**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 |  | Коршков А.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:

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.

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

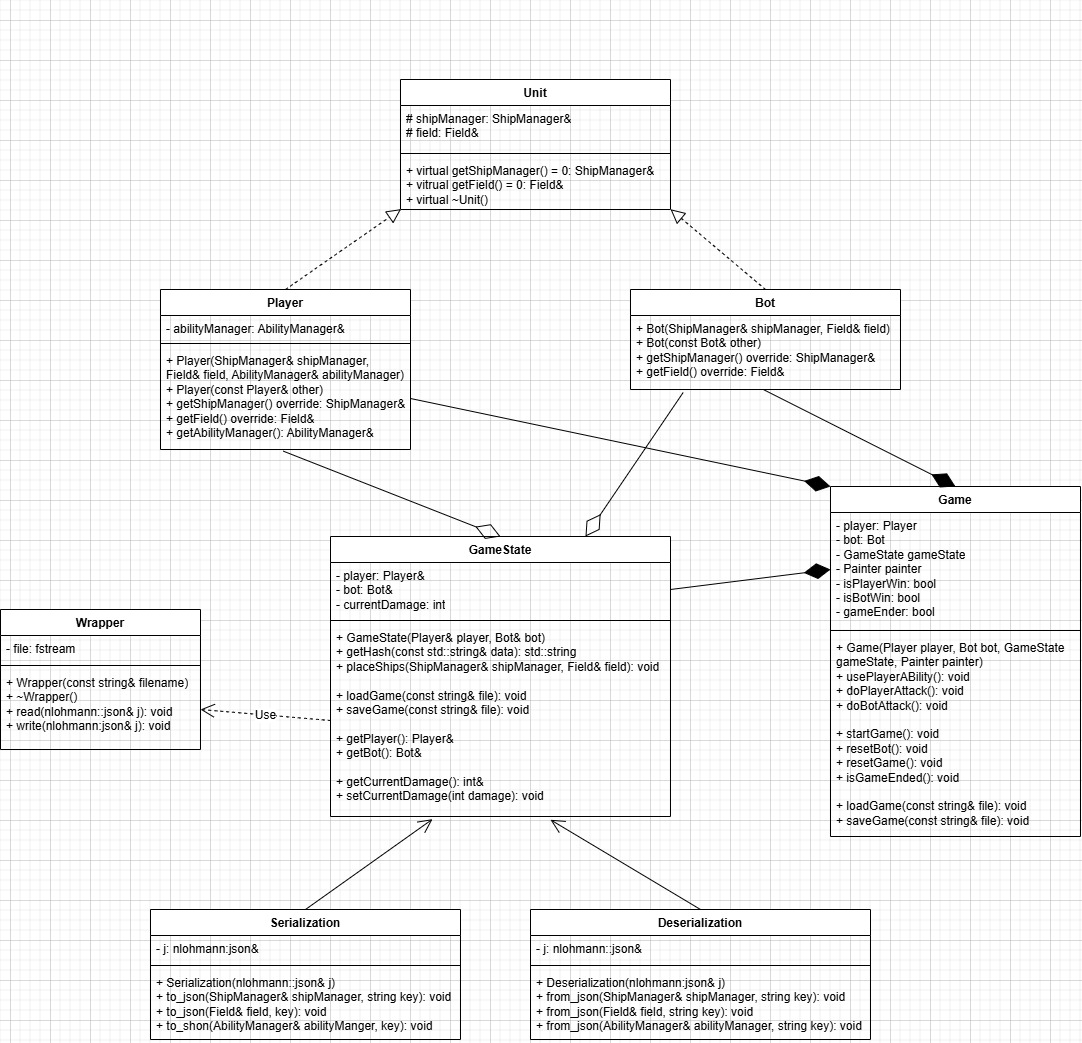


Рисунок 1 – UML-диаграмма классов

Код программы содержит реализацию классов: *Game*, *Unit*, *PlayerUnit*, *BotUnit*, *Serialization*, *Deserialization*, *Wrapper* и *GameState*.

