**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. 3343 |  | Силяев Р.А. |
| Преподаватель |  | Жангиров Т. Р. |

Санкт-Петербург

2024

# Цель работы

Изучить связывание классов, путём усовершенствования программы из предыдущей лабораторной работы. Необходимо создать: класс игры и класс состояния игры.

# 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.

## Выполнение работы

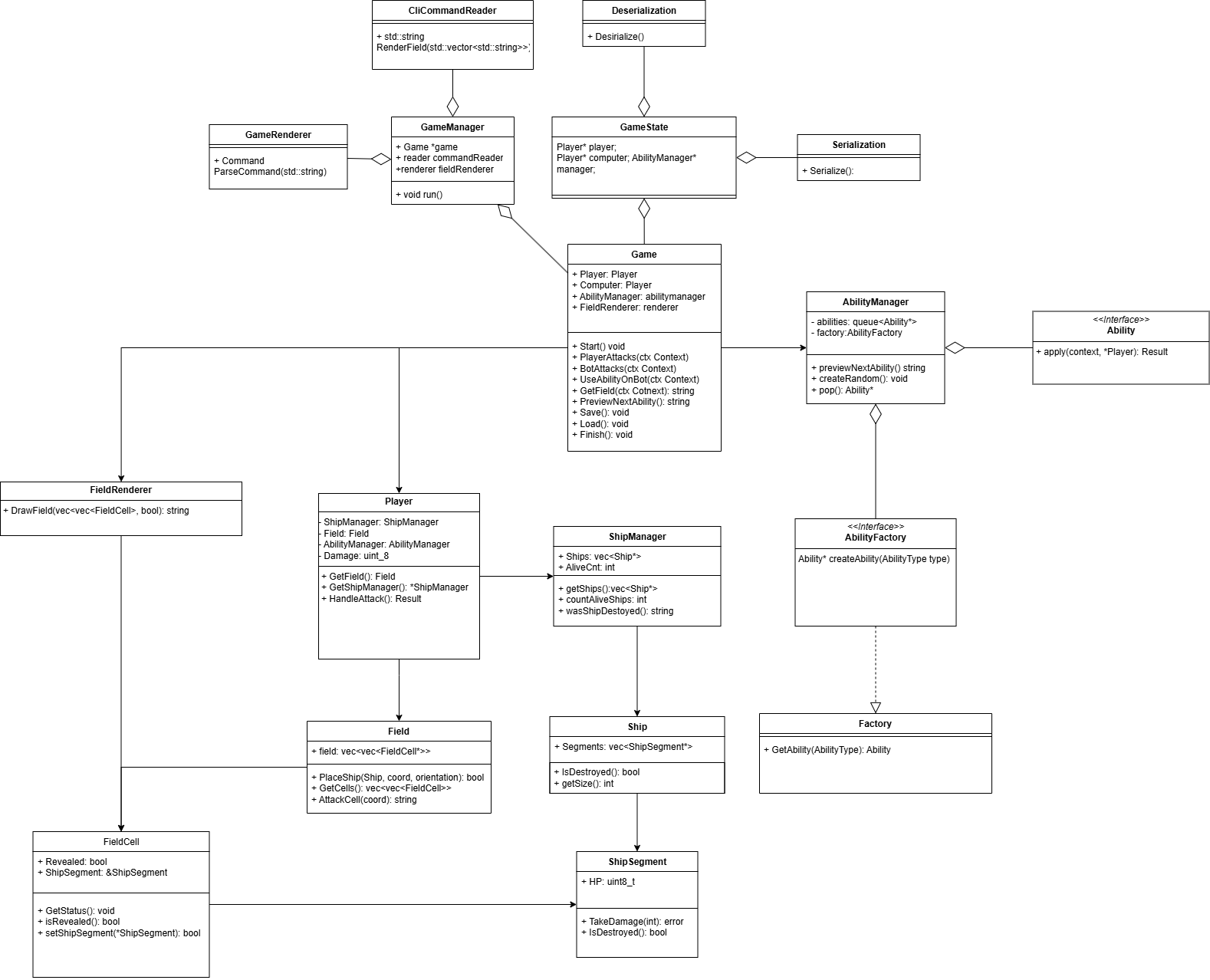


Рисунок 1 – UML-диаграмма классов

Код программы содержит реализацию классов: *Game*, *Player*, *Serialization*, *Deserialization* и *GameStatus*.

Классы *Game* и *GameState* были добавлены согласно заданию. *Game* связывает классы и работает с ними. Класс *GameState* отвечает за связывание классов *Serialization*, *Deserialization*, которые в сумме дают возможность работать с json файлом и совершать загрузку/сохранение игры.

