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

ОТЧЕТ

по лабораторной работе №3

по дисциплине «Объектно-ориентированное программирование»

Тема: Связывание классов

Студент гр. 3381 _____ Дудин Д. С.

Преподаватель _____ Жангиров Т.Р.

Санкт-Петербург

2024

Цель работы.

Создание класса игры, реализующего игровой цикл с чередующимися ходами игрока и компьютерного врага. Реализация системы управления игрой, позволяющей начать новую игру, выполнять ходы и сохранять/загружать состояние игры. Включение в игру механизма победы и поражения, а также перенос состояния игры между раундами. Реализация сохранения и загрузки игры с использованием идиомы RAII.

Задание.

a. Создать класс игры, который реализует следующий игровой цикл:
   i. Начало игры
   ii. Раунд, в котором чередуются ходы пользователя и компьютерного врага. В свой ход пользователь может применить способность и выполняет атаку. Компьютерный враг только наносит атаку.
   iii. В случае проигрыша пользователь начинает новую игру
   iv. В случае победы в раунде, начинается следующий раунд, причем состояние поля и способностей пользователя переносятся.

Класс игры должен содержать методы управления игрой, начало новой игры, выполнить ход, и т.д., чтобы в следующей лаб. работе можно было выполнять управление исходя из ввода игрока.

b. Реализовать класс состояния игры, и переопределить операторы ввода и вывода в поток для состояния игры. Реализовать сохранение и загрузку игры. Сохраняться и загружаться можно в любой момент, когда у пользователя приоритет в игре. Должна быть возможность загружать сохранение после перезапуска всей программы.

Примечание:

- Класс игры может знать о игровых сущностях, но не наоборот
- Игровые сущности не должны сами порождать объекты состояния

- Для управления самое игрой можно использовать обертки над командами
- При работе с файлом используйте идиому RAII.

Основные теоретические положения:

Классы и интерфейсы**:** в рамках работы создаются несколько классов, включая класс игры и класс состояния игры. Класс игры управляет игровым процессом, включая выполнение ходов и переход между раундами, а класс состояния игры отвечает за сохранение и восстановление состояния игры. Переопределение операторов ввода и вывода в поток позволяет удобно работать с сохранением и загрузкой данных, обеспечивая правильное взаимодействие с пользователем.

Игровой цикл и раунды**:** игровой цикл включает чередование ходов игрока и компьютерного врага, что требует динамического изменения состояния игры. Важной частью работы является реализация механизма победы и поражения, который позволяет начать новый раунд или перезапустить игру в случае проигрыша игрока. Состояние игры должно переноситься между раундами, что требует тщательной организации данных о поле, способностях игрока и других игровых объектах.

Сохранение и загрузка состояния игры**:** для сохранения прогресса игры реализуются методы сохранения и загрузки состояния игры, что позволяет пользователю продолжить игру после перезапуска программы. Использование идиомы RAII (Resource Acquisition Is Initialization) позволяет автоматически управлять ресурсами, такими как файлы, обеспечивая корректную работу с ними. Важно, чтобы сохранение происходило только в моменты, когда у игрока есть приоритет в игре, например, в его ход.

Управление состоянием**:** управление состоянием игры важно для правильного отображения данных на экране и обеспечения правильной логики переходов между раундами. Классы состояния игры будут содержать информацию о текущем ходе, текущем состоянии поля и способностях игрока. Это позволяет эффективно отслеживать изменения в ходе игры и обеспечивать сохранение всей необходимой информации для продолжения игры после перезапуска программы.

Выполнение работы.

В данной лабораторной работе была разработана система для сохранения и загрузки состояния игры "Морской бой" с использованием формата JSON. Помимо работы с JSON, были реализованы несколько ключевых компонентов игры, таких как: файлы game.h, game.cpp, gameState.h, gameState.cpp, input.h, input.cpp, output.h, output.cpp. Для обработки данных в формате JSON был использован внешний библиотечный модуль nlohmann::json, который значительно упрощает работу с этим форматом в языке C++. Все эти элементы работы совместно обеспечивают функциональность игры, включая возможность сохранения и загрузки состояния, управление игровым процессом, а также взаимодействие с пользователем.

1. Разработка класса FileHandler

Для работы с файлами был реализован класс FileHandler, который инкапсулирует операции чтения и записи данных в файл. Этот класс включает методы:

- open_for_read() — открывает файл для чтения;
- open_for_write() — открывает файл для записи;
- write() — записывает данные в файл в формате JSON;

- read() — считывает данные из файла в формате JSON;
- close_read() и close_write() — закрывают файлы после завершения операций.

Использование этого класса позволяет избежать дублирования кода и эффективно управлять файловыми операциями.

