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

ОТЧЕТ

по лабораторной работе №1
по дисциплине «Объектно-ориентированное программирование»
Тема: Создание классов

Студент гр. 3341 Шаповаленко Е.В.

Преподаватель Жангиров Т.Р.

Санкт-Петербург
2024

Цель работы

Цель работы – создание классов корабля, менеджера кораблей и поля и связей между ними.

Для достижения поставленной цели требуется:

1. Ознакомиться с понятием класс и его структурой;
2. Реализовать указанные классы согласно требованиям из условия;
3. Написать программу, в которой бы проводилась проверка работоспособности классов.

Задание

А) Создать класс корабля, который будет размещаться на игровом поле. Корабль может иметь длину от 1 до 4, а также может быть расположен вертикально или горизонтально. Каждый сегмент корабля может иметь три различных состояния: целый, поврежден, уничтожен. Изначально у корабля все сегменты целые. При нанесении 1 урона по сегменту, он становится поврежденным, а при нанесении 2 урона по сегменту, уничтоженным. Также добавить методы для взаимодействия с кораблем.

Б) Создать класс менеджера кораблей, хранящий информацию о кораблях. Данный класс в конструкторе принимает количество кораблей и их размеры, которые нужно расставить на поле.

В) Создать класс игрового поля, которое в конструкторе принимает размеры. У поля должен быть метод, принимающий корабль, координаты, на которые нужно поставить, и его ориентацию на поле. Корабли на поле не могут соприкасаться или пересекаться. Для игрового поля добавить методы для указания того, какая клетка атакуется. При попадании в сегмент корабля изменения должны отображаться в менеджере кораблей.

Каждая клетка игрового поля имеет три статуса:

- неизвестно (изначально вражеское поле полностью неизвестно),
- пустая (если на клетке ничего нет)
- корабль (если в клетке находится один из сегментов корабля).

Для класса игрового поля также необходимо реализовать конструкторы копирования и перемещения, а также соответствующие им операторы присваивания.

Примечания:

- Не забывайте для полей и методов определять модификаторы доступа
- Для обозначения переменной, которая принимает небольшое ограниченное количество значений, используйте enum
- Не используйте глобальные переменные

- При реализации копирования нужно выполнять глубокое копирование
- При реализации перемещения, не должно быть лишнего копирования
- При выделении памяти делайте проверку на переданные значения
- У поля не должно быть методов, возвращающих указатель на поле в явном виде, так как это небезопасно

Выполнение работы

Реализован класс *Ship*, моделирующий корабль в игре, представляя его размеры, состояние самого корабля и его сегментов:

1) Конструктор *Ship(int size)* позволяет создать корабль с определенным размером. Согласно условию размер корабля может варьироваться от 1 до 4.

2) Метод *getSize()* позволяет получить размер корабля, что может быть необходимо для итерации по его сегментам.

3) Метод *getSegmentStatus(int index)* возвращает состояние сегмента по индексу (*intact*, *damaged*, *destroyed*). Состояние сегмента зависит от его здоровья. Это необходимо для вывода информации пользователю.

4) Метод *getShipStatus()* возвращает состояние корабля (*dead*, *alive*). Это необходимо для вывода информации пользователю.

5) Метод *damageSegment(int index, int damage)* симулирует нанесение определенного урона выбранному сегменту.

6) Метод *healSegment(int index, int heal)* симулирует лечение на определенную величину выбранного сегмента. Данный метод противоположен по действию предыдущему.

7) Главным полем *Ship* является *vector<ShipSegment> segments_*. Оно хранит сегменты корабля. Для простоты работы реализовано поле *size_* для хранения размера корабля.

8) Переменные *kMinSize* и *kMaxSize* задают минимально и максимально допустимые размеры корабля соответственно.

Таким образом, класс *Ship* моделирует корабль в целом. Для моделирования его сегментов в классе *Ship* реализован подкласс *ShipSegment*:

1) Поле *health_* хранит здоровье сегмента. При создании объекта класса полю присваивается значение максимального здоровья *kMaxHealth*.