Классы *Game* и *GameState* были добавлены согласно заданию. *Game* связывает классы и работает с ними, описывает игровой цикл и выполнение ходов. Класс *GameState* отвечает за связывание классов *Serialization*, *Deserialization* и *Wrapper*, которые в сумме дают возможность работать с json файлом и совершать загрузку/сохранение игры. В нём также происходит хэширование json файла для его защиты от внешнего вмешательства.

Классы *Unit,* *PlayerUnit* и *BotUnit* являются дата-классами, *Unit* – абстрактный класс, который хранит общие для игрока и бота поля и методы; *Player* и *Bot* – наследуемые от *Unit* классы, представляющие собой игрока и бота соответственно, могут только возвращать значения полей.

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.

Класс *Wrapper* реализован как обёртка над файлом с использованием идиомы RAII для более удобной работы. В конструкторе происходит открытие файла, а в деструкторе его закрытие.

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

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

* *PlayerUnit player* – класс игрока.
* *BotUnit bot* – класс бота.
* *GameState gameState* – класс состояния игры.
* *Painter painter* – класс отрисовщика для исключений и поля.
* *bool is\_player\_win\_cond* – выиграл ли игрок.
* *bool is\_player\_bot\_cond* – выиграл ли бот.
* *bool gameEnder* – закончилась ли игра.

И следующие методы:

* *void usePlayerAbility()* – использовать способность игрока.
* *void doPlayerMove()* – провести атаку игрока по полю бота.
* *void doBotMove()* – провести атаку бота по полю игрока.
* *void StartGame()* – метод начала игры, в котором, в зависимости от решения игрока, происходит непосредственно игра, загрузка/сохранение или выход.
* *void resetBot()* – обнуление бота (после победы над ним и при желании продолжить игру).
* *void resetGame()* – обнуление всей игры (после проигрыша и при желании продолжить игру).
* *void isGameEnded()* – проверка, завершилась ли игра и требуется ли продолжение.
* *void loadGame()* – вызов загрузки игры у класса состояния.
* *void saveGame()* – вызов сохранения игры у класса состояния.

Класс *Unit* является абстрактным классом для игрока и бота. Он имеет следующие protected поля:

* *ShipManager& ship\_manager* – ссылка на менеджер кораблей.
* *Board& board* – ссылка на поле.

И следующие виртуальные методы:

* *virtual ShipManager& getShipManager() = 0* – возвращает ссылку на менеджер кораблей.
* *virtual Board& getBoard() = 0* – возвращает ссылку на поле.

Класс *Player* является реализацией дата-класса игрока, который наследуется от класса *Unit*. Он имеет следующие поля:

* *ShipManager& shipManager* – ссылка на менеджер кораблей, наследуется от *Unit*.
* *Board& board* – ссылка на поле, наследуется от *Unit*.
* *Ability\_manager& ability\_manager* – ссылка на менеджер способностей.

И соответствующие методы для получения полей.

Класс *BotUnit* является реализацией дата-класса бота, он тоже наследуется от класса *Unit*. Он имеет следующие поля:

* *Board& board* – ссылка на поле, наследуется от *Unit*.

И соответствующие методы для получения полей.

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 *FileWrapper* является обёрткой над файлом с использованием идиомы RAII. Он имеет следующее поле:

* *fstream file* – поток для работы с файлом.

И следующие методы:

* *read(nlohmann::json& j)* – записывает содержимое файла в структуру json.
* *write(nlohmann::json& j)* – записывает содержимое структуры json в файл.

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.
* *Bot& bot* – ссылка на бота.
* *bool isAbilityUsed* – была ли использована способность (случай сохранения до атаки).
* *int currentDamage* – текущий урон (случай сохранения до атаки для двойного урона).

