**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. 3341 |  | Перевалов П.И. |
| Преподаватель |  | Жангиров Т.Р. |

Санкт-Петербург

2024

## **Цель работы**

Реализовать классы игрового процесса и сохранения игры.

## **Задание**

Создать класс игры, который реализует следующий игровой цикл:

* Начало игры
* Раунд, в котором чередуются ходы пользователя и компьютерного врага. В свой ход пользователь может применить способность и выполняет атаку. Компьютерный враг только наносит атаку.
* В случае проигрыша пользователь начинает новую игру
* 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.

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.

## **Выполнение работы**

В процессе продолжения разработки игры "Морской бой" были реализованы классы для организации игрового процесса: Game, GameState.

**Класс GameState**

Этот класс был разработан для управления игровыми полями, кораблями и способностями игроков, обеспечивая их сохранение и восстановление из файлов. Рассмотрим методы класса GameState и их алгоритмы.

Конструктор GameState

Конструктор принимает указатели на игровые поля, менеджеры способностей и кораблей игроков и инициализирует их в соответствующих полях объекта.

Алгоритм:

* Сохраняет указатели на игровые поля (игрока и врага), менеджеры способностей и кораблей в приватных переменных объекта.

Метод save

Отвечает за сохранение текущего состояния игры в бинарный файл.

Алгоритм:

* Открывает файл для записи в бинарном режиме.
* Проверяет успешность открытия файла.
* Сохраняет размеры игрового поля игрока.
* Вызывает методы для сохранения состояния полей (save\_player\_field и save\_enemy\_field).
* Сохраняет информацию о кораблях игроков (save\_player\_ships и save\_enemy\_ships).
* Сохраняет список способностей игрока, записывая их имена.

Метод load

Загружает состояние игры из файла, восстанавливая состояние полей, кораблей и способностей.

Алгоритм:

* Открывает файл для чтения в бинарном режиме.
* Проверяет успешность открытия файла.
* Читает размеры игрового поля.
* Очищает текущие состояния полей.
* Вызывает методы для загрузки состояния полей (load\_player\_field и load\_enemy\_field).
* Восстанавливает состояние кораблей игроков (load\_player\_ships и load\_enemy\_ships).
* Пересвязывает корабли с клетками поля.
* Загружает способности игрока через метод load\_player\_abilities.

Методы для сохранения и загрузки полей

save\_player\_field и save\_enemy\_field

Сохраняют состояние клеток игрового поля в файл.

Алгоритм:

* Итерируются по каждой клетке игрового поля.
* Записывают состояние клетки (cell\_state), индекс корабля (ship\_index) и ориентацию корабля (vertical\_orientation).
* После записи строки делают переход на новую строку.

load\_player\_field и load\_enemy\_field

Загружают состояние клеток игрового поля из файла.

Алгоритм:

* Последовательно читают данные клетки.
* Заполняют состояние, индекс корабля и ориентацию клетки на поле.

Методы для сохранения и загрузки кораблей

save\_player\_ships и save\_enemy\_ships

Сохраняют параметры каждого корабля (длину, состояние, идентификатор и состояние HP).

Алгоритм:

* Записывают количество кораблей.
* Итерируются по каждому кораблю, записывая:
* Длину (length).
* Состояние (condition).
* Идентификатор (id).
* Состояние каждой части корпуса (hp\_bar).

load\_player\_ships и load\_enemy\_ships

Загружают параметры кораблей из файла.

Алгоритм:

* Читают количество кораблей.
* Для каждого корабля восстанавливают:
* Длину, состояние, идентификатор.
* Состояние корпуса, записывая данные в массив hp\_bar.

Методы для работы со способностями

save\_player\_abilities

Сохраняет список способностей игрока в файл, записывая их имена.

Алгоритм:

* Записывает количество способностей.
* Итерируется по способностям игрока, определяя их тип с помощью typeid и записывая имя типа в файл.

load\_player\_abilities

Загружает способности игрока на основе имен, сохраненных в файле.

Алгоритм:

* Читает количество способностей.
* Удаляет существующие способности игрока.
* По очереди считывает имя способности и добавляет соответствующую способность с помощью менеджера способностей.

**Класс Game**

Класс реализует размещение кораблей, ходы игрока и врага, проверку состояния игры, управление сохранением и загрузкой, а также контроль игрового цикла. Рассмотрим основные методы и их алгоритмы.

