**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. 3381 |  | Щеглов М.А. |
| Преподаватель |  | Жангиров Т.Р. |

Санкт-Петербург

2024

## Цель работы.

Необходимо разработать систему управления игровым процессом, включающую реализацию класса игрового цикла, обеспечивающего чередование ходов игрока и компьютера, переход между раундам. Реализовать сохранение и загрузку состояния игры через систему сериализации с использованием класса GameState, позволяя игроку продолжить игру после перезапуска программы. Обеспечить обработку исключений для ошибок при сохранении/загрузке.

## Задание.

## Лабораторная работа №3 - Связывание классов

1. Создать класс игры, который реализует следующий игровой цикл:
   1. Начало игры
   2. Раунд, в котором чередуются ходы пользователя и компьютерного врага. В свой ход пользователь может применить способность и выполняет атаку. Компьютерный враг только наносит атаку.
   3. В случае проигрыша пользователь начинает новую игру
   4. В случае победы в раунде, начинается следующий раунд, причем состояние поля и способностей пользователя переносятся.

Класс игры должен содержать методы управления игрой, начало новой игры, выполнить ход, и т.д., чтобы в следующей лаб. работе можно было выполнять управление исходя из ввода игрока.

1. Реализовать класс состояния игры, и переопределить операторы ввода и вывода в поток для состояния игры. Реализовать сохранение и загрузку игры. Сохраняться и загружаться можно в любой момент, когда у пользователя приоритет в игре. Должна быть возможность загружать сохранение после перезапуска всей программы.

**Примечание:**

* Класс игры может знать о игровых сущностях, но не наоборот
* Игровые сущности не должны сами порождать объекты состояния
* Для управления самое игрой можно использовать обертки над командами
* При работе с файлом используйте идиому RAII.

**Выполнение работы**

**Описание класса GameLoop.**

GameLoop реализует основной игровой цикл, обеспечивая управление игровым процессом. Он включает в себя логику сохранения/загрузки игры, чередование ходов игрока и компьютера, а также переходы между раундами.

**Методы класса**

**GameLoop()**

* Конструктор класса.
* Устанавливает начальные значения раунда и очередности ходов.
* Вызывает initializeGame() для создания начального состояния игры.

**void initializeGame()**

* Отвечает за создание начальных игровых сущностей: поля размером 10x10. Менеджеров кораблей для игрока и компьютера. Менеджера способностей.
* Выводит в консоль сообщение о завершении инициализации и промежуточных этапов для отладки.

**void startGame()**

* Запускает игровой процесс, который состоит из раундов.
* Цикл игры продолжает выполняться до тех пор, пока игрок не проиграет или не завершится последний раунд.
* Проверяет условия завершения игры: все корабли игрока уничтожены — игра заканчивается. Все корабли компьютера уничтожены — начинается новый раунд.

**void playRound()**

Отвечает за выполнение текущего раунда:

* Чередует ходы игрока и компьютера.
* Проверяет условия завершения раунда: если у компьютера нет кораблей, игрок выигрывает раунд. Если у игрока нет кораблей, игрок проигрывает.

**void playPlayerTurn()**

* Реализует логику хода игрока: предлагает сохранить игру. Запрашивает координаты для атаки. Выполняет атаку и отображает результат. Если у игрока есть доступные способности, предлагает их использовать.
* Обрабатывает исключения для корректной работы с пользовательским вводом.

**void playEnemyTurn()**

Реализует ход компьютера:

* Генерирует случайные координаты атаки, пока не будет выбрана пустая клетка.
* Атакует игрока по сгенерированным координатам и выводит результат атаки.

**void resetComputer()**

Сбрасывает состояние компьютера после завершения раунда:

* Удаляет старое поле, менеджеры кораблей и способностей.
* Создает новое поле, новые корабли и способности.
* Выводит сообщение о завершении обновления состояния.

**void saveGame()**

Сохраняет текущее состояние игры:

* Создает объект GameState с текущими данными (полями, кораблями, способностями, раундом).
* Сериализует данные в файл savegame.dat с помощью метода GameState::saveToFile.
* Выводит сообщение об успешном сохранении.

**void loadGame()**

Загружает сохраненное состояние игры:

* Десериализует данные из файла savegame.dat с помощью метода GameState::loadFromFile.
* Восстанавливает поля, менеджеры кораблей и способностей.
* Обновляет указатели на корабли в полях (updateShipPointers).
* Проверяет целостность данных после загрузки.
* Выводит сообщение об успешной загрузке.

