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. 3382 _____ Царегородцев Д.И.

Преподаватель _____ Жангиров Т. Р.

Санкт-Петербург

2024

Цель работы

Изучить язык C++, научиться создавать классы, согласно парадигме объектно-ориентированного программирования.

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

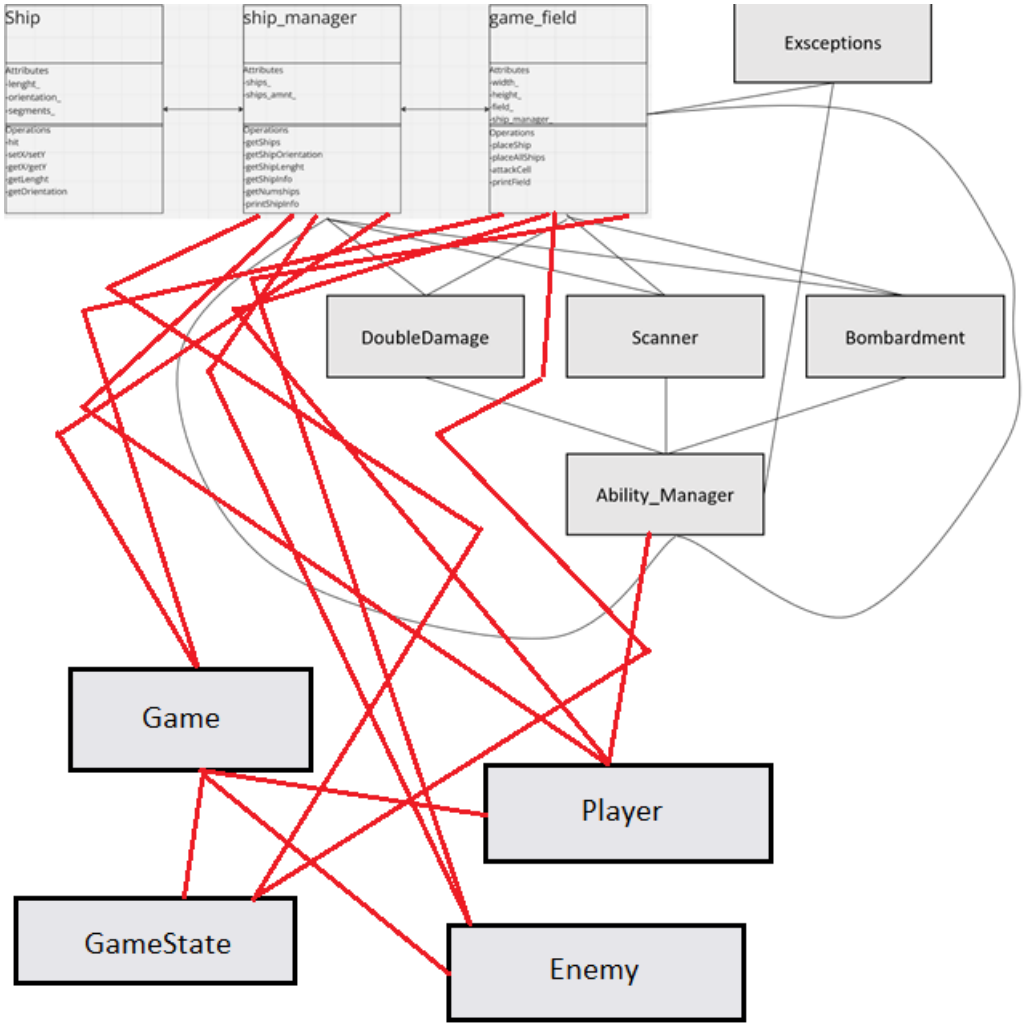
Для управления самое игрой можно использовать обертки над командами

При работе с файлом используйте идиому RAII.

Выполнение работы

Разработанный программный код см. в приложении А

В написанной программе уже присутствовали три класса: корабль, менеджер кораблей, поле, исключения, три способности и класс их класс- менеджер. Были добавлены класс игрока, противника, игры, класс состояния игры.



Game

Данный класс является основным управляющим компонентом. Он объединяет в себе управление игровым процессом, состоянием игры и взаимодействием с игровыми объектами, такими как поле и корабли. Класс включает методы для запуска новой игры startNewGame, обработки ходов p performEnemyTurn и противника performEnemyTurn, а также проведение полного игрового раунда playRound. Класс также предоставляет

функциональность сохранение и загрузки игры через методы saveGame и loadGame.

GameState

Этот класс предназначен для хранения текущего состояния игры, включая информацию о игроке, противнике, игровом поле, а также управления способностями и кораблями. Данный класс служит хранилищем данных, которые необходимо сохранить или загрузить в процессе игры. Так же данные классы game_field и ship_manager взаимодействуют с игровым процессом, управляя расположением кораблей и обеспечивая логику взаимодействия на игровом поле

Player

Этот класс представляет собой игрока в игре, который управляет своими кораблями и способностями, а также выполняет действия в игровом процессе, такие как атака и использование способностей. Класс включает в себя переменные для характеристики игрока: здоровье игрока int health, а также методы для выполнения различных действий.

Enemy

Класс является основным инструментом управления противником в игре. Он отвечает за генерацию флота, атаку игрока и управлением состоянием своих кораблей. Благодаря случайному размещению кораблей с помощью метода placeShipsRandomly и логике атаки с помощью метода attak , противник добавляет элемент стратегии и динамики в игровой процесс.

Тестирование

Результаты тестирования представлены в табл. 1.

Таблица 1 – Результаты тестирования

№ п/п	Входные данные	Выходные данные	Комментарии
1.	main запрашивает у пользователя координаты для атаки и способностей, так что входные данные всегда разные	ожидаемые и приемлемые	Вывод верный

Выводы

Были созданы классы соответствующие условию работы, заложен фундамент для написания последующих лабораторных работ, а также функция main отвечающая за проверку их работоспособности.

