**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. 3342 |  | Иванов Д. М. |
| Преподаватель |  | Жангиров Т. Р. |

Санкт-Петербург

2024

## Цель работы

Изучить основы RAII. Через эту идиому прописать сохранение и загрузку игры. Также связать все раннее созданные классы в игровой цикл.

## 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.

## Выполнение работы

1) Реализация класса для работы с файлом

Создадим класс, который будет соответствовать работе с файлом по идиоме RAII.

class FileWrapper

std::fstream file; - файл, в котором будет храниться игра. Тип fstream для того, чтобы была возможность и читать данные из файла, и записывать в него значения.

public:

FileWrapper(const std::string& filename, std::ios::openmode mode) — конструктор, прининмающий название файла и его тип взаимодействия с файлом. Если нужно сохранить данные, то передается std::ios::out | std::ios::trunc. Если прочитать — std::ios::in. Проверяется успешность открытия файла.

~FileWrapper(); - деструктор, закрывающий файл

template <typename T>

void write(const T& data); - метод, записывающий некоторые данные в файл через оператор << .

template <typename T>

void read(T& data); - метод, сохраняющий данные из файла в переменную data через оператор >> .

2) Создание класса игрока и компьютера(врага)

В данных классах будут содержаться поля и методы, соответствовующие участнику игры. Для компьютера будут храниться игровое поле, менджер кораблей, набор значений длины для каждого корабля, координаты первых сегментов кораблей на поле и их ориентация.

Playground\* playground;

Manager\_of\_ships\* ships\_manager

std::vector<Length\_of\_the\_ship> length\_of\_ships;

std::vector<Coords> coords\_of\_ships;

std::vector<Orientation> orientations\_of\_ships;

Те же типы полей будут храниться в поле игрока. Тольок помимо них добавим поля для работы со способностями.

Coords coords\_for\_scanner; - координата, куда мы будем записывать точку применения сканнера

Get\_coords\* get\_cor; - команда для работы с координатой сканнера

Ability\_maker\* maker; - мейкер для создания билдеров способностей

Add\_ability\* add\_abil; - команда, добавляемая в игровое поле, для добавления способности при уничтожении корабля

Manager\_of\_abilities\* ability\_manager; - менеджер способностей.

Опишем схожие методы, которые будут иметь участники игры.

void set\_arguments(int height, int width, std::vector<Length\_of\_the\_ship> length\_of\_ships); - метод, получающий из класса игры данные для инициализации полей класса. Для компьютера содержимое некоторых полей определяется рандомным образом через random\_device. Игрок эти значения получает в качетсве дополнительных аргументов метода. Игрок сам вводит эти данные.

void perform\_shoot(Playground& enemy\_playground, Coords coord); - выполнения удара по координате поля соперника

void put\_ships(); - расстановка кораблей на поле. Для врага происходит рандомный выбор координат кораблей и их ориентации. Игрок в свою очередь пользуется данными, введенными в классе игры. Проверяется корабль на правильность его расположения и затем добавляется сначала в менеджер, затем в игровое поле.

Playground& get\_playground(); - получение поля

void serialize(FileWrapper& file) const; - сериализация, процесс сохранения данных в файл при сохранении игры. Запись идет через команду write.

void deserialize(FileWrapper& file); - десериализация, процесс получения данных из файла при загрузке игры. Идет инициализация нужных полей в классе. Получение данных идет через команду read.

Класс врага будет также иметь метод void clear\_ships(); для очистки поля и создания новых кораблей при начале следующего раунда.

Класс игрока в свою очередь будет дополнительно иметь следующий функционал.

void use\_ability(Playground& enemy\_playground); - использование способности из менеджера способностей. Берется билдер, проверяется, нужны ли ему координаты, а затем выполняется способность для поля врага.

3) Создание класса состояния игры

class Game\_state {

private:

std::shared\_ptr<Player> player; - указатель на игрока

std::shared\_ptr<Enemy> enemy; - указатель на врага

int current\_round; - номер текущего раунда

public:

friend FileWrapper& operator<<(FileWrapper& file, const Game\_state& state); - Переопределенный оператор ввода. Идет добавление номера раунда в переданный файл и вызов сериализаций для игрока и врага.

friend FileWrapper& operator>>(FileWrapper& file, Game\_state& state); - Переопределенный оператор ввода. Идет чтение номера раунда из переданного файла и вызов десериализаций для игрока и врага.

std::shared\_ptr<Player> getPlayer(); - получение указателя на игрока для работы с ним в классе игры

std::shared\_ptr<Enemy> getEnemy(); - получение указателя на врага для работы с ним в классеигры

int& get\_current\_round(); - получение ссылки на номер раунда

Также создадим класс сохранения загрузки игры SaveLoad. Которому будем передавать название файла и объект состояния игры. Ну а также напишем функцию вычисления хэша (int count\_hash), для проверки верной загрузки файла.

4) Класс игры

В нем будет содержаться цикл самой игры и работа со всеми созданными раннее файлами.

