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

ОТЧЕТ

по лабораторной работе №4
по дисциплине «Объектно-ориентированное программирование»
Тема: Шаблонные классы.

Студент гр. 3382 Брызгалов Н. А.
Преподаватель Жангиров Т. Р.

Санкт-Петербург
2024

Цель работы:

Разработка шаблонных классов для управления игрой и её отображения в процессе создание класса, связывающего пользовательский ввод с командами игры, и класса для отрисовки игрового поля. Будет реализован класс для считывания ввода из терминала с настройкой соответствия команд через файл.

Задание:

a) Создать шаблонный класс управления игрой. Данный класс должен содержать ссылку на игру. В качестве параметра шаблона должен указываться класс, который определяет способ ввода команда, и переводящий введенную информацию в команду. Класс управления игрой, должен получать команду для выполнения, и вызывать соответствующий метод класса игры.

b) Создать шаблонный класс отображения игры. Данный класс реагирует на изменения в игре, и производит отрисовку игры. То, как происходит отрисовка игры определяется классом переданном в качестве параметра шаблона.

c) Реализовать класс считывающий ввод пользователя из терминала и преобразующий ввод в команду. Соответствие команды введенному символу должно задаваться из файла. Если невозможно считать из файла, то управление задается по умолчанию.

d) Реализовать класс, отвечающий за отрисовку поля.

Примечание:

- Класс отслеживания и класс отрисовки рекомендуется делать отдельными сущностями. Таким образом, класс отслеживания инициализирует отрисовку, и при необходимости можно заменить отрисовку (например, на GUI) без изменения самого отслеживания
- После считывания клавиши, считанный символ должен сразу обрабатываться, и далее работа должна проводить с сущностью, которая представляет команду.
- Для представления команды можно разработать системы классов или использовать перечисление enum.
- Хорошей практикой является создание “прослойки” между считыванием/обработкой команды и классом игры, которая сопоставляет

команду и вызываемым методом игры. Существуют альтернативные решения без явной “прослойки”

- При считывания управления необходимо делать проверку, что на все команды назначена клавиша, что на одну клавишу не назначено две команды, что на одну команду не назначено две клавиши.

Теоретические сведения:

Классы в C++ представляют собой пользовательский тип данных, который объединяет данные и функции, работающие с этими данными. Они являются основой объектно-ориентированного программирования (ООП) и позволяют моделировать объекты.

Структура класса

Класс состоит из полей и методов. Поля хранят состояние объекта, а методы определяют его поведение. Обычно поля объявляются как private, чтобы защитить данные от прямого доступа извне. Методы могут быть как public, так и private, что позволяет контролировать доступ к функциональности класса.

Конструктор класса

Конструктором называют специальный метод, который вызывается при создании объекта и используется для инициализации его полей. Конструкторы могут принимать параметры, что позволяет создавать разные варианты инициализации.

Ключевые компоненты класса

- Поля хранят состояние объекта и обычно защищены от внешнего доступа.
- Методы определяют поведение объекта и могут быть открыты или закрыты для доступа.
- Конструкторы и деструкторы используются для создания и очистки объектов.

- Наследование позволяет создавать новые классы на основе существующих, что способствует повторному использованию кода.
- Инкапсуляция скрывает внутреннее состояние объекта и предоставляет доступ к нему через методы, что улучшает безопасность кода.
- Полиморфизм позволяет объектам разных классов обрабатывать вызовы методов по-разному, увеличивая гибкость программы.

Исключения в C++ представляют собой механизм обработки ошибок, позволяющий отделить код, вызывающий ошибку, от кода, который её обрабатывает. Когда возникает исключение, выполнение текущей функции прерывается, и управление передаётся блоку catch, соответствующему типу исключения. Это позволяет избежать распространения ошибок по коду, улучшая структуру и читаемость программы.

Рекомендуется использовать пользовательские типы исключений для более точной обработки ошибок. Иерархия классов для исключений позволяет эффективно обрабатывать различные типы ошибок, а переброс исключений - передавать управление обработкой на более высокий уровень.

Шаблонные классы в C++ представляют собой мощный инструмент, позволяющий создавать обобщенные классы и функции, которые могут работать с различными типами данных без необходимости дублирования кода. Они обеспечивают гибкость и повторное использование, позволяя разработчикам определять структуру и поведение класса или функции с использованием параметров типа. Шаблоны компилируются на этапе компиляции, что обеспечивает высокую производительность и типобезопасность. Основные преимущества шаблонных классов включают возможность создания контейнеров (например, в стандартной библиотеке

STL), а также упрощение разработки сложных алгоритмов, которые могут работать с различными типами данных, сохраняя при этом читаемость и поддержку кода.

Данные аспекты помогают эффективно использовать классы в C++, создавая более структурированные и поддерживаемые программы.

Выполнение работы

В начале работы был создан класс GameControl, являющийся шаблонным классом, который управляет игровым процессом, используя переданный обработчик ввода (InputHandler). Он принимает ссылку на объект Game и обрабатывает игровой цикл, включая загрузку сохраненной игры или размещение кораблей игроком. Внутри игрового цикла происходит чередование ходов игрока и противника с проверкой условий победы и поражения. Класс также обрабатывает размещение кораблей и действия игрока, такие как атаки и использование способностей, с обработкой исключений для управления ошибками. Использование шаблонов позволяет легко адаптировать класс для различных типов обработчиков ввода, обеспечивая гибкость и повторное использование кода.