Классы *Serialization* и *Deserialization* отвечают за считывание и запись из json файла. Прописаны методы для менеджера кораблей, поля и менеджера способностей, чтобы реализовать загрузку и сохранение игры. Обработка json файла организована с использованием библиотеки nlohmann/json.

Помимо обозначенных классов, реализованы и интегрированы в код новые классы-исключения для обработки различных исключительных случаев работы с файлом и игрой.

*Game* является классом для реализации логики игры. Он имеет следующие поля:

* *Player\* player* – класс игрока.
* *Player\* computer* – класс бота.
* *AbilityManager\* manager* – класс менеджера способностей.
* *FieldRenderer renderer* – класс отрисовщика для исключений и поля.

И следующие методы:

* *std::string MakeTurn(Context ctx)* – Сделать ход.
* *std::string UseAbilityOnComputer(Context ctx)* – использовать способность на боте.
* *void NewGame()* – метод начала игры, в котором, в зависимости от решения игрока, происходит непосредственно игра, загрузка/сохранение или выход.
* *void NewRound()* – восстанавливает компьютер и начинает новый раунд.
* *void loadGame()* – вызов загрузки игры у класса состояния.
* *void saveGame()* – вызов сохранения игры у класса состояния.

Класс *Serialization* служит для записи информации в json файл с использованием библиотеки nlohmann/json. Он имеет следующее поле:

* *nlohmann::json& j* – ссылка на структуру данных для работы с json.

Он имеет три одинаковых по структуре метода (*to\_json*) для подготовки к записи в файл менеджера кораблей, поля и менеджера способностей.

Класс *Deserialization* служит для загрузки информации из json файла. Он имеет следующее поле:

* *nlohmann::json& j* – ссылка на структуру данных для работы с json.

Он имеет три одинаковых по структуре метода (*from\_json*) для загрузки из файла менеджера кораблей, поля и менеджера способностей.

Класс *GameState* является классом состояния для связывания других классов и для реализации полной логики загрузки/сохранения игры. Он имеет следующие поля:

* *Player& player* – ссылка на игрока.
* *Playert& player* – ссылка на бота.

И следующие методы:

* *int& getCurrentDamage()* – возвращает урон.
* *void setCurrentDamage(int damage)* – выставляет урон.

# Тестирование:

Происходит симуляция игры между игроком (слева) и ботом (справа), для этого используется большая часть реализованных методов внутри классов. Поле игрока изначально открыто, а вражеское скрыто. В начале хода игрок может использовать одну случайную способность или сразу перейти к атаке вражеского поля.

В классе *Game* реализована логика игры, которая позволяет выбирать действия в зависимости от команд пользователя. Он может: запустить игру, реализовав игровой цикл, с возможностью выйти обратно после использования способности; загрузить игру, получив состояния кораблей, поля и способностей; сохранить игру, уже записав состояния игровых сущностей; выйти из игры.

При победе игроку предлагается продолжить игру с сохранением его поля и с новым противником. В случае победы бота, игру можно продолжить, обнулив вообще всё.

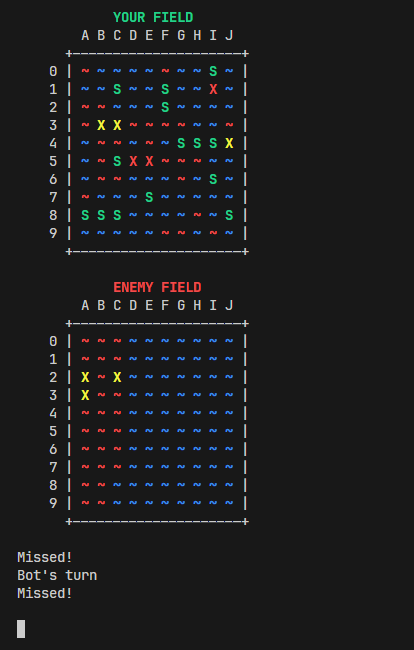
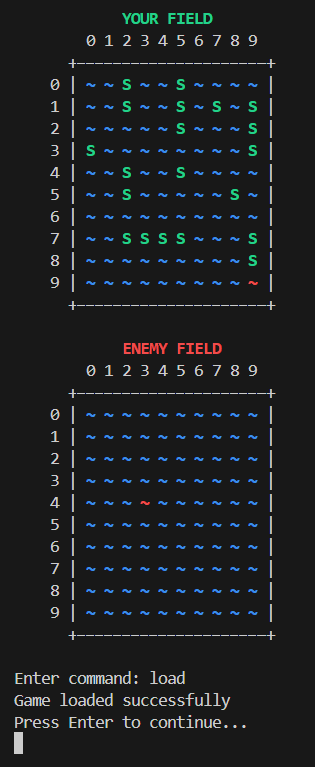
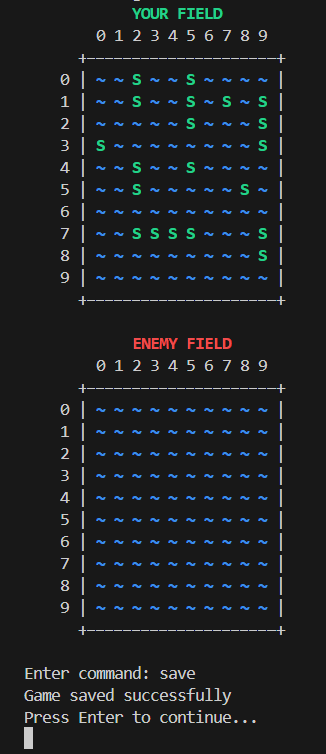


