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

ОТЧЕТ

по лабораторной работе №2
по дисциплине «Полиморфизм»
Тема: Создание классов

Студент гр. 3385 _____ Гребенщиков А.А.

Преподаватель _____ Жангиров Т.Р.

Санкт-Петербург
2024

Цель работы.

Разработать системы управления способностями игрока в игре Морской бой. В рамках этой работы необходимо создать классы, реализующие интерфейс способностей, а также менеджер способностей, который будет обеспечивать их применение и управление.

Задание.

Создать класс-интерфейс способности, которую игрок может применять. Через наследование создать 3 разные способности:

Двойной урон - следующая атак при попадании по кораблю нанесет сразу 2 урона (уничтожит сегмент).

Сканер - позволяет проверить участок поля 2x2 клетки и узнать, есть ли там сегмент корабля. Клетки не меняют свой статус.

Обстрел - наносит 1 урон случайному сегменту случайного корабля. Клетки не меняют свой статус.

Создать класс менеджер-способностей. Который хранит очередь способностей, изначально игроку доступно по 1 способности в случайном порядке. Реализовать метод применения способности.

Реализовать функционал получения одной случайной способности при уничтожении вражеского корабля.

Реализуйте набор классов-исключений и их обработку для следующих ситуаций (можно добавить собственные):

Попытка применить способность, когда их нет

Размещение корабля вплотную или на пересечении с другим кораблем

Атака за границы поля

Примечания:

Интерфейс события должен быть унифицирован, чтобы их можно было единообразно использовать через интерфейс

Не должно быть явных проверок на тип данныхас

Выполнение работы.

Класс **Ability** является абстрактным базовым классом, который определяет интерфейс для различных способностей в игре. Он предназначен для использования в контексте игрового поля (GameField) и управления кораблями (ManagerShip). Интерфейс Ability позволяет обеспечить единообразие между различными способностями и упрощает управление ими.

Класс Ability является абстрактным, поскольку содержит чисто виртуальные функции (printAbility и use). Это означает, что он не может быть инстанцирован напрямую и должен быть унаследован от других классов, которые реализуют эти функции.

Виртуальный деструктор ~Ability() гарантирует, что при удалении объекта производного класса через указатель базового класса будет вызван корректный деструктор производного класса. Это важный аспект для управления памятью и предотвращения утечек.

Методы:

virtual void printAbility() = 0; Чисто виртуальная функция, предназначенная для вывода информации о способности. Каждый производный класс должен реализовать этот метод, чтобы предоставить конкретное представление о своей способности.

virtual bool use(GameField& gameField, ManagerShip& manager) = 0;: Чисто виртуальная функция, которая отвечает за использование способности в контексте игрового поля и управления кораблем. Она принимает ссылки на объекты GameField и ManagerShip, что позволяет взаимодействовать с ними. Возвращает true или false, указывая на выполнение конкретной способности, чья логика действия отличается от работы остальных.

Класс **Bombardment** является производным от класса-интерфейса Ability и представляет собой способность, позволяющую выполнять обстрел в случайного корабля. Он реализует методы для использования способности и вывода информации о ней.

Метод use класса Bombardment выполняет логику обстрела в бесконечном цикле, который продолжается до выхода. В каждом цикле случайным образом выбирается корабль из менеджера и его сегмент. Если сегмент не уничтожен, наносится урон. После этого проверяется, уничтожены ли все корабли методом менеджера кораблей. Если да, выводится сообщение о завершении игры, и программа завершает выполнение. Если обстрел успешен, метод возвращает false. Таким образом, метод реализует способность атаковать случайные цели на игровом поле.

Класс **DoubleDamage** является производным от класса-интерфейса Ability и представляет собой способность, которая при следующем выстреле игрока наносит двойной урон. Метод use вовзвращает true, что в классе игры Player вызывает дополнительную атаку при обычном выстреле.

Класс **Scanner** является производным от класса-интерфейса Ability и представляет собой способность, проверяющую квадрат 2 на 2 на наличие в нем коробля.

Метод use класса Scanner выполняет следующие шаги: запрашивает у пользователя ввод координат верхнего левого угла квадрата 2x2, получает значения координат, проверяет, находятся ли координаты в пределах игрового поля; если нет, выводит сообщение об ошибке и возвращает false. Затем перебирает ячейки в квадрате 2x2 и если находит корабль (CellStatus::Ship), выводит сообщение и возвращает false. Если корабли не обнаружены, выводит соответствующее сообщение и завершает работу, возвращая false.