Класс DefaultInputHandler представляет собой обработчик ввода для игрового приложения, который позволяет пользователю взаимодействовать с игрой через текстовые команды. Он использует библиотеку nlohmann/json для загрузки команд из JSON-файла, что делает систему команд гибкой и легко настраиваемой. При инициализации класса происходит загрузка команд из файла commands.json, который содержит соответствие между строковыми командами и их числовыми значениями, что упрощает обработку пользовательского ввода.

Метод loadCommandMappings отвечает за считывание JSON-файла и заполнение внутренней карты commandMap, которая хранит соответствия между текстовыми командами и их числовыми представлениями. Это позволяет легко добавлять новые команды или изменять существующие, просто редактируя JSON-файл, без необходимости вносить изменения в код. Такой подход повышает модульность и удобство поддержки кода.

Методы getActionChoice и getActionChoiceNoAbility позволяют пользователю выбирать действия в игре. Первый метод предоставляет полный набор действий, включая использование способностей, тогда как второй ограничивает выбор до атак и операций сохранения/загрузки. Оба метода используют цикл для проверки корректности ввода, обеспечивая устойчивость к ошибкам: если введенная команда не распознана, пользователю будет предложено повторить ввод. Это делает интерфейс более дружелюбным и интуитивно понятным.

Метод getAttackCoordinates запрашивает у пользователя координаты атаки, обеспечивая дополнительную валидацию ввода. Он проверяет корректность введенных значений и повторяет запрос в случае ошибки, что предотвращает возможные сбои в игре из-за некорректного ввода.

В целом, класс DefaultInputHandler обеспечивает эффективное и безопасное взаимодействие пользователя с игровой системой, делая игровой процесс более удобным и интуитивно понятным.

Класс FieldTracking представляет собой шаблонный класс, который отвечает за отображение игровых полей (поля игрока и поля противника) в текстовом формате. Он использует объект рендерера, который реализует метод renderFieldLine, позволяя отделить логику отображения от самой структуры данных полей. Такой подход делает класс гибким и расширяемым, так как позволяет использовать различные реализации рендеринга без изменения основной логики.

Конструктор класса принимает ссылки на два объекта типа GameField (для игрока и противника) и объект рендерера. Это обеспечивает возможность работы с полями, которые могут иметь разные размеры и состояния, а также позволяет легко интегрировать различные методы визуализации. Передавая рендерер как шаблонный параметр, класс становится универсальным и может

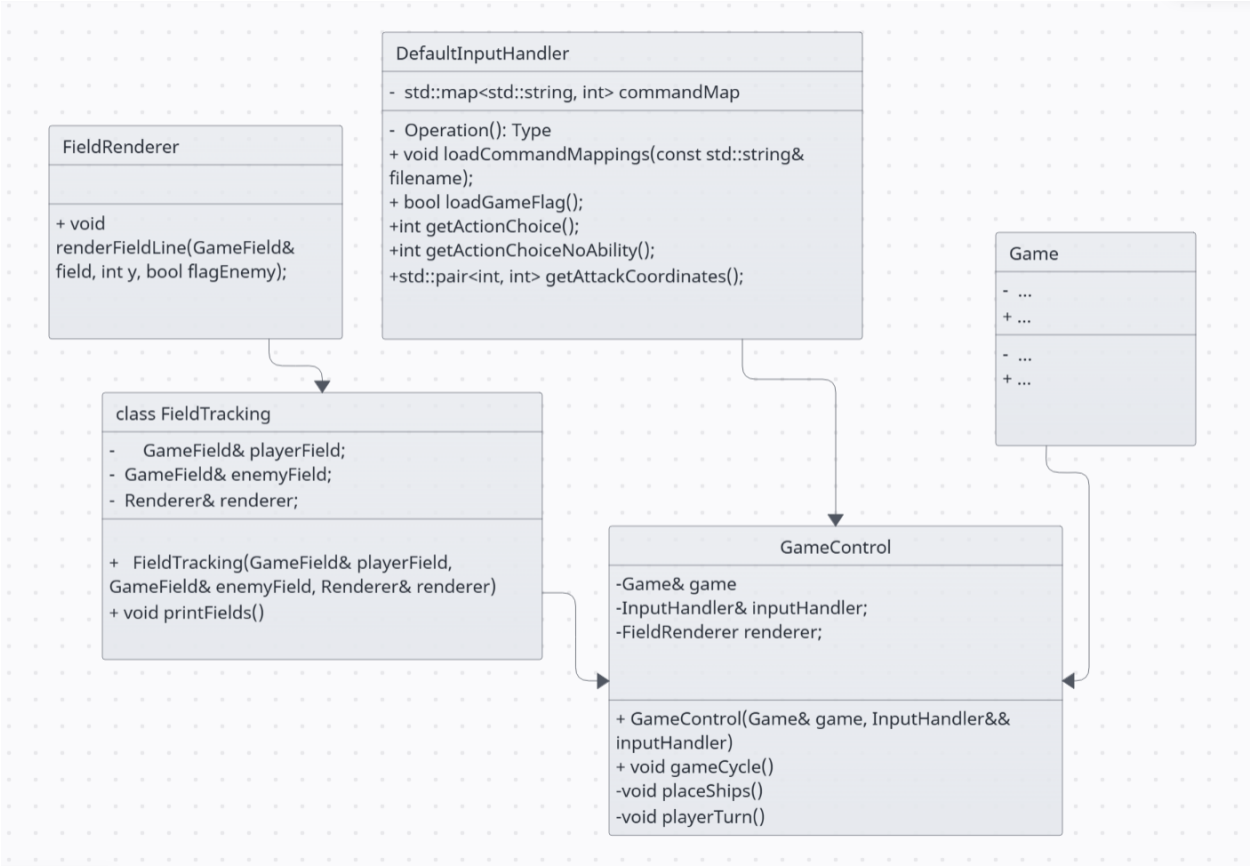
работать с любыми типами рендереров, что делает его применение более широким.