Конструктор Game

Инициализирует основные объекты, необходимые для игры, такие как поля, менеджеры кораблей, способностей, состояние игры и отслеживание текущего раунда.

Алгоритм:

* Устанавливает начальное значение для переменной round (равное 1).
* Инициализирует указатели на:

1. Поля игрока и противника.
2. Менеджеры кораблей игрока и противника.
3. Менеджер способностей игрока.
4. Состояние игры (GameState).

Метод place\_ships\_randomly

Размещает корабли на игровом поле случайным образом, избегая конфликтов с другими объектами.

Алгоритм:

* Очищает игровое поле (clear\_field) и восстанавливает менеджер кораблей (restore\_manager).
* Итерируется по каждому кораблю:
* Генерирует случайные координаты и ориентацию (вертикальная или горизонтальная).
* Проверяет, помещается ли корабль в указанные координаты.
* Пытается разместить корабль с помощью метода place\_ship.
* Если размещение не удалось, повторяет попытку.

Метод enemy\_turn

Реализует логику хода противника.

Алгоритм:

* Генерирует случайные координаты для атаки, проверяя, что клетка не является пустой или уже уничтоженной.
* Печатает координаты атаки.
* Атакует клетку с помощью метода attack\_cell и возвращает результат.

Метод user\_turn

Реализует логику хода игрока.

Алгоритм:

* Отображает текущие игровые поля:
* Поле игрока (display\_your\_playing\_field).
* Поле противника (display\_playing\_field\_for\_enemy).
* Спрашивает пользователя, хочет ли он сохранить игру:
* Если да, сохраняет игру с помощью метода save объекта GameState.
* Осуществляет атаку с использованием способностей игрока (ability\_attack\_cell).
* Если во время атаки возникает исключение, выводит его сообщение.

Метод is\_game\_over

Проверяет, остались ли на поле целые или поврежденные корабли.

Алгоритм:

Итерируется по всем клеткам игрового поля.

* Если находит клетку с состоянием UNDAMAGED\_SHIP или DAMAGED\_SHIP, возвращает false.
* Если ни одной такой клетки не найдено, возвращает true.

Метод start

Управляет запуском игры, включая возможность загрузки сохранения или начала новой партии.

Алгоритм:

* Спрашивает у пользователя, хочет ли он загрузить сохранение.
* Если да, загружает игру с помощью метода load объекта GameState и устанавливает флаг game\_loaded в true.
* Запускает игровой цикл:
* Если загружено сохранение, игра продолжается с текущего состояния.
* Если игра начинается с начала:

1. В первом раунде размещает корабли игрока случайным образом.
2. Размещает корабли противника на каждом этапе.
3. Запускает метод play\_round, который определяет результат текущего раунда.

Метод play\_round

Реализует игровой процесс в рамках одного раунда.

Алгоритм:

* Выводит сообщение о начале раунда.
* Организует чередующиеся ходы игрока и противника:
* Игрок делает ход, используя метод user\_turn.
* Если противник больше не имеет кораблей (is\_game\_over), выводится сообщение о победе, и раунд завершается.
* Противник делает ход, используя метод enemy\_turn.
* Если у игрока больше не осталось кораблей, выводится сообщение о поражении, и игра завершается.
* В случае окончания раунда увеличивает счетчик раундов (round).

**Архитектурные решения**

Класс GameState

Назначение: Отвечает за сохранение и восстановление состояния игры. Основной акцент сделан на сериализации и десериализации объектов.

Архитектурные решения:

* Инкапсуляция состояния: Использует поля для хранения указателей на игровые элементы (поля, менеджеры кораблей, способностей).
* Сериализация/десериализация: Методы save и load делят процесс на отдельные этапы для работы с игровыми полями, кораблями и способностями.
* Гибкость форматов: Состояние сохраняется в бинарном файле с использованием стандартных потоков ввода/вывода C++ (std::ofstream, std::ifstream).
* Модульность: Методы разделены по областям ответственности — работа с полями, кораблями, способностями.

Преимущества:

* Четкое разделение логики.
* Простота расширения для добавления новых компонентов игры.

Класс Game

Назначение: Управляет игровым процессом, включая игровой цикл, размещение кораблей, ходы игроков и проверку окончания игры.