**Общая логика**

1. **Очередность ходов**. Она меняется после каждого хода игрока или компьютера.
2. **Сохранение/загрузка игры.** Состояние игры может быть сохранено в любой момент, когда игрок делает ход. Игра может быть восстановлена после перезапуска программы.
3. **Обработка исключений**. Все взаимодействия с пользователем защищены от некорректного ввода.
4. **Менеджеры сущностей**. Корабли, поля и способности управляются через соответствующие менеджеры.
5. **RAII**. Используются умные указатели (std::unique\_ptr), которые автоматически управляют временем жизни объектов.

**Описание класса GameState.**

Класс GameState представляет состояние игры, включающее в себя информацию о текущем раунде, игровых полях, менеджерах кораблей и способностях. Он обеспечивает сериализацию и десериализацию состояния игры, что позволяет сохранять и восстанавливать игру в любой момент.

**Методы класса**

**Конструкторы.**

1. GameState() - Конструктор по умолчанию. Устанавливает currentRound в 1.
2. GameState(int round, std::unique\_ptr<GameField> playerField, std::unique\_ptr<GameField> enemyField, std::unique\_ptr<ShipManager> playerShips, std::unique\_ptr<ShipManager> enemyShips, std::unique\_ptr<AbilityManager> abilities) - Конструктор, принимающий состояние игры: текущий раунд, игровые поля, менеджеры кораблей и способностей. Использует std::move для передачи владения объектами.

**Методы получения данных**

1. **int getRound() const** - возвращает текущий номер раунда.
2. **const GameField& getPlayerField() const** - возвращает ссылку на поле игрока.
3. **const GameField& getEnemyField() const** - возвращает ссылку на поле компьютера.
4. **const ShipManager& getPlayerShips() const** - возвращает ссылку на менеджер кораблей игрока.
5. **const ShipManager& getEnemyShips() const** - возвращает ссылку на менеджер кораблей компьютера.
6. **const AbilityManager& getAbilities() const** - возвращает ссылку на менеджер способностей.

**Методы сериализации**

1. **void saveToFile(const std::string& filename) const**

* Сохраняет текущее состояние игры в файл.
* Преобразует данные в JSON с помощью toJson() и записывает их в файл.
* Проверяет успешность открытия файла и выводит сообщения об успехе или ошибке.

1. **void loadFromFile(const std::string& filename)**

* Загружает состояние игры из файла.
* Читает данные из файла, десериализует их с помощью fromJson().
* Проверяет успешность открытия файла и выводит сообщения об успехе или ошибке.

**Методы преобразования**

1. **json toJson() const**

* Преобразует текущее состояние игры в объект JSON.
* Сериализует каждое поле: раунд, игровое поле, менеджеры кораблей и способностей.

1. **void fromJson(const json& j)**

* Восстанавливает состояние игры из объекта JSON.
* Создает объекты игровых полей, менеджеров кораблей и способностей.
* Выводит сообщения об успехе или ошибке загрузки.