ПРИЛОЖЕНИЕ А

ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: LABA1

```
#ifndef GAME_H
#define GAME_H

#include "GameState.h"
#include <string>
#include "Enemy.h"

class Game {
private :
    bool isSaved;
public:
    GameState state;

    Game();
    ship_manager sm;  // или ship_manager_instance если ваше имя другое
    game_field gf;
    void startNewGame();
    void performPlayerTurn();
    void performEnemyTurn();
    void playRound();
    //void saveGame(const std::string& filename);
   // void loadGame(const std::string& filename);
   // Game() : isSaved(false) {} // Изначально игра не сохранена

   // Метод для сохранения игры
    void saveGame(const std::string& filename);
    // Метод для загрузки игры
    void loadGame(const std::string& filename);
};


#endif
#include "Game.h"
#include "Player.h"
#include <iostream>
#include "Ability_Manager.h"
#include "ship_manager.h"

Game::Game() : state() {}

void Game::startNewGame() {
    // Инициализация кораблей для игрока и врага
   // std::vector<int> playerShipLengths = { 2, 3, 3 };
    std::vector<int> enemyShipLengths = { 2, 3, 3 };


    state.enemy.placeShipsRandomly(10, enemyShipLengths);

    state.playerTurn = true;

    // Вывод состояния кораблей
  //  std::cout << "Корабли игрока: " << state.player.getShipCounts() <<
std::endl;
    std::cout << "Корабли противника: " << state.enemy.getShipCounts() <<
std::endl;
}

void Game::performPlayerTurn() {
    if (state.player.health > 0 && state.enemy.health > 0) {
```

```
        std::cout << "Твоя очередь!" << std::endl;
        std::cout << "1. Атака\n2. Способность" << std::endl;
        int choice;
        std::cin >> choice;

        if (choice == 1) {
            state.player.attack(state.enemy, sm, gf);
        }
        else if (choice == 2) {
            state.player.useAbility(state.enemy, state.abilityManager,
state.field, state.shipManager);
        }

        state.playerTurn = false;
    }
}

void Game::performEnemyTurn() {
    if (state.enemy.health > 0 && state.player.health > 0) {
        std::cout << "Очередь врага!" << std::endl;
        state.enemy.attack(state.player);
        state.playerTurn = true;
    }
}

void Game::playRound() {
    while (state.player.health > 0 && state.enemy.health > 0) {
        state.displayState();
        if (state.playerTurn) {
            performPlayerTurn();
        }
        else {
            performEnemyTurn();
        }
    }

    if (state.player.health <= 0) {
        std::cout << "Вы проиграли этот раунд! Начинаю новую игру..." <<
std::endl;
        startNewGame();
    }
    else {
        std::cout << "Вы выиграли раунд! Переходим к следующему раунду..." <<
std::endl;
    }
}


void Game::saveGame(const std::string& filename) {
    state.saveState(filename);
    isSaved = true; // Устанавливаем флаг сохранения
}

void Game::loadGame(const std::string& filename) {
    if (!isSaved) { // Проверяем, была ли игра сохранена
        std::cerr << "Ошибка: Игра не сохранена. Загрузка невозможна!" <<
std::endl;
        return; // Прерываем выполнение, если игра не сохранена
    }
    state.loadState(filename);
}
#ifndef GAMESTATE_H
#define GAMESTATE_H
#include "ship_manager.h"
#include "game_field.h"
#include "Player.h"
#include <string>
```

```
#include "Enemy.h"

class GameState {
public:
    Player player;
    Enemy enemy;
    bool playerTurn;
    Ability_Manager abilityManager; // Убедитесь, что этот класс существует
    game_field field;                 // Убедитесь, что этот класс существует
    ship_manager shipManager;

    GameState();


    void displayState() const;
    void saveState(const std::string& filename);

    void loadState(const std::string& filename);
};


#endif
#include "GameState.h"
#include <fstream>
#include <iostream>
#include "Ability_Manager.h"
GameState::GameState() : player(), enemy(), playerTurn(true) {}

void GameState::saveState(const std::string& filename) {
    // Логика сохранения состояния игры в файл
    std::cout << "Игра сохранена в " << filename << std::endl;
}

void GameState::displayState() const {
    std::cout << "Здоровье игрока: " << player.health << std::endl;
    std::cout << "Здоровье врага: " << enemy.health << std::endl;
}

void GameState::loadState(const std::string& filename) {
    // Логика загрузки состояния игры из файла
    std::cout << "Игра загружена из " << filename << std::endl;
}
#ifndef PLAYER_H
#define PLAYER_H
#include "ship.h"
#include <string>
#include <random>
#include <iostream>
#include <algorithm>

#include "Ability_Manager.h"
class Enemy;  // Предварительное объявление класса Enemy

class Player {
private:
    int attackPower;
    ship_manager manager;
    int count;
    std::vector<Ability*> abilityList;
public:
    int health;
    //int attackPower;
    std::string ability;
    Player(int attackPower) : attackPower(attackPower) {}
    Player();
```

```
    void attack(Enemy& target, ship_manager& sm, game_field& field);   // Изменили
параметр с Player на Enemy
    void useAbility(Enemy& target, Ability_Manager& abilityManager, game_field&
field, ship_manager& sm);  // Способность, которая будет работать с Enemy
    // Список кораблей игрока
    std::vector<Ship*> ships;

    // Другие методы и переменные



    // Метод для добавления способности
    void addRandomAbility() {
        // Массив указателей на способности
        Ability* abilities[] = { new DoubleDamage(), new Scanner(), new
Bombardment() };

        // Генерируем случайный индекс
        int randomIndex = rand() % 3; // 0, 1 или 2

        // Добавляем случайную способность в список
        abilityList.push_back(abilities[randomIndex]);
        std::cout << "Получена новая способность: " <<
typeid(*abilities[randomIndex]).name() << std::endl;

