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 Мокрушина В. Л.
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

Структура GameState содержит информацию о текущем состоянии игры:

UserScore и EnemyScore: очки игрока и противника.

RoundNumber: номер текущего раунда.

EnemyTurn: флаг, указывающий, чей ход (враг или игрок).

Методы:

Init() — инициализирует начальное состояние игры.

Save() и Load() — сохраняют и загружают состояние игры из потока данных.

2. Класс Game (файл Player.h)

Реализует весь процесс игры, включая управление досками, раундами, ходами и специальными способностями.

Приватные члены:

BoardWidth и BoardHeight: размеры игровой доски (по умолчанию 10x10).

PlayerBoard и EnemyBoard: доски для игрока и противника.

State: состояние игры (объект типа GameState).

Abilities: управление специальными способностями (объект типа AbilityManager).

Основные методы:

InitializeGame() — инициализирует игру, включая доски и способности.

InitializePlayerBoard() — запрашивает у игрока расположение кораблей (или случайно генерирует доску).

InitializeEnemyBoard() — случайно генерирует доску противника.

InitializeAbilities() — добавляет случайные способности в игру.

RunGame() — запускает основной цикл игры.

Round() — управление одним раундом, включая ходы игрока и противника.

UserTurn() и EnemyTurn() — реализуют логику ходов игрока и противника.

SaveGame() и LoadGame() — сохраняют/загружают состояние игры в/из файла.

Методы структуры GameState:

1.Init().Устанавливает начальное состояние игры:

Очки игрока и противника (UserScore, EnemyScore) равны нулю.

Номер раунда (RoundNumber) устанавливается на 0.

Флаг хода противника (EnemyTurn) сбрасывается в false.

2.Save(std::ostream& str).Записывает текущее состояние (UserScore, EnemyScore, RoundNumber) в переданный поток вывода.

3.Load(std::istream& str).Считывает состояние (UserScore, EnemyScore, RoundNumber) из переданного потока ввода и сбрасывает флаг хода противника (EnemyTurn) в false.

Методы класса Game:

InitializeGame()

Вызывает методы:

InitializeEnemyBoard() — создаёт доску противника.

InitializePlayerBoard() — создаёт доску игрока.

InitializeAbilities() — добавляет случайные способности.

Сбрасывает состояние игры (State.Init()).

InitializePlayerBoard()

Предлагает игроку выбрать способ генерации доски:

Если игрок выбирает случайную генерацию, вызывается GenerateRandomBoard().Если игрок размещает корабли вручную: Инициализируется объект ShipManager, содержащий список кораблей.

Для каждого корабля предлагается ввести:

Координаты начала (x, y), ориентацию (h или v для горизонтальной или вертикальной).Проверяется корректность размещения корабля. Если попытка неудачна, ввод повторяется.

InitializeEnemyBoard().Генерирует случайную доску противника с помощью GenerateRandomBoard().

InitializeAbilities().Случайно добавляет три способности в AbilityManager:

"Двойной урон" (DoubleDamage).

"Бомбардировка" (Bombing).

"Сканер" (Scanner), сгенерированный для случайной точки (x, y).

GenerateRandomBoard()

Инициализирует объект ShipManager с заданным набором размеров кораблей. Размещает корабли случайным образом: Выбирает ориентацию (горизонтальную или вертикальную), вычисляет координаты с учётом размера корабля. Проверяет, можно ли разместить корабль в указанной позиции. Если нельзя, повторяет попытку. Возвращает готовую доску (GameBoard).

Игровой процесс

RunGame()

Главный цикл игры:

Инициализирует игру через InitializeGame().

В цикле вызывает Round():Если игрок завершает игру, цикл прерывается. Если раунд завершается победой игрока, начинается новый раунд через InitializeRound().

Round() Организует один игровой раунд: Увеличивает номер раунда (State.RoundNumber).Выполняется цикл: Отображаются текущая доска и очки.Игрок вводит команду:

Q — завершить игру.

L — загрузить состояние игры.

S — сохранить состояние игры.

T — выполнить атаку:

Игрок вводит координаты и указывает, активируется ли способность. Вызывается UserTurn() для выполнения атаки. Проверяется, уничтожены ли все корабли противника. Если да, игрок выигрывает. Выполняется ход противника через EnemyTurn(). Если игрок теряет все корабли, игра завершается поражением.

