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. 3385 _____ Савранский Д.С.

Преподаватель _____ Жангиров Т. Р.

Санкт-Петербург
2024

Цель работы

Целью работы является создание классов с реализацией основного игрового цикла, включающего чередование ходов игрока и компьютерного врага, а также возможность сохранения и загрузки состояния игры.

Задание

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

i) Начало игры

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

iii) В случае проигрыша пользователь начинает новую игру

iv) 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.

b) 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

Класс GameState состояния игры хранит информацию об игре и имеет публичные поля:

int roundCount - хранящий номер текущего раунда

str::unique_ptr<GameField> playerField, enemyField - отвечающие за хранение игрового поля игрока и противника.

str::unique_ptr<ShipManager> playerShips, enemyShips - отвечающие за хранение менеджера кораблей игрока и противника.

str::unique_ptr<SkillManager> skillManager - отвечающие за хранение менеджера способностей игрока.

В классе переопределить операторы ввода и вывода в поток. Реализация сохранения и загрузки состояния игры основана на записи данных в поток. Сначала записывается количество раундов, затем размеры поля, двумерный массив состояния видимости клеток, количество кораблей и их параметры (длина, координаты, состояние сегментов, ориентация), а также информация о способностях игрока. Аналогичная структура используется для данных вражеского поля и кораблей.

Загрузка выполняется путем последовательного считывания данных из потока, в ходе этого происходит инициализация полей и объектов GameField, ShipManager и SkillManager, через создание и заполнение данных из потока, во временном объекте newGameState. Если во время записи, были получены все корректные данные, то данные перекидываются с временного на текущий GameState.

Класс Game хранит в себе приватное поле gameState, хранящий полную информацию о текущей игре и который он модифицирует, осуществляя тем самым игровой цикл.

Публичный метод gameStart используется для изначальной инициализации игры, он использует приватные методы playerInitialize и enemyInitialize, в которых инициализируются поля, корабли и способности, а

также идет расстановка самих кораблей. Игроку предлагается расположить свои корабли на поле.

Публичный метод startGameLoop начинает игровой цикл, в нем чередуются ходы игрока и противника. В ход игрока предлагается сделать одно из 4 действий:

0 - атаковать клетку, игрок вводит координаты для атаки клетки и вызывается приватный метод playerAttack, реализующий атаку с помощью метода игрового поля.

1 - использовать способность, используется первая в очередь способность с помощью метода менеджера способностей, продолжает ход игрока дальше.

2 - сохранение, текущее состояние игры потоком записывается в файл, который укажет игрок.