        // Освобождение памяти - если не использовать умные указатели, освободите
память по окончании
        delete abilities[randomIndex];
    }
    void destroyEnemyShip(Enemy& target, Ship* destroyedShip) {
        // Логика для уничтожения корабля противника
        std::cout << "Корабль противника уничтожен!" << std::endl;

        // Добавление новой способности
        addRandomAbility();
    }
    // Проверка, остались ли корабли у игрока
    // Проверка, остались ли корабли у игрока
    bool hasShips() const {
        return std::any_of(ships.begin(), ships.end(), [](const Ship* ship) {
            return !ship->isDestroyed(); // Проверяет, есть ли не уничтоженные
корабли
            });
    }
};


#endif
#include "Player.h"
#include <iostream>
#include "EnemyShipInitializer.h"
#include "Enemy.h"
Player::Player() : health(100), attackPower(10), ability("Fireball") {}

void Player::attack(Enemy& target, ship_manager& sm, game_field& field) {
    // Размещение кораблей противника на поле
    std::vector<int> shipLengths = { 2, 3, 3 }; // Пример кораблей
   target.placeShipsRandomly(10, shipLengths);

    int x, y;
    std::cout << "Введите координаты атаки (x y): ";
    while (!(std::cin >> x >> y) || x < 0 || x >= 10 || y < 0 || y >= 10) {
        std::cout << "Неверный ввод. Пожалуйста, введите числа в диапазоне 0-9: ";
        std::cin.clear();
        std::cin.ignore(std::numeric_limits<std::streamsize>::max(), '\n');
    }
```

```
    bool attackSuccessful = false;

    // Перебор кораблей для проверки попадания
    for (const Ship* ship : target.ships) {
        if (ship && ship->isHit(x, y)) {
            attackSuccessful = true;
            break; // Завершить цикл при первом попадании
        }
    }

    // Вывод результатов
    if (attackSuccessful) {
        std::cout << "Попадание!" << std::endl;
    }
    else {
        std::cout << "Промах!" << std::endl;
    }
}

void Player::useAbility(Enemy& target, Ability_Manager& abilityManager,
game_field& field, ship_manager& sm) {
    std::cout << "Игрок пытается использовать способность!" << std::endl;

    // Вызываем метод использования способности из Ability_Manager
    abilityManager.useAbility(field, sm);

    // Примерная сила способности, например, 20
    std::cout << "Игрок использует способность на враге!" << std::endl;
    target.health -= 20; // Предполагаемый урон от способности
}
Enemy::Enemy() : health(50), attackPower(10) {}

void Enemy::attack(Player& target) {
    target.health -= attackPower;
    std::cout << "Враг атакует игрока за " << attackPower << " ущерб!" <<
std::endl;
}
#pragma once
#include "ship.h"
#include <string>
#include <random>
#include <iostream>
#include <algorithm>

#include "Ability_Manager.h"
#include "Player.h"
class Enemy {
public:

    // int health;
    int attackPower;
    //  int health;
    std::vector<Ship*> ships; // Противник имеет вектор кораблей
   // Enemy(int hp, int ap) : health(hp), attackPower(ap) {}
    Enemy(int health) : health(health) {}
    //  Enemy() : attackPower(10) {}
    void addShip(Ship* ship) {
        ships.push_back(ship);
    }
    Enemy();
    void setShipManager(const ship_manager& manager) {
        this->manager = manager; // Храните экземпляр ship_manager
    }
```

```
    void attack(Player& target);
    //  std::vector<Ship*> ships; // Вектор для хранения указателей на корабли
    int health = 100;         // Здоровье противника

    void placeShipsRandomly(int boardSize, const std::vector<int>& shipLengths);

     // Другие методы и переменные

     // Метод для проверки количества кораблей остающихся у врага
    std::string getShipCounts() const;




    std::vector<int> getShipLengths() const {
        return manager.getShipLengths(); // Получаем длины кораблей
    }

private:
    bool canPlaceShip(int x, int y, int length, bool horizontal, int boardSize) {
        if (horizontal) {
            if (x + length > boardSize) return false; // Проверка по горизонтали
        }
        else {
            if (y + length > boardSize) return false; // Проверка по вертикали
        }

        // Проверка на пересечение с уже размещенными кораблями
        for (const Ship* ship : ships) {
            for (int i = 0; i < length; ++i) {
                int checkX = horizontal ? x + i : x; // координаты для проверки
                int checkY = horizontal ? y : y + i; // " "
                if (ship->isAt(checkX, checkY)) {
                    return false; // Если координаты заняты, вернем false
                }
            }
        }
        return true; // Если свободно, возвращаем true
    }
    ship_manager manager;

};
#include "Enemy.h"
std::string Enemy:: getShipCounts() const {
    int twoDeckCount = 0, threeDeckCount = 0, fourDeckCount = 0;

    for (const Ship* ship : ships) {
        if (ship->isDestroyed()) continue; // Пропустить потопленные корабли
        int length = ship->getLength();
        if (length == 2) twoDeckCount++;
        else if (length == 3) threeDeckCount++;
        else if (length == 4) fourDeckCount++;
    }

    return "Двухпалубных: " + std::to_string(twoDeckCount) +
        ", Трехпалубных: " + std::to_string(threeDeckCount) +
        ", Четырехпалубных: " + std::to_string(fourDeckCount);
}

void Enemy:: placeShipsRandomly(int boardSize, const std::vector<int>&
shipLengths) {
    ships.clear(); // Очистка предыдущих кораблей

    std::random_device rd; // Получаем случайное seed
    std::mt19937 gen(rd()); // Инициализация генератора
```

```
    std::uniform_int_distribution<> dis(0, boardSize - 1); // Распределение для
координат

    for (int length : shipLengths) {
        bool placed = false;
        while (!placed) {
