ~  2  ~  ~  ~
7 ~  ~  ~  ~  2  ~  ~  ~  ~  ~
8 ~  ~  ~  ~  ~  ~  ~  ~  ~  ~
9 ~  ~  2  2  2  2  ~  ~  ~  ~
Enemy field:
  0  1  2  3  4  5  6  7  8  9
0 ~  ~  ~  ~  ~  ~  ~  ~  ~  ~
1 ~  ~  ~  ~  ~  ~  ~  ~  ~  ~
2 ~  ~  ~  ~  ~  ~  ~  ~  ~  ~
3 ~  ~  ~  ~  ~  ~  ~  ~  ~  ~
4 ~  ~  ~  ~  ~  ~  ~  ~  ~  ~
5 ~  ~  ~  ~  ~  ~  ~  ~  ~  ~
6 ~  ~  ~  ~  ~  ~  ~  ~  ~  ~
7 ~  ~  ~  ~  ~  ~  ~  ~  ~  ~
8 ~  ~  ~  ~  ~  ~  ~  ~  ~  ~
9 ~  ~  ~  ~  ~  ~  ~  ~  ~  ~
```

UML диаграмма классов отображена ниже. На ней зеленым цветом отображены поля классов, синим цветом конструкторы/деструкторы, а фиолетовым – методы. Также на диаграмме отображены связи между классами.



Разработанный программный код см. в приложении А.

Выводы

В ходе разработки были созданы классы, которые помогли организовать управление игровым процессом.

ПРИЛОЖЕНИЕ А
ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: main.cpp

```
#include <iostream>
#include <exception>
#include <vector>
#include <ctime>

#include "sea_battle_playground.h"
#include "ship_manager.h"
#include "ability_manager.h"
#include "game_state.h"
#include "game.h"
#include "game_controller.h"
#include "game_display.h"
#include "terminal_input_processor.h"
#include "terminal_renderer.h"

int main() {
    std::srand(std::time(0));
    try {
        std::cout << "a - attack\nu - use ability\nl - load game save\ns - save game\nq - quit\n";

        int field_width = 10;
        int field_length = 10;
        std::vector<int> ship_sizes = {1, 2, 3, 4};

        SeaBattlePlayground user_field(field_width, field_length);
        SeaBattlePlayground enemy_field(field_width, field_length);
        ShipManager user_ships(ship_sizes.size(), ship_sizes);
        ShipManager enemy_ships(ship_sizes.size(), ship_sizes);
        AbilityManager user_abilities;
        GameState game_state(&user_field, &enemy_field, &user_abilities, &user_ships, &enemy_ships);
        Game sea_battle(&user_field, &enemy_field, &user_ships, &enemy_ships, &user_abilities, &game_state);
```

```
            TerminalInputProcessor input_processor("commands.txt");
            TerminalRenderer renderer;

            GameController<TerminalInputProcessor>
controller(&sea_battle, input_processor);
            GameDisplay<TerminalRenderer> display(&sea_battle, renderer);

            while (true) {
                try {
                    display.render();
                    bool action = controller.process_command();
                    if(!action) break;
                }
                catch (const std::exception& e) {
                    std::cout << "Exception: " << e.what() << std::endl;
                }
                catch (const char* e) {
                    std::cerr << "Error: " << e << std::endl;
                }
            }
        }
        catch (const std::exception& e) {
            std::cout << "Exception: " << e.what() << std::endl;
        }
        catch (const char* e) {
            std::cerr << "Error: " << e << std::endl;
        }
        return 0;
    }
```

Название файла: game_controller.h

```
    #ifndef GAME_CONTROLLER_H
    #define GAME_CONTROLLER_H

    #include <iostream>
    #include "game.h"

    template<typename InputProcessor>
    class GameController {
```

```
    public:
        Game* game;
        InputProcessor input_processor;

        GameController(Game* game_instance, InputProcessor processor) :
game(game_instance), input_processor(processor) {}
        bool process_command() {
            char command = input_processor.get_command();
            switch (command) {
                case 'a': game->attack(); return true;
                case 's': game->save(); return true;
                case 'l': game->load(); return true;
                case 'u': game->use_ability(); return true;
                case 'q': return false;
                default: std::cout << "Unknown command!" << std::endl;
            }
            return true;
        }
    };

    #endif
```

Название файла: terminal_input_processor.h

```
    #ifndef TERMINAL_INPUT_PROCESSOR_H
    #define TERMINAL_INPUT_PROCESSOR_H

    #include <iostream>
    #include <fstream>
    #include <unordered_map>

    class TerminalInputProcessor {
    private:
        std::unordered_map<char, std::string> command_map;
        void load_commands_from_file(const std::string& filename){
            std::ifstream file(filename);
            if (!file.is_open()) {
                std::cerr << "Error opening command file, using default
controls." << std::endl;
```

```
                command_map = {{'a', "attack"}, {'l', "load"}, {'s',
"save"}, {'u', "use_ability"}, {'q', "quit"}};
                return;
            }

            char key;
            std::string command;
            while (file >> key >> command) {
                command_map[key] = command;
            }
        }

    public:
        TerminalInputProcessor(const std::string& filename){
            load_commands_from_file(filename);
        }

        char get_command(){
            char input;
            std::cin >> input;
            return input;
        }
    };

    #endif
```

Название файла: terminal_renderer.cpp

```
#include "terminal_renderer.h"

void TerminalRenderer::render(Game* game) {
     std::cout << "Rendering game field..." << std::endl;
     game->display_playing_fields();
}
```

Название файла: terminal_renderer.h

```
#ifndef TERMINAL_RENDERER_H
#define TERMINAL_RENDERER_H

#include "game.h"
```

```
class TerminalRenderer {
public:
    void render(Game* game);
};

#endif
```

Название файла: Makefile

```
all : game
main.o : main.cpp
	g++ -c main.cpp
ship.o : ship.cpp
	g++ -c ship.cpp
field_cell.o : field_cell.cpp
	g++ -c field_cell.cpp
ship_manager.o : ship_manager.cpp
	g++ -c ship_manager.cpp
sea_battle_playground.o : sea_battle_playground.cpp
	g++ -c sea_battle_playground.cpp
exceptions.o : exceptions.cpp
	g++ -c exceptions.cpp
ability.o : ability.cpp
	g++ -c ability.cpp
double_damage.o : double_damage.cpp
	g++ -c double_damage.cpp
random_bombardment.o : random_bombardment.cpp
	g++ -c random_bombardment.cpp
scanner.o : scanner.cpp
	g++ -c scanner.cpp
ability_manager.o : ability_manager.cpp
	g++ -c ability_manager.cpp
game_state.o : game_state.cpp
	g++ -c game_state.cpp
game.o : game.cpp
	g++ -c game.cpp
terminal_renderer.o : terminal_renderer.cpp
	g++ -c terminal_renderer.cpp
game : main.o ship.o field_cell.o ship_manager.o
sea_battle_playground.o exceptions.o ability.o double_damage.o
```

```
random_bombardment.o scanner.o ability_manager.o game_state.o game.o
terminal_renderer.o
	g++ main.o ship.o field_cell.o ship_manager.o
sea_battle_playground.o exceptions.o ability.o double_damage.o
random_bombardment.o scanner.o ability_manager.o game_state.o game.o
terminal_renderer.o -o game

	rm *.o
```