2. Работа с игровым полем (GameField)

Класс GameField отвечает за представление игрового поля и взаимодействие с ним. В этом классе были реализованы методы для преобразования состояния поля в формат JSON и наоборот:

- to_json() — преобразует текущее состояние поля (ширину, высоту и состояние клеток) в JSON-формат.
- from_json_size() — создает объект поля из полученного JSON, задавая размер поля.
- from_json_coord() — заполняет данные поля координатами и состоянием клеток.

Эти методы позволяют сохранять и восстанавливать состояние игрового поля.

3. Управление кораблями (ShipManager)

Класс ShipManager управляет кораблями игрока и врага. Для работы с JSON были реализованы методы:

- to_json() — сохраняет координаты и состояния кораблей в формате JSON.
- from_json() — загружает данные о кораблях, включая их координаты и состояния, а также размещает их на игровом поле.

Эти методы обеспечивают возможность сериализации и десериализации данных о кораблях.

4. Управление способностями (AbilityManager)

Класс AbilityManager управляет способностями, доступными игроку. В нем реализованы методы:

- to_json() — сохраняет типы способностей и их параметры в формате JSON.
- from_json() — восстанавливает способности из данных JSON.

Эти методы обеспечивают сохранение и восстановление очереди способностей.

5. Сохранения

Для лабораторной работы была реализована система сохранения и загрузки состояния игры. В процессе выполнения реализовано:

1. Инициализировалось игровое поле и размещались корабли.
2. Состояние игры сохранялось в файл.
3. Затем игра загружалась из файла, и проверялась корректность восстановления всех данных.

Тестирование показало, что все данные (размеры поля, состояния клеток, расположение кораблей, очереди способностей) корректно сохранялись и восстанавливались.

6. Класс Input

Класс Input отвечает за обработку различных типов ввода от пользователя. Он предоставляет несколько методов для получения данных в различных форматах, обеспечивая правильность ввода с помощью встроенных проверок.

Класс использует объект Output для отображения сообщений об ошибках при некорректном вводе.

Основные методы класса:

1. **input_single_number()**

   Метод предназначен для ввода одного целого числа. Если ввод некорректен (не число), он повторяет запрос.
2. **input_two_ints()**

   Метод для ввода пары целых чисел. Пользователь должен ввести два числа, разделённых пробелом. Если ввод некорректен, метод повторно запрашивает данные.
3. **input_orientation()**

   Метод для ввода строки, представляющей ориентацию (например, направление в игре или выбор пользователя). Ввод проверяется на правильность, и в случае ошибок выводится сообщение об ошибке.
4. **input_flag()**

   Метод для ввода флага (булев тип), который интерпретирует ввод как "да" (y, Y) или "нет" (n, N). В случае некорректного ввода метод продолжает запрашивать данные до получения правильного ответа.

7. Класс Output

Файл output.cpp реализует класс Output, который отвечает за взаимодействие с пользователем через консоль. Класс включает функции для вывода:

- Сообщений о ходе игры, победителе, атаке и других событий.
- Состояния игрового поля, включая позиции кораблей, попадания и промахи.
- Состояния кораблей, таких как их повреждения и положение.

8. Работа с game.cpp

Файл game.cpp содержит основную логику игры. В нем реализован класс Game, который управляет игровым процессом, включая:

- Инициализацию игры, загрузку состояния и размещение кораблей.
- Основной цикл игры, в котором игрок и AI поочередно делают ходы.
- Сохранение и загрузку состояния игры в любой момент.

Класс взаимодействует с классами GameField, ShipManager, AbilityManager и другими, обеспечивая полный процесс игры.