Рисунок 2 – Пример игры



# Рисунок 3 – Сохранение и загрузка

# Выводы

Во время выполнения лабораторной работы, было изучено связывание классов и созданные соответствующие заданию классы.

ПРИЛОЖЕНИЕ А  
ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: Game.cpp

#include "Game.hpp"

Game::Game() {

NewGame();

}

std::string Game::MakeTurn(Context ctx) {

std::stringstream ss;

int x, y;

// Player attacks

try {

x = std::stoi(ctx.GetParam("x"));

y = std::stoi(ctx.GetParam("y"));

} catch (std::exception& e) {

return "Invalid coordinates";

}

Result result = computer->HandleAttack(x, y);

ss << result.Message() << std::endl;

// if attack wasn't valid, computer doesn't do anything

if (!result.Success()) {

return ss.str();

}

// if all computer ships are destroyed

if (result.AllShipsDestroyed()) {

StartNewRound();

ss << "Player wins!" << std::endl;

return ss.str();

}

// Add random ability, if ship was destroyed;

for (int i = 0; i < result.ShipsDestroyed(); i++) {

abilityManager->addRandom();

ss << "New ability acquired" << std::endl;

}

if (result.Hit()) {

return ss.str();

}

// Bot's turn

ss << "Bot's turn" << std::endl;

while (true) {

int x = rand() % (computer->GetFieldCells()[0].size());

int y = rand() % (computer->GetFieldCells().size());

result = player->HandleAttack(x, y);

// if attack was valid, leave this stupid cycle

if (result.Success()) {

ss << result.Message() << std::endl;

if (!result.Hit()) {

break;

}

}

}

// if all player ships are destroyed

if (result.AllShipsDestroyed()) {

Game();

NewGame();

ss << "Computer wins!" << std::endl;

return ss.str();

}

if (result.ShipsDestroyed() > 0) {

ss << "Computer destroyed a ship!" << std::endl;

}

return ss.str();

};

std::string Game::UseAbilityOnComputer(Context ctx) {

Ability\* ability;

try {

ability = abilityManager->GetAblity();

} catch (NoAbilitiesException& e) {

return e.what();

}

Result result = ability->Apply(computer, ctx);

if (result.Success()) {

try {

abilityManager->RemoveAbilityAtStart();

} catch (NoAbilitiesException& e) {

return e.what();

}

}

// Add random ability, if ship was destroyed;

for (int i = 0; i < result.ShipsDestroyed(); i++) {

abilityManager->addRandom();

}

return result.Message();

}

std::string Game::PreviewNextAbility(Context ctx) {

return abilityManager->previewNextAbility();

}

std::vector<std::vector<std::string>> Game::GetField(bool isPlayerField){

if (isPlayerField) {

return renderer->RenderCells(player->GetFieldCells(), false);

}

return renderer->RenderCells(computer->GetFieldCells(), true);

}

void Game::NewGame() {

// initialize player

Field\* field = new Field(10, 10);

ShipManager\* playerManager = new ShipManager(7, std::vector<int>({4, 3, 3, 2, 2, 2, 1, 1, 1, 1}));

player = new Player(field, playerManager, false);

player->PlaceAllShips();

// initialize computer

Field\* computerField = new Field(10, 10);

ShipManager\* computerManager = new ShipManager(7, std::vector<int>({4, 3, 3, 2, 2, 2, 1, 1, 1, 1}));

computer = new Player(computerField, computerManager, false);

computer->PlaceAllShips();