Все классы способностей имеют общую базу (интерфейс), что позволяет использовать полиморфизм для управления ими через указатели или ссылки на базовый интерфейс.

Класс **AbilityManager** управляет набором способностей, которые могут быть использованы в игре. Внутри класса хранится вектор уникальных указателей на объекты типа Ability. Этот класс обеспечивает хранение и использование способностей игроком.

Метод addAbility добавляет новую способность в менеджер. Он создает три уникальные способности аналогично конструктору, помещает их в временный вектор и случайным образом выбирает одну из них для добавления в основной вектор abilities. Для выбора используется функция randOnSection, которая возвращает случайный индекс на переданном промежутке.

Метод getAbility извлекает первую способность из вектора abilities. Если вектор пуст, он выбрасывает исключение NoAbilitiesError, указывая, что больше нет доступных способностей. Если способности есть, метод перемещает первый элемент (первую способность) из вектора и возвращает её, при этом удаляя её из списка доступных способностей.

Конструктор AbilityManager инициализирует вектор способностей, добавляя три типа: Bombardment, DoubleDamage и Scanner. Эти способности создаются с помощью std::make_unique, что обеспечивает безопасное управление памятью. После добавления способностей в вектор, они перемешиваются с использованием генератора случайных чисел, чтобы обеспечить случайный порядок их использования.

Таким образом, класс AbilityManager предоставляет функциональность для управления способностями, позволяя добавлять новые способности и извлекать их по мере необходимости, обеспечивая при этом безопасность работы с памятью.

Класс **Player** отвечает за взаимодействие игрока с игровым полем и расстановку кораблей в игре. Класс является «высшим» в игре, объединяя в себя игровое поле, менеджер кораблей и менеджер способностей, в свою очередь включающие в себя остальные классы. В реализации класса игры будет важным компонетом, отвечающим за интерфейс игрока.

Метод initializeGameField запрашивает у пользователя размеры игрового поля, обеспечивая при этом проверку на положительность введенных значений. Внутри этого метода происходит бесконечный цикл, который продолжает запрашивать ввод, пока пользователь не введет корректные (положительные)

значения для высоты и ширины поля. После успешного ввода метод возвращает объект типа GameField, инициализированный с указанными размерами.

Метод initializeManagerShip позволяет игроку задать количество кораблей каждой длины, начиная с 1 и заканчивая 4. В этом методе создается вектор пар, где каждая пара содержит длину корабля и соответствующее количество. Игроку предлагается ввести количество кораблей для каждой длины, и введенные данные используются для создания объекта ManagerShip, который управляет расстановкой и состоянием кораблей.

Конструктор класса Player обеспечивает создание игрока с необходимыми для игры атрибутами (поля с заданными пользователем размерами, менеджер кораблей, хранящий в себе корабли, которые расствляются на поле с помощью метода placeShips).

Метод placeShips управляет процессом расстановки кораблей на игровом поле. Он использует цикл для итерации по всем кораблям, управляемым объектом managerShip. Для каждого корабля метод запрашивает у пользователя координаты его размещения и ориентацию (горизонтальная или вертикальная). После получения необходимых данных метод вызывает placeShip у объекта gameField, передавая ему информацию о корабле, его координатах и ориентации.

Метод movePlayer реализует логику хода игрока. В начале метода игроку предоставляется возможность воспользоваться специальной способностью, выйти из игры или ввести координаты для выстрела. Если игрок выбирает использование способности, происходит попытка получить ее через менеджер способностей, и если она доступна, выводится информация о ней и применяется к игровому полю. После этого игроку предлагается ввести координаты для выстрела. Вводимые координаты проверяются на корректность с помощью метода checkCoordAtack. Если координаты выходят за пределы игрового поля, выбрасывается исключение OutOfBoundsAttackError, и игроку предлагается ввести их заново. Если выстрел успешен и корабль был поражен,

добавляется новая способность. Также проверяется, уничтожены ли все корабли; если да, игра завершается с соответствующим сообщением.

Метод printField выводит поле с расставленными на нем кораблями.

Классы исключений:

Эти классы предназначены для обработки ошибок, которые могут возникнуть в процессе игры. Они помогают обеспечить более надежное и предсказуемое поведение программы. Все классы наследуются от std::runtime_error, что позволяет использовать стандартные механизмы обработки исключений в C++.

