МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра математического обеспечения и применения ЭВМ

ОТЧЕТ

по лабораторной работе №3
по дисциплине «Объектно-ориентированное программирование»
Тема: Связывание классов.

Студент гр. 3382 Самойлова Е. М..
Преподаватель Жангиров Т. Р.

Санкт-Петербург
2024

Цель работы: Изучить принцип связывания классов на языке C++. Создать UML диаграмму.

Задание:

1) Создать класс игры, который реализует следующий игровой цикл:

2) Начало игры

3) Раунд, в котором чередуются ходы пользователя и компьютерного врага. В свой ход пользователь может применить способность и выполняет атаку. Компьютерный враг только наносит атаку.

4) В случае проигрыша пользователь начинает новую игру

5) В случае победы в раунде, начинается следующий раунд, причем состояние поля и способностей пользователя переносятся.

6) 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.

7) 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.

8) Примечание:

9) Класс игры может знать о игровых сущностях, но не наоборот

10) Игровые сущности не должны сами порождать объекты состояния

11) Для управления самое игрой можно использовать обертки над командами

12) При работе с файлом используйте идиому RAII.

Выполнение работы

1. GameState и методы:

GameState управляет состоянием игры. Это включает в себя:

- Очки игрока и противника.
- Номер текущего раунда.
- Флаг, показывающий, чья сейчас очередь (игрока или противника).

Методы в GameState:

- initialize(): Сбрасывает все значения (очки, раунд, ход) в начальное состояние.
- save() и load(): Используются для сохранения и загрузки состояния игры в/из потоков (файлы, например).

2. Конструктор и деструктор Game:

- Конструктор: Создает объект игры, но без дополнительных операций (в вашем случае).
- Деструктор: Также не выполняет никаких дополнительных действий, так как ресурсы, скорее всего, управляются автоматически через смарт-пойнтеры.

3. Инициализация игры:

- Метод initializeGame() инициирует все важные компоненты игры:
    - Инициализирует поля игрока и противника.
    - Инициализирует способности (если они присутствуют).
    - Инициализирует состояние игры (очки, раунд, чей ход).

4. Инициализация поля игрока и противника:

- Игрок: Может выбрать, хочет ли он сгенерировать поле случайным образом или настроить его вручную. Если поле генерируется вручную, игрок размещает корабли один за другим на своем поле, вводя их координаты и ориентацию.
- Противник: Генерирует поле случайным образом. Это поле создается с использованием предварительно заданных размеров кораблей и их случайного размещения.

5. Инициализация способностей:

- Способности случайным образом присваиваются игроку. Это может быть что-то вроде "двойного урона", "бомбардировки" или "сканирования" поля противника для нахождения кораблей.

6. Основной игровой цикл:

- В методе playRound() происходит основной игровой процесс. Игрок и противник по очереди делают ходы, используя свои способности и атакуя.
- Игроку предлагаются команды для управления:
  - q: Выход из игры.
  - l: Загрузка игры.
  - s: Сохранение игры.
  - c: Продолжение раунда (в этом случае игрок выбирает координаты для атаки и решает, использовать ли способности).