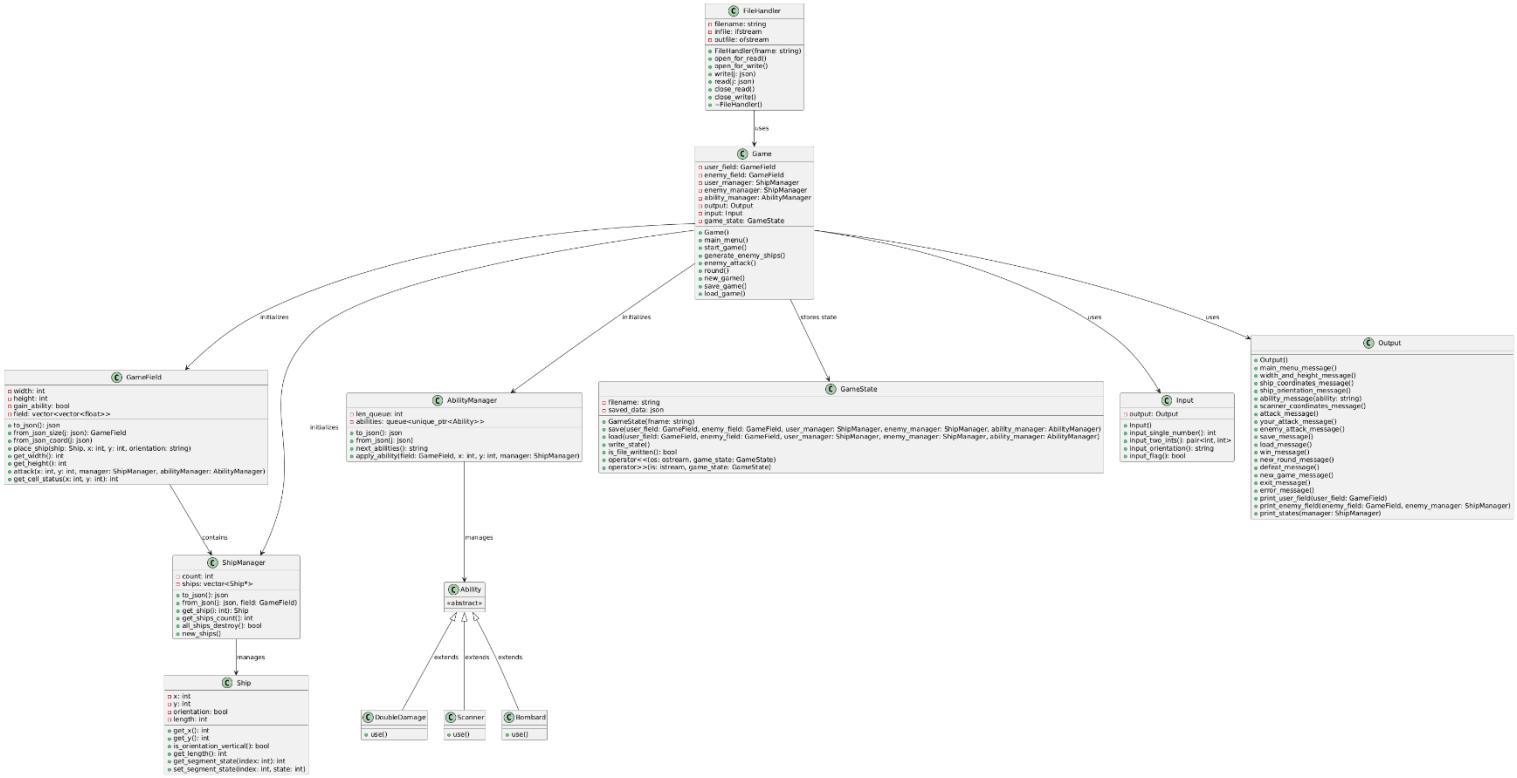
9. Использование идиомы RAII

В работе активно используется идиома RAII (Resource Acquisition Is Initialization), что позволяет эффективно управлять ресурсами (файлы, память) без явных вызовов для освобождения. Примеры использования RAII:

- Класс FileHandler автоматически открывает и закрывает файлы при создании и уничтожении объектов.
- Классы GameField, ShipManager и другие управляют памятью объектов через их создание и уничтожение в пределах области видимости, что исключает утечки памяти.

Разработанный программный код см. в приложении А.

Диаграмма классов.



Вывод

В рамках лабораторной работы был реализован функциональный проект игры "Морской бой" с возможностью сохранения и загрузки состояния игры в формате JSON. В проекте успешно использовались принципы объектно-ориентированного программирования и идиома RAII для эффективного управления ресурсами. Каждый компонент игры — от отображения информации до управления состоянием игры — был разработан с учетом модульности и четкой ответственности классов.

ПРИЛОЖЕНИЕ А
ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: main.cpp

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

class Game;

int main() {
    Game game;
    game.main_menu();
    return 0;
}
```

Название файла: abilityManager.h

```
#ifndef ABILITY_MANAGER_H
#define ABILITY_MANAGER_H

#include <iostream>
#include <queue>
#include <memory>
#include <string>
#include <random>
#include <vector>
#include <algorithm>
#include "doubleDamage.h"
#include "bombard.h"
#include "scanner.h"
#include "nlohmann/json.hpp"

using json = nlohmann::json;

class GameField;
class ShipManager;

class AbilityManager {
private:
```

```
        std::queue<std::unique_ptr<Ability>> abilities;
        int len_queue = 3;

    public:
        AbilityManager();

        void apply_ability(GameField& field, int x, int y, ShipManager&
manager);

        std::string next_abilities();

        void gain_random_ability();

        int get_len_queue();
        void set_len_queue(int value);

        void from_json(const json& j);
        json to_json();
    };

    #endif
```

Название файла: abilityManager.cpp

```
    #include "abilityManager.h"
    #include "ship.h"
    #include "shipManager.h"
    #include "ability.h"
    #include "exception.h"
    #include <algorithm>
    #include <random>