NoAbilitiesError: конструктор вызывает базовый конструктор с сообщением "Нет доступных способностей для применения." Это исключение может использоваться, когда игрок пытается применить способности, которые отсутствуют или недоступны.

ShipPlacementError: конструктор вызывает базовый конструктор с сообщением "Некорректные координаты расстановки корабля." Это исключение будет выбрасываться, если игрок пытается разместить корабль в недопустимых координатах на игровом поле.

OutOfBoundsAttackError: конструктор вызывает базовый конструктор с сообщением "Атака выходит за границы игрового поля." Это исключение может возникнуть, когда игрок пытается атаковать клетку, которая находится за пределами игрового поля.

Для визуализации отношений между классами были созданы две UML диаграммы:

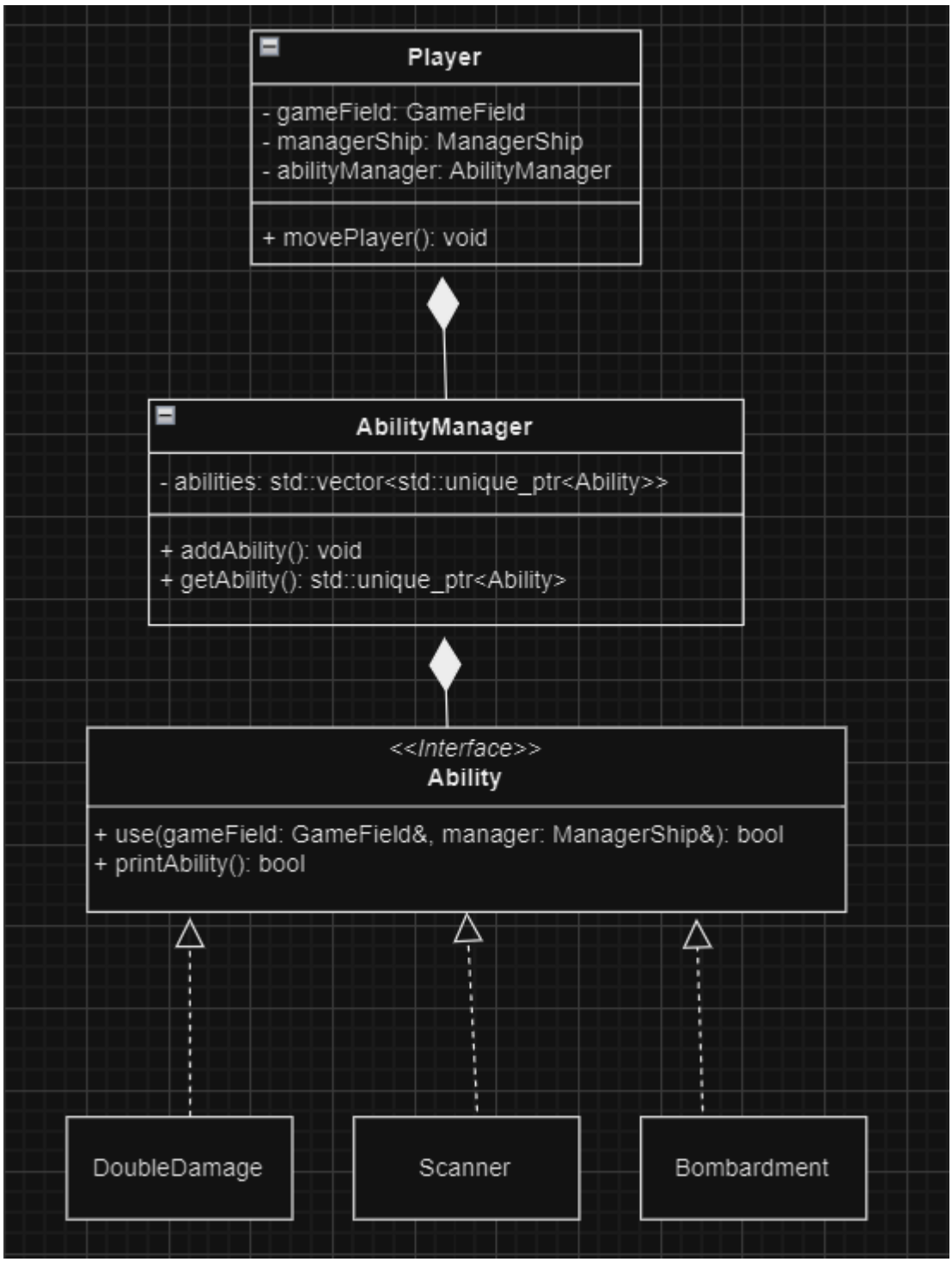

Рис.1 UML-диаграмма классов №1

Выводы.