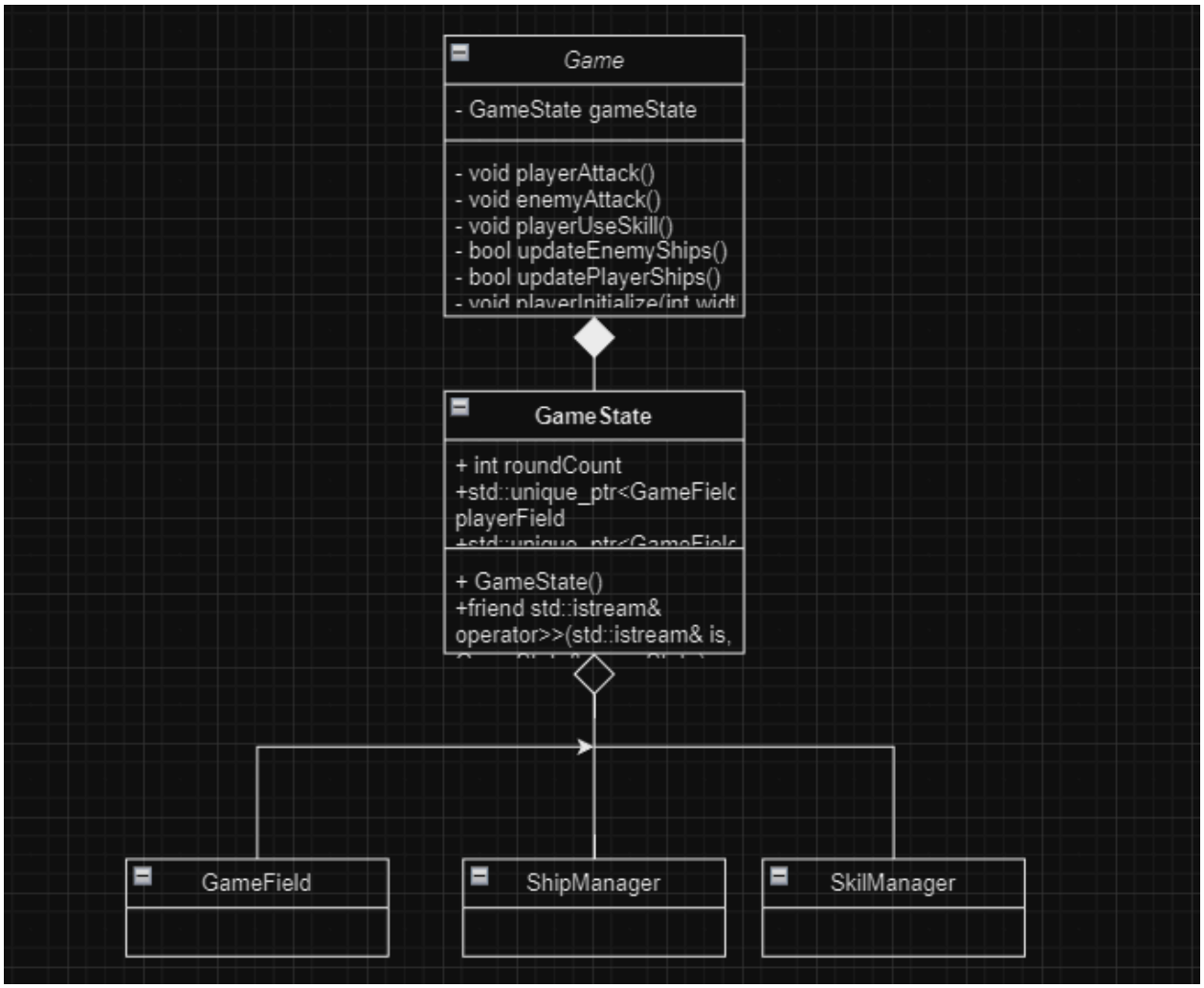
3 - загрузка, текущее состояние игры перезапишется из потока файла, указанным игроком.

4 - выход, прекращает выполнение gameLoop.

После того, как игрок атаковал, ход передается врагу. Используется приватный метод enemyAttack.

После каждой атаки или способности игрока, вызывается приватный метод checkEnemyShips, который, с помощью методов ShipManager, проверяет не были ли затоплены вражеские корабли, с момента прошлого вызова метода, если да, то игроку выдается случайная способность, если были потоплены все корабли, то метод возвращает значение true, и в GameLoop вызывается приватный метод, playerWin, который сообщает о победе, увеличивает кол-во раундов и вызывает метод softReset, который инициализирует нового противника.

После атаки противника также вызывается метод checkPlayerShips, который возвращает true, если все корабли игрока были затоплены, в таком случае выводится сообщение о поражении и вызывается метод gameStart и игры обнуляется.


Game
- GameState gameState
- void playerAttack()
- void enemyAttack()
- void playerUseSkill()
- bool updateEnemyShips()
- bool updatePlayerShips()
GameState
+ int roundCount
playerField
+ GameState()
operator>>(std::istream& is,
GameField
ShipManager
SkilManager

Выводы

В ходе работы были созданы классы с реализованным игровым циклом, обеспечивающим чередование ходов игрока и компьютерного врага. Реализована возможность сохранения и загрузки состояния игры, что позволяет пользователю продолжить игру с момента последнего сохранения.