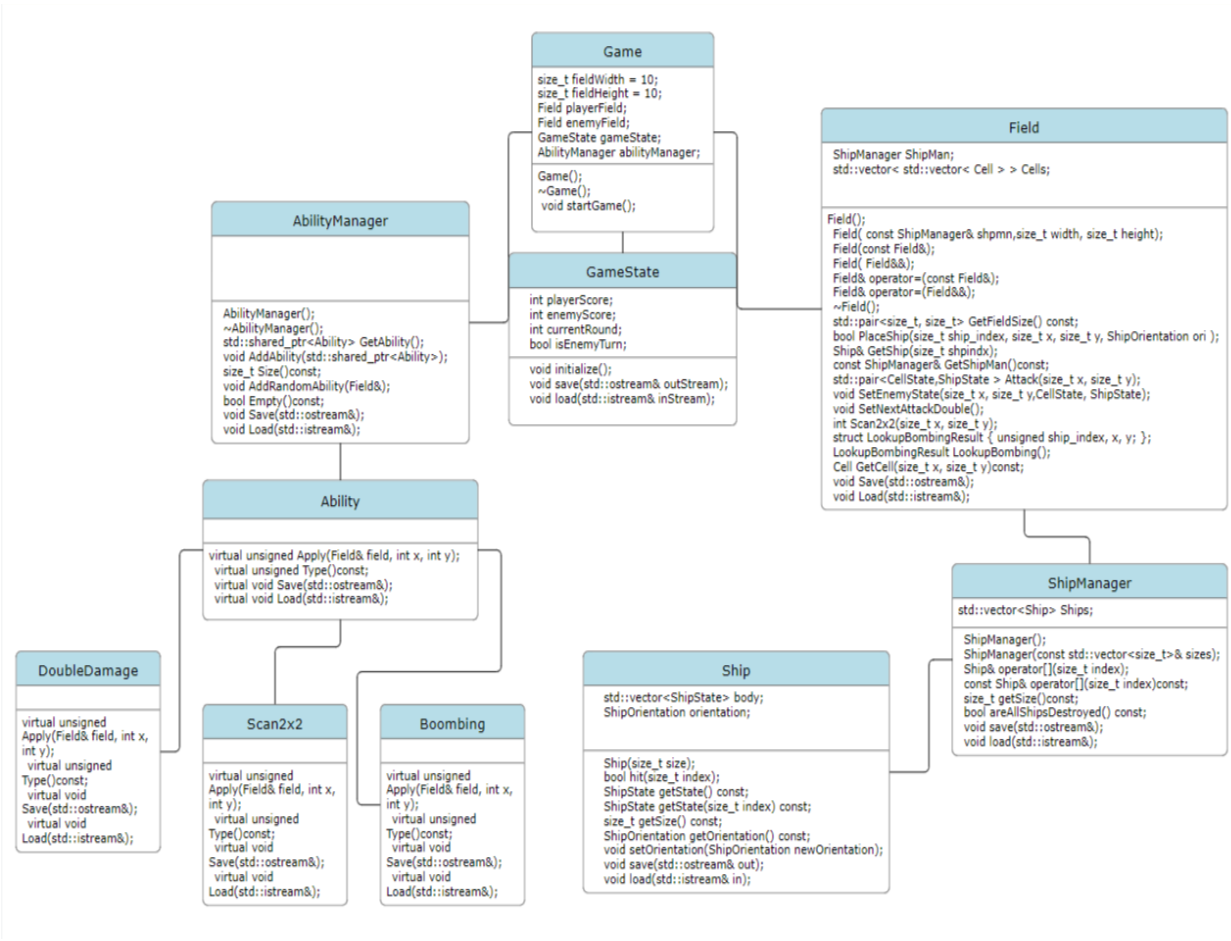
7. Ход игрока и противника:

- Ход игрока: Игрок выбирает, где атаковать (по координатам) и может использовать способности, если они есть. Также есть проверка, не были ли уничтожены все корабли противника.
- Ход противника: Противник выбирает случайные координаты для атаки на поле игрока.

8. Сохранение и загрузка игры:

- Сохранение: Состояние игры (включая размеры поля, расположение кораблей, очки и способности) сохраняется в файл.
- Загрузка: Загружает игру из сохраненного состояния, восстанавливая все параметры (поля, очки, способности).

UML - диаграмма:



Вывод

В ходе лабораторной работы удалось создать новые классы, реализовано взаимодействие между классами таким образом, чтобы обеспечить корректную работу программы. Связи между объектами организованы логично и соответствуют их роли в игре.