UserTurn(size_t x, size_t y, bool use_ability)

Проверяет, что сейчас ход игрока. Если активирована способность: Берёт способность из AbilityManager и применяет её через ApplyAbility(). Выполняет атаку: Вызывает EnemyBoard.Attack(x, y), чтобы атаковать указанную ячейку. Обновляет состояние доски игрока (PlayerBoard.SetEnemyState). Если ячейка занята (CellState::Occupied), увеличивает очки игрока. Если корабль противника уничтожен, добавляет случайную способность игроку.

EnemyTurn()

Проверяет, что сейчас ход противника.Случайно генерирует координаты для атаки. Выполняет атаку: Вызывает PlayerBoard.Attack(x, y), чтобы атаковать указанную ячейку. Обновляет состояние доски противника

(EnemyBoard.SetEnemyState). Если ячейка занята, увеличивает очки противника.

Сохранение и загрузка

SaveGame().Открывает файл saved_game для записи.Сохраняет:

Размеры доски, доски игрока и противника (Save() у досок), состояние игры., состояние способностей.

LoadGame().Открывает файл saved_game для чтения. Загружает:Размеры доски, доски игрока и противника (Load() у досок), состояние игры, состояние способностей.

SetNextAttackDouble().Активирует способность "Двойной урон" для следующей атаки.

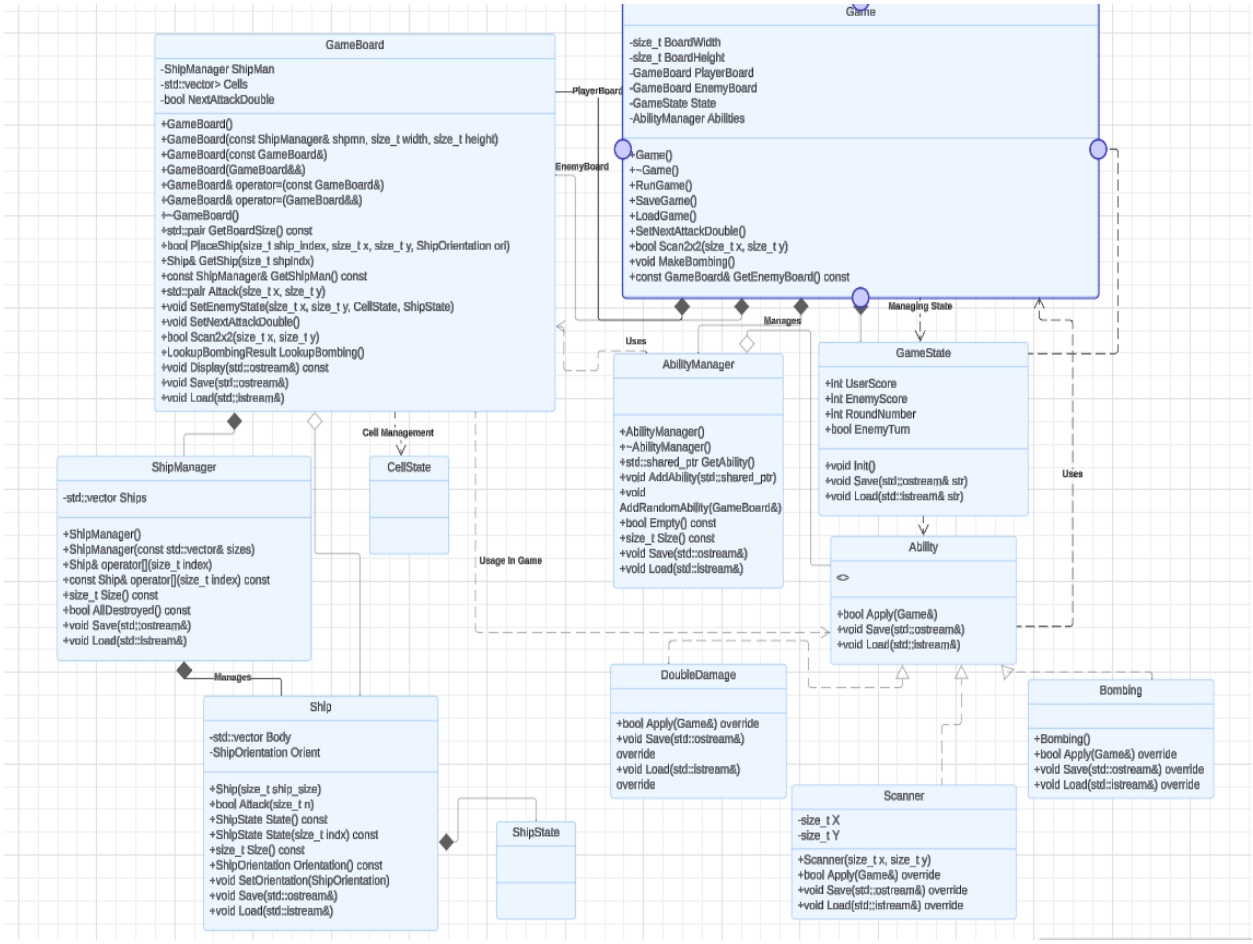
Scan2x2(size_t x, size_t y).Использует "Сканер" для проверки 2x2 области на наличие кораблей.