**Операторы ввода/вывода**

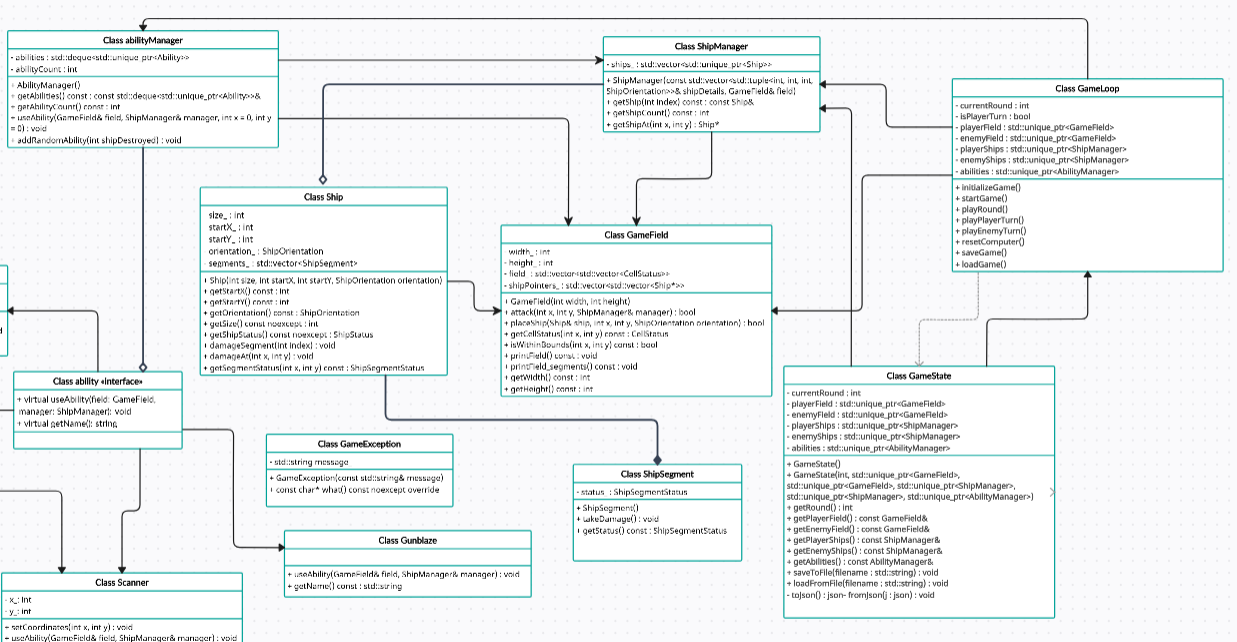
1. **std::ostream& operator<<(std::ostream& os, const GameState& state)**

* Сериализует объект GameState в поток вывода.
* Использует метод toJson() для преобразования в формат JSON.

1. **std::istream& operator>>(std::istream& is, GameState& state)**

* Десериализует объект GameState из потока ввода.
* Использует метод fromJson() для восстановления состояния.

Ниже представлена UML-диаграмма реализованных в данной работе классов:



Разработанный программный код см. в приложении А.

Тестирование программы см. в приложении Б.

## Выводы

В ходе третьей лабораторной работы был реализован игровой цикл для "Морского боя" с чередованием ходов игрока и компьютера, переходами между раундами и завершением игры. Создан класс GameState для сохранения и загрузки состояния игры, включая возможность продолжения после перезапуска программы. Реализована обработка исключений для корректного ввода, сохранения и загрузки.

# Приложение А Исходный код программы

**Название файла: exceptions.h**

#ifndef EXCEPTIONS\_H

#define EXCEPTIONS\_H

#include <exception>

#include <string>

class GameException : public std::exception {

public:

explicit GameException(const std::string& message) : message\_(message) {}

const char\* what() const noexcept override {

return message\_.c\_str();

}

private:

std::string message\_;

};

// СоБаКа

class sobaka : public GameException {

public:

sobaka() : GameException("СОБАКА (dog)") {}

};

// Исключение: попытка разместить корабль вплотную к другому кораблю

class ShipOverlapException : public GameException {

public:

ShipOverlapException() : GameException("Ошибка: попытка разместить корабль вплотную или на пересечении с другим кораблем!") {}

};

// Исключение: попытка атаки за границы поля

class OutOfBoundsAttackException : public GameException {

public:

OutOfBoundsAttackException() : GameException("Ошибка: попытка атаки за пределы поля!") {}

};

// Исключение для некорректного размера корабля

class InvalidShipSizeException : public GameException {

public:

InvalidShipSizeException() : GameException("Размер корабля должен быть от 1 до 4") {}

};

// Исключение для недопустимого индекса сегмента корабля

class InvalidSegmentIndexException : public GameException {

public:

InvalidSegmentIndexException() : GameException("Индекс сегмента корабля за границей") {}

};

// Исключение для недопустимого индекса корабля

class InvalidShipIndexException : public GameException {

public:

InvalidShipIndexException() : GameException("Индекс корабля за границей") {}

};

// Исключение для некорректных размеров игрового поля

class InvalidFieldSizeException : public GameException {

public:

InvalidFieldSizeException() : GameException("Неверные размеры поля, прямоугольные или отрицательные") {}

};

// Исключение для координат, выходящих за границы игрового поля

class CoordinatesOutOfBoundsException : public GameException {

public:

CoordinatesOutOfBoundsException() : GameException("Coordinates are out of bounds!") {}

};