Архитектурные решения:

* Организация игрового цикла: Методы start и play\_round структурируют игровой процесс, поддерживая возможность сохранения и загрузки.
* Разделение логики ходов: Методы user\_turn и enemy\_turn обрабатывают действия игроков отдельно, включая отображение состояния игры и выполнение атак.
* Случайное размещение: Метод place\_ships\_randomly генерирует координаты с учетом валидности позиции для кораблей.
* Интеграция с GameState: Взаимодействует с классом GameState для загрузки и сохранения игры, минимизируя дублирование кода.

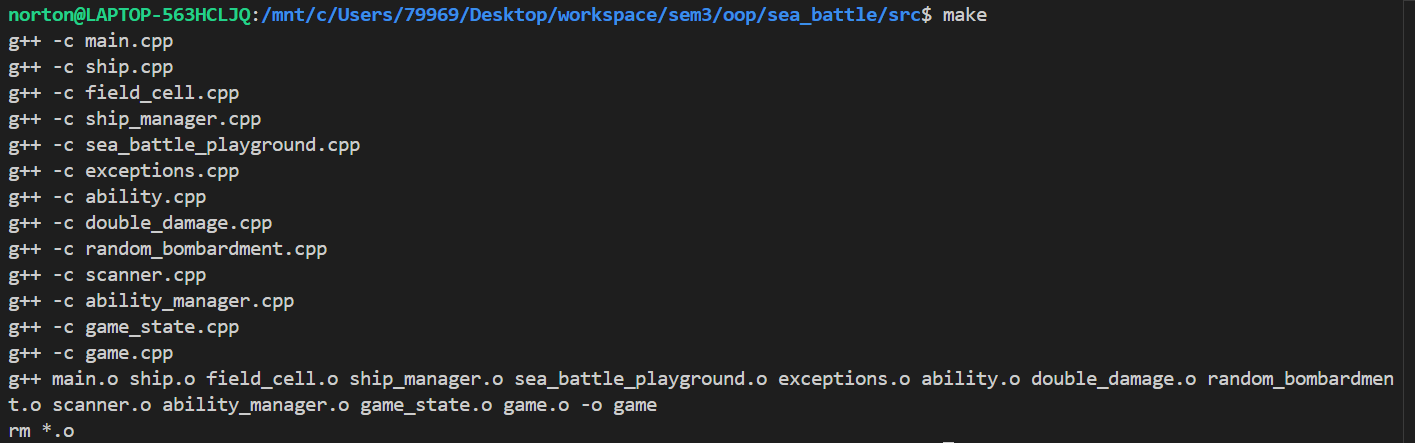
Преимущества:

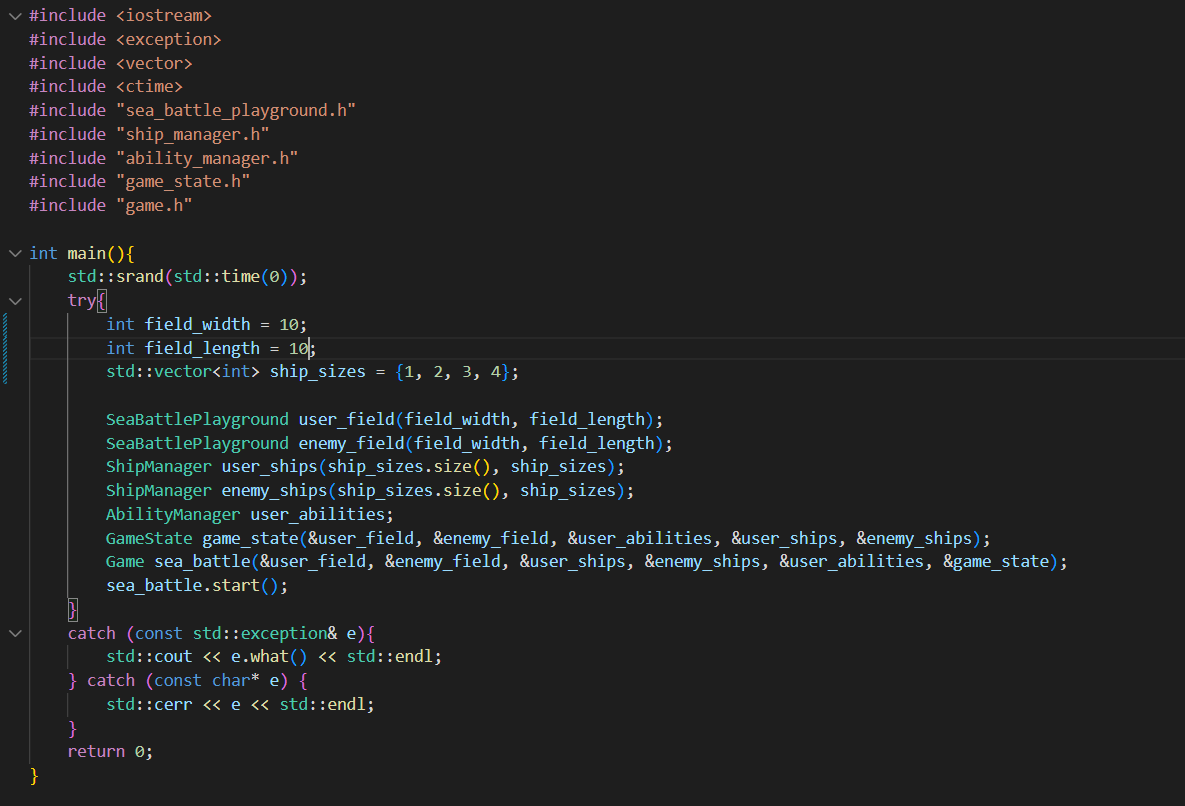
* Управление игровым процессом централизовано.
* Логика хорошо разделена по методам.
* Поддерживается сохранение и продолжение игры.