Также создадим дополнительный класс Painter для отрисовки поля. Для игрока будет видно все его поле со всеми сегментами кораблей и их жизнями. А поле врага будет скрыто.

class Game{

private:

Game\_state\* game\_state; - указатель на состояние игры

Paint painter; - класс отрисовки поля

bool is\_player\_turn; - флаг, определяющий, чей сейчас ход

bool is\_game\_end; - флаг окончания игры

public:

Game();

~Game();

void input\_ships(); - ввод данных от пользователя для отрисовки кораблей на поле и сохранения их в состоянии игры.

void play(); - методы цикла игры. На каждом ходе рисуются поля и идет ожидание ввода пользователя. Либо сохранить игру (вызов save()), либо окончание игры (установка нового значения для флага), либо применения удара. Также возможно при этом применить способность. Идет переход к player\_turn(). Если игрок проигрывает, то игра заканчивается. Если враг — создается новый раунд.

void start\_new\_game(); - метод начала новой игры. Устанавливаются начальные значения. Также предлагается выбор игроку. Либо загрузить игру, либо создать новую с вводом данных (переход к input\_ships()).

void player\_turn(bool is\_need\_ability); - ход игрока. Если он выбрал применить способность, то сначала вызывается она (use\_ability()), а затем удар (perform\_shoot()).

void enemy\_turn(); - ход врага. Берется рандомная координата поля и идет по ней удар в поле игрока.

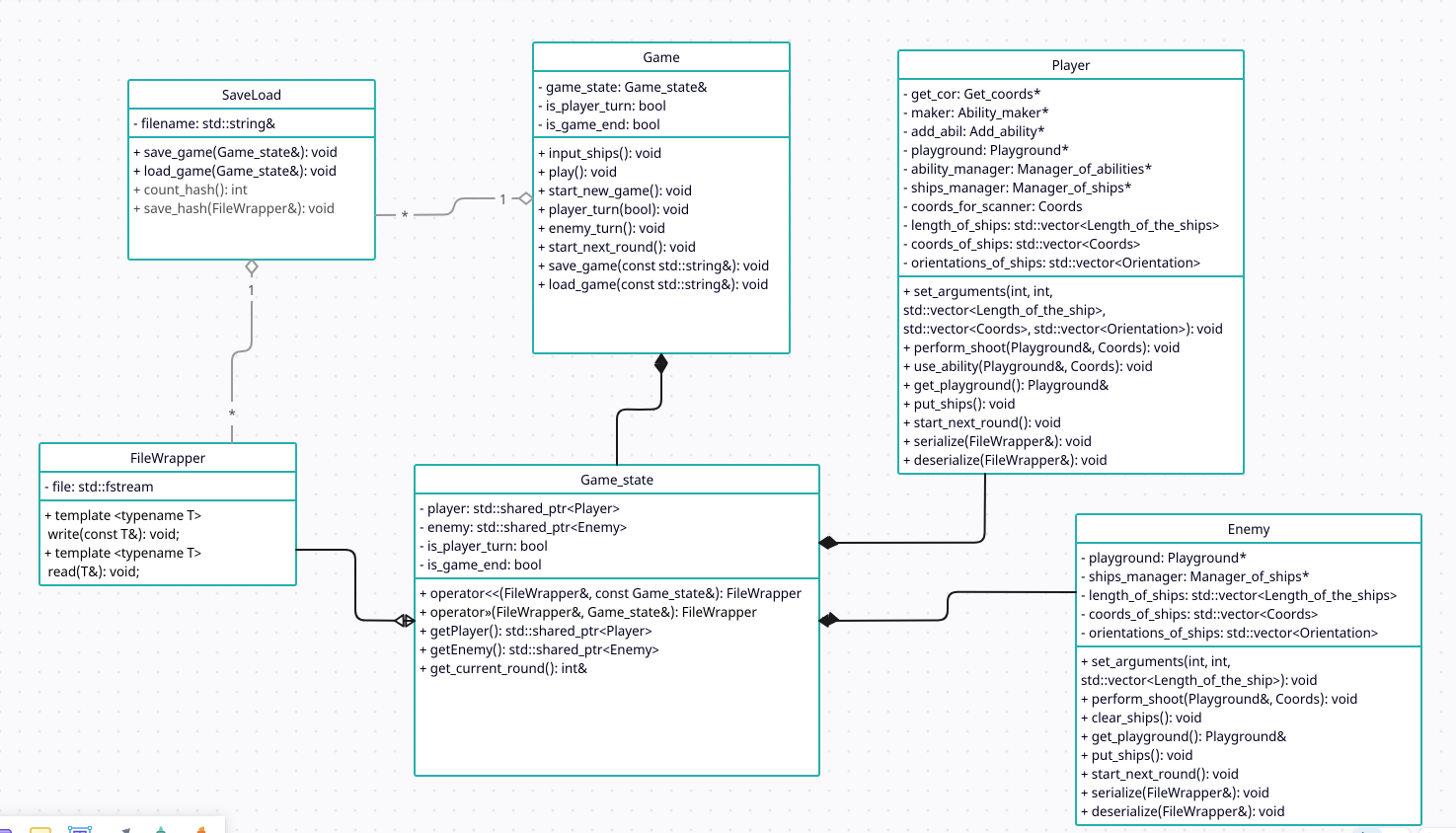
void start\_next\_round(); - переход к следующему раунду. Счетчик раундов увеличивается, очищается поле для врага и заполнение новыми (clear\_ships() и put\_ships())

void save\_game(const std::string& filename); - сохранение игры

void load\_game(const std::string& filename); - загрузка игры. В этих методах вызываются написанные раннее методы для класса состояния игры.