Метод printFields отвечает за вывод полей на экран. Он сначала вычисляет ширину и высоту игрового поля, чтобы корректно отобразить заголовки и индексы столбцов. Заголовки для полей выводятся с отступом, чтобы визуально отделить их друг от друга. Затем метод выводит индексы столбцов, что помогает игроку ориентироваться в поле. После этого идет основной цикл, который проходит по всем строкам высоты поля и вызывает метод renderFieldLine для каждого из полей, обеспечивая их одновременное отображение.

Таким образом, класс FieldTracking обеспечивает удобный и эффективный способ визуализации игровых полей, отделяя логику отображения от данных. Это улучшает читаемость кода и упрощает его поддержку, позволяя легко добавлять новые функции или изменять существующие без вмешательства в основную архитектуру приложения.

Класс FieldRenderer отвечает за визуализацию строк игрового поля в текстовом формате, используя цветовое кодирование для различных состояний ячеек. Метод renderFieldLine принимает объект GameField, координату строки и булевый флаг, указывающий, является ли поле полем противника. Он получает ширину поля и выводит номер строки, после чего проходит по всем столбцам, извлекая состояние каждой ячейки. В зависимости от состояния (от 0 до 5) выводятся соответствующие символы: пустая ячейка отображается пробелами, ячейки с состояниями 1, 2, 3, 4 и 5 отображаются с использованием ANSI-кодов для изменения цвета текста (например, серый, зеленый, желтый, оранжевый и красный), что делает информацию о состоянии игрового поля более интуитивно понятной и визуально привлекательной для игроков.

UML - диаграмма:



Результат работы программы

Результатом работы программы является полноценная игра. При старте пользователь может выбрать, хочет ли он загрузить сохраненную игру или начать новую. Затем, при старте новой игры пользователю предлагается разместить свои корабли на игровом поле. Все некорректные попытки ввода пропускаются и запрашиваются заново. Затем начинается сама игра. В свой ход пользователь может при помощи команд стрелять, использовать способности, сохранять и загружать игру. В ответ на действия игрока по его полю будет производиться обстрел со стороны компьютерного врага. При победе игрока игра перезапускается с сохранением состояния поля, способностей и кораблей игрока, но с новым врагом. При проигрыше игрока игра перезапускается с самого начала. Кроме того, в любой момент в ходе игры пользователь может сохраниться и выйти, чтобы при перезапуске программы продолжить игру с данного момента.

Вывод

В ходе выполнения работы был успешно доработана игра, похожая по своим правилам на морской бой. Были добавлены шаблонные классы для управления и отображения игры. Для данных шаблонных классов были написаны отдельные классы – параметры. Кроме того, был разработан класс игрового процесса, который объединяет всю вышеописанную логику.

Приложение 1

Файл main.cpp:

```
#include <iostream>
#include <vector>
#include <stdexcept>
#include "ShipManager.h"
#include "GameField.h"
#include "AbilityManager.h"
#include "Game.h"

int main() {
    while (true) {
        Game game(10, 10, { 1 });
        game.startGame();
    }

    return 0;
}
```

Файл Ship.cpp:

```
#include "Ship.h"

Ship::Ship(int length)
    : length(length), state(length, ShipState::Whole) {
    if (length < 1 || length > 4)
        throw std::invalid_argument("Ship length must be between 1 and 4.");
}

void Ship::damage(int segment) {
    if (segment < 0 || segment >= length)
        throw std::out_of_range("Segment index out of range.");

    if (state[segment] == ShipState::Whole) {
        state[segment] = ShipState::Damaged;
    }
```

```
    else if (state[segment] == ShipState::Damaged) {
        state[segment] = ShipState::Destroyed;
    }
}

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

Ship::ShipState Ship::getSegmentState(int segment) {
    return state[segment];
}

bool Ship::isDestroyed() {
    for (const auto& s : state) {
        if (s != ShipState::Destroyed) {
            return false;
        }
    }
    return true;
}

void Ship::restore() {
    for (int i = 0; i < length; ++i) {
        state[i] = ShipState::Whole;
    }
}
```

Файл Ship.h:

```
#pragma once
#include <iostream>
#include <vector>
#include <stdexcept>

class Ship {
public:
    enum class ShipState {
```

```
        Whole,
        Damaged,
        Destroyed
    };
    Ship(int length);
    void damage(int segment);
    int getLength();
    ShipState getSegmentState(int segment);
    bool isDestroyed();
    void restore();

private:
    int length;
    std::vector<ShipState> state;
};
```