2) Метод *takeDamage(int damage)* симулирует нанесение определенного урона выбранному сегменту. Урон не может быть отрицательным.

3) Метод *takeHeal(int heal)* симулирует лечение на определенную величину выбранного сегмента. Величина лечения не может быть отрицательной. Данный метод противоположен по действию предыдущему.

4) Метод *getStatus()* возвращает состояние сегмента (*intact*, *damaged*, *destroyed*). Состояние сегмента зависит от значения *health_*. Это необходимо для вывода информации пользователю.

Реализован класс ShipManager. Этот класс создает корабли и расставляет их на указанном поле. Также через него можно получить информацию о состоянии кораблей. Реализованы следующие поля и методы:

1) Конструктор *ShipManager(std::initializer_list<int> ship_sizes)* создает корабли в соответствии с поданными на вход размерами и сохраняет их в поле *list<pair<Ship, bool>> ships_*. В этом поле хранятся пары: корабль и был ли он размещен на поле.

2) Метод *addShip(int ship_size)* создает корабль в соответствии с поданным на вход размером и сохраняет его в поле *list<pair<Ship, bool>> ships_*.

3) Метод *placeShip(Field& field, int index, int x, int y, ShipOrientation orientation)* размещает корабль по индексу *index* в *ships_* на игровом поле *field* в соответствии с координатами *x* и *y* и ориентацией корабля *orientation*. Корабль отмечается размещенным на игровом поле в *ships_*.

4) Методы *getUsedShips()* и *getUnusedShips()* позволяют получить векторы, содержащие копии использованных и неиспользованных кораблей соответственно. Это может понадобиться для отрисовки кораблей в GUI/CLI.

5) Методы *getUsedShipsSize()* и *getUnusedShipsSize()* позволяют получить количество использованных и неиспользованных кораблей соответственно, что может быть необходимо для итерации по ним.

6) Отладочный метод *printShips()* выводит на экран *ships_*. С помощью него можно отслеживать какие корабли были размещены на поле, а какие нет, и состояние их сегментов.

Таким образом, при помощи класса *ShipManager* реализовано взаимодействие между кораблями и полем, а также в будущем между пользователем и кораблями.

Реализован класс *Field*, моделирующий игровое поле, представляя его размеры и состояние его клеток:

1) Конструктор *Field(int size_x, int size_y)* создает игровое поле в соответствии с размерами *size_x* и *size_y* и сохраняет его в поле *vector<vector<FieldCell>> field_*. Также сохраняются размеры в *size_x_* и *size_y_* соответственно.

2) Реализованы конструкторы копирования и перемещения и соответствующие им операторы.

3) Метод *placeShip(Ship* ship, int x, int y, ShipOrientation orientation)* размещает на игровом поле переданный корабль ship в соответствии с координатами *x* и *y* и ориентацией корабля *orientation*. Корабли не могут выходить за рамки игрового поля и пересекаться или касаться друг друга.

4) С помощью метода *attackCell(int x, int y, int damage)* атакуется выбранная клетка игрового поля.

5) Отладочный метод *printField()* выводит на экран *field_*. С помощью него можно отслеживать состояние клеток игрового поля.

Таким образом, класс *Field* моделирует игровое полке в целом. Для моделирования его клеток в классе *Field* реализован подкласс *FieldCell*:

1) Конструктор *FieldCell()* инициализирует поля клетки игрового поля *status_*, *ship_* и *ship_segment_index_* значениями *unknown*, *nullptr* и -1 соответственно. Поле *status_* хранит состояние клетки (*unknown*, *empty*, *ship*). Поля *ship_* и *ship_segment_index_* хранят указатель на корабль и индекс сегмента в этом корабле соответственно. Они нужны для реализации атаки корабля с игрового поля.

2) Метод *getStatus()* возвращает состояние клетки (*unknown*, *empty*, *ship*), в зависимости от того, была ли атакована клетка или нет и есть ли на ней корабль или нет.

3) Метод *setStatus(FieldCellStatus status)* задает статус клетке игрового поля.

4) С помощью метода *attackCell(int damage)* атакуется сегмент корабля, расположенный на данной клетке игрового поля.