Был разработан интерфейс способностей, на основе которых созданы требуемые условием задания способности игрока. Был создан менеджер способностей, для хранения доступных игроку способностей и для их добавления. Разработаны классы ошибок, отлавливающие некорректные ситуации.

Приложение

Abilty.h

```
#ifndef ABILITY_H
#define ABILITY_H


#include "GameField.hpp"
#include "ManagerShip.hpp"



class Ability {
public:
    virtual ~Ability() = default;
    virtual void printAbility() = 0;
    virtual bool use(GameField& gameField, ManagerShip& manager) = 0;
};


#endif
```

AbilityManager.h

```
#ifndef ABILITY_MANAGER_H
#define ABILITY_MANAGER_H

#include "Ability.h"
#include "memory"



class AbilityManager {
private:
    std::vector<std::unique_ptr<Ability>> abilities;
public:
    AbilityManager();
    void addAbility();
    std::unique_ptr<Ability> getAbility();
};


#endif
```

AbilityManager.cpp

```
#include <random>
#include <algorithm>
#include <vector>
#include "AbilityManager.h"
#include "Bombardment.h"
#include "DoubleDamage.h"
#include "RandOnSection.h"
#include "Scanner.h"
#include "BattleshipException.h"



AbilityManager::AbilityManager() {
    abilities.emplace_back(std::make_unique<Bombardment>());
    abilities.emplace_back(std::make_unique<DoubleDamage>());
```

```
        abilities.emplace_back(std::make_unique<Scanner>());

        std::random_device rd;
        std::mt19937 generator(rd());
        std::shuffle(abilities.begin(), abilities.end(), generator);
    }



    void AbilityManager::addAbility() {
        std::unique_ptr<Ability>                bombardment                =
std::make_unique<Bombardment>();
        std::unique_ptr<Ability>                doubleDamage               =
std::make_unique<DoubleDamage>();
        std::unique_ptr<Ability> scanner = std::make_unique<Scanner>();

        std::vector<std::unique_ptr<Ability>> randAbilities;
        randAbilities.push_back(std::move(bombardment));
        randAbilities.push_back(std::move(doubleDamage));
        randAbilities.push_back(std::move(scanner));

        int randomIndex = randOnSection(0, randAbilities.size());
        abilities.push_back(std::move(randAbilities[randomIndex]));
    }


    std::unique_ptr<Ability> AbilityManager::getAbility() {
        if (abilities.empty()) {
            throw NoAbilitiesError();
        }

        std::unique_ptr<Ability> ability = std::move(abilities.front());
        abilities.erase(abilities.begin());
        return ability;
    }
```

BattleshipException.h

```
          #ifndef BATTLESHIP_EXCEPTION_H
    #define BATTLESHIP_EXCEPTION_H

    #include <stdexcept>
    #include <iostream>



    class NoAbilitiesError : public std::runtime_error {
    public:
        NoAbilitiesError()  :  std::runtime_error("Нет  доступных  способностей
для применения.") {}
    };


    class ShipPlacementError : public std::runtime_error {
    public:
        ShipPlacementError()   :   std::runtime_error("Некорректные   координаты
расстановки корабля") {}
    };


    class OutOfBoundsAttackError : public std::runtime_error {
    public:
```

```
        OutOfBoundsAttackError()   :   std::runtime_error("Атака   выходит   за
границы игрового поля.") {}
    };


#endif
```

Bombardment.h

```
        #ifndef BOMBARDMENT_H
    #define BOMBARDMENT_H

    #include "Ability.h"



    class Bombardment : public Ability {
    public:
        bool use(GameField& gameField, ManagerShip& manager) override;
        void printAbility() override;
    };


#endif
```