Приложение

Файл Game.cpp:

```
#include "Game.h"

#include "Field.h"
#include "ShipManager.h"
#include "Ship.h"
#include "AbilityManager.h"
#include "Exceptions.h"
#include <iostream>
#include <random>
#include <fstream>
#include <algorithm>

void GameState::initialize()
{
    playerScore = enemyScore = currentRound = 0;
    isEnemyTurn = false;
}

void GameState::save(std::ostream& str)
{
    str << playerScore << " " << enemyScore << " " << currentRound << std::endl;
}

void GameState::load(std::istream& str)
{
    str >> playerScore >> enemyScore >> currentRound;
    isEnemyTurn = false;
}

Game::Game()
{

}

Game::~Game()
{
}

void Game::initializeGame()
{
    initializeEnemyField();
    initializePlayerField();
    initializeAbilities();
    gameState.initialize();
}
```

```
void Game::initializeRound()
{
    initializeEnemyField();
}

Field Game::generateRandomField()
{
    ShipManager sm({4,3,3,2,2,2,1,1,1,1});
    Field b(sm, fieldWidth, fieldHeight);
    for (int i = 0; i < 10; i++) {
        bool flag;
        do {
            auto ori = std::rand() & 1 ? ShipOrientation::Horizontal : ShipOrientation::Vertical;
            auto s = sm[i].getSize();
            unsigned x, y;
            if (ori == ShipOrientation::Horizontal) {
                x = std::rand() % (fieldWidth - 1 - s);
                y = std::rand() % (fieldHeight - 1);
            }
            else {
                x = std::rand() % (fieldWidth - 1);
                y = std::rand() % (fieldHeight - 1 - s);
            }
            flag = false;
            try {
                flag = b.PlaceShip(i, x, y, ori);
            }
            catch (...) {
            }
        } while (!flag);
    }
    return b;
}

void Game::initializePlayerField()
{
    std::string ans;
    std::cout << "Do you want to generate a random field? (y/n): ";
    std::cin >> ans;

    if (ans == "y") {
        playerField = generateRandomField();
        return;
    }

    ShipManager sm({ 4, 3, 3, 2, 2, 2, 1, 1, 1, 1 });
    Field b(sm, fieldWidth, fieldHeight);

    // Цикл по всем кораблям
    for (int i = 0; i < 10; i++) {
        bool flag;
```

```
            int attempts = 0;
            do {
                auto s = sm[i].getSize();
                unsigned x, y;
                char o;
                std::cout << b << std::endl;
                std::cout << s << "-segment ship; enter X Y Orient(h or v): ";
                std::cin >> x >> y >> o;

                // Проверка на допустимость ориентации
                auto ori = (o == 'h' || o == 'H') ? ShipOrientation::Horizontal : ShipOrientation::Vertical;

                flag = false;
                try {
                    flag = b.PlaceShip(i, x, y, ori);
                }
                catch (const std::exception& e) {
                    std::cout << "Error: " << e.what() << std::endl;
                }

                attempts++;

                if (attempts >= 100) {
                    std::cout << "Too many failed attempts to place the ship. Exiting." << std::endl;
                    return;
                }

            } while (!flag);
        }

        playerField = b;
    }

    void Game::initializeEnemyField()
    {
        enemyField = generateRandomField();
    }

    void Game::initializeAbilities()
    {
        unsigned x = std::rand() % (fieldWidth - 1);
        unsigned y = std::rand() % (fieldHeight - 1);

        // Массив указателей на способности
        std::vector<std::shared_ptr<Ability>> abilities = {
            std::make_shared<DoubleDamage>(),
            std::make_shared<Bombing>(),
            std::make_shared<Scanner>(x, y)
        };

        std::random_device rd;
```

```
    std::default_random_engine rng(rd());
    std::shuffle(abilities.begin(), abilities.end(), rng);

    for (int i = 0; i < 3; ++i) {
        abilityManager.AddAbility(abilities[i]);
    }
}

RoundOutcome Game::playRound()
{
    gameState.currentRound += 1;

    while (true) {
        std::cout << std::endl;
        std::cout << playerField;
        std::cout << "Round: " << gameState.currentRound << std::endl;
        std::cout << "Your Score: " << gameState.playerScore << " Enemy score: " <<
gameState.enemyScore << std::endl;
        std::cout << "Abilities: " << abilityManager.Size() << std::endl;

        char cmd = getCommandFromUser();
        if (cmd == 'q') return RoundOutcome::Quit;
        if (cmd == 'l') { loadGame(); continue; }
        if (cmd == 's') { saveGame(); continue; }

        if (cmd == 'c') {
            int x, y;
            char useAbility = 'n';

            if (!abilityManager.Empty()) {
                std::cout << "Enter x y ability (y or n): ";
                std::cin >> x >> y >> useAbility;
            } else {