#endif

**Название файла: abilityManager.h**

#ifndef ABILITY\_MANAGER\_H

#define ABILITY\_MANAGER\_H

#include <deque>

#include <algorithm>

#include <memory>

#include <random>

#include "abilities.h"

class AbilityManager {

private:

std::deque<std::unique\_ptr<Ability>> abilities; // Очередь способностей

int abilityCount;

public:

AbilityManager();

// Получение очереди способностей

const std::deque<std::unique\_ptr<Ability>>& getAbilities() const;

// Получение количества способностей

int getAbilityCount() const;

// Применение первой способности из очереди

void useAbility(GameField& field, ShipManager& manager, int x = 0, int y = 0);

// Добавление случайной способности в очередь

void addRandomAbility(int shipDestroyed);

};

#endif // ABILITY\_MANAGER\_H

**Название файла: abilityManager.cpp**

#include "abilityManager.h"

#include "DoubleDamage.h"

#include "Scanner.h"

#include "GunBlaze.h"

#include <iostream>

#include <random>

#include <algorithm>

// Конструктор: создаем случайную очередь способностей

AbilityManager::AbilityManager() {

std::random\_device rd;

std::mt19937 gen(rd());

abilities.push\_back(std::make\_unique<DoubleDamage>());

abilities.push\_back(std::make\_unique<Scanner>());

abilities.push\_back(std::make\_unique<GunBlaze>());

std::shuffle(abilities.begin(), abilities.end(), gen);

abilityCount = abilities.size();

}

// Получение очереди способностей (константная ссылка)

const std::deque<std::unique\_ptr<Ability>>& AbilityManager::getAbilities() const {

return abilities;

}

// Получение количества способностей

int AbilityManager::getAbilityCount() const {

return abilityCount;

}

// Применение первой способности из очереди

void AbilityManager::useAbility(GameField& field, ShipManager& manager, int x, int y) {

if (!abilities.empty()) {

// Проверка, требуется ли установка координат для способности

SetCoordinates\* coordAbility = dynamic\_cast<SetCoordinates\*>(abilities.front().get());

if (coordAbility) {

coordAbility->setCoordinates(x, y); // Установка координат, если это поддерживается

}

abilities.front()->useAbility(field, manager);

abilities.pop\_front();

abilityCount = abilities.size();

} else {

std::cout << "Нет доступных способностей для использования!" << std::endl;

}

}

// Добавление случайной способности в очередь, если корабль уничтожен

void AbilityManager::addRandomAbility(int shipDestroyed) {

if (shipDestroyed == 1) {

std::random\_device rd;

std::mt19937 gen(rd());

std::uniform\_int\_distribution<> dis(0, 2);

int randomNum = dis(gen);

switch (randomNum) {

case 0:

abilities.push\_back(std::make\_unique<DoubleDamage>());

break;

case 1:

abilities.push\_back(std::make\_unique<Scanner>());

break;

case 2:

abilities.push\_back(std::make\_unique<GunBlaze>());

break;

}

std::cout << "Добавлена новая случайная способность!" << std::endl;

abilityCount = abilities.size();

}

}

**Файл abilities.h**

#ifndef ABILITIES\_H

#define ABILITIES\_H

#include <string>

#include "field.h"

#include "shipManager.h"

class Ability {

public:

virtual ~Ability() = default;

virtual void useAbility(GameField& field, ShipManager& manager) = 0;

virtual std::string getName() const = 0;

};

class SetCoordinates : public Ability {

public:

virtual void setCoordinates(int x, int y) = 0;

};

#endif

# Приложение Б Тестирование программы

Название файла: main.cpp

#include "ship.h"

#include "shipManager.h"

#include "field.h"

#include "DoubleDamage.h"

#include "Scanner.h"

#include "GunBlaze.h"

#include "abilityManager.h"

#include <iostream>

#include <tuple>

#include <vector>

#include <memory>

int main() {

// Инициализация параметров кораблей: {size, startX, startY, orientation}

std::vector<std::tuple<int, int, int, ShipOrientation>> shipDetails = {

{4, 1, 8, ShipOrientation::Horizontal},

{1, 1, 1, ShipOrientation::Horizontal},

{3, 7, 3, ShipOrientation::Vertical}

};

GameField field(10, 10);

ShipManager manager(shipDetails, field);

field.printField();

std::cout << std::endl;