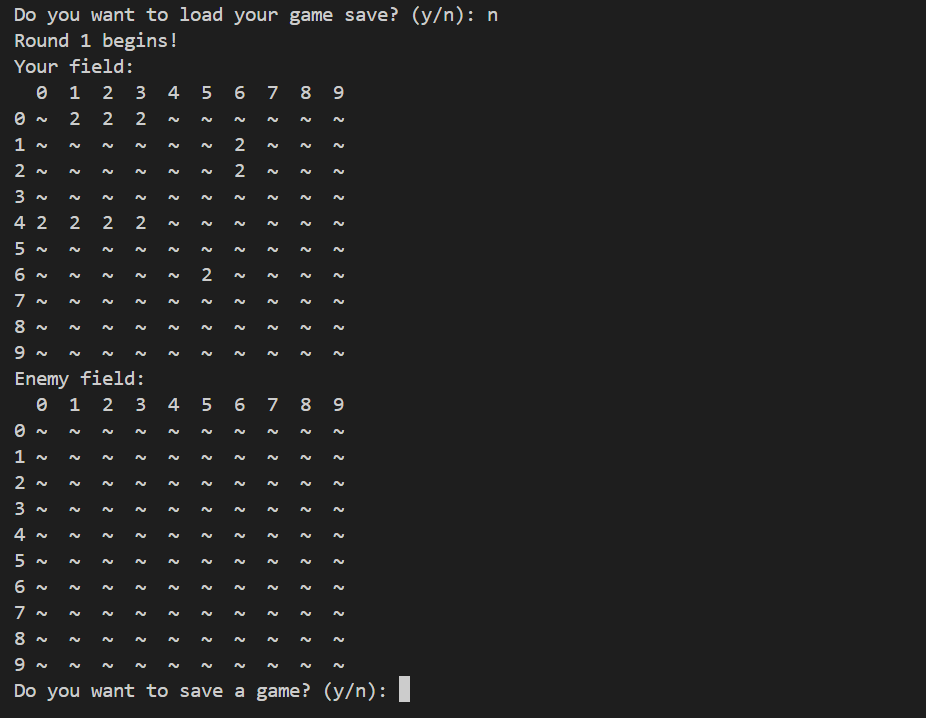
Взаимодействие классов

Класс Game использует GameState для управления состоянием игры, предоставляя модульное и легко расширяемое решение:

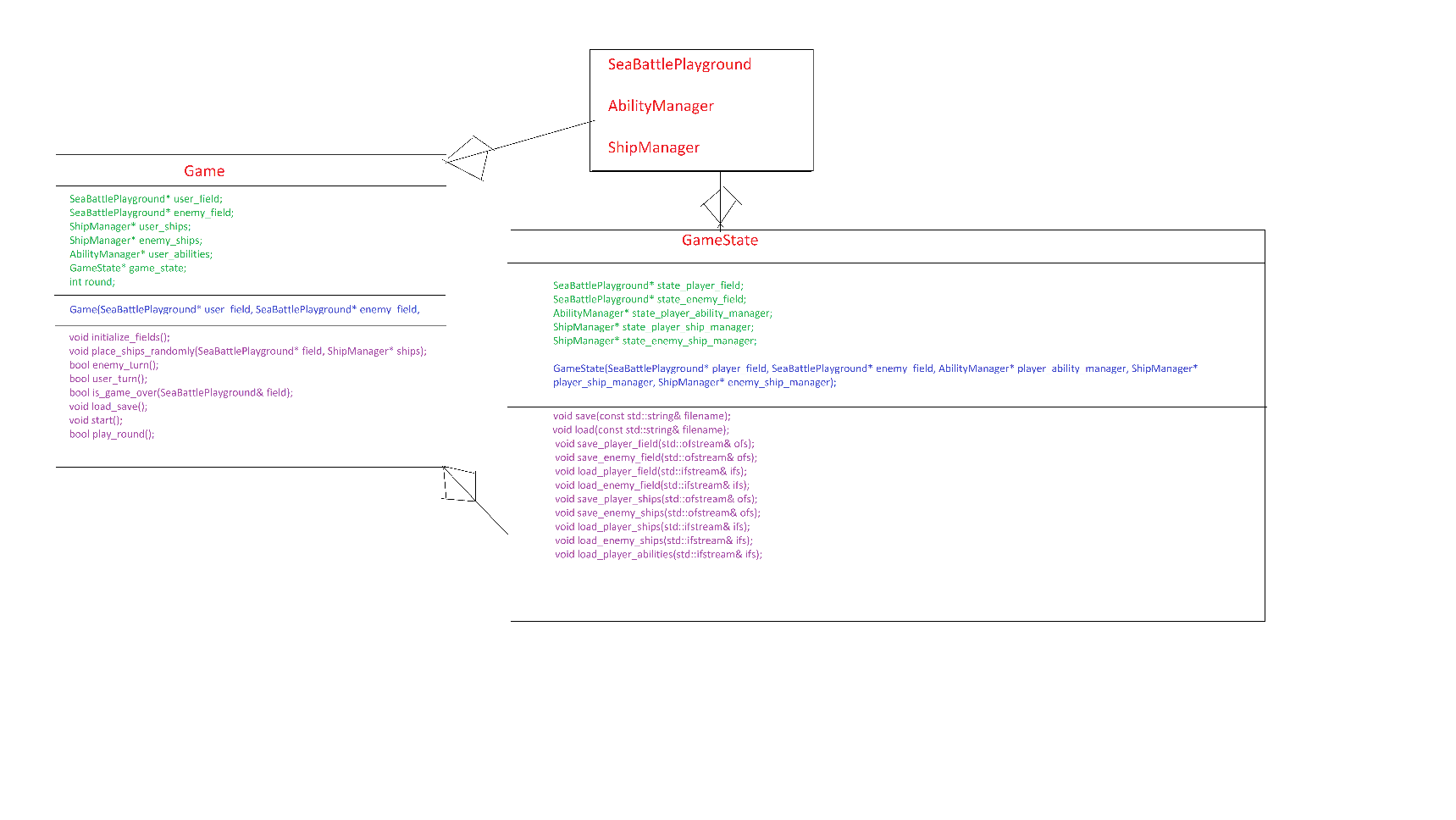
* GameState отвечает за сохранение/загрузку.
* Game реализует логику игрового процесса.

Такое разделение упрощает тестирование, поддержку и добавление новых функций. 





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 игровой процесс.

# **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"

int main(){

std::srand(std::time(0));

try{

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);

sea\_battle.start();

}

catch (const std::exception& e){

std::cout << e.what() << std::endl;

} catch (const char\* e) {

std::cerr << e << std::endl;

}

return 0;

}

Название файла: game.cpp

#include <vector>

#include <time.h>

#include "game.h"

#include "game\_state.h"

enum CellState {

EMPTY\_CELL = '.',

UNKNOWN\_CELL = '~',

DAMAGED\_SHIP = '1',

UNDAMAGED\_SHIP = '2',

DESTROYED\_SHIP = 'x'

};

void Game::place\_ships\_randomly(SeaBattlePlayground\* field, ShipManager\* ships) {

field->clear\_field();

ships->restore\_manager();

for (int i = 0; i < ships->quantity; ++i) {

bool placed = false;

time\_t counter = 0;

while (!placed) {

srand(static\_cast<unsigned int>(time(&counter)));

int x = rand() % field->width;

int y = rand() % field->length;

bool vertical = rand() % 2;

if((vertical)&&(y + ships->ships[i].length > field->length)) continue;

else{

if(x + ships->ships[i].length > field->width) continue;