Файл ShipManager.cpp:

```
#include "ShipManager.h"


ShipManager::ShipManager(const std::vector<int>& sizes) {
    for (int size : sizes) {
        ships.emplace_back(size);
    }
}

Ship& ShipManager::getShip(int index) {
    if (index < 0 || index >= ships.size()) {
        throw std::out_of_range("Ship index out of range.");
    }
    return ships[index];
}

int ShipManager::getCount() {
```

```
    return ships.size();
}

int ShipManager::getShipIndex(const Ship& ship) const {
    for (int i = 0; i < ships.size(); ++i) {
        if (&ships[i] == &ship) {
            return i;
        }
    }
    return -1; // Корабль не найден
}

bool ShipManager::allShipDestroyed() {
    for (int i = 0; i < ships.size(); ++i) {
        if (!ships[i].isDestroyed())
            return false;
    }
    return true;
}

void ShipManager::restoreAllShips() {
    for (int i = 0; i < ships.size(); ++i) {
        ships[i].restore();
    }
}
```

Файл ShipManager.h:

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


class ShipManager {
```

```
public:
    struct ShipPlacement {
        int x;
        int y;
        bool vertical;
    };

    ShipManager(const std::vector<int>& sizes);
    Ship& getShip(int index);
    int getCount();
    int getShipIndex(const Ship& ship) const;
    bool allShipDestroyed();
    void restoreAllShips();


private:
    std::vector<Ship> ships;
};
```

Файл GameField.cpp:

```
#include "GameField.h"


GameField::GameField(int width, int height)
    : width(width), height(height), field(height, std::vector<Cell>(width, { CellStatus::Unknown, nullptr })) {}


void GameField::placeShip(Ship& ship, int x, int y, bool vertical, int shipIndex) {
    int length = ship.getLength();
    if (vertical) {
        if (y + length >= height || x < 0 || x >= width || y < 0) {
            throw InvalidShipPlacementException("Ship is out of bounds.");
        }
        int y0 = y - 1;
        int y1 = y + length;
        int x0 = x - 1;
        int x1 = x + 1;
```

```
        if (y0 < 0) {
            y0 = 0;
        }
        if (y1 >= height) {
            y1 = height - 1;
        }
        if (x0 < 0) {
            x0 = 0;
        }
        if (x1 >= width) {
            x1 = width - 1;
        }
        for (int i = x0; i <= x1; ++i) {
            for (int j = y0; j <= y1; ++j) {
                if (field[j][i].status != CellStatus::Unknown) {
                    throw InvalidShipPlacementException("Ship intersects with another ship.");
                }
            }
        }
        for (int i = 0; i < length; ++i) {
            field[y + i][x] = { CellStatus::Ship, &ship };
        }
    }
    else {
        if (x + length >= width || x < 0 || y >= height || y < 0) {
            throw InvalidShipPlacementException("Ship is out of bounds.");
        }
        int y0 = y - 1;
        int y1 = y + 1;
        int x0 = x - 1;
        int x1 = x + length;
        if (y0 < 0) {
            y0 = 0;
        }
        if (y1 >= height) {
            y1 = height - 1;
        }
```

```
        if (x0 < 0) {
            x0 = 0;
        }
        if (x1 >= width) {
            x1 = width - 1;
        }
        for (int i = x0; i <= x1; ++i) {
            for (int j = y0; j <= y1; ++j) {
                if (field[j][i].status != CellStatus::Unknown) {
                    throw InvalidShipPlacementException("Ship intersects with another ship.");
                }
            }
        }
        for (int i = 0; i < length; ++i) {
            field[y][x + i] = { CellStatus::Ship, &ship };
        }
    }
}


bool GameField::attack(int x, int y, ShipManager& manager) {
    if (x < 0 || x >= width || y < 0 || y >= height)
        throw OutOfBoundsAttackException();

    if (field[y][x].status == CellStatus::Ship) {
        Ship& attackedShip = *field[y][x].ship; // Получаем ссылку на корабль из клетки
        // Определяем сегмент
        int segmentIndex = 0;

        // Ищем начало корабля по горизонтали (слева)
        int x0 = x;
        while (x0 >= 1 && field[y][x0 - 1].status == CellStatus::Ship) {
            x0--;
        }

        // Ищем начало корабля по вертикали (сверху)
```

```
        int y0 = y;
        while (y0 >= 1 && field[y0 - 1][x].status == CellStatus::Ship) {
            y0--;
        }

        // Определяем сегмент, исходя из позиции начала корабля
        if (x - x0 > 0) {
            segmentIndex = x - x0; // Горизонтально
        }
        else {
            segmentIndex = y - y0; // Вертикально
        }

        // Наносим урон
        attackedShip.damage(segmentIndex); // Урон сегменту



        // Вывод информации об атаке
        if (attackedShip.isDestroyed()) {
            std::cout << "Hit on ship " << manager.getShipIndex(attackedShip)
<< " at (" << x << ", " << y << "), ship destroyed\n";
            return true;
        }
        else      if      (attackedShip.getSegmentState(segmentIndex)      ==
Ship::ShipState::Destroyed) {
            std::cout << "Hit on ship " << manager.getShipIndex(attackedShip)
<< " at (" << x << ", " << y << "), segment destroyed\n";
        }
        else {
            std::cout << "Hit on ship " << manager.getShipIndex(attackedShip)
<< " at (" << x << ", " << y << ")\n";
        }