MakeBombing().Находит целевой корабль для бомбардировки.

Выполняет атаку по случайной ячейке корабля, не уничтоженной ранее.

ApplyAbility(Ability& ab). Применяет переданную способность (ab.Apply(*this)).

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 1

Файл Player.cpp:

```
#include "Player.h"
#include "GameBoard.h"
#include "ShipManager.h"
#include "Ship.h"
#include "AbilityManager.h"
#include "Exceptions.h"
#include <iostream>
#include <random>
#include <fstream>


void GameState::Init()
{
    UserScore = EnemyScore = RoundNumber=0;
    EnemyTurn = false;
}


void GameState::Save(std::ostream& str)
{
    str << UserScore << " " << EnemyScore << " " << RoundNumber << std::endl;
}


void GameState::Load(std::istream& str)
{
    str >> UserScore >> EnemyScore >> RoundNumber;
    EnemyTurn = false;
}
```

```
Game::Game()
{

}

Game::~Game()
{
}

void Game::InitializeGame()
{
    InitializeEnemyBoard();
    InitializePlayerBoard();
    InitializeAbilities();
    State.Init();
}

void Game::InitializeRound()
{
    InitializeEnemyBoard();
}

GameBoard Game::GenerateRandomBoard()
{
    ShipManager sm({ 4,3,3,2,2,2,1,1,1,1 });
    GameBoard b(sm, BoardWidth, BoardHeight);
    for (int i = 0; i < 10; i++) {
        bool newship;
        do {
```

```
            auto ori = std::rand() & 1 ? ShipOrientation::Horisontal :
ShipOrientation::Vertical;
            auto s = sm[i].Size();   //ship size
            unsigned x, y;
            if (ori == ShipOrientation::Horisontal) {
                x = std::rand() % (BoardWidth - 1 - s);
                y = std::rand() % (BoardHeight - 1);
            }
            else {
                x = std::rand() % (BoardWidth - 1);
                y = std::rand() % (BoardHeight - 1 - s);
            }
            newship = false;
            try {
                newship = b.PlaceShip(i, x, y, ori);
            }
            catch (...) {
            }
        } while (!newship);
    }
    return b;
}

void Game::InitializePlayerBoard()
{
    std::cout << "Do you want to generate random board?(y,n)";
    char ans;
    std::cin >> ans;
    if (ans == 'y') {
        PlayerBoard = GenerateRandomBoard();
```

```
        return;
    }
    ShipManager sm({ 4,3,3,2,2,2,1,1,1,1 });
    GameBoard b(sm, BoardWidth, BoardHeight);
    for (int i = 0; i < 10; i++) {
        bool newship;
        do {
         //
            auto s = sm[i].Size();   //ship size
            unsigned x, y;
            char o;
            std::cout << b<<std::endl<<s<<"-segment ship; enter X Y O(h or
v): ";
            std::cin >> x >> y >> o;
            auto ori = o=='h' ? ShipOrientation::Horisontal :
ShipOrientation::Vertical;
            newship = false;
            try {
                newship = b.PlaceShip(i, x, y, ori);
            }
            catch (...) {
            }
        } while (!newship);
    }
    PlayerBoard = b;
}

void Game::InitializeEnemyBoard()
{
    EnemyBoard =GenerateRandomBoard();
```

```
}

void Game::InitializeAbilities()
{
    unsigned x = std::rand() % (BoardWidth - 1 );
    unsigned y = std::rand() % (BoardHeight - 1);
    switch (std::rand() % 6) {
        case 0:
            Abilities.AddAbility(std::make_shared<DoubleDamage>());
            Abilities.AddAbility(std::make_shared<Bombing>());
            Abilities.AddAbility(std::make_shared<Scanner>(x,y));
            break;
        case 1:
            Abilities.AddAbility(std::make_shared<DoubleDamage>());
            Abilities.AddAbility(std::make_shared<Scanner>(x, y));
            Abilities.AddAbility(std::make_shared<Bombing>());
            break;
        case 2:
            Abilities.AddAbility(std::make_shared<Bombing>());
            Abilities.AddAbility(std::make_shared<Scanner>(x, y));
            Abilities.AddAbility(std::make_shared<DoubleDamage>());
            break;
        case 3:
            Abilities.AddAbility(std::make_shared<Bombing>());
            Abilities.AddAbility(std::make_shared<DoubleDamage>());
            Abilities.AddAbility(std::make_shared<Scanner>(x, y));
            break;
        case 4:
            Abilities.AddAbility(std::make_shared<Scanner>(x, y));
```

```
            Abilities.AddAbility(std::make_shared<Bombing>());
            Abilities.AddAbility(std::make_shared<DoubleDamage>());
            break;
        case 5:
            Abilities.AddAbility(std::make_shared<Scanner>(x, y));
            Abilities.AddAbility(std::make_shared<DoubleDamage>());
            Abilities.AddAbility(std::make_shared<Bombing>());
            break;
    }
}

RoundResult Game::Round()
{
    State.RoundNumber += 1;
    for (;;) {
        std::cout << std::endl;
        std::cout << PlayerBoard;
        std::cout << "Round: " << State.RoundNumber << std::endl;