}

try {

field->place\_ship(ships->ships[i], x, y, vertical, i, true);

placed = true;

} catch (...) {

counter++;

continue;

}

}

}

}

bool Game::enemy\_turn() {

int x, y;

do {

x = rand() % user\_field->width;

y = rand() % user\_field->length;

} while (user\_field->field[x][y].cell\_state == EMPTY\_CELL ||

user\_field->field[x][y].cell\_state == DESTROYED\_SHIP);

std::cout << "Enemy attacks (" << x << ", " << y << ")" << std::endl;

return user\_field->attack\_cell(x, y);

}

bool Game::user\_turn() {

user\_field->display\_your\_playing\_field();

enemy\_field->display\_playing\_field\_for\_enemy();

std::cout << "Do you want to save a game? (y/n): ";

std::string choice;

std::cin >> choice;

if(choice == "y"){

game\_state->save("game.txt");

}

try {

return enemy\_field->ability\_attack\_cell(\*user\_abilities, \*enemy\_ships);

} catch (const std::exception& e) {

std::cout << e.what() << std::endl;

return false;

}

}

bool Game::is\_game\_over(SeaBattlePlayground& field) {

for(int i = 0; i < field.length; i++){

for(int k = 0; k < field.width; k++){

if(field.field[k][i].cell\_state == UNDAMAGED\_SHIP || field.field[k][i].cell\_state == DAMAGED\_SHIP)

return false;

}

}

return true;

}

Game::Game(SeaBattlePlayground\* user\_f, SeaBattlePlayground\* enemy\_f, ShipManager\* user\_s, ShipManager\* enemy\_s, AbilityManager\* user\_a, GameState\* game\_s){

round = 1;

user\_field = user\_f;

enemy\_field = enemy\_f;

user\_ships = user\_s;

enemy\_ships = enemy\_s;

user\_abilities = user\_a;

game\_state = game\_s;

}

void Game::start() {

std::cout << "Do you want to load your game save? (y/n): ";

std::string choice;

std::cin >> choice;

bool game\_loaded = false;

if(choice == "y"){

game\_loaded = true;

game\_state->load("game.txt");

}

while (true) {

if(game\_loaded){

game\_loaded = false;

} else {

if(round == 1) place\_ships\_randomly(user\_field, user\_ships);

place\_ships\_randomly(enemy\_field, user\_ships);

}

bool result = play\_round();

if(result == false) break;

}

}

bool Game::play\_round() {

while (true) {

std::cout << "Round " << round << " begins!" << std::endl;

while (true) {

user\_turn();

if (is\_game\_over(\*enemy\_field)) {

std::cout << "You win this round!" << std::endl;

++round;

return true;

}

enemy\_turn();

if (is\_game\_over(\*user\_field)) {

std::cout << "You lose the game." << std::endl;

++round;

return false;

}

}

}

}

Название файла: game.h

#ifndef GAME\_H

#define GAME\_H

#include <vector>

#include <memory>

#include "sea\_battle\_playground.h"

#include "ship\_manager.h"

#include "ability\_manager.h"

#include "ship.h"

#include "exceptions.h"

#include "game\_state.h"

class Game {

private:

SeaBattlePlayground\* user\_field;

SeaBattlePlayground\* enemy\_field;

ShipManager\* user\_ships;

ShipManager\* enemy\_ships;

AbilityManager\* user\_abilities;

GameState\* game\_state;

int round;

void initialize\_fields();

void place\_ships\_randomly(SeaBattlePlayground\* field, ShipManager\* ships);

bool enemy\_turn();

bool user\_turn();

bool is\_game\_over(SeaBattlePlayground& field);

public:

Game(SeaBattlePlayground\* user\_field, SeaBattlePlayground\* enemy\_field, ShipManager\* user\_ships, ShipManager\* enemy\_ships, AbilityManager\* user\_abilities, GameState\* game\_state);

void load\_save();

void start();

bool play\_round();

};

#endif

Название файла: game\_state.cpp

#include <vector>

#include <iostream>

#include <typeinfo>

#include "game\_state.h"

#include "sea\_battle\_playground.h"

#include "ship\_manager.h"

#include "ability\_manager.h"