// initialize ability manager

Factory\* factory = new Factory();

abilityManager = new AbilityManager(factory);

std::vector<int> vec;

for (int i = 0; i < 3; i++){

vec.push\_back(i);

}

std::random\_shuffle(vec.begin(), vec.end());

for (int ability: vec) {

abilityManager->addAbility(ability);

}

// // initialize field renderer

renderer = new CellsRenderer();

}

void Game::StartNewRound() {

// Reset double damage flag

computer->TakeDoubleDamage=false;

// Ressurect computer

Field\* computerField = new Field(10, 10);

ShipManager\* computerManager = new ShipManager(7, std::vector<int>({4, 3, 3, 2, 2, 2, 1}));

computer = new Player(computerField, computerManager, false);

computer->PlaceAllShips();

}

std::string Game::Save(Context ctx) {

GameState\* state = new GameState(player, computer, abilityManager);

Serializer serializer;

nlohmann::json j = serializer.to\_json(state);

std::ofstream file("save.json");

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

file.close();

Load(ctx);

return "Game saved successfully";

}

std::string Game::Load(Context ctx) {

nlohmann::json j;

std::ifstream file2("save.json");

if (file2.is\_open()) {

file2 >> j;

file2.close();

} else {

return "Save file not found";

}

Deserializer d{};

GameState\* state = d.from\_json\_game\_state(j);

player = state->GetPlayer();

computer = state->GetComputer();

abilityManager = state->GetAbilityManager();

return "Game loaded successfully";

}

Название файла: Game.hpp

#include "Player.hpp"

#include "AbilityManager.hpp"

#include "UI.hpp"

#include "GameState.hpp"

#include "Deserializer.hpp"

#include "Serializer.hpp"

class Game {

public:

Game();

Game(GameState\* state){

player = state->GetPlayer();

computer = state->GetComputer();

abilityManager = state->GetAbilityManager();

renderer = new CellsRenderer();

};

// Requires ctx to have valid x and y coordinates

// Attacks computer and depending on outcome does smthin else

std::string MakeTurn(Context ctx);

std::string UseAbilityOnComputer(Context ctx);

// Shows current available ability

std::string PreviewNextAbility(Context ctx);

// If context has 'owner' param and it equals player, it returns player's field

// If 'owner' is equal to "Computer", it returns computer's field

std::vector<std::vector<std::string>> GetField(bool isPlayerField);

std::string Save(Context ctx);

std::string Load(Context ctx);

private:

void NewGame();

void StartNewRound();

Player\* player;

Player\* computer;

AbilityManager\* abilityManager;

CellsRenderer\* renderer;

};

Название файла: Serializer.cpp

#include "Serializer.hpp"

#include <fstream>

nlohmann::json Serializer::to\_json(std::vector<Ship\*> ships) {

nlohmann::json j = nlohmann::json{};

for (int i = 0; i < ships.size(); i++) {

Ship\* temp = ships[i];

std::string key = "ship" + std::to\_string(i);

j[key] = {

{"x", temp->getX()},

{"y", temp->getY()},

{"length", temp->getSize()},

{"vertical", temp->isVertical()},

{"segments", nlohmann::json::array()}

};

auto segments = temp->getSegments();

for (int k = 0; k < segments.size(); k++) {

auto tempSegment = segments[k];

j[key]["segments"].push\_back({

{"health", tempSegment->getHP()},

});

}

}

return j;

}

nlohmann::json Serializer::to\_json(std::vector<std::vector<FieldCell>> field) {

nlohmann::json j = nlohmann::json{};

for (int y = 0; y < field.size(); y++) {

for (int x = 0; x < field[0].size(); x++) {

std::string key = std::to\_string(y) + std::to\_string(x);

j[key] = {

{"revealed", field[y][x].isRevealed()},

};

}

}

return j;

}

nlohmann::json Serializer::to\_json(AbilityManager\* abilityManager) {

nlohmann::json j = nlohmann::json{};

for (auto type : abilityManager->GetAllAbilities()) {

j["abilities"].push\_back(

type

);

}

return j;

}

nlohmann::json Serializer::to\_json(Player\* player) {

nlohmann::json j = nlohmann::json{};

j["ships"] = to\_json(player->GetShips());

j["field"] = to\_json(player->GetFieldCells());

j["take\_double\_damage"] = player->TakeDoubleDamage;

return j;