Bombardment.cpp

```
    #include <iostream>
    #include "Bombardment.h"
    #include "RandOnSection.h"



    void Bombardment::printAbility() {
        std::cout << "Была выбрана способность: Обстрел" << std::endl;
    }


    bool Bombardment::use(GameField& gameField, ManagerShip& manager) {
        while (true) {
            Ship      *targetShip      =      manager.getShip(randOnSection(0,
manager.getCountShip()));
            int lenindex = randOnSection(0, targetShip->getLength());
            if            (targetShip->getSegmentStatus(lenindex)           !=
SegmentStatus::Destroyed) {
                targetShip->takeDamage(lenindex);
                if (manager.allShipDestroyed()) {
                    std::cout << "Все корабли уничтожены. Спасибо за игру!" <<
std::endl;
                    exit(0);
                }
                return false;
            }
        }
}
```

DoubleDamage.h

```
    #ifndef DOUBLEDAMAGE_H
    #define DOUBLEDAMAGE_H

    #include "Ability.h"
```

```
class DoubleDamage : public Ability {
public:
    void printAbility() override;
    bool use(GameField& gameField, ManagerShip& manager) override;
};


#endif
```

DoubleDamage.cpp

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



void DoubleDamage::printAbility() {
    std::cout << "Была выбрана способность: Двойной урон" << '\n' <<
"Введите координаты" << std::endl;
}


bool DoubleDamage::use(GameField& gameField, ManagerShip& manager) {
    return true;
}
```

Scanner.h

```
#ifndef SCANNER_H
#define SCANNER_H

#include "Ability.h"



class Scanner : public Ability {
public:
    void printAbility() override;
    bool use(GameField& gameField, ManagerShip& manager) override;
};


#endif
```

Scanner.cpp

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



void Scanner::printAbility() {
    std::cout << "Была выбрана способность: Сканер" << std::endl;
}


bool Scanner::use(GameField& gameField, ManagerShip& manager) {
    int x, y;
    std::cout << "Введите координаты верхнего левого угла квадрата 2x2" <<
std::endl;
```

```
            x = safeInput();
            y = safeInput();
            if (x < 0 || y < 0 || x >= gameField.getWidth() - 1 || y >=
gameField.getHeight() - 1) {
                std::cout << "Координаты вне границ поля" << std::endl;
                return false;
            }

            for (int dy = 0; dy < 2; dy++) {
                for (int dx = 0; dx < 2; dx++) {
                    int newX = x + dx;
                    int newY = y + dy;

                    if (gameField.getCellStatus(newY, newX) == CellStatus::Ship) {
                        std::cout << "Был найден корабль" << std::endl;
                        return false;
                    }
                }
            }

            std::cout << "Не был найден корабль" << std::endl;
            return false;
}
```

Player.h

```
      #ifndef PLAYER_H
      #define PLAYER_H

      #include <iostream>
      #include "GameField.hpp"
      #include "ManagerShip.hpp"
      #include "AbilityManager.h"



      class Player {
      private:
          GameField gameField;
          ManagerShip managerShip;
          AbilityManager abilityManager;
          ManagerShip initializeManagerShip();
          GameField initializeGameField();
          void placeShips();
      public:
          Player();
          void printField() const;
          void movePlayer();
      };


#endif
```