5) Метод *isShip()* возвращает информацию о том, есть ли на данной клетке игрового поля корабль или нет.

6) Метод *setShipSegment(Ship* ship, int index)* задает поля *ship_* и *ship_segment_index_*.

7) Метод *getShipSegmentStatus()* возвращает состояние сегмента корабля, который расположен в данной клетке игрового поля.

Таким образом, при помощи класса *Field* реализовано взаимодействие между кораблями и полем.

В местах, где может возникнуть ошибка (выход за допустимые границы, недопустимое значение переменной, нарушение логики программы), программа завершает работу с сообщением о соответствующей ошибке.

Ниже представлена UML-диаграмма реализованных в данной работе классов:

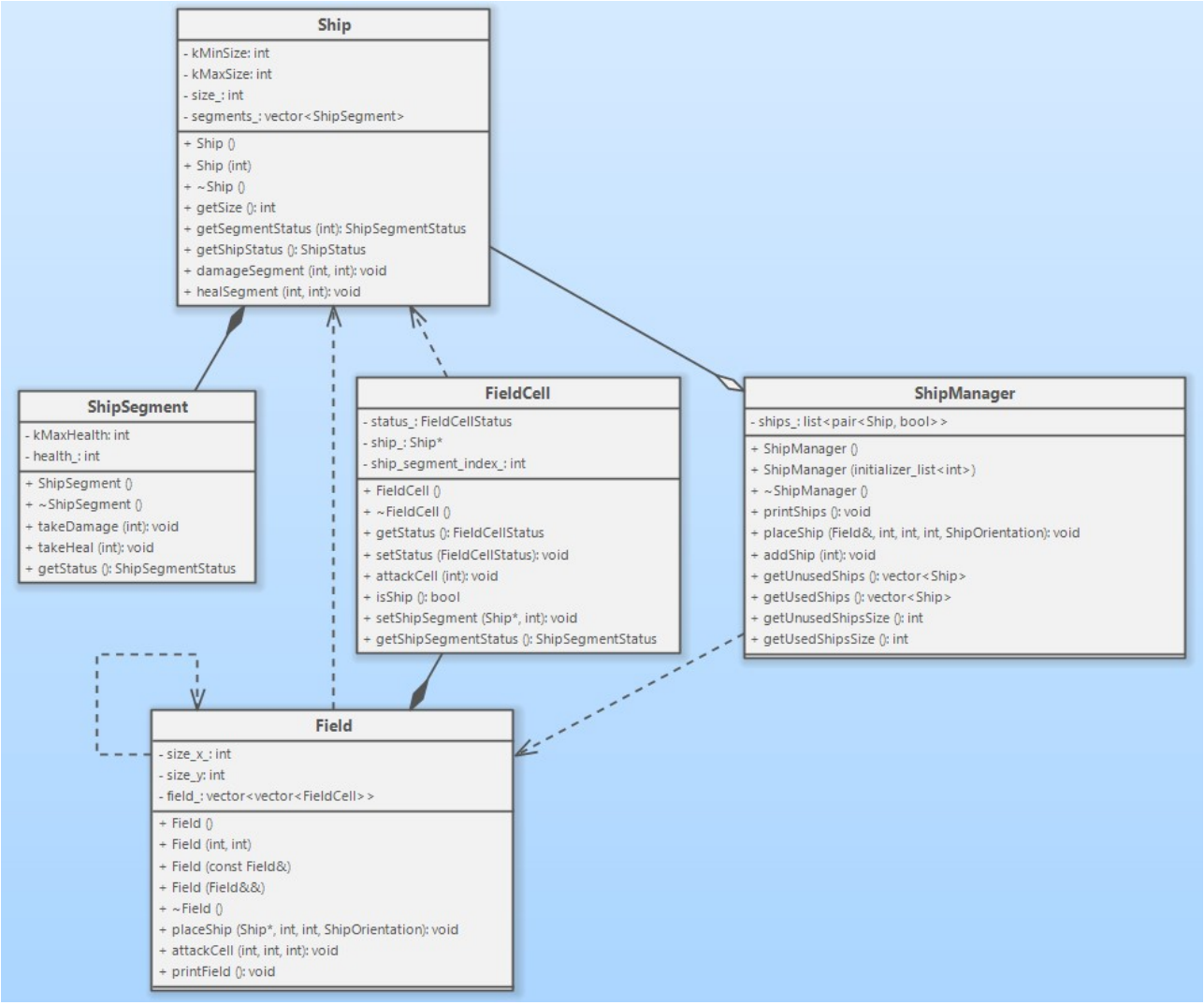

Рисунок 1: UML-диаграмма классов

Разработанный программный код см. в приложении А.