    AbilityManager::AbilityManager() {
        std::vector<std::unique_ptr<Ability>> available_abilities;

available_abilities.emplace_back(std::make_unique<DoubleDamage>());
        available_abilities.emplace_back(std::make_unique<Scanner>());
        available_abilities.emplace_back(std::make_unique<Bombard>());
```

```
        std::random_device rd;
        std::mt19937 g(rd());

        std::shuffle(available_abilities.begin(),
available_abilities.end(), g);

        for(auto &ability : available_abilities) {
            abilities.push(move(ability));
        }
    }

    void AbilityManager::apply_ability(GameField& field, int x, int y,
ShipManager& manager) {
        try {
            if (!abilities.empty()) {
                abilities.front()->apply(field, x, y, manager);
                abilities.pop();
                len_queue -= 1;
            } else {
                throw NoAbilitiesException("No abilities available.");
            }
        } catch (NoAbilitiesException& e) {
            std::cerr << e.what() << std::endl;
        }
    }

    std::string AbilityManager::next_abilities() {
        std::string ability = "";

        if (!abilities.empty()) {

            Ability* next_ability = abilities.front().get();

            if (dynamic_cast<DoubleDamage*>(next_ability)) {
                ability = "DoubleDamage";
            } else if (dynamic_cast<Scanner*>(next_ability)) {
                ability = "Scanner";
            } else if (dynamic_cast<Bombard*>(next_ability)) {
```

```
            ability = "Bombard";
        }
    }

    return ability;
}

void AbilityManager::gain_random_ability() {
    static std::random_device rd;
    static std::mt19937 gen(rd());
    std::uniform_int_distribution<> dist(0, 2);

    int random = dist(gen);
    std::unique_ptr<Ability> new_ability;

    len_queue += 1;

    switch (random) {
        case 0:
            new_ability = std::make_unique<DoubleDamage>();
            break;
        case 1:
            new_ability = std::make_unique<Scanner>();
            break;
        case 2:
            new_ability = std::make_unique<Bombard>();
            break;
    }

    abilities.push(std::move(new_ability));
    std::cout << "A new ability has been gained." << std::endl;
}

int AbilityManager::get_len_queue() {
    return len_queue;
}

void AbilityManager::set_len_queue(int value) {
    len_queue = value;
```

```
    }

    json AbilityManager::to_json() {
        json j;
        std::vector<json> ability_list;
        std::vector<std::string> temp_ability;

        for(int i = 0; i < len_queue; i++) {
            if  (dynamic_cast<DoubleDamage*>(abilities.front().get()))
{
                ability_list.push_back({"type", "DoubleDamage"});
                temp_ability.push_back("DoubleDamage");

            } else if (dynamic_cast<Scanner*>(abilities.front().get()))
{
                ability_list.push_back({"type", "Scanner"});
                temp_ability.push_back("Scanner");

            } else if (dynamic_cast<Bombard*>(abilities.front().get()))
{
                ability_list.push_back({"type", "Bombard"});
                temp_ability.push_back("Bombard");

            }
            abilities.pop();
        }

        j["abilities"] = ability_list;

        for(int i = 0; i < temp_ability.size(); i++) {
            if(temp_ability[i] == "Bombard") {

abilities.push(std::move(std::make_unique<Bombard>()));

            } else if (temp_ability[i] == "DoubleDamage") {

abilities.push(std::move(std::make_unique<DoubleDamage>()));

            } else if (temp_ability[i] == "Scanner") {
```

```
abilities.push(std::move(std::make_unique<Scanner>()));
            }
        }

        j["len_queue"] = len_queue;

        return j;
    }

    void AbilityManager::from_json(const json& j) {
        std::queue<std::unique_ptr<Ability>> empty;
        std::swap(abilities, empty);

        const auto& abilities_array = j["abilities"];
        for (const auto& ability_data : abilities_array) {
            std::string type = ability_data[1];

            if (type == "Bombard") {
                abilities.push(std::make_unique<Bombard>());

            } else if (type == "Scanner") {
                abilities.push(std::make_unique<Scanner>());

            } else if (type == "DoubleDamage") {
                abilities.push(std::make_unique<DoubleDamage>());

            } else {
                throw std::invalid_argument("Unknown ability type.");
            }
        }

        len_queue = j["len_queue"];
    }
```

Название файла: gameField.h

```
    #ifndef GAME_FIELD_H
    #define GAME_FIELD_H
```

```
#include <iostream>
#include <vector>
#include <limits>
#include "nlohmann/json.hpp"

using json = nlohmann::json;

class Ship;
class ShipManager;
class AbilityManager;

class GameField {
private:
enum cell {
    unknown_state,
    empty_state,
    ship_state
    };

    int width, height;
    cell** field;

    bool double_damage = false;
public:
    int ships_count;
    bool gain_ability;