        std::cout << "Your Score: " << State.UserScore << " Enemy score: " <<
State.EnemyScore << std::endl;
        std::cout << "Abilities: " << Abilities.Size() << std::endl;
        std::cin.clear();
        char cmd;
        std::cout << "Enter cmd (Q,T,L,S): ";
        std::cin >> cmd;
        std::cout << std::endl;
        if (cmd == 'Q')
            return RoundResult::Quit;
        if (cmd == 'L'){
            LoadGame();
```

```
        continue;
    }
    if (cmd == 'S') {
        SaveGame();
        continue;
    }
    if (cmd != 'T') {
        continue;
    }
    std::cin.clear();
    int x = -1, y = -1;
    char ability='n';
    if(!Abilities.Empty()){
        std::cout << "Enter x y ability(y or n): ";
        std::cin >> x >> y >> ability;
        std::cout << std::endl;
    }
    else {
        std::cout << "Enter x y: ";
        std::cin >> x >> y ;
        std::cout << std::endl;
    }
    if (x < 0 || y < 0)
        continue;
    UserTurn(x, y, ability == 'y');
    if (EnemyBoard.GetShipMan().AllDestroyed()) {
        std::cout << EnemyBoard << std::endl;
        std::cout << "Round is over. You won!"<<std::endl;
        return RoundResult::RoundOver;
```

```
        }
        EnemyTurn();
        if (PlayerBoard.GetShipMan().AllDestroyed()) {
            std::cout << PlayerBoard << std::endl;
            std::cout << "Game is over. You were defeated!" << std::endl;
            break;
        }
    }

    return RoundResult::GameOver;
}

void Game::RunGame()
{
    for(;;){
        std::cout << "Initialize GAME" << std::endl;
        InitializeGame();
        for(;;){
            auto res= Round();
            if (res == RoundResult::Quit) {
                std::cout << "Quit" << std::endl;
                return;
            }
            if (res == RoundResult::GameOver)
                break;
            if (res == RoundResult::RoundOver) {
                std::cout << "Initialize ROUND" << std::endl;
                InitializeRound();
            }
```

```
        }
    }
}

bool Game::UserTurn(size_t x, size_t y, bool use_ability)
{
    // check state
    if (State.EnemyTurn)
        throw InternalError();
    State.EnemyTurn = true;
    // use ability
    if (use_ability) {
        auto ability = Abilities.GetAbility();
        if(ApplyAbility(*ability))
            std::cout << "scanner found ship" << std::endl;
    }

    //perform attack
   auto [cs,ss] = EnemyBoard.Attack(x, y);
   PlayerBoard.SetEnemyState(x, y, cs, ss);
    //return result
    if (cs != CellState::Occupied)
        return false;
    State.UserScore += 1;
    if (ss == ShipState::Destroyed)
        Abilities.AddRandomAbility(PlayerBoard);
    return true;
}
```

```
void Game::SetNextAttackDouble()
{
    std::cout << "used double damage" << std::endl;
    EnemyBoard.SetNextAttackDouble();
}

bool Game::Scan2x2(size_t x, size_t y)
{
    std::cout << "used scanner: x=" << x << " y=" << y << std::endl;
    return EnemyBoard.Scan2x2(x, y);
}

void Game::MakeBombing()
{
    try {
        auto [ship_index, x, y] = EnemyBoard.LookupBombing();
        const Ship& ship = EnemyBoard.GetShip(ship_index);
        for (int j = 0; j < 10; ++j) {
            unsigned m = std::rand() % ship.Size();
            if (ship.Orientation() == ShipOrientation::Horisontal)
                x += m;
            else
                y += m;
            if (ship.State(m) != ShipState::Destroyed)
                break;
        }
        std::cout << "used bombing: " << x << " " << y << std::endl;
        auto [cs, ss] = EnemyBoard.Attack(x, y);
        PlayerBoard.SetEnemyState(x, y, cs, ss);
```

```
        State.UserScore += 1;
    }
    catch (...) {
        std::cout << "used bombing: error" << std::endl;
    }
}

bool Game::EnemyTurn()
{
    // check state
    if (!State.EnemyTurn)
        throw InternalError();
    State.EnemyTurn = false;
    auto [w, h] = PlayerBoard.GetBoardSize();
    //generate random x and y for attack
    unsigned x = std::rand() % (w - 1);
    unsigned y = std::rand() % (h - 1);
    //perform attack
    auto [cs, ss] = PlayerBoard.Attack(x, y);
    EnemyBoard.SetEnemyState(x, y, cs, ss);
    //return result
    if (cs != CellState::Occupied)
        return false;
    State.EnemyScore += 1;
    return true;
}

bool Game::ApplyAbility(Ability& ab)
{
```

```
    return ab.Apply(*this);
}

void Game::SaveGame()
{
    std::ofstream of("saved_game", std::ios_base::trunc);
    of << BoardWidth << " " << BoardHeight<<std::endl;
    PlayerBoard.Save(of);
    EnemyBoard.Save(of);
    State.Save(of);
    Abilities.Save(of);
}

void Game::LoadGame()
{
    std::ifstream f("saved_game");
    f>> BoardWidth>>BoardHeight;
    PlayerBoard.Load(f);
    EnemyBoard.Load(f);
    State.Load(f);
    Abilities.Load(f);
}
```