                std::cout << "Enter x y: ";
                std::cin >> x >> y;
            }

            if (x >= 0 && y >= 0) {
                UserTurn(x, y, useAbility == 'y');
                if (enemyField.GetShipMan().areAllShipsDestroyed()) {
                    std::cout << enemyField << std::endl;
                    std::cout << "Round is over. You won!" << std::endl;
                    return RoundOutcome::RoundComplete;
                }

                EnemyTurn();
                if (playerField.GetShipMan().areAllShipsDestroyed()) {
                    std::cout << playerField << std::endl;
                    std::cout << "Round is over. You were defeated!" << std::endl;
                    std::cout << "Game is over. You were defeated!" << std::endl;
                    return RoundOutcome::GameOver;
```

```
            }
        }
    }
}

char Game::getCommandFromUser()
{
    char cmd;
    std::cin.clear();
    std::cout << "Enter cmd (q - quit, s - save, l - load, c - continue): ";
    std::cin >> cmd;
    std::cout << std::endl;
    return cmd;
}

void Game::startGame()
{
    for(;;){
        std::cout << "Sea battle!" << std::endl;
        initializeGame();
        for(;;){
            auto res = playRound();
            if (res == RoundOutcome::Quit) {
                std::cout << "Quit" << std::endl;
                return;
            }
            if (res == RoundOutcome::GameOver)
                break;
            if (res == RoundOutcome::RoundComplete) {
                std::cout << "New round!" << std::endl;
                initializeRound();
            }
        }
    }
}

bool Game::UserTurn(size_t x, size_t y, bool use_ability)
{
    if (gameState.isEnemyTurn)
        throw InternalError();

    gameState.isEnemyTurn = true;

    if (use_ability) {
        auto ability = abilityManager.GetAbility();
        unsigned a = applyAbility(*ability, this->enemyField, x, y);
        switch (a)
        {
        case 1:
        {
```

```
            std::cout << "Scanner found ship!" << std::endl;
            break;
        }
        case 2:
        {
            enemyField.Attack(x, y);
            break;
        }
        case 3:
            enemyField.LookupBombing();
            break;
        default:
            break;
        }
    }

    auto [cs, ss] = enemyField.Attack(x, y);
    playerField.SetEnemyState(x, y, cs, ss);

    if (cs == CellState::Occupied) {
        gameState.playerScore += 1;
        if (ss == ShipState::Destroyed) {
            abilityManager.AddRandomAbility(playerField);
        }
    }

    return cs == CellState::Occupied;
}

bool Game::EnemyTurn()
{
    if (!gameState.isEnemyTurn)
        throw InternalError();
    gameState.isEnemyTurn = false;
    auto [w, h] = playerField.GetFieldSize();
    unsigned x = std::rand() % (w - 1);
    unsigned y = std::rand() % (h - 1);
    auto [cs, ss] = playerField.Attack(x, y);
    enemyField.SetEnemyState(x, y, cs, ss);
    if (cs != CellState::Occupied)
        return false;
    gameState.enemyScore += 1;
    return true;
}

unsigned Game::applyAbility(Ability& ab, Field& field, int x, int y)
{
    return ab.Apply(field, x, y);
}

void Game::saveGame()
```

```
{
    std::ofstream of("saved_game", std::ios_base::trunc);
    of << fieldWidth << " " << fieldHeight<<std::endl;
    playerField.Save(of);
    enemyField.Save(of);
    gameState.save(of);
    abilityManager.Save(of);
}

void Game::loadGame()
{
    std::ifstream f("saved_game");
    f>> fieldWidth>>fieldHeight;
    playerField.Load(f);
    enemyField.Load(f);
    gameState.load(f);
    abilityManager.Load(f);
}
```

Файл Game.h:

```
#pragma once

#include "Field.h"
#include "ShipManager.h"
#include "Ship.h"
#include "AbilityManager.h"
#include <ostream>
#include <istream>

struct GameState {
    int playerScore;
    int enemyScore;
    int currentRound;
    bool isEnemyTurn;
    void initialize();
    void save(std::ostream& outStream);
    void load(std::istream& inStream);
};

enum class RoundOutcome { Quit, GameOver, RoundComplete };

class Game {
private:
    size_t fieldWidth = 10;
    size_t fieldHeight = 10;
    Field playerField;
    Field enemyField;
    GameState gameState;
```

```
    AbilityManager abilityManager;

private:
    void initializeGame();
    void initializeRound();
    Field generateRandomField();
    void initializePlayerField();
    void initializeEnemyField();
    void initializeAbilities();
    RoundOutcome playRound();
    bool UserTurn(size_t x, size_t y, bool useAbility);
    bool EnemyTurn();
    unsigned applyAbility(Ability& ability, Field& field, int x, int y);
    char getCommandFromUser();
    void saveGame();
    void loadGame();

public:
    Game();
    ~Game();
    void startGame();
};
```