    GameField(int width, int height);

    GameField(const GameField& other);
    GameField(GameField&& other) noexcept;

    GameField& operator=(const GameField& other);

    GameField& operator=(GameField&& other) noexcept;

    void place_ship(Ship& ship, int x, int y, std::string
orientation);
```

```
        ShipManager ship_quantity_preset();

        void draw_all_field();
        void draw_enemy_field(ShipManager& manager);
        int get_cell_status(int x, int y);
        void clean();

        void attack(int x, int y, ShipManager& manager, AbilityManager&
ability_manager);

        int get_height() const;
        int get_width() const;

        bool get_double_damage();
        void set_double_damage(bool value);

        json to_json() const;

        GameField from_json_size(const json& j);

        void from_json_coord(const json& j);

        ~GameField();
    };

#endif
```

Название файла: gameField.cpp

```
    #include "gameState.h"


    bool GameState::is_file_written() const {
            std::ifstream file(filename);
            return file.peek() != std::ifstream::traits_type::eof();
    }

    void GameState::save(GameField& user_field, GameField& enemy_field,
```

```
            ShipManager& user_manager, ShipManager& enemy_manager,
            AbilityManager& ability_manager) {
    json j;

    j["user_field"] = user_field.to_json();
    j["enemy_field"] = enemy_field.to_json();
    j["user_manager"] = user_manager.to_json();
    j["enemy_manager"] = enemy_manager.to_json();
    j["ability_manager"] = ability_manager.to_json();

    FileHandler file_handler(filename);
    file_handler.open_for_write();
    file_handler.write(j);
}

void GameState::load(GameField& user_field, GameField& enemy_field,
            ShipManager& user_manager, ShipManager& enemy_manager,
            AbilityManager& ability_manager) {
    FileHandler file_handler(filename);
    file_handler.open_for_read();

    json j;
    file_handler.read(j);

    user_field = user_field.from_json_size(j["user_field"]);
    enemy_field = enemy_field.from_json_size(j["enemy_field"]);
    enemy_manager = enemy_field.ship_quantity_preset();
    user_manager = user_field.ship_quantity_preset();
    user_manager.from_json(j["user_manager"], user_field);
    enemy_manager.from_json(j["enemy_manager"], enemy_field);
    user_field.from_json_coord(j["user_field"]);
    enemy_field.from_json_coord(j["enemy_field"]);
    ability_manager.from_json(j["ability_manager"]);
}

void GameState::write_state() {
    FileHandler file_handler(filename);
    file_handler.open_for_write();
    file_handler.write(saved_data);
```

```
}
```

Название файла: shipManager.h

```
#ifndef SHIP_MANAGER_H
#define SHIP_MANAGER_H

#include <iostream>
#include <vector>
#include "ship.h"
#include "gameField.h"
#include "nlohmann/json.hpp"

using json = nlohmann::json;

class Ship;

class ShipManager {
private:
    std::vector<std::unique_ptr<Ship>> ships;
    int count;
public:
    ShipManager(int count, const std::vector<int>& sizes);

    bool all_ships_destroy();

    int get_ships_count();

    Ship& get_ship(int index) const;

    std::vector<std::unique_ptr<Ship>>& get_ships();

    void new_ships();

    json to_json() const;

    void from_json(const json& j, GameField& field);
};
```

```
#endif
```

Название файла: shipManager.cpp

```
#include "shipManager.h"


ShipManager::ShipManager(int count, const std::vector<int>& sizes) : count(count) {
    if (count != sizes.size()) {
        throw std::invalid_argument("Count of ships must match sizes vector.");
    }

    for (int size : sizes) {
        ships.emplace_back(std::make_unique<Ship>(size, size % 2 == 0));
    }
}

bool ShipManager::all_ships_destroy() {
    int count_destroy_ships = 0;
    for(int i = 0; i < ships.size(); i++) {
        Ship& ship = *ships[i];

        int len_ship = ship.get_length();
        int count_destroy = 0;
        for(int i = 0; i < len_ship; i++) {
            int state = ship.get_segment_state(i);
            if (state == 2) {
                count_destroy++;
            }
        }
        if(count_destroy == len_ship) {
            count_destroy_ships++;
        }
    }

    if(count == count_destroy_ships) {
```

```
            return true;
        }

        return false;
    }

    int ShipManager::get_ships_count() {
        return count;
    }

    Ship& ShipManager::get_ship(int index) const {
        return *ships[index];
    }

    std::vector<std::unique_ptr<Ship>>& ShipManager::get_ships(){
            return ships;
    }

    void ShipManager::new_ships() {
        for(int i = 0; i < ships.size(); i++) {
            Ship& ship = *ships[i];
            int len_ship = ship.get_length();
            for(int j = 0; j < len_ship; j++) {
                ship.set_segment_state(j, 0);
            }
        }
    }

    json ShipManager::to_json() const {
        json j;
        j["count"] = count;

        std::vector<json> coord_ships;

        for(int i = 0; i < ships.size(); i++) {
            Ship& ship = *ships[i];
            int x = ship.get_x();
            int y = ship.get_y();
            bool orint = ship.is_orientation_vertical();
```