И следующие методы:

* Wrapper& operator<<(Wrapper& fileWrapper, GameState& state) – переопределяет оператор << следующим образом: сначала происходит сериализация и вся необходимая информация по кораблям, полям и способностям сохраняется в библиотечную структуру, которая потом переносится в обёртку и она возвращается.
* *Wrapper& operator>>(Wrapper& fileWrapper, GameState& state)* – переопределяет оператор >> следующим образом: сначала происходит считывание информации из обёртки в структуру json, затем десериализация, информация записывается в временные объекты и позже переносится на используемые, в конце возвращается обёртка.
* *void placeShips(ShipManager& shipManager, Board& board)* – расставляет корабли обратно после загрузки из файла.
* *void loadGame(const string& file)* – создаёт обертку и заполняет объект класса информацией из файла.
* void saveGame(const string file)– очищает файл, создаёт обёртку и загружает в неё информацию из объекта класса.
* *bool& getIsAbilityUsed()* – возвращает информацию о том, была ли использована способность.
* *void setIsAbilityUsed(bool value)* – выставляет информацию о использовании способности.
* *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.

# 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 "../include/Game.hpp"

void Game::usePlayerAbility() {

int x, y;

std::cout << "You have " << player.getAbility\_manager().getAbilityCount() << " abilities available." << std::endl;

std::cout << "Use random ability? 'y'" << std::endl;

std::string result;

std::cin >> result;

if (result == "y" || result == "Y") {

Coordinate coordinate = {-1, -1};

AbilityParameters ap(player.getBoard(), player.getShipManager(), coordinate, gameState.getCurrentDamage());

player.getAbility\_manager().checkIfEmpty();

painter.printAbilityName(player.getAbility\_manager().getCreator(0).getName());

try {

if (player.getAbility\_manager().getCreator(0).isUsingCoordinate()) {

std::cout << "Give coordinates for ability." << std::endl;

std::cin >> x >> y;

ap.coordinate = {x, y};

}

player.getAbility\_manager().useAbility(ap);

}

catch (RevealedCellAttackException& e) {

player.getAbility\_manager().popAbility();

}

}

this->gameState.setIsAbilityUsed(true);

}

void Game::doPlayerAttack() {

int x, y;

int successAttack = false;

while (true) {

try {

std::cin >> x >> y;

if (std::cin.fail()) {

std::cin.clear();

std::cin.ignore(std::numeric\_limits<std::streamsize>::max(), '\n');

throw InvalidCoordinateException();

}

for (int i = 0; i < gameState.getCurrentDamage(); i++) {

player.getBoard().attack({x, y});

successAttack = true;

}

}

catch (InvalidCoordinateException& e) {

painter.printException(e);

continue;

}

catch (RevealedCellAttackException& e) {

if (successAttack) {

break;

}

painter.printException(e);

continue;

}

catch (OutOfRangeException& e) {

painter.printException(e);

continue;

}

break;

}

this->gameState.setCurrentDamage(1);

Ship\* enemyShip = player.getShipManager().getShipByCoordinate({x, y});

if (enemyShip != nullptr && enemyShip->isDestroyed()) {

player.getBoard().revealCoordinatesAround(enemyShip);

player.getShipManager().setShipsAlive(player.getShipManager().getShipsAlive() - 1);

std::cout << "Ability added." << std::endl;

player.getAbility\_manager().giveRandomAbility();

}

if (player.getShipManager().getShipsAlive() == 0) {

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

this->isPlayerWin = true;

}

this->gameState.setIsAbilityUsed(false);

return;

}

void Game::doBotAttack() {

Coordinate coords = {-1, -1};

try {

coords = bot.getBoard().attackRandomly();

}

catch (MultipleMissesException& e) {

painter.printException(e);

return;

}

Ship\* selfShip = bot.getShipManager().getShipByCoordinate(coords);

if (selfShip != nullptr && selfShip->isDestroyed()) {

bot.getBoard().revealCoordinatesAround(selfShip);

bot.getShipManager().setShipsAlive(bot.getShipManager().getShipsAlive() - 1);

}

if (bot.getShipManager().getShipsAlive() == 0) {

std::cout << "You lose!" << std::endl;

this->isBotWin = true;

}

return;