ПРИЛОЖЕНИЕ А

ИСХОДНЫЙ КОД ПРОГРАММЫ

Barrage.h:

```
#pragma once
#include "GameField.h"
#include "Skill.h"

class Barrage : public Skill {
public:
    void use(GameField& field) override;
    std::string getName() override;
};
```

Barrage.cpp:

```
#include "Barrage.h"
#include <random>
#include <iostream>

void Barrage::use(GameField& field) {
                  std::vector<std::pair<int,    int>>    occupied    =
field.getOccupiedCells();
    if (occupied.size() == 0) {
        std::cout << "No ships left" << std::endl;
        return;
    }
    std::random_device rd;
    std::mt19937 gen(rd());
    std::uniform_int_distribution<> dis(0, occupied.size() - 1);

    int index = dis(gen);

                  field.attackCellWithoutVision(occupied[index].first,
occupied[index].second);
    std::cout << "Barrage used" << std::endl;
}

std::string Barrage::getName() {
    return "Barrage";
}
```

CustomExceptions.h:

```
#pragma once
#include <stdexcept>

class NoSkillAvailable : public std::runtime_error {
public:
         explicit  NoSkillAvailable(const  std::string&  message)  :
std::runtime_error(message) {}
};

class InvalidShipPlacement : public std::runtime_error {
public:
```

```
        explicit InvalidShipPlacement(const std::string& message) :
std::runtime_error(message) {}
};

class OutOfBounds : public std::runtime_error {
public:
            explicit OutOfBounds(const std::string& message) :
std::runtime_error(message) {}
};
```

DoubleDamage.h:

```
#pragma once
#include <iostream>
#include "GameField.h"
#include "Skill.h"

class DoubleDamage : public Skill {
public:
    void use(GameField& field) override;
    std::string getName() override;
};
```

DoubleDamage.cpp:

```
#include <iostream>
#include "DoubleDamage.h"

void DoubleDamage::use(GameField& field) {
    field.activateDoubleDamage();
    std::cout << "Double damage activated" << std::endl;
}

std::string DoubleDamage::getName() {
    return "Double Damage";
}
```

GameField.h:

```
#pragma once
#include <vector>
#include "Ship.h"

enum class CellState {unknown, empty, ship};

struct Cell {
    CellState cellState;
    SegmentState* segmentState;
};

class GameField {
    private:
        int width;
        int height;
        std::vector<std::vector<Cell>> cells;
        bool doubleDamage = false;
    public:
        GameField(int width, int height);
```

```
        GameField(const GameField& other);
        GameField& operator=(const GameField& other);
        GameField(GameField&& other);
        GameField& operator=(GameField&& other);

              void placeShip(Ship& ship, int x, int y, Direction
direction);
        void attackCellWithoutVision(int x, int y);
        void attackCell(int x, int y);
        std::vector<std::pair<int, int>> getOccupiedCells();
        void activateDoubleDamage();
        bool isCellEmpty(int x, int y);


        void printField();
};
```