Тестирование программы см. в приложении Б.

Выводы

Цель работы – создание классов корабля, менеджера кораблей и поля и связей между ними – была достигнута.

В ходе выполнения работы были выполнены следующие задачи:

1. Было проведено ознакомление с понятием класса и его структурой;
2. Были реализованы указанные классы согласно требованиям из условия;
3. Написана программа, в которой проводится проверка работоспособности классов.

.

ПРИЛОЖЕНИЕ А
ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: ship.h

```
#ifndef SHIP
#define SHIP

#include <vector>
#include <stdexcept>

enum class ShipStatus : int
{
    dead,
    alive
};

enum class ShipSegmentStatus : int
{
    destroyed,
    damaged,
    intact
};

class Ship
{
public:
    Ship();

    explicit Ship(int size);

    ~Ship();

    int getSize() const noexcept;

    ShipSegmentStatus getSegmentStatus(int index) const;

    ShipStatus getShipStatus() const noexcept;

    void damageSegment(int index, int damage);

    void healSegment(int index, int heal);

private:
    class ShipSegment
    {
    public:
        ShipSegment();

        ~ShipSegment();

        void takeDamege(int damage);
```

```
        void takeHeal(int heal);

        ShipSegmentStatus getStatus() const noexcept;

    private:
        int kMaxHealth = 2;
        int health_;
    };

    int kMinSize = 1;
    int kMaxSize = 4;
    int size_;
    std::vector<ShipSegment> segments_;
};

#endif
```

Название файла: ship.cpp

```
#include "ship.h"

#include <vector>
#include <stdexcept>

Ship::Ship() = default;

Ship::Ship(int size)
{
    if (size < kMinSize || size > kMaxSize) {
        throw std::logic_error("Ship size can be from 1 to 4");
    }

    size_ = size;

    for (int i = 0; i < size_; i++) {
        segments_.push_back(ShipSegment());
    }
}

Ship::~Ship()
{
    segments_.clear();
}

int Ship::getSize() const noexcept
{
    return size_;
}

ShipSegmentStatus Ship::getSegmentStatus(int index) const
{
    if (index < 0 || index >= size_) {
        throw std::out_of_range("Ship segment index out of range");
    }

    return segments_[index].getStatus();
}
```

```
ShipStatus Ship::getShipStatus() const noexcept
{
    int dead_flag = 1;
    for (int i = 0; i < size_; i++) {
        if (segments_[i].getStatus() != ShipSegmentStatus::destroyed) {
            dead_flag = 0;
            break;
        }
    }

    if (dead_flag) {
        return ShipStatus::dead;
    } else {
        return ShipStatus::alive;
    }
}

void Ship::damageSegment(int index, int damage)
{
    if (index < 0 || index >= size_) {
        throw std::out_of_range("Ship segment index out of range");
    }

    segments_[index].takeDamege(damage);
}

void Ship::healSegment(int index, int heal)
{
    if (index < 0 || index >= size_) {
        throw std::out_of_range("Ship segment index out of range");
    }

    segments_[index].takeHeal(heal);
}

Ship::ShipSegment::ShipSegment()
{
    health_ = kMaxHealth;
}

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

void Ship::ShipSegment::takeDamege(int damage)
{
    if (damage < 0) {
        throw std::invalid_argument("Damage can't be negative");
    }

    health_ = std::max(0, health_ - damage);
}

void Ship::ShipSegment::takeHeal(int heal)
{
    if (heal < 0) {
        throw std::invalid_argument("Heal can't be negative");
    }
```

```
        health_ = std::min(kMaxHealth, health_ + heal);
    }

    ShipSegmentStatus Ship::ShipSegment::getStatus() const noexcept
    {
        if (health_ == kMaxHealth) {
            return ShipSegmentStatus::intact;
        } else if (health_ == 0) {
            return ShipSegmentStatus::destroyed;
        } else {
            return ShipSegmentStatus::damaged;
        }
    }
```