Player.cpp

```
      #include "Player.h"
      #include "BattleshipException.h"
      #include "SafeInput.h"


      Player::Player():                          gameField(initializeGameField()),
managerShip(initializeManagerShip()) {
          placeShips();
```

```
    }


    void Player::printField() const {
        for (int y = 0; y < gameField.getHeight(); y++) {
            for (int x = 0; x < gameField.getWidth(); x++) {
                if (gameField.getCellStatus(y, x) == CellStatus::Empty) {
                    std::cout << '[' << ' ' << ']';
                }    else    if    (gameField.getStatusShipCell(y,    x)    ==
SegmentStatus::Damaged) {
                    std::cout << '[' << '&' << ']';
                }    else    if    (gameField.getStatusShipCell(y,    x)    ==
SegmentStatus::Destroyed) {
                    std::cout << '[' << 'x' << ']';
                } else {
                    std::cout << '[' << gameField.getCellIndex(y, x) << ']';
                }
            }
            std::cout << std::endl;
        }
    }


    void Player::movePlayer() {
        int x, y;
        bool flagDoubleDamage = false;
        while (true) {
            std::cout   <<   "\nВведите   -1,   если   хотите   воспользоваться
способностью \n"
                         "Введите -2, если хотите выйти из игры \n"
                         "Иначе введите координаты для выстрела" << std::endl;
            x = safeInput();
            if (x == -3) {
                printField();
                std::cout << "Можете продолжить ввод" << std::endl;
                x = safeInput();
            }
            if (x == -1) {
                std::unique_ptr<Ability> ability;
                try {
                    ability = abilityManager.getAbility();
                    ability->printAbility();
                    flagDoubleDamage = ability->use(gameField, managerShip);
                } catch (const NoAbilitiesError& e) {
                    std::cout << e.what() << std::endl;
                }
                std::cout  <<  '\n'  <<  "Введите  координаты  для  выстрела:"  <<
std::endl;
                x = safeInput();
            } else if (x == -2) {
                std::cout << "Спасибо за игру!" << std::endl;
                break;
            }
            y = safeInput();
            while (true) {
                try {
                    if (!gameField.checkCoordAtack(y, x)) {
                        throw OutOfBoundsAttackError();
                    }
                    break;
                } catch (const OutOfBoundsAttackError& e) {
                    std::cout  <<  e.what()  <<  "\nВведите  координаты  повторно:
";
```

```
                    x = safeInput();
                    y = safeInput();
                }
            }
            if (flagDoubleDamage) {
                if (gameField.attackCell(y, x)){
                    abilityManager.addAbility();
                    std::cout << "Была добавлена способность" << std::endl;
                }
                flagDoubleDamage = false;
            }
            if (gameField.attackCell(y, x)){
                abilityManager.addAbility();
                std::cout << "Была добавлена способность" << std::endl;
            }
            if (managerShip.allShipDestroyed()) {
                std::cout << "Все корабли уничтожены. Спасибо за игру!" <<
std::endl;
                return;
            }
        }
    }


    ManagerShip Player::initializeManagerShip() {
        std::cout << "Введите количество кораблей для соответвующей длины:" <<
std::endl;
        std::vector<std::pair<int, int>> lenAndCount(4);
        for (int i = 0; i < 4; i++) {
            lenAndCount[i].first = i + 1;
            std::cout << "Для длины " << i + 1 << ": ";
            lenAndCount[i].second = safeInput();
        }
        return ManagerShip(lenAndCount);
    }

    GameField Player::initializeGameField() {
        std::cout << "Введите размеры поля: ";
        int width, height;
        while (true) {
            height = safeInput();
            width = safeInput();
            if (height > 0 && width > 0) {
                break;
            }
            std::cout << "Размеры поля должны быть положительными. Попробуйте
снова: ";
        }
        return {width, height};
    }


    void Player::placeShips() {
        for (int i = 0; i < managerShip.getCountShip(); i++) {
            std::cout << "Для расстановки кораблей на поле введите координаты
и ориентацию корабля длины: "
                      << managerShip.getShip(i)->getLength() << std::endl;
            int x, y, isHoriz;
            x = safeInput();
            y = safeInput();
            isHoriz = safeInput();
            gameField.placeShip(managerShip.getShip(i), y, x, isHoriz);
        }
```

```
}
```

Ship.hpp

```
#ifndef SHIP_HPP
#define SHIP_HPP

#include <vector>



enum class SegmentStatus {
    Whole,
    Damaged,
    Destroyed
};


class Ship{
public:
    Ship(int length, bool ishorizontally);
    void setHorizontally(bool isHorisontally);
    bool getHorizontally() const;
    int getLength() const;
    SegmentStatus getSegmentStatus(int i);
    bool takeDamage(int indexsegment);
    bool isDestroyed() const;
    ~Ship();
private:
    int length;
    bool horizontally;
    std::vector<SegmentStatus> segments;
};


#endif
```

Ship.cpp

```
#include <iostream>
#include "Ship.hpp"



Ship::Ship(int len, bool ishorizontally) : length(len),
horizontally(ishorizontally), segments(len, SegmentStatus::Whole) {}


bool Ship::takeDamage(int indexSegment) {
    switch (segments[indexSegment]) {
        case SegmentStatus::Whole:
            segments[indexSegment] = SegmentStatus::Damaged;
            std::cout << "Сегмент корабля был поврежден" << std::endl;
            break;
        case SegmentStatus::Damaged:
            segments[indexSegment] = SegmentStatus::Destroyed;
            std::cout << "Сегмент корабля был уничтожен" << std::endl;
            if (isDestroyed()) {
                std::cout << "Корабль уничтожен" << std::endl;
                return true;
            }
            break;
        case SegmentStatus::Destroyed:
```

```
                std::cout << "Этот сегмент корабля уже уничтожен" <<
std::endl;
                break;
            }
        return false;
    }


    SegmentStatus Ship::getSegmentStatus(int i) {
        return segments[i];
    }


    bool Ship::isDestroyed() const {
        for (const auto &segment: segments)
            if (segment != SegmentStatus::Destroyed) {
                return false;
            }
        return true;
    }



    void Ship::setHorizontally(bool isHorisontally){
        horizontally = isHorisontally;
    }


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


    bool Ship::getHorizontally() const{
        return horizontally;
    }


Ship::~Ship() = default;
```