```
            coord_ships.push_back({x, y, orint});
        }

        j["coordinate_ships"] = coord_ships;

        json ships_array = json::array();
        for(int i = 0; i < ships.size(); i++) {
            Ship& ship = *ships[i];
            int len_ship = ship.get_length();
            std::vector<int> segment_array;
            for(int i = 0; i < len_ship; i++) {
                segment_array.push_back(ship.get_segment_state(i));
            }
            ships_array.push_back(segment_array);
        }
        j["ships"] = ships_array;
        return j;
    }

    void ShipManager::from_json(const json& j, GameField& field) {
        count = j["count"];
        const auto& ships_array = j["ships"];
        const auto& coord_ships = j["coordinate_ships"];

        for(int i = 0; i < count; i++) {
            Ship& ship = *ships[i];
            std::string orient;
            if(coord_ships[i][2] == true) {
                orient = "v";
            } else {
                orient = "h";
            }
            field.place_ship(ship,                coord_ships[i][0],
coord_ships[i][1], orient);
        }

        for(int i = 0; i < count; i++) {
            Ship& ship = *ships[i];
            int len_ship = ship.get_length();
```

```
            for(int l = 0; l < len_ship; l++) {
                ship.set_segment_state(l, ships_array[i][l]);
            }
        }
    }
```

Название файла: game.h

```
#ifndef GAME_H
#define GAME_H

#include <iostream>
#include <vector>
#include <tuple>
#include <random>
#include "exception.h"
#include "gameField.h"
#include "ship.h"
#include "abilityManager.h"
#include "shipManager.h"
#include "gameState.h"
#include "output.h"
#include "input.h"


class Game{
private:
    GameField user_field;
    GameField enemy_field;

    ShipManager user_manager;
    ShipManager enemy_manager;

    AbilityManager ability_manager;

    Output output;
    Input input;

    GameState game_state;
```

```
public:
    Game()
        : user_field(5, 5),
          enemy_field(5, 5),
          enemy_manager(enemy_field.ship_quantity_preset()),
          user_manager(user_field.ship_quantity_preset()),
          game_state("data_game.json") {}

    void main_menu();
    void start_game();

    void generate_enemy_ships();
    void enemy_attack();

    void round();
    void new_game();

    void save_game();
    void load_game();
};

#endif
```

Название файла: game.cpp

```
#include "game.h"


void Game::main_menu() {
    output.main_menu_message();

    bool input_flag = input.input_flag();
    if (input_flag) {
        load_game();
        round();
    } else{
        start_game();
        generate_enemy_ships();
        round();
```

```
    }
}

void Game::start_game() {
    int width_field, height_field;
    output.width_and_height_message();
    std::pair<int, int> size = input.input_two_ints();

    std::tie(width_field, height_field) = size;

    user_field = GameField(width_field, height_field);
    enemy_field = GameField(width_field, height_field);

    user_field.gain_ability = false;
    enemy_field.gain_ability = true;

    enemy_manager = enemy_field.ship_quantity_preset();
    user_manager = user_field.ship_quantity_preset();
    int ships_count = user_field.ships_count;

    for (int i = 0; i < ships_count; ++i) {
        Ship& ship = user_manager.get_ship(i);
        int x, y;
        std::string orientation;

        output.print_user_field(user_field);

        output.ship_coordinates_message();
        std::pair<int, int> coordinates = input.input_two_ints();

        std::tie(x, y) = coordinates;

        output.ship_orientation_message();
        orientation = input.input_orientation();

        user_field.place_ship(ship, x, y, orientation);
    }

    output.print_user_field(user_field);
```

```
    }


    void Game::generate_enemy_ships() {
        srand(static_cast<unsigned int>(time(NULL)));
        int width = enemy_field.get_width();
        int height = enemy_field.get_height();
        std::vector<std::vector<int>>              cell_status(height,
std::vector<int>(width, 0));

        for (int i = 0; i < enemy_manager.get_ships_count(); i++) {
            Ship& ship = enemy_manager.get_ship(i);
            int length = ship.get_length();
            bool placed = false;

            while (!placed) {
                int x = rand() % width;
                int y = rand() % height;
                std::string orientation = (rand() % 2 == 0) ? "h" :
"v";
                bool can_place = true;

                if (x > width || y > height || x < 0 || y < 0 || x +
length > width || y + length > height) {
                    continue;
                }

                for (int j = 0; j < length; j++) {
                    int check_x = (orientation == "h") ? x + j : x;
                    int check_y = (orientation == "h") ? y : y + j;

                    if (check_x >= 0 && check_x < width && check_y >=
0 && check_y < height) {
                        if (cell_status[check_y][check_x] != 0) {
                            can_place = false;
                            break;
                        }
                    } else {
                        can_place = false;
```