Файл Player.h:

```
#pragma once
#include "GameBoard.h"
#include "ShipManager.h"
#include "Ship.h"
#include "AbilityManager.h"
#include <ostream>
```

```
#include <istream>

struct GameState {
    int UserScore;
    int EnemyScore;
    int RoundNumber;
    bool EnemyTurn;
    //    bool AbilityFlag;
    //int AbilityCount;
    void Init();
    void Save(std::ostream& str);
    void Load(std::istream& str);
};

enum class RoundResult{Quit,GameOver,RoundOver};
class Game {
private:
    size_t BoardWidth = 10;
    size_t BoardHeight = 10;
    GameBoard   PlayerBoard;
    GameBoard   EnemyBoard;
    GameState State;
    AbilityManager Abilities;

private:
    void InitializeGame();
    void InitializeRound();
    GameBoard GenerateRandomBoard();
    void InitializePlayerBoard();
    void InitializeEnemyBoard();
    void InitializeAbilities();
    //раунд
    RoundResult Round();
    bool UserTurn(size_t x, size_t y, bool use_ability);
    bool EnemyTurn();
    bool ApplyAbility(Ability&);
```

```
public:
    Game();
    ~Game();
    //вся игра
    void RunGame();
    //сохранение
    void SaveGame();
    //загрузка
    void LoadGame();
    //для использования ability
    void SetNextAttackDouble();
    bool Scan2x2(size_t x, size_t y);
    void MakeBombing();

public:  // для тестов
    const GameBoard& GetEnemyBoard()const { return EnemyBoard; }

};
```