GameField.cpp:

```
#include "GameField.h"
#include "CustomExceptions.h"
#include <random>
#include <algorithm>
#include <iostream>

GameField::GameField(int width, int height) {
    this->width = width;
    this->height = height;
             this->cells   =   std::vector<std::vector<Cell>>(height,
std::vector<Cell>(width, Cell{CellState::unknown, nullptr}));
}

GameField::GameField(const GameField& other) {
    this->width = other.width;
    this->height = other.height;
    this->cells = other.cells;
}

GameField& GameField::operator=(const GameField& other) {
    if (this != &other) {
        this->width = other.width;
        this->height = other.height;
        this->cells = other.cells;
    }
    return *this;
}

GameField::GameField(GameField&& other) {
    this->width = other.width;
    this->height = other.height;
    this->cells = std::move(other.cells);
}

GameField& GameField::operator=(GameField&& other) {
    if (this != &other) {
        this->width = other.width;
        this->height = other.height;
```

```
        this->cells = std::move(other.cells);
    }
    return *this;
}

void GameField::placeShip(Ship& ship, int x, int y, Direction
direction) {
    int length = ship.getLength();

    int dx = direction == Direction::horizontal ? 1 : 0;
    int dy = direction == Direction::vertical ? 1 : 0;

    if (x < 0 || x + length * dx >= width || y < 0 || y + length * dy
>= height) {
        throw InvalidShipPlacement("Invalid ship placement");
    }

    int mn_x = std::max(x - 1, 0);
    int mx_x = std::min(x + length * dx + 1, width);
    int mn_y = std::max(y - 1, 0);
    int mx_y = std::min(y + length * dy + 1, height);

    for (int i = mn_y; i < mx_y; i++) {
        for (int j = mn_x; j < mx_x; j++) {
            if (cells[i][j].segmentState != nullptr) {
                throw InvalidShipPlacement("Invalid ship placement");
            }
        }
    }


    for (int i = 0; i < length; i++) {
                cells[y + i * dy][x + i * dx].segmentState =
ship.getSegmentState(i);
    }
}

void GameField::attackCellWithoutVision(int x, int y) {
    if (x < 0 || x >= width || y < 0 || y >= height) {
        throw OutOfBounds("Invalid cell coordinates");
    }
    if (cells[y][x].segmentState == nullptr) {
        return;
    }

    SegmentState* segmentState = cells[y][x].segmentState;

    if (doubleDamage) {
        *segmentState = SegmentState::destroyed;
        doubleDamage = false;
        return;
    }
    switch (*segmentState)
    {
        case SegmentState::intact:
            *segmentState = SegmentState::damaged;
            break;
```

```
        case SegmentState::damaged:
            *segmentState = SegmentState::destroyed;
            break;
        default:
            break;
    }
}
void GameField::attackCell(int x, int y) {

    attackCellWithoutVision(x, y);

    if (cells[y][x].segmentState == nullptr) {
        cells[y][x].cellState = CellState::empty;
    }
    else {
        cells[y][x].cellState = CellState::ship;
    }
}

std::vector<std::pair<int, int>> GameField::getOccupiedCells() {
    std::vector<std::pair<int, int>> occupiedSectors;
    for (int x = 0; x < width; ++x) {
        for (int y = 0; y < height; ++y) {
              if (!isCellEmpty(x, y) && *cells[y][x].segmentState !=
SegmentState::destroyed) {
                occupiedSectors.emplace_back(x, y);
            }
        }
    }

    return occupiedSectors;
}

void GameField::activateDoubleDamage() {
    doubleDamage = true;
}

bool GameField::isCellEmpty(int x, int y) {
    if (x < 0 || x >= width || y < 0 || y >= height) {
        throw OutOfBounds("Invalid cell coordinates");
    }
    return cells[y][x].segmentState == nullptr;
}

void GameField::printField() {
    for (int y = 0; y < height; y++) {
        for (int x = 0; x < width; x++) {
            if (cells[y][x].segmentState == nullptr) {
                std::cout << ".";
                continue;
            }
            switch (*cells[y][x].segmentState) {
                case SegmentState::intact:
                    std::cout << "2";
                    break;
                case SegmentState::damaged:
                    std::cout << "1";
```

```
                    break;
                case SegmentState::destroyed:
                    std::cout << "0";
                    break;
                default:
                    break;
            }
        }
        std::cout << std::endl;
    }
}
```

Scanner.cpp:

```
#pragma once
#include "GameField.h"
#include "Skill.h"

class Scanner : public Skill {
public:
    void use(GameField& field) override;
    std::string getName() override;
};
```

Scanner.h:

```
#include "Scanner.h"
#include <iostream>
#include <limits>

void Scanner::use(GameField& field) {
    int x, y;
    while (true) {
        std::cout << "Enter coordinates for scanner (x, y): ";
        std::cin >> x >> y;
        if (std::cin.fail()) {
            std::cin.clear();

std::cin.ignore(std::numeric_limits<std::streamsize>::max(), '\n');
                std::cout << "Invalid input. Please try again." <<
std::endl;
            continue;
        }
        break;
    }

    for (int i = 0; i <= 1; i++) {
        for (int j = 0; j <= 1; j++) {
            if (!field.isCellEmpty(x + j, y + i)) {
                    std::cout << "Scanner has detected a ship" <<
std::endl;
                return;
            }
        }
    }
    std::cout << "Scanner has detected no ships" << std::endl;
}
```

```
std::string Scanner::getName() {
    return "Scanner";
}
```

Ship.h:

```
#pragma once

enum class SegmentState { intact = 2, damaged = 1, destroyed = 0 };
enum class Direction { horizontal, vertical };
class Ship {
    private:
        int length;
        SegmentState segments[4];
    public:
        Ship(int length);
        bool isDestroyed();
        int getLength();
        SegmentState* getSegmentState(int index);
};
```