    }
    else {
        std::cout << "Missed :(\n";
        field[y][x].status = CellStatus::Empty; // Помечаем клетку как пустую
    }
    return false;
```

```
}


GameField::GameField(const GameField& other)
    :     width(other.width),     height(other.height),     field(other.height,
std::vector<Cell>(other.width)) {
    for (size_t i = 0; i < height; ++i) {
        for (size_t j = 0; j < width; ++j) {
            field[i][j].status = other.field[i][j].status;
            field[i][j].ship = other.field[i][j].ship; // Копируем указатель
        }
    }
}


GameField::GameField(GameField&& other) noexcept
    : width(other.width), height(other.height), field(std::move(other.field))
{
    other.width = 0;
    other.height = 0;
}


GameField& GameField::operator=(const GameField& other) {
    if (this != &other) {
        width = other.width;
        height = other.height;
        field.resize(height);
        for (size_t i = 0; i < height; ++i) {
            field[i].resize(width);
            for (size_t j = 0; j < width; ++j) {
                field[i][j].status = other.field[i][j].status;
                field[i][j].ship   =   other.field[i][j].ship;   //   Копируем
указатель
            }
        }
    }
    return *this;
}


GameField& GameField::operator=(GameField&& other) noexcept {
```

```
    if (this != &other) {
        width = other.width;
        height = other.height;
        field = std::move(other.field);
        other.width = 0;
        other.height = 0;
    }
    return *this;
}


void GameField::printField() const {
    // Вывод заголовка столбцов
    std::cout << "  ";
    for (int i = 0; i < width; ++i) {
        std::cout << i << " ";
    }
    std::cout << std::endl;

    // Вывод поля
    for (int y = 0; y < height; ++y) {
        // Вывод номера строки
        std::cout << y << " ";

        for (int x = 0; x < width; ++x) {
            switch (field[y][x].status) {
            case CellStatus::Unknown:
                std::cout << "  ";
                break;
            case CellStatus::Empty:
                std::cout << ". ";
                break;
            case CellStatus::Ship:
                Ship& attackedShip = *field[y][x].ship; // Получаем ссылку на
корабль из клетки
                if (attackedShip.isDestroyed()) {
                    std::cout << "# ";
                }
```

```
                else {
                    // Определяем сегмент
                    int segmentIndex = 0;

                    // Ищем начало корабля по горизонтали (слева)
                    int x0 = x;
                    while (x0 >= 1 && field[y][x0 - 1].status == CellStatus::Ship) {
                        x0--;
                    }

                    // Ищем начало корабля по вертикали (сверху)
                    int y0 = y;
                    while (y0 >= 1 && field[y0 - 1][x].status == CellStatus::Ship) {
                        y0--;
                    }

                    // Определяем сегмент, исходя из позиции начала корабля
                    if (x - x0 > 0) {
                        segmentIndex = x - x0; // Горизонтально
                    }
                    else {
                        segmentIndex = y - y0; // Вертикально
                    }


                    if (attackedShip.getSegmentState(segmentIndex) == Ship::ShipState::Destroyed) {
                        std::cout << "X ";
                    } else if (attackedShip.getSegmentState(segmentIndex) == Ship::ShipState::Damaged) {
                        std::cout << "x ";
                    } else {
                        std::cout << "! ";
                    }

                }
                break;
```

```
            }
        }
        std::cout << std::endl;
    }
}

void GameField::removeShipsLocations() {
    for (int y = 0; y < height; ++y) {
        for (int x = 0; x < width; ++x) {
            field[y][x].status = CellStatus::Unknown;
            field[y][x].ship = nullptr;
        }
    }
}
```

Файл GameField.h:

```
#pragma once
#include <vector>
#include <stdexcept>
#include "ShipManager.h"
#include "InvalidShipPlacementException.h"
#include "OutOfBoundsAttackException.h"



class GameField {
public:
    enum class CellStatus {
        Unknown,
        Empty,
        Ship
    };

    struct Cell {
        CellStatus status;
        Ship* ship; // Указатель на корабль, если он есть в этой клетке
    };
```

```
    GameField(int width, int height);
    void placeShip(Ship& ship, int x, int y, bool vertical, int shipIndex);
    bool attack(int x, int y, ShipManager& manager);
    GameField(const GameField& other); // Конструктор копирования
    GameField(GameField&& other) noexcept; // Конструктор перемещения
    GameField& operator=(const GameField& other); // Оператор присваивания копированием
    GameField& operator=(GameField&& other) noexcept; // Оператор присваивания перемещением
    void printField() const;
    int getWidth() const { return width; }
    int getHeight() const { return height; }
    CellStatus getCellStatus(int x, int y) const { return field[y][x].status; }
    void removeShipsLocations();


private:
    int width;
    int height;
    std::vector<std::vector<Cell>> field;
};
```

Файл AbilityManager.h:

```
#pragma once
#include <vector>
#include <queue>
#include <random>
#include "Ability.h"
#include "DoubleDamage.h"
#include "Scanner.h"
#include "RandomShot.h"
#include "NoAbilitiesException.h"

class AbilityManager {
public:
    AbilityManager();
```

```
    void applyAbility(GameField& field, ShipManager& manager);
    void addAbility(Ability* ability);
    void generateInitialAbilities();
    void giveRandomAbility();

private:
    std::queue<Ability*> abilities;
};
```

Файл AbilityManager.cpp:

```
#include "AbilityManager.h"

AbilityManager::AbilityManager() {
    generateInitialAbilities();
}

void AbilityManager::applyAbility(GameField& field, ShipManager& manager) {
    if (!abilities.empty()) {
        Ability* ability = abilities.front();
        abilities.pop();
        ability->apply(field, manager);
        delete ability;
    }
    else {
        throw NoAbilitiesException();
    }
}

void AbilityManager::addAbility(Ability* ability) {
    abilities.push(ability);
}

void AbilityManager::generateInitialAbilities() {
    srand(time(NULL)); // Инициализация генератора случайных чисел
    // Массив с типами способностей (0 - DoubleDamage, 1 - Scanner, 2 - RandomShot)
    int abilityTypes[3] = { 0, 1, 2 };
```

```
    // Перемешивание массива для случайного порядка способностей
    for (int i = 2; i > 0; --i) {
        int j = rand() % (i + 1);
        std::swap(abilityTypes[i], abilityTypes[j]);
    }

    // Создание и добавление способностей в очередь
    for (int i = 0; i < 3; ++i) {
        switch (abilityTypes[i]) {
        case 0:
            addAbility(new DoubleDamage());
            break;
        case 1:
            addAbility(new Scanner());
            break;
        case 2:
            addAbility(new RandomShot());
            break;
        }
    }
}

void AbilityManager::giveRandomAbility() {
    srand(time(NULL)); // Инициализация генератора случайных чисел
    int abilityType = rand() % 3; // Случайное число от 0 до 2

    switch (abilityType) {
    case 0:
        std::cout << "\nAdded double damage ability!\n\n";
        addAbility(new DoubleDamage());
        break;
    case 1:
        std::cout << "\nAdded scanner ability!\n\n";
        addAbility(new Scanner());
        break;
    case 2:
        std::cout << "\nAdded RandomShot ability!\n\n";
```

```
        addAbility(new RandomShot());
        break;
    }
}
```

Файл Ability.h:

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

class Ability {
public:
      virtual void apply(GameField& field, ShipManager& manager) = 0;
      virtual ~Ability() = default;
};
```

Файл RandomShot.h:

```
#pragma once
#include "Ability.h"
#include <random>

class RandomShot : public Ability {
public:
    void apply(GameField& field, ShipManager& manager) override;
};
```

Файл RandomShot.cpp:

```
#include "RandomShot.h"
#include <random>

void RandomShot::apply(GameField& field, ShipManager& manager) {
    srand(time(NULL));
    if (manager.getCount() > 0) {
        Ship* ship = nullptr;
        int random_index;
```

```
        do {
            random_index = rand() % manager.getCount();
            ship = &manager.getShip(random_index);
        } while (ship->isDestroyed());

        int segment_index;
        do {
            segment_index = rand() % ship->getLength();
        }         while         (ship->getSegmentState(segment_index)         !=
Ship::ShipState::Destroyed);
        ship->damage(segment_index);
        std::cout << "\nRandom shot hit the ship!\n\n";
    }
}
```

Файл Scanner.h:

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

class Scanner : public Ability {
public:
      void apply(GameField& field, ShipManager& manager) override;
};
```

Файл Scanner.cpp:

```
#include "Scanner.h"

void Scanner::apply(GameField& field, ShipManager& manager) {
    int x, y;
    bool flag = true;
    std::cout << "Scanner activated! (x y): ";
    std::cin >> x >> y;

    for (int i = -1; i <= 1; ++i) {
        for (int j = -1; j <= 1; ++j) {
            int scanX = x + i;
```

```
            int scanY = y + j;
            if (scanX >= 0 && scanX < field.getWidth() && scanY >= 0 && scanY
< field.getHeight()) {
                if        (field.getCellStatus(scanX,        scanY)        ==
GameField::CellStatus::Ship) {
                    std::cout << "Ship detected at (" << scanX << ", " << scanY
<< ")\n";
                    flag = false;
                }
            }
        }
    }
    if (flag) {
        std::cout << "Scanner didn't see the ships\n";
    }
}
```

Файл DoubleDamage.h:

```
#pragma once
#include "Ability.h"
class DoubleDamage : public Ability {
public:
      void apply(GameField& field, ShipManager& manager) override;
};
```

Файл DoubleDamage.cpp:

```
#include "DoubleDamage.h"

void DoubleDamage::apply(GameField& field, ShipManager& manager) {
    field.setDoubleDamage(true);
    std::cout << "Ability 'Double Damage' activated!\n";
}
```

Файл GameException.h:

```
#pragma once
#include <string>
#include <stdexcept>

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;
};
```

Файл NoAbilitiesException.h:

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

class NoAbilitiesException : public GameException {
public:
    NoAbilitiesException() : GameException("No abilities available.") {}
};
```

Файл OutOfBoundsAttackException.h:

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

class OutOfBoundsAttackException : public GameException {
public:
    OutOfBoundsAttackException() : GameException("Attack coordinates out of
bounds.") {}
};
```

Файл InvalidShipPlacementException.h:

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

class InvalidShipPlacementException : public GameException {
public:
    InvalidShipPlacementException(const std::string& reason) : GameException("Invalid ship placement: " + reason) {}
};
```

Файл GeneralException.h:

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

class GeneralException : public GameException {
public:
    GeneralException(const std::string& message) : GameException(message) {}
};
```

Файл Game.cpp:

```
#include "Game.h"

Game::Game(int width, int height, const std::vector<int>& shipSizes)
    : field(width, height), shipManager(shipSizes), abilityManager(), enemyShipManager(shipSizes), enemyField(width, height) {
    generateEnemyShips();
}

void Game::startGame() {
    // Размещение кораблей игрока
    std::cout << "Place your ships:\n";
    for (int i = 0; i < shipManager.getCount(); ++i) {
```

```
        int x, y;
        bool vertical = true;
        std::cout << "Place a ship with length " << shipManager.getShip(i).getLength() << " (x y): ";
        std::cin >> x >> y;
        if (shipManager.getShip(i).getLength() > 1) {
            std::cout << "Place the ship vertically? (1 - yes, 0 - no, 2 - hohoho) : ";
            std::cin >> vertical;
        }
        try {
            field.placeShip(shipManager.getShip(i), x, y, vertical, i);
        }
        catch (const std::exception& e) {
            std::cout << e.what() << "\n";
            --i;
        }
    }
    // Генерация вражеских кораблей
    //generateEnemyShips();

    // Основной цикл игры
    std::cout << "\nThe game has begun!\n";
    while (true) {
        playRound();
        if (enemyShipManager.allShipDestroyed()) {
            std::cout << "Round won!\n\n\n";
            std::cout << "New round!\n";
            resetEnemyShips();
        }
        else if (shipManager.allShipDestroyed()) {
            std::cout << "You lose!\n\n\n";
            std::cout << "New game!\n";
            break;
        }
    }
}
```

```
void Game::playRound() {
    std::cout << "\nYour field:\n";
    field.printField();
    std::cout << "\nEnemy field:\n";
    enemyField.printField();

    playerTurn();
    if (!enemyShipManager.allShipDestroyed()) {
        enemyTurn();
    }
}

void Game::playerTurn() {
    std::cout << "\nYour move:\n";
    int task = 0;
    if (abilityManager.getAbilityCount()) {
        std::cout << "Enter 0 if attack, enter 1 if ability: ";
        std::cin >> task;
    }
    if (task) {
        abilityManager.applyAbility(enemyField, enemyShipManager);

    } else {

        int x, y;
        std::cout << "Enter coordinates to attack (x y): ";
        std::cin >> x >> y;
        bool attacked;
        if (abilityManager.isDoubleDamage()) {
            enemyField.attack(x, y, enemyShipManager);
            abilityManager.delDoubleDamage();
        }
        attacked = enemyField.attack(x, y, enemyShipManager);
        if (attacked) {
            abilityManager.giveRandomAbility();
        }
    }
}
```

```
void Game::enemyTurn() {
    srand(time(NULL));
    int x = rand() % field.getWidth();
    int y = rand() % field.getHeight();
    std::cout << "\nEnemy move: ";
    bool attacked = field.attack(x, y, shipManager);
}

void Game::generateEnemyShips() {
    srand(time(NULL));
    for (int i = 0; i < enemyShipManager.getCount(); ++i) {
        int x = rand() % enemyField.getWidth();
        int y = rand() % enemyField.getHeight();
        bool vertical = rand() % 2;
        try {
            enemyField.placeShip(enemyShipManager.getShip(i), x, y, vertical,
i);
        }
        catch (const std::exception& e) {
            --i;
        }
    }
}

void Game::resetEnemyShips() {
    enemyShipManager.restoreAllShips();
    enemyField.removeShipsLocations();
    generateEnemyShips();
}
```

Файл Game.h:

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

```
class Game {
public:
    Game(int width, int height, const std::vector<int>& shipSizes);
    void startGame();

private:
    void playRound();
    void playerTurn();
    void enemyTurn();
    void generateEnemyShips();
    void resetEnemyShips();

    GameField field;
    ShipManager shipManager;
    AbilityManager abilityManager;

    GameField enemyField;
    ShipManager enemyShipManager;
};
```

Файл FieldRender.h:

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

class FieldRenderer {
public:
    void renderFieldLine(GameField& field, int y, bool flagEnemy);
};
```

Файл FieldRender.cpp:

```
#include "FieldRender.h"


void FieldRenderer::renderFieldLine(GameField & field, int y, bool flagEnemy)
{
```

```
    int width = field.getWidth();
    std::cout << y << " ";
    for (int x = 0; x < width; ++x) {
        int state = field.getState(x, y);
        switch (state) {
        case 0: std::cout << "  "; break;
        case 1: std::cout << "\033[1;30m. \033[0m"; break; // Серый цвет для .
        case 2: std::cout << (flagEnemy ? "  " : "\033[1;32mo \033[0m"); break;
// Зеленый цвет для !
        case 3: std::cout << "\033[1;33mx \033[0m"; break; // Желтый цвет для
x
        case 4: std::cout << "\033[1;31mX \033[0m"; break; // Оранжевый цвет
для X  (рыжий в ASCII не поддерживается)
        case 5: std::cout << "\033[1;31m# \033[0m"; break; // Красный цвет для
#
        }
    }
}
```

Файл FieldTracking.h:

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

template<typename Renderer>
class FieldTracking {
public:
    FieldTracking(GameField& playerField, GameField& enemyField, Renderer&
renderer)
        : playerField(playerField), enemyField(enemyField), renderer(renderer)
{}

    void printFields() {
        int width = playerField.getWidth();
        int height = playerField.getHeight();

        // Отрисовка заголовков
        for (int i = 0; i < width/2 + 3; ++i) {
```

```
            std::cout << " ";
        }
        std::cout << "Your field";
        for (int i = 0; i < (width); ++i) {
            std::cout << "  ";
        }
        std::cout << "Enemy field" << std::endl;

        // Отрисовка индексов столбцов
        std::cout << "    ";
        for (int i = 0; i < width; ++i) {
            std::cout << i << " ";
        }
        std::cout << "          ";
        for (int i = 0; i < width; ++i) {
            std::cout << i << " ";
        }
        std::cout << std::endl;

        // Отрисовка строк поля
        for (int y = 0; y < height; ++y) {
            std::cout << "  ";
            renderer.renderFieldLine(playerField, y, false);
            std::cout << "        ";
            renderer.renderFieldLine(enemyField, y, true);
            std::cout << std::endl;
        }
    }

private:
    GameField& playerField;
    GameField& enemyField;
    Renderer& renderer;
};
```

Файл GameControl.h:

```
#pragma once
#include "Game.h"
#include "FieldTracking.h"



template <typename InputHandler>
class GameControl {
private:
    Game& game;
    InputHandler inputHandler;
    FieldRenderer renderer;

public:
    GameControl(Game& game, InputHandler&& inputHandler)
        : game(game), inputHandler(inputHandler), renderer() {}

    void gameCycle() {
        FieldTracking<FieldRenderer>      fieldTracking(game.getPlayerField(),
game.getEnemyField(), renderer);

        // Ask the player if they want to load a saved game
        std::cout << "Would you like to load a saved game? ";
        bool flag_loadGame = inputHandler.loadGameFlag();

        if (flag_loadGame) {
            game.loadGame();
        }
        else {
            placeShips();
        }

        std::cout << "\nThe game has started!\n";
        while (true) {
            std::cout << std::endl;
            fieldTracking.printFields();

            playerTurn();
```

```
            if (!game.isWin()) {
                game.enemyTurn();
            }

            // Check if the player has won or lost
            if (game.isWin()) {
                std::cout << "\nYour turn: F9 + ???\n";
                std::cout << "Congratulations! You won this round!\n\n";
                std::cout << "Starting a new round...\n";
                game.resetEnemyShips();
            }
            else if (game.isLoss()) {
                std::cout << "\nEnemy turn: F9 + ???\n";
                std::cout << "Game over! You have lost.\n\n";
                std::cout << "Starting a new game...\n";
                break;
            }
        }
    }

private:
    void placeShips() {
        FieldTracking<FieldRenderer>      fieldTracking(game.getPlayerField(),
game.getEnemyField(), renderer);
        std::cout << "Time to place your ships:\n";
        for (int i = 0; i < game.getShipCount(); ++i) {
            int x, y;
            bool vertical = true;

            std::cout << "Place your ship of length " << game.getShipLength(i)
<< ". ";
            std::tie(x, y) = inputHandler.getAttackCoordinates();

            // Ask if the ship should be placed vertically if it has a length
greater than 1
            if (game.getShipLength(i) > 1) {
                std::cout << "Should the ship be placed vertically? ";
                vertical = inputHandler.loadGameFlag();
            }
```

```
            // Attempt to place the ship and handle any exceptions
            try {
                //field.placeShip(shipManager.getShip(i), x, y, vertical, i);
                game.placeShip(x, y, vertical, i);
                std::cout << std::endl;
                fieldTracking.printFields();
                std::cout << std::endl;
            }
            catch (const std::exception& e) {
                std::cout << "Error: " << e.what() << "\nPlease try again.\n";
                --i; // Decrease index to repeat placement
            }

        }
        game.generateEnemyShips();
    }

    void playerTurn() {
        std::cout << "\nYour turn:\n";

        // Variable for player's action choice
        int actionChoice = 0;

        // Check if there are available abilities
        if (game.haveAbility()) {
            actionChoice = inputHandler.getActionChoice();
        }
        else {
            actionChoice = inputHandler.getActionChoiceNoAbility();
        }

        // Handle the action choice
        switch (actionChoice) {
        case 1: // Use Ability
            try {
                game.useAbility();
            }
```

```
        catch (const std::exception& e) {
            std::cout << "Error: " << e.what() << "\n";
            playerTurn();
        }
        break;

    case 0: // Attack
    {
        int x, y;
        try {
            std::cout << "Attack! ";
            std::tie(x, y) = inputHandler.getAttackCoordinates();
            game.attack(x, y);
        }
        catch (const std::exception& e) {
            std::cout << "Error: " << e.what() << "\n";
            playerTurn();
        }
        break;
    }

    case 2: // Save Game
        game.saveGame();
        std::cout << "Game successfully saved.\n";
        playerTurn();
        break;

    case 3: // Load game
        try {
            game.loadGame();
            std::cout << "Game loaded successfully.\n";
        }
        catch(const std::exception & e) {
            std::cout << "Error: " << e.what() << "\n";
        }
        playerTurn();
        break;
```

```
        default:
            std::cout << "Invalid choice, form again!\n";
            playerTurn();
        }
    }
};
```