            // Случайно определяем координаты и ориентацию
(горизонтально/вертикально)
            bool horizontal = dis(gen) % 2 == 0;

            int x = dis(gen);
            int y = dis(gen);

            // Проверка возможности размещения
            if (canPlaceShip(x, y, length, horizontal, boardSize)) {
                ships.push_back(new Ship(x, y, length, horizontal)); // Создание
нового корабля
                placed = true; // Установка флага о размещении
            }
        }
    }
}
#pragma once
#include <vector>
#include <iostream>
#include "string"
using namespace std;

class ShipSegment {
private:


public:

    int x;  // X-координата сегмента
    int y;  // Y-координата сегмента
    bool isHit; // Флаг, указывающий, был ли сегмент поражен

    ShipSegment(int x, int y) : x(x), y(y), isHit(false) {}
   // bool isHit = false;  // Сегмент может быть поврежден
};
class Ship
{
public:
    enum Orientation { h, v }; // Ориентация корабля: горизонтальная (h) или
вертикальная (v)

private:
    //int length_; // Длина корабля
   // char orientation_; // Ориентация корабля ('h' - горизонтально, 'v' -
вертикально)
    vector<int*> segments_; // Сегменты корабля, указатели типа int
    std::vector<ShipSegment> segments;
    std::vector<std::pair<int, int>> segmentCoordinates; // Хранит координаты
сегментов
    bool sunk;
    std::vector<std::pair<int, int>> shipCoordinates;
  //  int length;
    int startX;      // Начальная координата X
    int startY;      // Начальная координата Y
    int length;      // Длина корабля
    bool horizontal;  // Ориентация: горизонтально (true) или вертикально (false)
    int hits;
public:
    Ship(int length, char orientation); // Конструктор, принимающий длину и
ориентацию
    int length_;
```

```
    int orientation_;
  /*  bool isHit(int x, int y) const {
        // Реализация метода
        // Для демонстрации даже можно просто вернуть true
        return (x == 1 && y == 1); // Просто пример условия
    }*/
    Ship(int length, int orientation) : length_(length), orientation_(orientation)
{}
    int* getSegment(int i); // Метод для получения сегмента по индексу

    void hit(int* segment); // Метод, который регистрирует попадание в сегмент
корабля

    bool isDestroyed(); // Метод, проверяющий, уничтожен ли корабль

    int getLength(); // Метод для получения длины корабля
    //bool isSunk() const;
    // Предполагаемые функции:

    char getOrientation(); // Метод для получения ориентации корабля

    string toString(); // Метод для получения строкового представления корабля
    void hit(ShipSegment* segment);
    Ship(const std::vector<std::pair<int, int>>& segmentCoordinates) {
        for (const auto& coord : segmentCoordinates) {
            segments.emplace_back(coord.first, coord.second); // Добавляем
сегменты в корабль
        }
    }

    int getSegmentIndex(int x, int y) {
        for (size_t i = 0; i < segments.size(); ++i) {
            if (segments[i].x == x && segments[i].y == y) {
                return static_cast<int>(i);  // Возвращаем индекс сегмента
            }
        }
        return -1; // Возврат -1, если сегмент не найден
    }

    /*Ship(const std::vector<std::pair<int, int>>& segmentCoordinates) {
        for (const auto& coord : segmentCoordinates) {
            segments.emplace_back(coord.first, coord.second);
        }
    }*/

  /*  bool isHit(int x, int y) {
        for (auto& segment : segments) {
            if (segment.x == x && segment.y == y && !segment.isHit) {
                segment.isHit = true;
                return true; // Попадание
            }
        }
        return false; // Промах
    }*/

    bool isSunk() const {
        for (const auto& segment : segments) {
            if (!segment.isHit) {
                return false; // Найден непораженный сегмент
            }
        }
        return sunk; // Все сегменты поражены
    }

    const std::vector<ShipSegment>& getSegments() const { return segments; }
    // Метод для добавления сегмента (координаты)
    void addSegment(int x, int y) {
```

```
        shipCoordinates.push_back({ x, y });
    }

    const std::vector<std::pair<int, int>>& getSegment() const {
        return shipCoordinates;
    }
    Ship(int length) : length(length), sunk(false) {}
    int getLength() const { return length; }
 //   bool isSunk() const { return sunk; }

    Ship(int x, int y, int length, bool horizontal)
        : startX(x), startY(y), length(length), horizontal(horizontal) {}

    bool isHit(int x, int y) const {
        std::cout << "Проверка попадания: (" << x << ", " << y << ") для корабля
на ("
            << startX << ", " << startY << "), длина: " << length << ",
ориентация: "
            << (horizontal ? "горизонтально" : "вертикально") << std::endl;

        if (horizontal) {
            return (y == startY && x >= startX && x < startX + length);
        }
        else {
            return (x == startX && y >= startY && y < startY + length);
        }
    }

    // Метод для проверки, содержится ли точка в корабле
    bool isAt(int x, int y) const {
        if (horizontal) {
            return (y == startY && x >= startX && x < startX + length);
        }
        else {
            return (x == startX && y >= startY && y < startY + length);
        }
    }
    bool isDestroyed() const {
        return hits >= length; // hits - количество повреждений
    }

   /* int getLength() const {
        return length;
    }*/
};
#include "Ship.h"
Ship::Ship(int length, char orientation)
{
    length_ = length;
    orientation_ = orientation;
    segments_.resize(length);
    for (int i = 0; i < length; ++i)
    {
        segments_[i] = new int(2); // Инициализируем каждый сегмент значением "2"