## Представление в UML-диаграмме

Рисунок 1 — UML диаграмма



## Выводы

Была изучена идиома RAII. С ее помощью был реализован полноценный цикл игры с сохранением и загрузкой путем связывания всех раннее созданных классов.

# Приложение А Исходный код программы

Название файла: Game.h

#ifndef GAME\_H

#define GAME\_H

#include <iostream>

#include <ctime>

#include <random>

#include "Playground.h"

#include "Manager\_of\_abilities.h"

#include "Player.h"

#include "Paint.h"

#include "Enemy.h"

#include "Get\_coords.h"

#include "Add\_ability.h"

#include "Game\_state.h"

class Game{

private:

Game\_state\* game\_state;

Paint painter;

bool is\_player\_turn;

bool is\_game\_end;

public:

Game();

~Game();

void input\_ships();

void play();

void start\_new\_game();

void player\_turn(bool is\_need\_ability);

void enemy\_turn();

void start\_next\_round();

void save\_game(const std::string& filename);

void load\_game(const std::string& filename);

};

#endif

Название файла: Game\_state.h

#ifndef GAME\_STATE\_H

#define GAME\_STATE\_H

#include <iostream>

#include <fstream>

#include <memory>

#include <string>

#include "Player.h"

#include "Enemy.h"

#include "FileWrapper.h"

class Game\_state {

private:

std::shared\_ptr<Player> player;

std::shared\_ptr<Enemy> enemy;

int current\_round;

public:

Game\_state();

void save(const std::string& filename) const;

void load(const std::string& filename);

friend FileWrapper& operator<<(FileWrapper& file, const Game\_state& state);

friend FileWrapper& operator>>(FileWrapper& file, Game\_state& state);

std::shared\_ptr<Player> getPlayer();

std::shared\_ptr<Enemy> getEnemy();

int& get\_current\_round();

};

#endif

Название файла: Player.h

#ifndef PLAYER\_H

#define PLAYER\_H

#include <iostream>

#include "Playground.h"

#include "Manager\_of\_abilities.h"

#include "Manager\_of\_ships.h"

#include "Get\_coords.h"

#include "Add\_ability.h"

#include "FileWrapper.h"

class Player{

private:

Get\_coords\* get\_cor;

Ability\_maker\* maker;

Add\_ability\* add\_abil;

Playground\* playground;

Manager\_of\_abilities\* ability\_manager;

Manager\_of\_ships\* ships\_manager;

Coords coords\_for\_scanner;

std::vector<Length\_of\_the\_ship> length\_of\_ships;

std::vector<Coords> coords\_of\_ships;

std::vector<Orientation> orientations\_of\_ships;

public:

Player();

~Player();

void set\_arguments(int height, int width, std::vector<Length\_of\_the\_ship> length\_of\_ships,

std::vector<Coords> coords\_of\_ships, std::vector<Orientation> orientations\_of\_ships);

void perform\_shoot(Playground& enemy\_playground, Coords coord);

void use\_ability(Playground& enemy\_playground);

Playground& get\_playground();

void put\_ships();

void serialize(FileWrapper& file) const;

void deserialize(FileWrapper& file);

};

#endif

Название файла: Enemy.h

#ifndef ENEMY\_H

#define ENEMY\_H

#include <iostream>

#include "Playground.h"

#include "Manager\_of\_abilities.h"

#include "Manager\_of\_ships.h"

#include "Add\_ability.h"

#include "FileWrapper.h"

class Enemy{

private:

Playground\* playground;

Manager\_of\_ships\* ships\_manager;

std::vector<Length\_of\_the\_ship> length\_of\_ships;

std::vector<Coords> coords\_of\_ships;

std::vector<Orientation> orientations\_of\_ships;

public:

Enemy();

~Enemy();

void set\_arguments(int height, int width, std::vector<Length\_of\_the\_ship> length\_of\_ships);

void perform\_shoot(Playground& enemy\_playground, Coords coord);

void put\_ships();

void clear\_ships();

Playground& get\_playground();

void serialize(FileWrapper& file) const;

void deserialize(FileWrapper& file);

};

#endif

Название файла: FileWrapper.h

#ifndef WRAPPER\_H

#define WRAPPER\_H

#include <iostream>

#include <fstream>

class FileWrapper {

std::fstream file;

public:

FileWrapper(const std::string& filename, std::ios::openmode mode);

~FileWrapper();

template <typename T>

void write(const T& data);

template <typename T>

void read(T& data);

};

#endif