```
                        break;
                    }
                }

                if (can_place) {
                    for (int j = 0; j < length; j++) {
                        int place_x = (orientation == "h") ? x + j :
x;
                        int place_y = (orientation == "h") ? y : y +
j;
                        cell_status[place_y][place_x] = 1;
                    }

                    for (int j = -1; j <= length; j++) {
                        for (int k = -1; k <= 1; k++) {
                            int startX = (orientation == "v") ? x + k :
x + j;
                            int startY = (orientation == "v") ? y + j :
y + k;

                            if (startX >= 0 && startX < width &&
startY >= 0 && startY < height) {
                                cell_status[startY][startX] = 1;
                            }
                        }
                    }

                    enemy_field.place_ship(ship, x, y, orientation);
                    placed = true;
                }
            }
        }
        output.print_user_field(enemy_field);
    }

    void Game::round() {
        bool save_flag;
        bool load_flag;
        bool ability_flag;
```

```
        int x = 0, y = 0;


        while(!enemy_manager.all_ships_destroy()
&& !user_manager.all_ships_destroy()) {
            std::string ability = ability_manager.next_abilities();
            output.print_user_field(user_field);
            output.print_states(user_manager);
            output.print_enemy_field(enemy_field, enemy_manager);
            output.ability_message(ability);
            ability_flag = false;

            if (ability != ""){
                ability_flag = input.input_flag();
            }
            if (ability_flag) {
                if (ability == "Scanner") {
                    output.scanner_coordinates_message();
                    std::pair<int,      int>      coordinates      =
input.input_two_ints();

                    std::tie(x, y) = coordinates;
                }
                ability_manager.apply_ability(enemy_field,    x,    y,
enemy_manager);
                if (enemy_manager.all_ships_destroy()) {
                    new_game();
                }
            } else {
                output.attack_message();
                std::pair<int,        int>        coordinates        =
input.input_two_ints();

                std::tie(x, y) = coordinates;

                output.your_attack_message();
                enemy_field.attack(x,        y,        enemy_manager,
ability_manager);
```

```
                if (enemy_manager.all_ships_destroy()) {
                    new_game();
                }
            }

            output.enemy_attack_message();
            enemy_attack();

            save_flag = false;
            output.save_message();
            save_flag = input.input_flag();
            if (save_flag) {
                save_game();
            }

            load_flag = false;
            output.load_message();
            load_flag = input.input_flag();
            if (load_flag) {
                load_game();
            }

        }
        new_game();
    }

    void Game::enemy_attack() {
        std::random_device rd;
        std::mt19937 gen(rd());
        std::uniform_int_distribution<int>                  dist_x(0,
user_field.get_width() - 1);
        std::uniform_int_distribution<int>                  dist_y(0,
user_field.get_height() - 1);

        int x, y;

        do {
            x = dist_x(gen);
```

```
        y = dist_y(gen);
    } while (user_field.get_cell_status(x, y) == 1);

    user_field.attack(x, y, user_manager, ability_manager);
}

void Game::new_game() {
    bool new_game_flag;
    if (enemy_manager.all_ships_destroy()) {
        output.win_message();
        new_game_flag = input.input_flag();
        if (new_game_flag) {
            output.new_round_message();
            enemy_manager.new_ships();
            enemy_field.clean();
            generate_enemy_ships();
            round();
        } else {
            output.exit_message();
            exit(0);
        }

    } else if (user_manager.all_ships_destroy()) {
        output.defeat_message();
        new_game_flag = input.input_flag();
        if (new_game_flag) {
            output.new_game_message();
            start_game();
            generate_enemy_ships();
            round();
        } else{
            output.exit_message();
            exit(0);
        }
    }
}

void Game::save_game() {
```

```
        game_state.save(user_field,     enemy_field,     user_manager,
enemy_manager, ability_manager);
    }

    void Game::load_game() {
        game_state.load(user_field,     enemy_field,     user_manager,
enemy_manager, ability_manager);
    }
```

Название файла: gameState.h