(представление состояния сегмента, например, "whole")
    }
}

int* Ship::getSegment(int i)
{
    return segments_[i];
}

void Ship:: hit(int* segment)
{
    if (segment == nullptr)
    {
```

```
        cout << "Невозможно обработать запрос, так как сегмент не существует" <<
endl;
        return;
    }

    if (*segment == 2) // whole
    {
        *segment = 1; // damaged
    }
    else if (*segment == 1) // damaged
    {
        *segment = 0; // destroyed
    }
}

bool Ship::isDestroyed()
{
    for (int* segment : segments_)
    {
        if (*segment > 0) // Проверка не уничтожен ли сегмент
        {
            return false;
        }
    }
    return true;
}


int Ship::getLength()
{
    return length_;
}

char Ship:: getOrientation()
{
    return orientation_;
}

string Ship:: toString()
{
    string orientation_str;
    if (orientation_ == 'h')
    {
        orientation_str = "горизонталь";
    }
    else if (orientation_ == 'v')
    {
        orientation_str = "вертикаль";
    }

    string segments_str = "";
    for (int* segment : segments_)
    {
        switch (*segment)
        {
        case 2:
            segments_str += "whole, ";
            break;
        case 1:
            segments_str += "damaged, ";
            break;
        default:
            segments_str += "destroyed, ";
            break;
        }
    }
```

```
    if (!segments_str.empty())
    {
        segments_str.pop_back(); // Проверка если сегмент не уничтожен
        segments_str.pop_back();
    }

     return "Корабль длиной  " + to_string(length_) + " в " + orientation_str + "
ориентации: " + segments_str;
}


#ifndef GAME_FIELD_H
#define GAME_FIELD_H

#include <vector>
#include "ship.h"          // Класс Ship
#include "ship_manager.h"  // Класс ship_manager
#include "OutOfBoundsAttackException.h" // Исключения
#include "ShipPlacementException.h"

enum CellState { s, n, u }; // s - корабль, n - нет корабля, u - неизвестно
//enum class CellState { Empty, Ship, Hit };
struct Cell {
    CellState state;  // Состояние клетки (s - корабль, n - нет корабля, u -
неизвестно)
    Ship* ship;       // Указатель на объект Ship, который размещен в данной
клетке
    int* shipSegment; // Указатель на сегмент корабля, который находится в данной
клетке

    Cell() : state(u), ship(nullptr), shipSegment(nullptr) {} // Конструктор,
инициализирующий состояние клетки как "неизвестно"
};
// Убедитесь, что вы меняете это на:
enum class DetailedCellState {
    EMPTY, // Пустая клетка
    SHIP,  // Корабль
    HIT,   // Попадание
    MISS   // Промах
};
typedef CellState SimpleCellState;
class game_field{
private:
    int width_; // Ширина поля
    int height_; // Высота поля
    std::vector<std::vector<Cell>> field_; // Игровое поле
    ship_manager* ship_manager_; // Менеджер кораблей
    bool ability_possibility_; // Возможность использования способности

public:
    game_field(int width, int height, ship_manager* ship_manager);
    game_field()
        : width_(10), height_(10), ship_manager_(nullptr),
ability_possibility_(false) {
        field_ = std::vector<std::vector<Cell>>(width_,
std::vector<Cell>(height_)); // Инициализация поля
    }
    int getWidth();
    int getHeight();
    bool getAbilityPossibility();
    CellState getCellState(int x, int y);
    bool placeShip(Ship* ship, int x, int y, char orientation);
    bool placeAllShips();
    void attackCell(int x, int y);
    void printField();

};
```

```
#endif // GAME_FIELD_H
#include "game_field.h"
game_field::game_field(int width, int height, ship_manager* ship_manager) {
    width_ = width;
    height_ = height;
    field_.resize(height, vector<Cell>(width)); // Инициализация игрового поля
размером width x height
    ship_manager_ = ship_manager;
    ability_possibility_ = false;
}

int game_field::getWidth()
{
    return width_;
}

int game_field::getHeight()
{
    return height_;
}

bool  game_field:: getAbilityPossibility()
{
    if (ability_possibility_ == true)
    {
        ability_possibility_ = false;
        return true;
    }
    return false;
}

CellState game_field:: getCellState(int x, int y)
{
    if (!(x < 0 || x >= width_ || y < 0 || y >= height_))
    {
        return field_[y][x].state;
    }
}

bool game_field::placeShip(Ship* ship, int x, int y, char orientation)
{
    // Проверка, что корабль помещается в поле
    if (x < 0 || x >= width_ || y < 0 || y >= height_)
    {
        throw ShipPlacementException();
    }

    // Проверка, что корабль не выходит за пределы поля и не перекрывает другие
корабли
    for (int i = 0; i < ship->getLength(); i++) {
        int dx = (orientation == 'h') ? i : 0;
        int dy = (orientation == 'v') ? i : 0;
        if (x + dx < 0 || x + dx >= width_ || y + dy < 0 || y + dy >= height_)
        {
            throw ShipPlacementException();
        }
        if (field_[y + dy][x + dx].state == s) // Проверка на занятость клетки
        {
            throw ShipPlacementException();
        }
    }

    // Проверка, что вокруг корабля нет других кораблей
    for (int i = 0; i < ship->getLength(); i++)
    {
        int dx = (orientation == 'h') ? i : 0;
```

```
            int dy = (orientation == 'v') ? i : 0;
            for (int j = -1; j <= 1; j++)
            {
                for (int k = -1; k <= 1; k++)
                {
                    int nx = x + dx + j;
                    int ny = y + dy + k;
                    if (nx >= 0 && nx < width_ && ny >= 0 && ny < height_)
                    {
                        if (field_[ny][nx].state == s)
                        {
                            throw ShipPlacementException();
                        }
                    }
                }
            }
        }

        // Установка корабля на поле
