**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 |  | Марков М.М. |
| 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.

1. Создать класс игры, который реализует следующий игровой цикл:
   1. Начало игры
   2. Раунд, в котором чередуются ходы пользователя и компьютерного врага. В свой ход пользователь может применить способность и выполняет атаку. Компьютерный враг только наносит атаку.
   3. В случае проигрыша пользователь начинает новую игру
   4. 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.

1. 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.

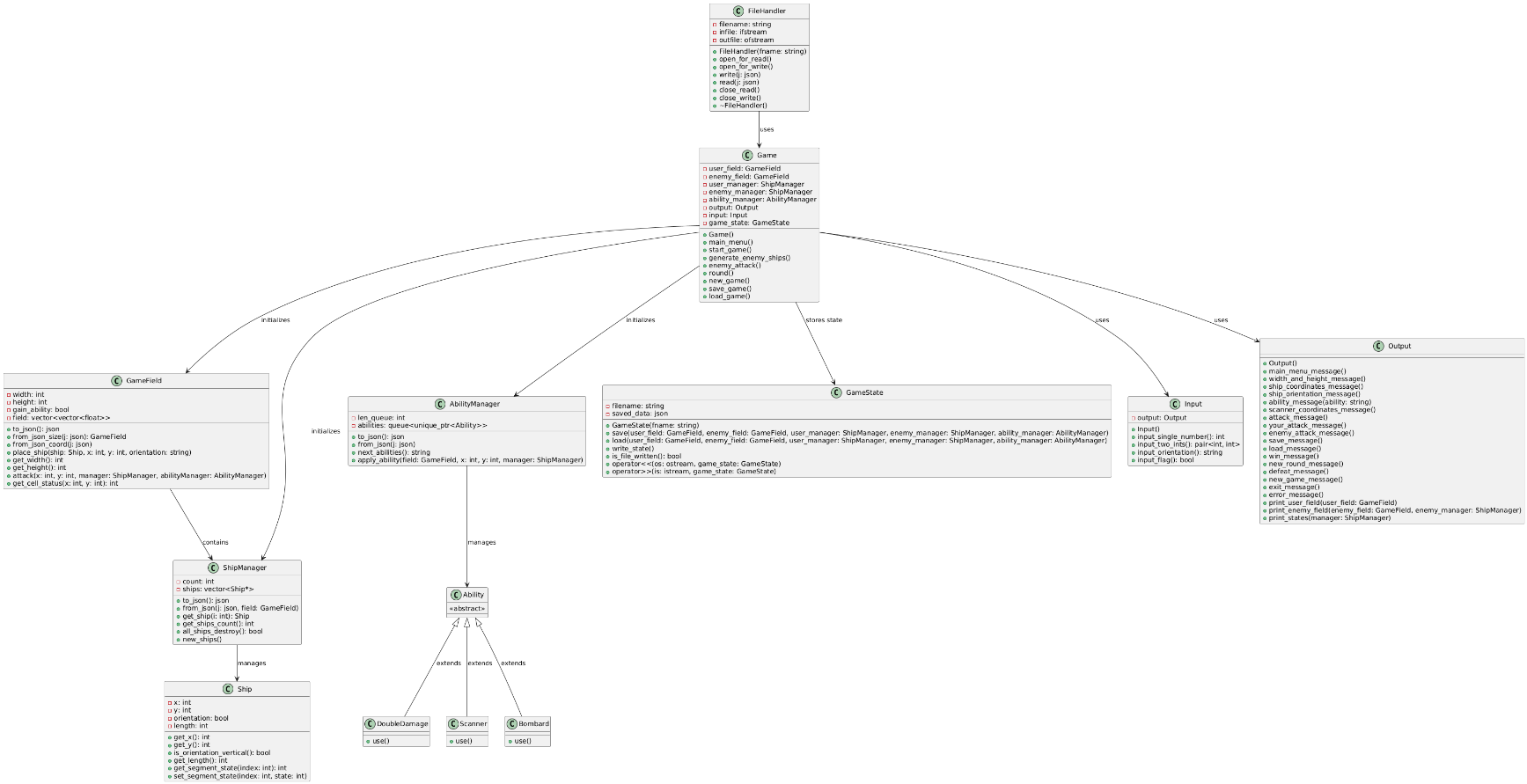
**9. Использование идиомы RAII**

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();

}

}