GameState::GameState(SeaBattlePlayground\* player\_f, SeaBattlePlayground\* enemy\_f, AbilityManager\* player\_ability\_m, ShipManager\* player\_ship\_m, ShipManager\* enemy\_ship\_m){

state\_player\_field = player\_f;

state\_enemy\_field = enemy\_f;

state\_player\_ability\_manager = player\_ability\_m;

state\_player\_ship\_manager = player\_ship\_m;

state\_enemy\_ship\_manager = enemy\_ship\_m;

}

void GameState::save(const std::string& filename){

std::ofstream ofs(filename, std::ios::binary);

if (!ofs.is\_open()) {

throw std::runtime\_error("Unable to open file for saving");

}

ofs << state\_player\_field->width << " " << state\_player\_field->length << "\n";

save\_player\_field(ofs);

save\_enemy\_field(ofs);

save\_player\_ships(ofs);

save\_enemy\_ships(ofs);

ofs << state\_player\_ability\_manager->abilities.size() << "\n";

for (auto ability : state\_player\_ability\_manager->abilities) {

ofs << typeid(\*ability).name() << "\n";

}

}

void GameState::load(const std::string& filename) {

std::ifstream ifs(filename, std::ios::binary);

if (!ifs.is\_open()) {

throw std::runtime\_error("Unable to open file for loading");

}

int field\_width, field\_length;

ifs >> field\_width >> field\_length;

state\_player\_field->clear\_field();

state\_enemy\_field->clear\_field();

load\_player\_field(ifs);

load\_enemy\_field(ifs);

load\_player\_ships(ifs);

load\_enemy\_ships(ifs);

for(int i = 0; i < state\_player\_field->length; i++){

for(int k = 0; k < state\_player\_field->width; k++){

int ship\_index = state\_player\_field->field[k][i].ship\_index;

if(ship\_index != -1){

state\_player\_field->tie\_the\_ship(&state\_player\_ship\_manager->ships[i], k, i);

}

}

}

for(int i = 0; i < state\_enemy\_field->length; i++){

for(int k = 0; k < state\_enemy\_field->width; k++){

if(state\_enemy\_field->field[k][i].ship\_index != -1){

int ship\_id = state\_enemy\_field->field[k][i].ship\_index;

state\_enemy\_field->tie\_the\_ship(&state\_enemy\_ship\_manager->ships[i], k, i);

}

}

}

load\_player\_abilities(ifs);

}

void GameState::save\_player\_field(std::ofstream& ofs) {

for(int i = 0; i < state\_player\_field->width; i++){

for(int k = 0; k < state\_player\_field->length; k++){

ofs << state\_player\_field->field[i][k].cell\_state << " " << state\_player\_field->field[i][k].ship\_index << " " << state\_player\_field->field[i][k].vertical\_orientation << " ";

}

ofs << "\n";

}

}

void GameState::save\_enemy\_field(std::ofstream& ofs) {

for(int i = 0; i < state\_enemy\_field->width; i++){

for(int k = 0; k < state\_enemy\_field->length; k++){

ofs << state\_enemy\_field->field[i][k].cell\_state << " " << state\_enemy\_field->field[i][k].ship\_index << " " << state\_enemy\_field->field[i][k].vertical\_orientation << " ";

}

ofs << "\n";

}

}

void GameState::load\_player\_field(std::ifstream& ifs){

char cell\_state;

int ship\_index;

bool vertical\_orientation;

for(int i = 0; i < state\_player\_field->width; i++){

for(int k = 0; k < state\_player\_field->length; k++){

ifs >> cell\_state >> ship\_index >> vertical\_orientation;

state\_player\_field->field[i][k].cell\_state = cell\_state;

state\_player\_field->field[i][k].ship\_index = ship\_index;

state\_player\_field->field[i][k].vertical\_orientation = vertical\_orientation;

}

}

}

void GameState::load\_enemy\_field(std::ifstream& ifs){

for(int i = 0; i < state\_enemy\_field->width; i++){

for(int k = 0; k < state\_enemy\_field->length; k++){

char cell\_state;

int ship\_index;

bool vertical\_orientation;

ifs >> cell\_state >> ship\_index >> vertical\_orientation;

state\_enemy\_field->field[i][k].cell\_state = cell\_state;

state\_enemy\_field->field[i][k].ship\_index = ship\_index;

state\_enemy\_field->field[i][k].vertical\_orientation = vertical\_orientation;