        for (int i = 0; i < ship->getLength(); i++)
        {
            int dx = (orientation == 'h') ? i : 0;
            int dy = (orientation == 'v') ? i : 0;
            field_[y + dy][x + dx].state = s; // Помечаем клетки как занятые
            field_[y + dy][x + dx].ship = ship; // Связываем клетку с кораблем

            // Устанавливаем сегмент корабля на соответствующую клетку
            field_[y + dy][x + dx].shipSegment = ship->getSegment(i);
        }

        return true;
    }


    bool game_field:: placeAllShips()
    {
        for (int i = 0; i < ship_manager_->getNumShips(); i++)
        {
            Ship* ship = ship_manager_->getShips()[i];
            cout << "Введите координаты для корабля номер " << i + 1 << " длиной " <<
                ship->getLength() << " (x y): ";
            int x, y;
            cin >> x >> y;
            cout << endl;
            if (!placeShip(ship, x, y, ship->getOrientation()))
            {
                return false;
            }
        }

        return true;
    }

    void game_field::attackCell(int x, int y)
    {
        if (x < 0 || x >= width_ || y < 0 || y >= height_)
        {
            throw OutOfBoundsAttackException();
        }
        // проверка что клетка занята кораблем
        if (field_[y][x].state == s)
        {
            // получаем корабль, который атакован, и его сегмент
            Ship* attackedShip = field_[y][x].ship;
            int* segment = field_[y][x].shipSegment;

            // наносим повреждения по сегменту корабля
```

```
        attackedShip->hit(segment);

        if (attackedShip->isDestroyed())
        {
            ability_possibility_ = true; // условие для добавления способности
выполнено
        }
    }
}

void game_field::printField()
{
    cout << "Игровое поле:" << endl;
    for (int y = 0; y < height_; y++)
    {
        for (int x = 0; x < width_; x++)
        {
            char cell_char = (field_[y][x].state == s) ? 'S' : (field_[y][x].state
== n ? 'N' : '0');
            cout << cell_char << " ";
        }
        cout << endl;
    }
}
#pragma once
#include "vector"
#include <iostream>
#include "ship.h"
using namespace std;

class ship_manager
{
private:
    vector<Ship*> ships_; // Вектор указателей на корабли, который хранит все
корабли
    int ships_amnt_; // Количество кораблей
    vector<int> shipLengths_; // Длины кораблей
    vector<char> orientation_; // Ориентации кораблей
    std::vector<int> lengths;
    int count;
public:
    // Конструктор по умолчанию
    ship_manager()
        : ships_amnt_(0)  // Количество кораблей = 0
    {
        // Здесь можно инициализировать пустые вектора
        shipLengths_ = vector<int>();
        orientation_ = vector<char>();
        ships_ = vector<Ship*>();  // Пустой вектор указателей на корабли
    }

    ship_manager(int ships_amnt, vector<int> shipLengths); // Конструктор для
инициализации менеджера кораблей

    int getNumShips(); // Метод для получения количества кораблей

    vector<Ship*> getShips(); // Метод для получения списка кораблей

    string getShipInfo(int index); // Метод для получения информации о конкретном
корабле по индексу

    void printShipsInfo(); // Метод для вывода информации о всех кораблях
    void inputShipOrientations(); // Метод для ввода ориентаций для всех кораблей
    std::vector<int> getShipLengths() const {
        return lengths; // Возвращаем длины кораблей
    }
    ship_manager(const std::vector<int>& shipLengths)
```

```
        : lengths(shipLengths) {
        count = shipLengths.size(); // Установим количество кораблей
    }

    int getShipCounts() const {
        return count; // Возвращаем количество кораблей
    }
};
#include "ship_manager.h"
ship_manager::ship_manager(int ships_amnt, vector<int> shipLengths)
{
    ships_amnt_ = ships_amnt;

    // Запрашиваем ориентацию для каждого корабля с проверкой на корректность
ввода
    char orientation;
    for (int i = 0; i < ships_amnt_; i++) {
        do {
            cout << "Введите ориентацию для корабля " << i + 1 << " длиной " <<
shipLengths[i]
                << " (h - горизонтально, v - вертикально): ";
            cin >> orientation;
            cout << "\n";

            // Проверка на корректность ввода
            if (orientation != 'h' && orientation != 'v') {
                cout << "Ошибка! Введите правильную ориентацию (h для
горизонтально, v для вертикально)." << endl;
            }
        } while (orientation != 'h' && orientation != 'v');  // Повторяем запрос,
если ориентация некорректна

        // Добавляем ориентацию в список и создаем корабль с указанной ориентацией
        orientation_.push_back(orientation);
        ships_.emplace_back(new Ship(shipLengths[i], orientation));
    }
}



int ship_manager::getNumShips()
{
    return ships_amnt_;
}

vector<Ship*> ship_manager::  getShips()
{
    return ships_;
}

string ship_manager::getShipInfo(int index)
{
    return  ships_[index]->toString();
}

void  ship_manager::printShipsInfo()
{
    cout << "Информация о кораблях:" << endl;
    for (int i = 0; i < ships_amnt_; i++)
    {
        cout << "Корабль номер " << i + 1 << ":" << endl;
        cout << "  Длина: " << ships_[i]->getLength() << endl;
        cout << "  Ориентация: " << ships_[i]->getOrientation() << endl;
        cout << endl;
    }
}
#pragma once
```

```
#include "ship_manager.h"
#include "game_field.h"

class Ability
{
public:
    virtual void use(game_field& field, ship_manager& sm) = 0; // Метод для
использования способности

    // Метод для вывода информации о способности
    virtual std::string getInfo() const {
        return "Информация о способности";  // Пример текста, который можно
заменить на реальные данные
    }
};
#pragma once