FieldCell.hpp

```
    #ifndef FIELD_CELL_HPP
    #define FIELD_CELL_HPP

    #include <memory>
    #include "Ship.hpp"



    enum class CellStatus {
        Unknown,
        Empty,
        Ship
    };


    class FieldCell{
    private:
        Ship* ship;
        CellStatus status;
        int indexseg;
    public:
```

```
        explicit FieldCell(Ship* ship = nullptr, CellStatus cellstatus =
CellStatus::Empty);
        void setShip(Ship* ship);
        void setStatus(CellStatus cellStatus);
        void setIndexseg(int index);
        CellStatus getStatus() const;
        Ship* getShip() const;
        int getIndexseg() const;
    };


#endif
```

FieldCell.cpp

```
    #include "FieldCell.hpp"



    FieldCell::FieldCell(Ship* ship, CellStatus status) : ship(ship),
status(status){}


    void FieldCell::setShip(Ship* ship) {
        this->ship = ship;
    }


    void FieldCell::setStatus(CellStatus cellStatus) {
        this->status = cellStatus;
    }


    void FieldCell::setIndexseg(int index) {
        indexseg = index;
    }


    CellStatus FieldCell::getStatus() const {
        return status;
    }


    Ship* FieldCell::getShip() const {
        return ship;
    }


    int FieldCell::getIndexseg() const {
        return indexseg;
}
```

RandOnSection.h

```
        #ifndef RAND_ON_SECTION_H
    #define RAND_ON_SECTION_H

    #include <random>



    int randOnSection(int begin, int end);
```

```
#endif
```

RandOnSection.cpp

```
      #include "RandOnSection.h"



      int randOnSection(int begin, int end) {
          std::random_device rd;
          std::mt19937 generator(rd());
          std::uniform_int_distribution<int> distribution(begin, end - 1);
          return distribution(generator);
}
```

ManagerShip.hpp

```
      #ifndef MANAGERSHIP_HPP
      #define MANAGERSHIP_HPP

      #include <vector>
      #include <memory>
      #include "Ship.hpp"



      class ManagerShip {
      public:
          explicit    ManagerShip(const    std::vector<std::pair<int,    int>>&
lenAndCount);
          void registerShip(std::pair<int, int>);
          int getCountShip() const;
          Ship* getShip(int index);
          void deleteShip(int index);
          bool allShipDestroyed();
          ~ManagerShip();
      private:
          std::vector<Ship*> ships;
      };



#endif
```

ManagerShip.cpp

```
      #include "ManagerShip.hpp"



      ManagerShip::ManagerShip(const      std::vector<std::pair<int,      int>>&
lensAndCounts) {
          for (const auto lensAndCount : lensAndCounts) {
              registerShip(lensAndCount);
          }
      }


      void ManagerShip::registerShip(std::pair<int, int> lenAndCount) {
          int len = lenAndCount.first;
          int count = lenAndCount.second;
          for (int i = 0; i < count; i++) {
              Ship* ship = new Ship(len, true);
```

```
            ships.push_back(ship);
        }
    }


    Ship* ManagerShip::getShip(int i) {
        return ships[i];
    }


    int ManagerShip::getCountShip() const {
        return (int)ships.size();
    }


    bool ManagerShip::allShipDestroyed() {
        for (int i = 0; i < getCountShip(); i++) {
            if (!ships[i]->isDestroyed()) {
                return false;
            }
        }
        return true;
    }

    ManagerShip::~ManagerShip(){
        for (auto x : ships)
            delete x;
}
```