}

void Game::startGame() {

std::string answer;

const std::string file = "/home/nhitar/oop-ships/savefile.json";

while (!this->gameEnder) {

std::cout << "Push 'p' to play, 'l' to load game, 's' to save game, 'q' to quit." << std::endl;

std::string line;

std::cin >> line;

if (line.size() == 1) {

switch (line[0]) {

case 'p':

painter.printBoards(bot.getBoard(), player.getBoard());

if (!this->gameState.getIsAbilityUsed()) {

try {

usePlayerAbility();

} catch (NoAbilitiesAvailableException& e) {

painter.printException(e);

break;

}

catch (OutOfRangeException& e) {

painter.printException(e);

break;

}

std::cout << "Do you want to quit/load/save the game? y/n" << std::endl;

std::cin >> answer;

if (answer == "y" || answer == "Y") {

break;

}

}

painter.printBoards(bot.getBoard(), player.getBoard());

doPlayerAttack();

doBotAttack();

// для отладки

for (int i = 0; i < 10; i++)

painter.printShip(player.getShipManager().getShipByIndex(i));

painter.printBoards(bot.getBoard(), player.getBoard());

this->isGameEnded();

break;

case 'l':

std::cout << "Loading the game." << std::endl;

this->loadGame(file);

painter.printBoards(bot.getBoard(), player.getBoard());

break;

case 's':

std::cout << "Saving the game." << std::endl;

this->saveGame(file);

break;

case 'q':

std::cout << "Quitting the game." << std::endl;

this->gameEnder = true;

break;

default:

std::cout << "Unknown command." << std::endl;

break;

}

continue;

}

std::cout << "Invalid command." << std::endl;

}

}

void Game::resetBot() {

std::vector<int> shipSizes = {4, 3, 3, 2, 2, 2, 1, 1, 1, 1};

Board newBoard = Board(10, 10);

ShipManager newShips = ShipManager(10, shipSizes);

for (size\_t i = 0; i < shipSizes.size(); i++) {

newBoard.placeShipRandomly(&newShips.getShipByIndex(i));

}

this->player = Player(newShips, newBoard, player.getAbility\_manager());

}

void Game::resetGame() {

resetBot();

std::vector<int> shipSizes = {4, 3, 3, 2, 2, 2, 1, 1, 1, 1};

Board newBoard = Board(10, 10);

ShipManager newShips = ShipManager(10, shipSizes);

for (size\_t i = 0; i < shipSizes.size(); i++) {

newBoard.placeShipRandomly(&newShips.getShipByIndex(i));

}

newBoard.revealCells();

this->bot = Bot(newShips, newBoard);

}

void Game::isGameEnded() {

if (!this->isPlayerWin && !this->isBotWin) {

this->gameEnder = false;

return;

}

std::cout << "Do you want to continue playing? y/n" << std::endl;

std::string line;

std::cin >> line;

if (line == "n" || line == "N") {

this->gameEnder = true;

return;

}

if (this->isPlayerWin) {

resetBot();

this->isPlayerWin = false;

painter.printBoards(bot.getBoard(), player.getBoard());

}

if (this->isBotWin) {

resetGame();

this->isBotWin = false;

painter.printBoards(bot.getBoard(), player.getBoard());

}

}

void Game::loadGame(const std::string& file) {

try {

this->gameState.loadGame(file);

} catch (nlohmann::json::exception& e) {

std::cerr << "Error parsing JSON: " << e.what() << std::endl;

return;

}

}

void Game::saveGame(const std::string& file) {

this->gameState.saveGame(file);

}

Название файла: Game.hpp

#pragma once

#include "Ship.hpp"

#include "ShipManager.hpp"

#include "Board.hpp"

#include "Painter.hpp"

#include "Player.hpp"

#include "GameState.hpp"

#include "Abilities.hpp"

#include "Ability\_manager.hpp"

#include "Exceptions/InvalidShipSizeException.hpp"

#include "Exceptions/InvalidCoordinateException.hpp"

class Game {

private:

Player player;

Bot bot;

GameState gameState;

Painter painter;

bool isPlayerWin;

bool isBotWin;

bool gameEnder;

public:

Game(Player player, Bot bot, GameState gameState, Painter painter)