```
    #ifndef GAME_STATE_H
    #define GAME_STATE_H

    #include <iostream>
    #include <vector>
    #include <stdexcept>
    #include <memory>
    #include <ctime>
    #include <iomanip>
    #include <sstream>
    #include <string>
    #include <fstream>
    #include "nlohmann/json.hpp"
    #include "ship.h"
    #include "shipManager.h"
    #include "abilityManager.h"
    #include "fileHandler.h"
    #include "gameField.h"

    using json = nlohmann::json;

    class GameState {
    private:
        std::string filename;
        json saved_data;

    public:
        GameState(const std::string& fname) : filename(fname) { }
```

```
        void save( GameField& user_field, GameField& enemy_field,
        ShipManager& user_manager, ShipManager& enemy_manager,
        AbilityManager& ability_manager);

        void load(GameField& user_field, GameField& enemy_field,
        ShipManager& user_manager, ShipManager& enemy_manager,
        AbilityManager& ability_manager);

        void write_state();

        bool is_file_written() const;

        friend std::ostream& operator<<(std::ostream& os, const
GameState& game_state) {
            FileHandler file_handler(game_state.filename);
            file_handler.open_for_read();

            json j;
            file_handler.read(j);

            os << j.dump(4) << std::endl;

            return os;
        }

        friend std::istream& operator>>(std::istream& is, GameState&
game_state) {
            json j;

            is >> j;
            game_state.saved_data = j;
            if (j.is_null()) {
                throw std::runtime_error("Failed to read valid JSON
data.");
            }

            game_state.write_state();
```

```
        return is;
    }

};

#endif
```

Название файла: gameState.cpp

```
#include "gameState.h"


bool GameState::is_file_written() const {
        std::ifstream file(filename);
        return file.peek() != std::ifstream::traits_type::eof();
}

void GameState::save(GameField& user_field, GameField& enemy_field,
            ShipManager& user_manager, ShipManager& enemy_manager,
            AbilityManager& ability_manager) {
    json j;

    j["user_field"] = user_field.to_json();
    j["enemy_field"] = enemy_field.to_json();
    j["user_manager"] = user_manager.to_json();
    j["enemy_manager"] = enemy_manager.to_json();
    j["ability_manager"] = ability_manager.to_json();

    FileHandler file_handler(filename);
    file_handler.open_for_write();
    file_handler.write(j);
}

void GameState::load(GameField& user_field, GameField& enemy_field,
            ShipManager& user_manager, ShipManager& enemy_manager,
            AbilityManager& ability_manager) {
    FileHandler file_handler(filename);
    file_handler.open_for_read();
```

```
    json j;
    file_handler.read(j);

    user_field = user_field.from_json_size(j["user_field"]);
    enemy_field = enemy_field.from_json_size(j["enemy_field"]);
    enemy_manager = enemy_field.ship_quantity_preset();
    user_manager = user_field.ship_quantity_preset();
    user_manager.from_json(j["user_manager"], user_field);
    enemy_manager.from_json(j["enemy_manager"], enemy_field);
    user_field.from_json_coord(j["user_field"]);
    enemy_field.from_json_coord(j["enemy_field"]);
    ability_manager.from_json(j["ability_manager"]);
}

void GameState::write_state() {
    FileHandler file_handler(filename);
    file_handler.open_for_write();
    file_handler.write(saved_data);
}
```

Название файла: fileHandler.h

```
#ifndef FILE_HANDLER_H
#define FILE_HANDLER_H

#include <iostream>
#include <vector>
#include <stdexcept>
#include <iomanip>
#include <sstream>
#include <string>
#include <fstream>
#include "nlohmann/json.hpp"

using json = nlohmann::json;

class FileHandler {
private:
    std::string filename;
```

```
    std::ifstream infile;
    std::ofstream outfile;

public:
    FileHandler(const std::string& fname) : filename(fname) {}

    void open_for_read();

    void open_for_write();

    void write(const json& j);

    void read(json& j);

    void close_read();

    void close_write();

    ~FileHandler() {
        close_read();
        close_write();
    }
};

#endif
```

Название файла: fileHandler.cpp

```
    #include "fileHandler.h"


    void FileHandler::open_for_read() {
        infile.open(filename);
        if (!infile.is_open()) {
            throw std::runtime_error("Could not open file for
reading.");
        }
    }
```

```
void FileHandler::open_for_write() {
    outfile.open(filename);
    if (!outfile.is_open()) {
        throw  std::runtime_error("Could  not  open  file  for
writing.");
    }
}

void FileHandler::write(const json& j) {
    if (outfile.is_open()) {
        outfile << j.dump(4);
    } else {
        throw std::runtime_error("File not open for writing.");
    }
}

void FileHandler::read(json& j) {
    if (infile.is_open()) {
        infile >> j;
    } else {
        throw std::runtime_error("File not open for reading.");
    }
}

void FileHandler::close_read() {
    if (infile.is_open()) {
        infile.close();
    }
}

void FileHandler::close_write() {
    if (outfile.is_open()) {
        outfile.close();
    }
}
```