GameField.hpp

```
    #ifndef GAMEFIELD_HPP
    #define GAMEFIELD_HPP

    #include <vector>
    #include <memory>
    #include "Ship.hpp"
    #include "FieldCell.hpp"



    class GameField {
    public:
        GameField(int width, int height);
        void placeShip(Ship*, int y, int x, int isHorizontally);
        bool attackCell(int y, int x);
        int getWidth() const;
        int getHeight() const;
        int getCellIndex(int y, int x) const;
        SegmentStatus getStatusShipCell(int y, int x) const;
        CellStatus getCellStatus(int y, int x) const;
        bool checkCoordAtack(int y, int x);

        GameField(const GameField& other);
        GameField(GameField&& other) noexcept;
        GameField& operator=(const GameField& other);
        GameField& operator=(GameField&& other) noexcept;
        ~GameField();
    private:
        std::vector<std::vector<FieldCell*>> field;
        int width, height;
        bool canPlaceShip(int y, int x, int length, bool isHorizontally)
const;
```

```
};

#endif
```

GameField.cpp

```
#include <iostream>
#include <memory>
#include "GameField.hpp"
#include "BattleshipException.h"


GameField::GameField(int width, int height) : width(width),
height(height), field(height, std::vector<FieldCell*>(width)) {
    for (int y = 0; y < height; ++y) {
        for (int x = 0; x < width; ++x) {
            FieldCell* cell = new FieldCell();
            field[y][x] = cell;
        }
    }
}


bool GameField::checkCoordAtack(int y, int x) {
    if (x < 0 || y < 0 || x >= width || y >= height) {
        return false;
    }
    return true;
}



void GameField::placeShip(Ship* ship, int y, int x, int isHorizontally) {
    ship->setHorizontally(isHorizontally > 0);
    try {
        if (!canPlaceShip(y, x, ship->getLength(), ship-
>getHorizontally())) {
            throw ShipPlacementError();
        }
    } catch (const ShipPlacementError& e) {
        std::cout << e.what() << std::endl;
    }
    for (int i = 0; i < ship->getLength(); i++) {
        int currentY = y + (ship->getHorizontally() ? 0 : i);
        int currentX = x + (ship->getHorizontally() ? i : 0);
        field[currentY][currentX]->setStatus(CellStatus::Ship);
        field[currentY][currentX]->setShip(ship);
        field[currentY][currentX]->setIndexseg(i);
    }

}


bool GameField::attackCell(int y, int x) {
    if (getCellStatus(y, x) == CellStatus::Ship) {
        return field[y][x]->getShip()->takeDamage(field[y][x]-
>getIndexseg());
    }
    std::cout << "По этим координатам корабля нет" << std::endl;
    return false;
}
```

```
      bool GameField::canPlaceShip(int y, int x, int length, bool
isHorizontally) const {
          for (int i = 0; i < length; ++i) {
              int checkX = x + (isHorizontally ? i : 0);
              int checkY = y + (isHorizontally ? 0 : i);
              if ((checkX < 0) || (checkX >= getWidth()) || (checkY < 0) ||
(checkY >= getHeight())) {
                  return false;
              }
          }
          int startX = x - 1;
          int endX = isHorizontally ? x + length : x + 1;
          int startY = y - 1;
          int endY = isHorizontally ? y + 1 : y + length;
          for (int dy = startY; dy <= endY; dy++) {
              for (int dx = startX; dx <= endX; dx++) {
                  if (dy >= 0 && dx >= 0 && dy < getHeight() && dx < getWidth())
{
                      if (getCellStatus(dy, dx) == CellStatus::Ship) {
                          return false;
                      }
                  }
              }
          }
          return true;
      }


      int GameField::getWidth() const {
          return width;
      }


      int GameField::getHeight() const {
          return height;
      }


      CellStatus GameField::getCellStatus(int y, int x) const {
          return field[y][x]->getStatus();
      }


      int GameField::getCellIndex(int y, int x) const {
          return field[y][x]->getIndexseg();
      }

      GameField::GameField(const GameField& other) = default;


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


      GameField& GameField::operator=(const GameField& other) {
          if (this != &other) {
              field = (other.field);
          }
          return *this;
      }
```

```
      GameField& GameField::operator=(GameField&& other) noexcept {
          if (this != &other) {
              field = std::move(other.field);
          }
          return *this;
      }


      SegmentStatus GameField::getStatusShipCell(int y, int x) const {
          return           field[y][x]->getShip()->getSegmentStatus(field[y][x]-
>getIndexseg());
      }


      GameField::~GameField() {
          for(const auto& y : field) {
              for (auto x : y) {
                  delete x;
              }
          }
}
```