: player(player), bot(bot), gameState(gameState), painter(painter), isPlayerWin(false), isBotWin(false), gameEnder(false) {}

void usePlayerAbility();

void doPlayerAttack();

void doBotAttack();

void startGame();

void resetBot();

void resetGame();

void isGameEnded();

void loadGame(const std::string& file);

void saveGame(const std::string& file);

};

Название файла: Player.hpp

#pragma once

#include "ShipManager.hpp"

#include "Board.hpp"

#include "Ability\_manager.hpp"

#include "Painter.hpp"

#include "AbilityParameters.hpp"

class Unit {

protected:

ShipManager& shipManager;

Board& board;

public:

Unit(ShipManager& shipManager, Board& board)

: shipManager(shipManager), board(board) {}

virtual ShipManager& getShipManager() = 0;

virtual Board& getBoard() = 0;

virtual ~Unit() {};

};

class Player : public Unit {

private:

Ability\_manager& ability\_manager;

public:

Player(ShipManager& shipManager, Board& board, Ability\_manager& ability\_manager)

: Unit(shipManager, board), ability\_manager(ability\_manager) {}

Player(const Player& other)

: Unit(other.shipManager, other.board), ability\_manager(other.ability\_manager) {}

Player& operator=(const Player& other) {

if (this != &other) {

this->shipManager = other.shipManager;

this->board = other.board;

this->ability\_manager = other.ability\_manager;

}

return \*this;

}

ShipManager& getShipManager() override { return shipManager; };

Board& getBoard() override { return board; };

Ability\_manager& getAbility\_manager() { return ability\_manager; };

};

class Bot : public Unit {

public:

Bot(ShipManager& shipManager, Board& board)

: Unit(shipManager, board) {}

Bot(const Bot& other)

: Unit(other.shipManager, other.board) {}

Bot& operator=(const Bot& other) {

if (this != &other) {

this->shipManager = other.shipManager;

this->board = other.board;

}

return \*this;

}

ShipManager& getShipManager() override { return shipManager; };

Board& getBoard() override { return board; };

};

Название файла: Serialization.cpp

#include "../include/Serialization.hpp"

#include <fstream>

void Serialization::to\_json(ShipManager& shipManager, std::string key) {

nlohmann::json jsm = nlohmann::json{};

for (int i = 0; i < shipManager.getShipCount(); i++) {

Ship& temp = shipManager.getShipByIndex(i);

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

jsm[key] = {

{"length", temp.getLength()},

{"horizontal", temp.isHorizontal()},

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

};

for (int j = 0; j < temp.getLength(); j++) {

Segment\* tempSegment = temp.getSegment(j);

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

{"x", tempSegment->coordinate.x},

{"y", tempSegment->coordinate.y},

{"health", tempSegment->health}

});

}

}

j[key] = jsm;

}

void Serialization::to\_json(Board& board, std::string key) {

nlohmann::json jf = nlohmann::json{};

jf["rows"] = board.getRows();

jf["columns"] = board.getColumns();

std::vector<Cell> temp = board.getBoard();

for (int y = 0; y < board.getRows(); y++) {

for (int x = 0; x < board.getColumns(); x++) {

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

jf[key] = {

{"x", temp[y\*board.getRows() + x].coordinate.x},

{"y", temp[y\*board.getRows() + x].coordinate.y},

{"state", temp[y\*board.getRows() + x].state},

{"value", temp[y\*board.getRows() + x].value}

};

}

}

j[key] = jf;

}

void Serialization::to\_json(Ability\_manager& ability\_manager, std::string key) {

nlohmann::json jam = nlohmann::json{};

for (int i = 0; i < ability\_manager.getAbilityCount(); i++) {

jam["abilities"].push\_back(

ability\_manager.getCreator(i).getName()

);

}

if (ability\_manager.getAbilityCount() == 0) {

jam["abilities"].push\_back("");

}

j[key] = jam;

}

Название файла: Serialization.hpp

#pragma once

#include "ShipManager.hpp"

#include "Board.hpp"

#include "Ability\_manager.hpp"

#include <string.h>

#include <nlohmann/json.hpp>

class Serialization {

private:

nlohmann::json& j;

public:

Serialization(nlohmann::json& j) : j(j) {};

void to\_json(ShipManager& shipManager, std::string key);

void to\_json(Board& board, std::string key);

void to\_json(Ability\_manager& ability\_manager, std::string key);

};

Название файла: Deserialization.cpp

#include "../include/Deserialization.hpp"

void Deserialization::from\_json(ShipManager& shipManager, std::string key) {

const auto& jsm = j.at(key);

std::vector<int> shipSizes;

for (const auto& jship : jsm) {

shipSizes.push\_back(jship.at("length"));

}

shipManager = ShipManager(shipSizes.size(), shipSizes);

for (size\_t i = 0; i < shipSizes.size(); i++) {

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

Ship& ship = shipManager.getShipByIndex(i);

if (jsm.at(key).at("horizontal") == true) {

ship.setOrientation(Orientation::Horizontal);

}

else {

ship.setOrientation(Orientation::Vertical);

}

for (int j = 0; j < shipSizes[i]; j++) {

Segment\* segment = ship.getSegment(j);

segment->health = jsm.at(key).at("segments").at(j).at("health");

segment->coordinate.x = jsm.at(key).at("segments").at(j).at("x");

segment->coordinate.y = jsm.at(key).at("segments").at(j).at("y");

}

}

int shipsAlive = shipSizes.size();

for (size\_t i = 0; i < shipSizes.size(); i++) {

Ship& ship = shipManager.getShipByIndex(i);

if (ship.isDestroyed()) {

shipsAlive--;

}

}

shipManager.setShipsAlive(shipsAlive);

}

void Deserialization::from\_json(Board& board, std::string key) {

const auto& jf = j.at(key);

board = Board(jf.at("rows"), jf.at("columns"));

for (int y = 0; y < board.getRows(); y++) {

for (int x = 0; x < board.getColumns(); x++) {

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

Cell& cell = board.getCell({x, y});

cell.coordinate.x = jf.at(key).at("x");

cell.coordinate.y = jf.at(key).at("y");

cell.state = jf.at(key).at("state");

cell.value = jf.at(key).at("value");

}

}

}

void Deserialization::from\_json(Ability\_manager& ability\_manager, std::string key) {

const auto& jam = j.at(key);

ability\_manager = Ability\_manager();

ability\_manager.popAbility();

for (const auto& jability : jam.at("abilities")) {

if (jability == "Double Damage") {

ability\_manager.addAbility(new DoubleDamageAbilityCreator());

}

else if(jability == "Scanner"){

ability\_manager.addAbility(new ScannerAbilityCreator());

}

else if (jability == "Gunblaze") {

ability\_manager.addAbility(new GunblazeAbilityCreator());

}

}

}

Название файла: Deserialization.hpp

#pragma once

#include "ShipManager.hpp"

#include "Board.hpp"

#include "Ability\_manager.hpp"

#include <string.h>

#include <nlohmann/json.hpp>

class Deserialization {

private:

nlohmann::json& j;

public:

Deserialization(nlohmann::json& j) : j(j) {};

void from\_json(ShipManager& shipManager, std::string key);

void from\_json(Board& board, std::string key);

void from\_json(Ability\_manager& ability\_manager, std::string key);

};

Название файла: Wrapper.cpp

#include "../include/Wrapper.hpp"

Wrapper::Wrapper(const std::string& filename) : file(filename) {};

Wrapper::~Wrapper() {

if (file.is\_open())

file.close();

}

void Wrapper::read(nlohmann::json& j) {

if (!file.is\_open() || !file.good()) {

throw UnableToOpenFileException();

}

file >> j;

}

void Wrapper::write(nlohmann::json& j) {

if (!file.is\_open() || !file.good()) {

throw UnableToOpenFileException();

}

file << j.dump(4);

}

Название файла: Wrapper.hpp

#pragma once

#include "Exceptions/UnableToOpenFileException.hpp"

#include <nlohmann/json.hpp>

#include <iostream>

#include <fstream>

class Wrapper {

private:

std::fstream file;

public:

Wrapper(const std::string& filename);

~Wrapper();

void read(nlohmann::json& j);

void write(nlohmann::json& j);

};

Название файла: GameState.cpp

#include "../include/GameState.hpp"

Wrapper& operator<<(Wrapper& fileWrapper, GameState& state) {

nlohmann::json j;

Serialization seri(j);

seri.to\_json(state.getPlayer().getShipManager(), "playerShipManager");

seri.to\_json(state.getPlayer().getBoard(), "playerBoard");

seri.to\_json(state.getPlayer().getAbility\_manager(), "playerAbility\_manager");

seri.to\_json(state.getBot().getShipManager(), "botShipManager");

seri.to\_json(state.getBot().getBoard(), "botBoard");

j["currentDamage"] = state.getCurrentDamage();

j["isAbilityUsed"] = state.getIsAbilityUsed();

try {

fileWrapper.write(j);

}

catch (UnableToOpenFileException& e){

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

}

return fileWrapper;

}

Wrapper& operator>>(Wrapper& fileWrapper, GameState& state) {

nlohmann::json j;

try {

fileWrapper.read(j);

}

catch (UnableToOpenFileException& e) {

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

return fileWrapper;

}

Deserialization deseri(j);

ShipManager shipManager;

Board board;

Ability\_manager ability\_manager;

ShipManager enemyShipManager;

Board enemyBoard;

deseri.from\_json(shipManager, "playerShipManager");

deseri.from\_json(board, "playerBoard");

deseri.from\_json(ability\_manager, "playerAbility\_manager");

deseri.from\_json(enemyShipManager, "botShipManager");

deseri.from\_json(enemyBoard, "botBoard");

state.setCurrentDamage(j.at("currentDamage"));

state.setIsAbilityUsed(j.at("isAbilityUsed"));

state.getPlayer().getShipManager() = shipManager;

state.getPlayer().getBoard() = board;

state.getPlayer().getAbility\_manager() = ability\_manager;

state.getBot().getShipManager() = enemyShipManager;

state.getBot().getBoard() = enemyBoard;

state.placeShips(state.getPlayer().getShipManager(), state.getPlayer().getBoard());

state.placeShips(state.getBot().getShipManager(), state.getBot().getBoard());

return fileWrapper;

}

void GameState::placeShips(ShipManager& shipManager, Board& board) {

for (int i = 0; i < shipManager.getShipCount(); i++) {

for (int j = 0; j < shipManager.getShipByIndex(i).getLength(); j++) {

Coordinate coordinate = shipManager.getShipByIndex(i).getSegment(j)->coordinate;

auto& cell = board.getCell(coordinate);

cell.segment = shipManager.getShipByIndex(i).getSegment(j);

}

}

}

void GameState::loadGame(const std::string& file) {

Wrapper fileWrapper(file);

fileWrapper >> \*this;

}

void GameState::saveGame(const std::string& file) {

std::ofstream ofs(file, std::ofstream::out | std::ofstream::trunc);

Wrapper fileWrapper(file);

fileWrapper << \*this;

}

Название файла: GameState.hpp

#pragma once

#include "Serialization.hpp"

#include "Deserialization.hpp"

#include "Painter.hpp"

#include "Player.hpp"

#include "Wrapper.hpp"

#include "Exceptions/UnableToOpenFileException.hpp"

#include <fstream>

class GameState {

private:

Player& player;

Bot& bot;

bool isAbilityUsed;

int currentDamage;

public:

GameState(Player& player, Bot& bot) : player(player), bot(bot), isAbilityUsed(false), currentDamage(1) {};

void placeShips(ShipManager& shipManager, Board& board);

void loadGame(const std::string& file);

void saveGame(const std::string& file);

Player& getPlayer() { return this->player; };

Bot& getBot() { return this->bot; };

bool& getIsAbilityUsed() { return this->isAbilityUsed; };

void setIsAbilityUsed(bool value) { this->isAbilityUsed = value; };

int& getCurrentDamage() { return this->currentDamage; };

void setCurrentDamage(int damage) { this->currentDamage = damage; };

};