// Проводим несколько атак по полю

field.attack(0, 0, manager);

field.attack(1, 0, manager);

field.attack(2, 0, manager);

field.attack(1, 1, manager);

field.attack(1, 1, manager);

field.attack(1, 8, manager);

// Использование способностей напрямую

DoubleDamage doubleDamage;

doubleDamage.setCoordinates(7, 3);

doubleDamage.useAbility(field, manager);

field.printField();

std::cout << std::endl;

Scanner scanner;

scanner.setCoordinates(0, 8);

scanner.useAbility(field, manager);

GunBlaze gunBlaze;

gunBlaze.useAbility(field, manager);

field.printField();

std::cout << std::endl;

// Создаем менеджер способностей

AbilityManager abilityManager;

// Используем первую способность из очереди менеджера способностей

if (!abilityManager.getAbilities().empty()) {

std::cout << "Применяем способность: " << abilityManager.getAbilities().front()->getName() << std::endl;

abilityManager.useAbility(field, manager, 2, 8);

} else {

std::cout << "Нет доступных способностей для использования." << std::endl;

}

//field.attack(2, 8, manager);

field.printField();

std::cout << std::endl;

Ship ship(1, 9, 9, ShipOrientation::Horizontal);

field.attack(9, 9, manager);

field.attack(9, 9, manager);

// Проверяем, уничтожен ли корабль и добавляем случайную способность при уничтожении

// смерть корабля можно проверять многими способами, например

// на каждой итерации игры считать количество кораблей на поле

bool isShipDestroyed = 1;

if (isShipDestroyed) {

std::cout << "Корабль уничтожен! Добавляем новую случайную способность." << std::endl;

abilityManager.addRandomAbility(1);

}

// Вывод количества доступных способностей после добавления

std::cout << "Количество доступных способностей: " << abilityManager.getAbilityCount() << std::endl;

field.printField();

std::cout << std::endl;

return 0;

}

**Файл DoubleDamage.h**

#include "DoubleDamage.h"

void DoubleDamage::setCoordinates(int x, int y) {

x\_ = x;

y\_ = y;

}

void DoubleDamage::useAbility(GameField& field, ShipManager& manager) {

std::cout << "Использована способность: Двойной урон!" << std::endl;

field.attack(x\_, y\_, manager);

field.attack(x\_, y\_, manager);

}

std::string DoubleDamage::getName() const {

return "Double Damage";

}

**Файл GunBlaze.h**

#include "GunBlaze.h"

void GunBlaze::useAbility(GameField& field, ShipManager& manager) {

std::cout << "Использована способность: Обстрел!" << std::endl;

std::random\_device rd;

std::mt19937 gen(rd());

std::uniform\_int\_distribution<> disX(0, field.getWidth() - 1);

std::uniform\_int\_distribution<> disY(0, field.getHeight() - 1);

int attempts = 0;

while (attempts < 100) {

int randomX = disX(gen);

int randomY = disY(gen);

if (field.getCellStatus(randomX, randomY) == CellStatus::Ship) {

field.attack(randomX, randomY, manager);

std::cout << "Урон нанесен по сегменту корабля на координатах (" << randomX << ", " << randomY << ")" << std::endl;

return;

}

attempts++;

}

std::cout << "Не удалось найти корабль для атаки после 100 попыток" << std::endl;

}

std::string GunBlaze::getName() const {

return "Gun Blaze";

}

**Файл Scanner.h**

#include "Scanner.h"

void Scanner::setCoordinates(int x, int y) {

x\_ = x;

y\_ = y;

}

void Scanner::useAbility(GameField& field, ShipManager& manager) {

std::cout << "Использована способность: Сканер!" << std::endl;

for (int dx = 0; dx <= 1; ++dx) {

for (int dy = 0; dy <= 1; ++dy) {

int currentX = x\_ + dx;

int currentY = y\_ + dy;

if (field.isWithinBounds(currentX, currentY) && field.getCellStatus(currentX, currentY) == CellStatus::Ship) {

std::cout << "Корабль найден на клетке (" << currentX << ", " << currentY << ")" << std::endl;

} else {

std::cout << "В клетке (" << currentX << ", " << currentY << ") ничего нет" << std::endl;

}

}

}

}

std::string Scanner::getName() const {

return "Scanner";

}

**Результат работы программы:**