Ship.cpp:

```
#include "Ship.h"
#include <stdexcept>
Ship::Ship(int length) {
    if (length < 1 || length > 4) {
          throw std::invalid_argument("Ship length must be between 1
and 4");
    }
    this->length = length;
    for (int i = 0; i < length; i++) {
        segments[i] = SegmentState::intact;
    }
}

bool Ship::isDestroyed() {
    for (SegmentState state : segments) {
        if (state != SegmentState::destroyed) {
            return false;
        }
    }
    return true;
}

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

SegmentState* Ship::getSegmentState(int index) {
    if (index < 0 || index >= length) {
        throw std::invalid_argument("Invalid segment index");
    }
    return &segments[index];
}
```

ShipManager.h:

```
#pragma once
#include <vector>
#include "Ship.h"

class ShipManager {
private:
    std::vector<Ship> ships;
public:
    ShipManager(std::vector<int> &lengths);
    Ship* getShip(int index);
    int getShipsCount();
    void addShip(int length);
    void printShips();
};
```

ShipManager.cpp:

```
#include "ShipManager.h"
#include "CustomExceptions.h"
#include <stdexcept>
#include <iostream>

ShipManager::ShipManager(std::vector<int> &lenghts) {
    for (int length : lenghts) {
        addShip(length);
    }
}

void ShipManager::addShip(int length) {
    ships.push_back(Ship(length));
}

Ship* ShipManager::getShip(int index) {
    if (index < 0 || index >= ships.size()) {
        throw std::invalid_argument("Invalid ship index");
    }
    return &ships[index];
}

int ShipManager::getShipsCount() {
    return ships.size();
}

void ShipManager::printShips() {
    for (int i = 0; i < ships.size(); i++) {
                std::cout << "Ship: " << i << " length: " <<
ships[i].getLength() << " State:";
        for (int j = 0; j < ships[i].getLength(); j++) {
            std::cout << " " << (int)*ships[i].getSegmentState(j);
        }
        std::cout << std::endl;
    }
}
```

Skill.h:

```
#pragma once
#include "GameField.h"
```

```
#include <string>

class Skill {
public:
    virtual void use(GameField& field) = 0;
    virtual std::string getName() = 0;
};
```

SkillManager.h:

```
#pragma once
#include "Skill.h"
#include <random>
#include <vector>
#include <memory>

class SkillManager {
    private:
        std::mt19937 gen;
        std::uniform_int_distribution<> dis;
        std::vector<std::shared_ptr<Skill>> skills;
        std::vector<std::shared_ptr<Skill>> skillsCopy;

        void initializeSkills();
    public:
        SkillManager();
        void addRandomSkill();
        void printSkills();
        void useSkill(GameField& field);
};
```

SkillManager.cpp:

```
#include <random>
#include <vector>
#include <algorithm>
#include <iostream>
#include "CustomExceptions.h"
#include "SkillManager.h"
#include "Barrage.h"
#include "DoubleDamage.h"
#include "Scanner.h"

void SkillManager::initializeSkills() {
    skillsCopy.push_back(std::make_shared<Barrage>());
    skillsCopy.push_back(std::make_shared<DoubleDamage>());
    skillsCopy.push_back(std::make_shared<Scanner>());
}

SkillManager::SkillManager() {
    initializeSkills();

    skills = skillsCopy;

    std::random_device rd;
    gen = std::mt19937(rd());
    dis = std::uniform_int_distribution<>(0,  skillsCopy.size() - 1);
    std::shuffle(skills.begin(), skills.end(), gen);
```

```
}

void SkillManager::addRandomSkill() {
    int index = dis(gen);
    skills.push_back(skillsCopy[index]);
}

void SkillManager::useSkill(GameField& field) {
    if (skills.size() == 0) {
        throw NoSkillAvailable("No skills available");
    }
    skills[0]->use(field);
    skills.erase(skills.begin(), skills.begin() + 1);
}

void SkillManager::printSkills() {
    if (skills.size() == 0) {
        std::cout << "No skills available" << std::endl;
        return;
    }
    std::cout << skills[0]->getName();
    for (int i = 1; i < skills.size(); i++) {
        std::cout << ", " << skills[i]->getName();
    }
    std::cout << std::endl;
}
```