#include "ability.h"
#include "doubleDamage.h"
#include "scanner.h"
#include "shelling.h"
#include "NoAbilitiesException.h"
#include <memory>

class Ability_Manager
{
private:
    // std::queue<std::unique_ptr<Ability>> abilities;
    std::vector<std::unique_ptr<Ability>> availableAbilities; // Доступные
способности
    queue<Ability*> abilities; // Очередь для способностей
   // std::vector<Ability*> abilities;
    void shuffleAbilities(vector<Ability*>& abilityList)
    {
        for (int i = 0; i < abilityList.size(); ++i)
        {
            int j = rand() % abilityList.size(); // Генерация случайного индекса
для обмена
            Ability* tmp = abilityList[j]; // Временная переменная для обмена
            abilityList[j] = abilityList[i]; // Меняем местами элементы
            abilityList[i] = tmp; // Завершаем обмен
        }
    }

public:

    Ability_Manager(); // Конструктор менеджера способностей
    void displayAbilities();
    // Метод для использования способности
    void useAbility(game_field& field, ship_manager& sm);

    void addAbility(std::unique_ptr<Ability> ability); // Метод для добавления
способности
    void addRandomAbilityIfPossible(game_field& field); // Метод для добавления
случайной способности, если возможно
    void addAbilityFromUser(); // Метод для добавления способности от пользователя
    void shuffleAbilities(std::vector<std::unique_ptr<Ability>>& abilities); //
Метод для перемешивания списка способностей
};
#include "Ability_Manager.h"
Ability_Manager::Ability_Manager()
{
    // Инициализация списка способностей
    vector<Ability*> abilityList;
    abilityList.push_back(new DoubleDamage());
    abilityList.push_back(new Scanner());
    abilityList.push_back(new Bombardment());
    // Перемешиваем список способностей
```

```
    shuffleAbilities(abilityList);
    // Добавляем способности в очередь для дальнейшего использования
    for (Ability* ability : abilityList)
    {
        abilities.push(ability);
    }
}
// Метод для отображения списка способностей
/*void Ability_Manager::displayAbilities() {
    std::cout << "Available abilities:" << std::endl;
    for (size_t i = 0; i < abilities.size(); ++i) {
        std::cout << i + 1 << ". " << abilities[i]->getName() << std::endl;
    }
}
*/

void Ability_Manager::useAbility(game_field& field, ship_manager& sm)
{
    // Проверка наличия способностей
    if (!abilities.empty()) {
        Ability* ability = abilities.front();  // Получаем способность из очереди

        // Если способность является nullptr
        if (ability == nullptr) {
            std::cerr << "Ошибка: ability == nullptr!" << std::endl;
            // Просто выходим из функции, не используя способность
            return;
        }

        // Если способность есть и она валидна
        abilities.pop();  // Удаляем способность из очереди
        ability->use(field, sm);  // Используем способность
        delete ability;  // Освобождаем память
    }
    else {
        // Если нет доступных способностей
        std::cerr << "Ошибка: Нет доступных способностей!" << std::endl;
        // Просто не выполняем действия и продолжаем игру
        return;
    }
}


void Ability_Manager::addRandomAbilityIfPossible(game_field& field)
{
    if (field.getAbilityPossibility())
    {
        int randomIndex = rand() % 3; // Генерация случайного индекса от 0 до 2

        Ability* randomAbility = nullptr;
        switch (randomIndex)
        {
        case 0:
            randomAbility = new Bombardment();
            break;
        case 1:
            randomAbility = new Scanner();
            break;
        case 2:
            randomAbility = new DoubleDamage();
            break;
        default:
            cout << "Произошла ошибка при генерации способности. " << endl;
        }
        abilities.push(randomAbility); // Добавляем случайную способность в
очередь способностей
```

```
    }
    else
    {
        cout << "-Невозможно добавить способность, так как не выполнены условия. "
<< endl;
    }
}
#pragma once
#include "ability.h"
class Scanner : public Ability
{
public:
    void use(game_field& field, ship_manager& sm) override
    {
        cout << "Использование способности: Сканирование!" << endl;
        int x, y;
        cout << "Введите координаты для сканирования области 2x2 (x y):";
        cin >> x >> y;


        for (int i = 0; i < 2; ++i)
        {
            for (int j = 0; j < 2; ++j)
            {
                if (x + j < field.getWidth() && y + i < field.getHeight())
                {
                    if (field.getCellState(x + j, y + i) == s)
                    {
                        cout << "Обнаружен корабль в клетке (" << x + j << ", " <<
y + i << ")" << endl;
                    }
                }
            }
        }
    }
    std::string getInfo() const override {
        return "Сканирование";
    }

};
#pragma once
#include "ability.h"

class Bombardment : public Ability {
public:
    void use(game_field& field, ship_manager& sm) override {
        std::cout << "Использование способности: Обстрел!" << std::endl;

        int numShips = sm.getNumShips();
        if (numShips == 0) {
            std::cout << "Ошибка: У вас нет доступных кораблей для обстрела!" <<
std::endl;
            return; // Выход из функции, если кораблей нет
        }

        // Выбираем случайный корабль и случайный сегмент
        int shipIndex = rand() % numShips; // Получаем индекс корабля
        Ship* randomShip = sm.getShips()[shipIndex]; // Доступ к кораблю

        if (randomShip == nullptr) {
            std::cout << "Ошибка: Выбранный корабль недоступен!" << std::endl;
            return; // Выход из функции, если корабль недоступен (например,
освобожден)