}

nlohmann::json Serializer::to\_json(GameState\* gameState) {

nlohmann::json j = nlohmann::json{};

j["player"] = to\_json(gameState->GetPlayer());

j["computer"] = to\_json(gameState->GetComputer());

j["ability\_manager"] = to\_json(gameState->GetAbilityManager());

return j;

}

Название файла: Serializer.hpp

#ifndef SERIALIZER\_HPP

#define SERIALIZER\_HPP

#include <string.h>

#include <nlohmann/json.hpp>

#include "Player.hpp"

#include "GameState.hpp"

#include "AbilityManager.hpp"

class Serializer {

public:

Serializer(){};

nlohmann::json to\_json(GameState\* gameState);

private:

nlohmann::json to\_json(std::vector<Ship\*> ships);

nlohmann::json to\_json(std::vector<std::vector<FieldCell>> field);

nlohmann::json to\_json(AbilityManager\* abilityManager);

nlohmann::json to\_json(Player\* player);

};

#endif SERIALIZER\_HPP

Название файла: Deserializer.cpp

#include "Deserializer.hpp"

std::vector<Ship\*> Deserializer::from\_json\_ships(const nlohmann::json& j) {

std::vector<Ship\*> ships;

for (auto& [key, value] : j.items()) {

int x = value["x"];

int y = value["y"];

bool vertical = value["vertical"];

std::vector<ShipSegment\*> segments;

for (auto& segment : value["segments"]) {

int health = segment["health"];

ShipSegment\* s = new ShipSegment(health);

segments.push\_back(s);

}

Ship\* ship = new Ship(segments, x, y, vertical);

ships.push\_back(ship);

}

return ships;

}

std::vector<std::vector<FieldCell>> Deserializer::from\_json\_field(const nlohmann::json& j) {

std::vector<std::vector<FieldCell>> field;

for (int y = 0; y < 10; y++) {

std::vector<FieldCell> row;

for (int x = 0; x < 10; x++) {

std::string key = std::to\_string(y) + std::to\_string(x);

bool revealed = j[key]["revealed"];

FieldCell cell;

if (revealed) {

cell.reveal();

}

row.push\_back(cell);

}

field.push\_back(row);

}

return field;

}

AbilityManager\* Deserializer::from\_json\_ability\_manager(const nlohmann::json& j) {

Factory\* f = new Factory();

AbilityManager\* abilityManager = new AbilityManager(f);

try {

for (int ability : j["abilities"]) {

abilityManager->addAbility(ability);

}

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

}

return abilityManager;

}

Player\* Deserializer::from\_json\_player(const nlohmann::json& j) {

Field\* field = new Field(from\_json\_field(j["field"]));

ShipManager\* sm = new ShipManager(from\_json\_ships(j["ships"]));

bool tdd = j["take\_double\_damage"];

auto ships = sm->getShips();

for (auto ship : ships) {

int x = ship->getX();

int y = ship->getY();

bool vertical = ship->isVertical();

field->PlaceShip(ship, x, y, vertical);

}

Player\* player = new Player(field, sm, tdd);

return player;

}

GameState\* Deserializer::from\_json\_game\_state(const nlohmann::json& j) {

Player\* p = from\_json\_player(j["player"]);

Player\* c = from\_json\_player(j["computer"]);

AbilityManager\* am = from\_json\_ability\_manager(j["ability\_manager"]);

GameState\* gameState = new GameState(p, c, am);

return gameState;

}

Название файла: Deserializer.hpp

#ifndef DESERIALIZER\_HPP

#define DESERIALIZER\_HPP

#include <GameState.hpp>

#include "nlohmann/json.hpp"

#include <fstream>

class Deserializer {

public:

Deserializer() {};

GameState\* from\_json\_game\_state(const nlohmann::json& j);

private:

std::vector<Ship\*> from\_json\_ships(const nlohmann::json& j);

std::vector<std::vector<FieldCell>> from\_json\_field(const nlohmann::json& j);

AbilityManager\* from\_json\_ability\_manager(const nlohmann::json& j);

Player\* from\_json\_player(const nlohmann::json& j);

};

#endif

Название файла: GameState.hpp

#ifndef GAMESTATE\_HPP

#define GAMESTATE\_HPP

#include "AbilityManager.hpp"

#include "Player.hpp"

#include <fstream>

class GameState {

private:

Player\* player;

Player\* computer;

AbilityManager\* manager;

public:

GameState(Player\* player, Player\* computer, AbilityManager\* manager) : player(player), computer(computer), manager(manager) {};

Player\* GetPlayer() { return player; };

Player\* GetComputer() { return computer; };

AbilityManager\* GetAbilityManager() { return manager; };

};

#endif