Название файла: shipManager.h

```
    #ifndef SHIP_MANAGER
    #define SHIP_MANAGER

    #include <iostream>
    #include <initializer_list>
    #include <vector>
    #include <list>
    #include <utility>
    #include "ship.h"
    #include "field.h"

    class ShipManager
    {
    public:
        ShipManager();

        explicit ShipManager(std::initializer_list<int> ship_sizes);

        ~ShipManager();

        void printShips() const noexcept;

        void placeShip(Field& field, int index, int x, int y,
ShipOrientation orientation);

        void addShip(int ship_size);

        std::vector<Ship> getUnusedShips() const noexcept;

        std::vector<Ship> getUsedShips() const noexcept;

        int getUnusedShipsSize() const noexcept;

        int getUsedShipsSize() const noexcept;

    private:
        std::list<std::pair<Ship, bool>> ships_;
    };

    #endif
```

Название файла: shipManager.cpp

```
#include "shipManager.h"
#include "ship.h"
#include "field.h"

#include <iostream>
#include <initializer_list>
#include <vector>
#include <utility>

ShipManager::ShipManager() = default;

ShipManager::ShipManager(std::initializer_list<int> ship_sizes)
{
    for (auto ship : ship_sizes) {
        ships_.push_back({Ship(ship), false});
    }
}

ShipManager::~ShipManager()
{
    ships_.clear();
}

void ShipManager::printShips() const noexcept
{

    int counter = 0;
    for (auto ship: ships_) {
        std::cout << "Ship " << counter++ << ": ";
        for (int i = 0; i < ship.first.getSize(); i++) {
            if          (ship.first.getSegmentStatus(i)          ==
ShipSegmentStatus::intact) {
                std::cout << "[2]";
            }    else    if    (ship.first.getSegmentStatus(i)    ==
ShipSegmentStatus::damaged) {
                std::cout << "[1]";
            } else {
                std::cout << "[0]";
            }
        }

        if (ship.second) {
            std::cout << " Used" << "\n";
        } else {
            std::cout << " Unused" << "\n";
        }
    }

    if (ships_.size() == 0) {
        std::cout << "None" << "\n";
    }

    std::cout << "\n";

    return;
}
```

```
    void ShipManager::placeShip(Field& field, int index, int x, int y,
ShipOrientation orientation)
    {
        if (index < 0 || index >= ships_.size()) {
            throw std::out_of_range("Index out of range");
        }

        std::list<std::pair<Ship, bool>>::iterator it = ships_.begin();
        std::advance(it, index);

        if (it->second) {
            throw std::logic_error("Ship was already placed to field");
        }

        it->second = true;
        field.placeShip(&(it->first), x, y, orientation);
    }

    void ShipManager::addShip(int ship_size)
    {
        ships_.push_back({Ship(ship_size), false});
    }

    std::vector<Ship> ShipManager::getUnusedShips() const noexcept
    {
        std::vector<Ship> unused_ships;
        for(auto ship: ships_)
            if(ship.second == false)
                unused_ships.push_back(ship.first);
        return unused_ships;
    }

    std::vector<Ship> ShipManager::getUsedShips() const noexcept
    {
        std::vector<Ship> used_ships;
        for(auto ship: ships_)
            if(ship.second == false)
                used_ships.push_back(ship.first);
        return used_ships;
    }

    int ShipManager::getUnusedShipsSize() const noexcept
    {
        int unused_ships_num;
        for(auto ship: ships_)
            if(ship.second == false)
                unused_ships_num++;
        return unused_ships_num;
    }

    int ShipManager::getUsedShipsSize() const noexcept
    {
        int used_ships_num;
        for(auto ship: ships_)
            if(ship.second == false)
                used_ships_num++;
        return used_ships_num;
    }
```