}

}

}

void GameState::save\_player\_ships(std::ofstream& ofs){

ofs << state\_player\_ship\_manager->quantity << "\n";

for(int i = 0; i < state\_player\_ship\_manager->quantity; i++){

ofs << state\_player\_ship\_manager->ships[i].length << " " << state\_player\_ship\_manager->ships[i].condition << " " << state\_player\_ship\_manager->ships[i].id << " ";

for(int k = 0; k < state\_player\_ship\_manager->ships[i].length; k++){

ofs << state\_player\_ship\_manager->ships[i].hp\_bar[k] << " ";

}

ofs << "\n";

}

}

void GameState::save\_enemy\_ships(std::ofstream& ofs){

ofs << state\_enemy\_ship\_manager->quantity << "\n";

for(int i = 0; i < state\_enemy\_ship\_manager->quantity; i++){

ofs << state\_enemy\_ship\_manager->ships[i].length << " " << state\_enemy\_ship\_manager->ships[i].condition << " " << state\_enemy\_ship\_manager->ships[i].id << " ";

for(int k = 0; k < state\_enemy\_ship\_manager->ships[i].length; k++){

ofs << state\_enemy\_ship\_manager->ships[i].hp\_bar[k] << " ";

}

ofs << "\n";

}

}

void GameState::load\_player\_ships(std::ifstream& ifs){

int quantity;

ifs >> quantity;

for (int i = 0; i < quantity; ++i) {

int length;

std::string condition;

int ship\_id;

ifs >> length >> condition >> ship\_id;

state\_player\_ship\_manager->ships[i].condition = condition;

for(int k = 0; k < state\_player\_ship\_manager->ships[i].length; k++){

ifs >> state\_player\_ship\_manager->ships[i].hp\_bar[k];

}

}

}

void GameState::load\_enemy\_ships(std::ifstream& ifs){

int quantity;

ifs >> quantity;

for (int i = 0; i < quantity; ++i) {

int length;

std::string condition;

int ship\_id;

ifs >> length >> condition >> ship\_id;

state\_enemy\_ship\_manager->ships[i].condition = condition;

for(int k = 0; k < state\_enemy\_ship\_manager->ships[i].length; k++){

ifs >> state\_enemy\_ship\_manager->ships[i].hp\_bar[k];

}

}

}

void GameState::load\_player\_abilities(std::ifstream& ifs) {

int ability\_count;

ifs >> ability\_count;

state\_player\_ability\_manager->delete\_abilities();

for (int i = 0; i < ability\_count; ++i) {

std::string ability\_name;

ifs >> ability\_name;

if (ability\_name == typeid(DoubleDamage).name()) {

state\_player\_ability\_manager->add\_double\_damage();

} else if (ability\_name == typeid(Scanner).name()) {

state\_player\_ability\_manager->add\_scanner();

} else if (ability\_name == typeid(RandomBombardment).name()) {

state\_player\_ability\_manager->add\_randbomb();

}

}

}

Название файла: game\_state.h

#ifndef GAME\_STATE\_H

#define GAME\_STATE\_H

#include <fstream>

#include "sea\_battle\_playground.h"

#include "ability\_manager.h"

#include "ship\_manager.h"

class GameState {

public:

SeaBattlePlayground\* state\_player\_field;

SeaBattlePlayground\* state\_enemy\_field;

AbilityManager\* state\_player\_ability\_manager;

ShipManager\* state\_player\_ship\_manager;

ShipManager\* state\_enemy\_ship\_manager;

GameState(SeaBattlePlayground\* player\_field, SeaBattlePlayground\* enemy\_field, AbilityManager\* player\_ability\_manager, ShipManager\* player\_ship\_manager, ShipManager\* enemy\_ship\_manager);

void save(const std::string& filename);

void load(const std::string& filename);

friend std::ostream& operator<<(std::ostream& os, GameState& state) {

os << "Player Field:\n";

state.state\_player\_field->display\_your\_playing\_field();

os << "Enemy Field:\n";

state.state\_enemy\_field->display\_playing\_field\_for\_enemy();

return os;

}

private:

void save\_player\_field(std::ofstream& ofs);

void save\_enemy\_field(std::ofstream& ofs);

void load\_player\_field(std::ifstream& ifs);

void load\_enemy\_field(std::ifstream& ifs);

void save\_player\_ships(std::ofstream& ofs);

void save\_enemy\_ships(std::ofstream& ofs);

void load\_player\_ships(std::ifstream& ifs);

void load\_enemy\_ships(std::ifstream& ifs);

void load\_player\_abilities(std::ifstream& ifs);

};

#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

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

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 -o game

rm \*.o