        }

        int segmentIndex = rand() % randomShip->getLength();
```

```
        // Попытка нанести удар по выбранному сегменту
        randomShip->hit(randomShip->getSegment(segmentIndex));
        std::cout << "Попадание в сегмент " << segmentIndex + 1 << " корабля " <<
shipIndex + 1 << std::endl;
    }

    std::string getInfo() const override {
        return "Обстрел";
    }
};
#pragma once
#include "ability.h"

class DoubleDamage : public Ability {
public:
    void use(game_field& field, ship_manager& sm) override {
        cout << "Использование способности: Двойной удар!" << endl;
        int x, y;
        cout << "Введите координаты для атаки (x y):";
        cin >> x >> y;

        // Делаем два удара по клетке
        field.attackCell(x, y);
        field.attackCell(x, y);
    }

    std::string getInfo() const override {
        return "Двойной удар";
    }

};
#pragma once
#include <iostream>
#include <string>
#include <vector>
#include <cstdlib>
#include <queue>
#include <exception>

using namespace std;

class NoAbilitiesException : public exception {
public:
    const char* what() const noexcept override {
        return "Нет доступных способностей!";
    }
};
#pragma once
#include <iostream>
#include <string>
#include <vector>
#include <cstdlib>
#include <queue>
#include <exception>

using namespace std;

class OutOfBoundsAttackException : public exception {
public:
    const char* what() const noexcept override {
        return "Атака выходит за пределы поля!";
    }
};
#pragma once

#include <iostream>
#include <string>
```

```
#include <vector>
#include <cstdlib>
#include <queue>
#include <exception>

using namespace std;

class ShipPlacementException : public exception {
public:
    const char* what() const noexcept override {
        return "Ошибка размещения корабля: Не удается разместить корабль на
поле!";
    }
};
#ifndef ENEMY_SHIP_INITIALIZER_H
#define ENEMY_SHIP_INITIALIZER_H

#include <vector>
#include "ship.h"

std::vector<Ship*> initializeEnemyShips(int shipCount, int fieldWidth, int
fieldHeight);

#endif // ENEMY_SHIP_INITIALIZER_H
#include "EnemyShipInitializer.h"
#include <cstdlib>

std::vector<Ship*> initializeEnemyShips(int shipCount, int fieldWidth, int
fieldHeight) {
    std::vector<Ship*> ships;

    for (int i = 0; i < shipCount; ++i) {
        // Случайное размещение кораблей с сегментами
        std::vector<std::pair<int, int>> segmentCoordinates;

        for (int j = 0; j < 3; ++j) { // Например, каждый корабль имеет 3 сегмента
            int x = rand() % fieldWidth;
            int y = rand() % fieldHeight;
            segmentCoordinates.emplace_back(x, y);
        }

        ships.push_back(new Ship(segmentCoordinates));
    }

    return ships;
}
#include "ship.h"
#include "ship_manager.h"
#include "ability.h"
#include "game_field.h"
#include "Ability_Manager.h"
#include "Game.h"
#include "GameState.h"
#include "EnemyShipInitializer.h"

int main() {
    setlocale(LC_ALL, "Russian");  // Устанавливаем локализацию на русский язык
    srand(static_cast<unsigned int>(time(0)));  // Инициализация генератора
случайных чисел

    // Инициализация ship_manager с 3 кораблями (длиной 2, 3 и 3)
    ship_manager manager(3, { 2, 3, 3 });
    // Вводим ориентации кораблей
    // manager.inputShipOrientations(); // Этот вызов был закомментирован
    // Печатаем информацию о кораблях
    manager.printShipsInfo();
```

```
    // Создаем игровое поле размером 10x10
    game_field field(10, 10, &manager);

    // Размещение всех кораблей на поле
    field.placeAllShips();

    // Печатаем игровое поле
    field.printField();

   /* // Создаем менеджер способностей (Ability Manager)
    Ability_Manager abilityManager;

    // Совершаем несколько атак по клеткам
    field.attackCell(0, 0);
    field.attackCell(0, 0);
    field.attackCell(0, 0);
    field.attackCell(0, 1);
    field.attackCell(0, 1);
    field.attackCell(0, 1);

    // Используем способности
    abilityManager.useAbility(field, manager);
    abilityManager.useAbility(field, manager);
    abilityManager.useAbility(field, manager);
    abilityManager.useAbility(field, manager);

    // Добавляем случайную способность, если возможно
    abilityManager.addRandomAbilityIfPossible(field);
    abilityManager.addRandomAbilityIfPossible(field);

    // Используем способность еще раз
   abilityManager.useAbility(field, manager);
   // srand(time(0));  // Инициализация генератора случайных чисел*/

   // Создание противника
    Enemy enemy(100); // Здоровье противника 100

    // Инициализация случайных кораблей у противника
    std::vector<Ship*> enemyShips = initializeEnemyShips(3, 10, 10); // 3 корабля,
10x10 поле
    for (Ship* ship : enemyShips) {
        enemy.addShip(ship); // Добавляем корабли противника
    }

    // Создаем менеджер способностей (Ability Manager)
    Ability_Manager abilityManager;

    Game game;
    game.startNewGame();

    std::string filename = "game_save.dat";

    while (true) {
        std::cout << "\nMenu:\n1. Начать новую игру\n2. Загрузить
игру\n3.Сохранить игру\n4. Выход\n";
        int choice;
        std::cin >> choice;

        if (choice == 1) {
            game.startNewGame();
            game.playRound();
        }
        else if (choice == 2) {
            game.loadGame(filename);
            game.playRound();
        }
        else if (choice == 3) {
```

```
            game.saveGame(filename);
        }
        else if (choice == 4) {
            break;
        }
    }

    std::cout << "Игра завершена." << std::endl;

    // Очистка динамически выделенной памяти
    for (Ship* ship : enemyShips) {
        delete ship; // Не забудьте освободить память для кораблей противника
    }

    return 0;
}
```