Название файла: field.h

```
#ifndef FIELD
#define FIELD

#include <iostream>
#include <vector>
#include <algorithm>
#include <stdexcept>
#include "ship.h"

enum class ShipOrientation : int
{
    horizontal,
    vertical
};

enum class FieldCellStatus : int
{
    unknown,
    empty,
    ship
};

class Field
{
public:
    Field();

    explicit Field(int size_x, int size_y);

    Field(const Field& other);

    Field& operator=(const Field& other);

    Field(Field&& other);

    Field& operator=(Field&& other);

    ~Field();

    void placeShip(Ship* ship, int x, int y, ShipOrientation
orientation);

    void attackCell(int x, int y, int damage);

    void printField() const noexcept;

private:
    class FieldCell
    {
    public:
        FieldCell();

        ~FieldCell() = default;

        FieldCellStatus getStatus() const noexcept;
```

```
            void setStatus(FieldCellStatus status) noexcept;

            void attackCell(int damage);

            bool isShip() const noexcept;

            void setShipSegment(Ship* ship, int index) noexcept;

            ShipSegmentStatus getShipSegmentStatus() const noexcept;

        private:
            FieldCellStatus status_;
            Ship* ship_;
            int ship_segment_index_;
        };

        int size_x_;
        int size_y_;
        std::vector<std::vector<FieldCell>> field_;
    };

    #endif
```

Название файла: field.cpp

```
    #include "field.h"
    #include "ship.h"

    #include <iostream>
    #include <vector>
    #include <algorithm>
    #include <stdexcept>

    Field::Field() = default;

    Field::Field(int size_x, int size_y)
    {
        if (size_x <= 0 || size_y <= 0) {
            throw std::invalid_argument("Field size must be grater than
0");
        }
        size_x_ = size_x;
        size_y_ = size_y;

        field_.resize(size_x_);
```

```
    for (int i = 0; i < size_x_; i++) {
        for (int j = 0; j < size_y_; j++) {
            field_[i].push_back(FieldCell());
        }
    }
}

Field::Field(const Field& other) :
    Field(other.size_x_, other.size_x_)
{}

Field& Field::operator=(const Field& other)
{
    if (this == &other) {
        return *this;
    }

    size_x_ = other.size_x_;
    size_y_ = other.size_y_;

    field_.resize(size_x_);

    for (int i = 0; i < size_x_; i++) {
        for (int j = 0; j < size_y_; j++) {
            field_[i].push_back(FieldCell());
        }
    }

    return *this;
}

Field::Field(Field&& other) :
    field_(std::move(other.field_)),
    size_x_(std::move(other.size_x_)),
    size_y_(std::move(other.size_y_))
{}

Field& Field::operator=(Field&& other)
{
```

```
        if (this == &other) {
            return *this;
        }

        field_ = std::move(other.field_);
        size_x_ = std::move(other.size_x_);
        size_y_ = std::move(other.size_y_);

        return *this;
    }

    Field::~Field()
    {
        for (int i = 0; i < size_x_; i++) {
            field_[i].clear();
        }

        field_.clear();
    }

    void Field::placeShip(Ship* ship, int x, int y, ShipOrientation
orientation)
    {
        if (ship == nullptr) {
            throw std::invalid_argument("Ship pointer is nullptr");
        }

        int offset_x, offset_y;
        if (orientation == ShipOrientation::horizontal) {
            offset_x = ship->getSize() - 1;
            offset_y = 0;
        } else {
            offset_x = 0;
            offset_y = ship->getSize() - 1;
        }

        if (x < 0 || x >= size_x_ - offset_x || y < 0 || y >= size_y_ -
offset_y) {
            throw std::out_of_range("Ship coordinates out of range");
```

```
        }

        for (int i = x - 1; i < x + offset_x + 1; i++) {
            if (i >= 0 && i < size_x_) {
                for (int j = y - 1; j < y + offset_y + 1; j++) {
                    if (j >= 0 && j < size_y_) {
                        if (field_[i][j].isShip()) {
                            throw   std::logic_error("Ships   may   not
contact each other");
                        }
                    }
                }
            }
        }

        int segment_index = 0;
        for (int i = x; i < x + offset_x + 1; i++) {
            for (int j = y; j < y + offset_y + 1; j++) {
                field_[i][j].setShipSegment(ship, segment_index++);
            }
        }
    }

    void Field::attackCell(int x, int y, int damage)
    {
        if (x < 0 || x >= size_x_ || y < 0 || y >= size_y_) {
            throw std::out_of_range("Coordinates out of range");
        }

        field_[x][y].attackCell(damage);

        if (field_[x][y].isShip()) {
            field_[x][y].setStatus(FieldCellStatus::ship);
        } else {
            field_[x][y].setStatus(FieldCellStatus::empty);
        }
    }

    void Field::printField() const noexcept
```

```
    {
        for (int y = 0; y < size_y_; y++) {
            for (int x = 0; x < size_x_; x++) {
                if (field_[x][y].getStatus() ==
FieldCellStatus::unknown) {
                    std::cout << "[ ]";
                } else if (field_[x][y].getStatus() ==
FieldCellStatus::empty) {
                    std::cout << "[.]";
                } else {
                    if (field_[x][y].getShipSegmentStatus() ==
ShipSegmentStatus::intact) {
                        std::cout << "[2]";
                    } else if (field_[x][y].getShipSegmentStatus() ==
ShipSegmentStatus::damaged) {
                        std::cout << "[1]";
                    } else {
                        std::cout << "[0]";
                    }
                }
            }
            std::cout << "\n";
        }
        std::cout << "\n";
    }

    Field::FieldCell::FieldCell()
    {
        status_ = FieldCellStatus::unknown;
        ship_ = nullptr;
        ship_segment_index_ = -1;
    }

    FieldCellStatus Field::FieldCell::getStatus() const noexcept
    {
        return status_;
    }

    void Field::FieldCell::setStatus(FieldCellStatus status) noexcept
```

```
{
    status_ = status;
}

void Field::FieldCell::attackCell(int damage)
{
    if (ship_ != nullptr) {
        ship_->damageSegment(ship_segment_index_, damage);
    }
}

bool Field::FieldCell::isShip() const noexcept
{
    return ship_ != nullptr;
}

void Field::FieldCell::setShipSegment(Ship* ship, int index)
noexcept
{
    ship_ = ship;
    ship_segment_index_ = index;
}

ShipSegmentStatus Field::FieldCell::getShipSegmentStatus() const
noexcept
{
    return ship_->getSegmentStatus(ship_segment_index_);
}
```

.

ПРИЛОЖЕНИЕ Б
ТЕСТИРОВАНИЕ ПРОГРАММЫ

Название файла: main.cpp

```
#include <iostream>
#include "ship.h"
#include "shipManager.h"
#include "field.h"

int main()
{
    Ship test_ship_1(2); // OK
    Ship test_ship_2(0); // Error
    Ship test_ship_3(5); // Error

    test_ship_1.damageSegment(0, 1); // OK
    test_ship_1.damageSegment(0, -1); // Error
    test_ship_1.damageSegment(3, 1); // Error

    ShipManager manager({1, 3, 4});
    manager.printShips();

    Field field(5, 5);

    manager.placeShip(field, 1, 0, 0, ShipOrientation::horizontal);
// OK
    manager.placeShip(field, 1, 0, 3, ShipOrientation::horizontal);
// Error
    manager.placeShip(field, 2, 0, 0, ShipOrientation::vertical);
// Error
    manager.placeShip(field, 2, 3, 0, ShipOrientation::vertical);
// Error

    manager.addShip(2);

    manager.printShips();
```

```
    field.attackCell(0, 0, 1); // OK
    field.attackCell(6, 10, 1); // Error

    field.attackCell(1, 0, 0);
    field.attackCell(2, 0, 1);
    field.attackCell(3, 0, 0);
    field.attackCell(0, 1, 0);
    field.attackCell(1, 1, 0);
    field.attackCell(2, 1, 0);
    field.attackCell(3, 1, 0);

    manager.printShips();
    field.printField();
}
```

Результат работы программы: