**МИНОБРНАУКИ РОССИИ**

**Санкт-Петербургский государственный**

**электротехнический университет**

**«ЛЭТИ» им. В. И. Ульянова (Ленина)**

**Кафедра МО ЭВМ**

отчет

**по лабораторной работе №2**

**по дисциплине «Объектно-ориентированное программирование»**

**Тема: Полиморфизм**

|  |  |  |
| --- | --- | --- |
| Студент гр. 3342 |  | Лапшов К.Н. |
| Преподаватель |  | Жангиров Т.Р. |

Санкт-Петербург

2024

## **Цель работы**

Разработать систему классов для способностей в игре "Морской бой", используя полиморфизм. Классы способностей должны наследоваться от единого класса-интерфейса.

## **Задание**

1. Создать класс-интерфейс способности, которую игрок может применять. Через наследование создать 3 разные способности:
2. Двойной урон - следующая атак при попадании по кораблю нанесет сразу 2 урона (уничтожит сегмент).
3. Сканер - позволяет проверить участок поля 2х2 клетки и узнать, есть ли там сегмент корабля. Клетки не меняют свой статус.
4. Обстрел - наносит 1 урон случайному сегменту случайного корабля. Клетки не меняют свой статус.
5. Создать класс менеджера способностей. Который хранит очередь способностей, изначально игроку доступно по 1 способности в случайном порядке. Реализовать метод применения способности.
6. Реализовать функционал получения одной случайной способности при уничтожении вражеского корабля.
7. Реализуйте набор классов-исключений и их обработку для следующих ситуаций (можно добавить собственные):
8. Попытка применить способность, когда их нет
9. Размещение корабля вплотную или на пересечении с другим кораблем
10. 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 данных

## **Выполнение работы**

Класс перечисления ‘AbilityName’ содержит в себе константные переменные идентифицирующие способности.

Класс ‘AbilityInterface’ представляет класс-интерфейс доступной пользователю игроку. Используются при наследовании для реализации конкретных способностей.

Методы класса:

* virtual void applyAbility(GameField &field) – Принимает ссылку на конкретное поле и реализует логику способности
* virtual AbilityName getName() – Возвращает член перечисления названий способностей.

Класс ‘DoubleDamage’ представляет реализацию логики способности Двойного урона. Наследуется от класса-интерфейса ‘AbilityInterface’.

Поля класса:

* Point coords - Приватное поле класса, хранящее координаты для выстрела

Методы класса:

* DoubleDamage(Point coordinates) – конструктор принимающий координаты выстрела.
* void applyAbility(GameField &field) override – метод, реализующий логику способности
* AbilityName getName() override – метод, возвращающий член перечисления способности

Класс ‘Scanner’ представляет реализацию логики способности Сканера. Наследуется от класса-интерфейса ‘AbilityInterface’.

Поля класса:

* Point coords - Приватное поле класса, хранящее координаты для использования сканера
* std::function<void (bool)> trigger\_func – поле, хранящее функцию, которая возвращает реакцию сканера на нахождение корабля

Методы класса:

* Scanner(Point coordinates, std::function<void (bool)> trigger) – конструктор принимающий координаты и функцию для работы сканера.
* void applyAbility(GameField &field) override – метод, реализующий логику способности
* AbilityName getName() override – метод, возвращающий член перечисления способности

Класс ‘RandomShot’ представляет реализацию логики способности случайного попадания по случайному кораблю противника. Наследуется от класса-интерфейса ‘AbilityInterface’.

Методы класса:

* RandomShot() – конструктор по умолчанию.
* void applyAbility(GameField &field) override – метод, реализующий логику способности
* AbilityName getName() override – метод, возвращающий член перечисления способности

Для корректного взаимодействия менеджера способностей и самих способностей был реализован производственный паттерн ‘Фабрика’.

Класс ‘AbilityFactory’ – класс интерфейс для описания фабрик способностей.

Методы класса:

* virtual std::shared\_ptr<AbilityInterface> create(Point cord = {0, 0}, std::function<void (bool)> trigger = [](bool) {}) – метод, позволяющий создавать объекты определенной способности.
* AbilityName getName()– метод, возвращающий член перечисления способности

Класс ‘DoubleDamageFactory’ представляет собой фабрику способности ‘DoubleDamage’. Наследуется от класса-интерфейса ‘AbilityFactory’.

Методы класса:

* virtual std::shared\_ptr<AbilityInterface> create(Point cord = {0, 0}, std::function<void (bool)> trigger = [](bool) {}) override – метод, позволяющий создавать объекты класса способности ‘DoubleDamage’.
* AbilityName getName()– метод, возвращающий член перечисления способности

Класс ‘ScannerFactory’ представляет собой фабрику способности ‘Scanner’. Наследуется от класса-интерфейса ‘AbilityFactory’.

Методы класса:

* virtual std::shared\_ptr<AbilityInterface> create(Point cord = {0, 0}, std::function<void (bool)> trigger = [](bool) {}) override – метод, позволяющий создавать объекты класса способности ‘Scanner’.
* AbilityName getName()– метод, возвращающий член перечисления способности

Класс ‘RandomShotFactory’ представляет собой фабрику способности ‘RandomShot’. Наследуется от класса-интерфейса ‘AbilityFactory’.

Методы класса:

* virtual std::shared\_ptr<AbilityInterface> create(Point cord = {0, 0}, std::function<void (bool)> trigger = [](bool) {}) override – метод, позволяющий создавать объекты класса способности ‘RandomShot’.
* AbilityName getName()– метод, возвращающий член перечисления способности

Класс ‘AbilityManger’ реализует логику менеджера способностей.

Поля класса:

* std::queue<std::shared\_ptr<AbilityFactory>> abilities – очередь фабрик способностей.
* AbilityProduction abilityProduction – объект класса AbilityProduction, хранящий в себе все фабрики способностей.

Методы класса:

* AbilityManager() – конструктор класса. Во время вызова создает 3 случайные способности.
* void produceAbility(AbilityName abilityName) – метод создания способности. Добавляет в очередь определенную по abilityName фабрику.
* std::vector<AbilityName> getRandomAbilities() – метод, возвращающий список способностей в случайном порядке.
* std::shared\_ptr<AbilityFactory> extractAbility() – метод, возвращающий из очереди фабрику способности
* AbilityName font\_skill() – метод, возвращающий название первой способности в очереди
* int queue\_size() – метод, возвращающий кол-во способностей в очереди

Класс ‘AbilityProduction’ класс хранения всех созданных фабрик способностей.

Поля класса:

* std::shared\_ptr<DoubleDamageFactory> double\_damage\_factory – фабрика способности ‘DoubleDamage’.
* std::shared\_ptr<ScannerFactory> double\_damage\_factory – фабрика способности ‘Scanner’
* std::shared\_ptr<RandomShotFactory> double\_damage\_factory – фабрика способности ‘RandomShot’

Методы класса:

* AbilityProduction() – конструктор, в котором инициализируются объекты фабрик.
* std::shared\_ptr<AbilityFactory> getFactory (AbilityName name) – метод, возвращающий указатель на определенную фабрику способности.

Класс ‘PlaceError’ – класс-исключение, которое обрабатывается при попытке установить корабль в неправильном для установке месте (Пересечение, слияние кораблей, а так же установка корабля за полем). Наследуется от std::invalid\_argument.

Класс ‘OutOfBoundsError’ – класс-исключение, которое обрабатывается при попытке провести атаку за пределами игрового поля. Наследуется от std::invalid\_argument.

Класс ‘AbilityExtractError’ – класс-исключение, которое обрабатывается при попытке получения способности из пустой очереди класса ‘AbilityManager’. Наследуется от std::runtime\_error.

Для реализации логики функционала получения одной случайной способности при уничтожении вражеского корабля был использован поведенческий паттерн ‘Команда’.

Класс Command – абстрактный класс для реализации паттерна.

Метод класса:

* virtual void execute() – логика реакции команды.

Класс AddNewAbility – класс, который позволяет добавить одну новую случайную способность. Наследуется от абстрактного класса ‘Command’

Поля класса:

* AbilityManager &ability\_manager – ссылка на менеджер способностей.

Методы класса:

* AddNewAbility(AbilityManager& ability\_manager) – конструктор привязывающий менеджер способностей
* void execute() – основная логика добавления случайной способности.

Добавление новой способности будет происходить при разрушении корабля в методе класса GameField – attackField. Для этого, нам необходимо привязать объект-команду AddNewAbility к объекту GameField с помощью отдельного метода у класса GameField:

* void setCommand(std::shared\_ptr<Command> command) – позволяет привязать определенную команду к объекту класса GameField

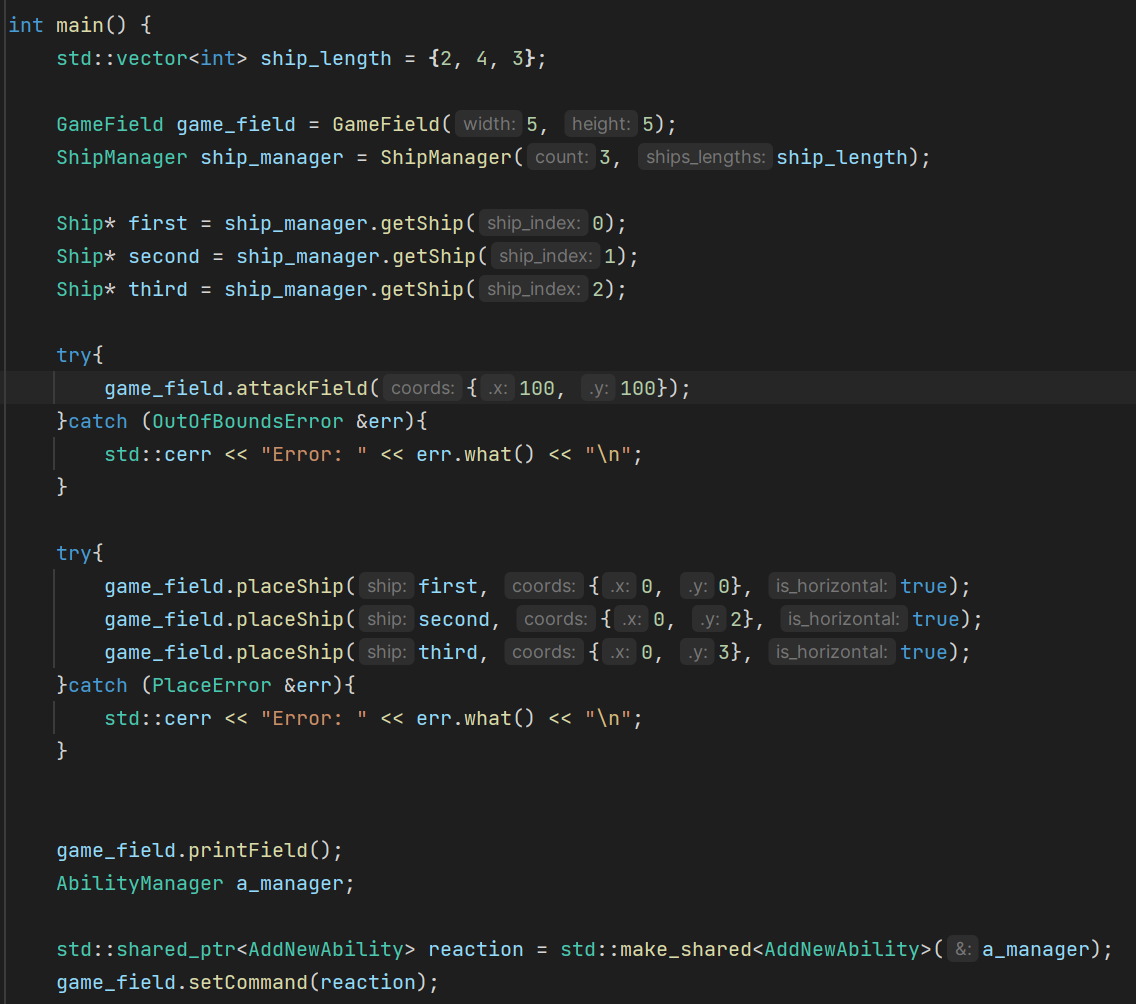
Помимо реакции на уничтожение корабля, у GameField появился метод, принимающий способность:

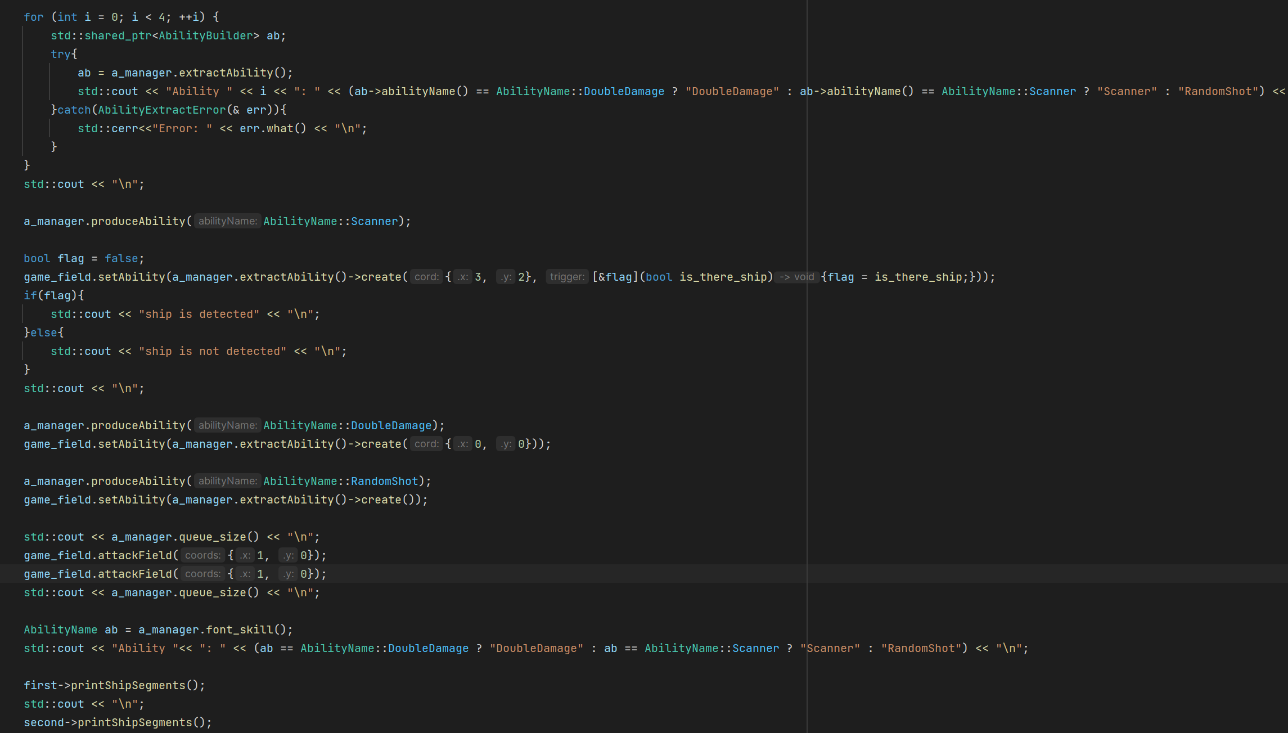
* void setAbility(std::shared\_ptr<AbilityInterface> ability).

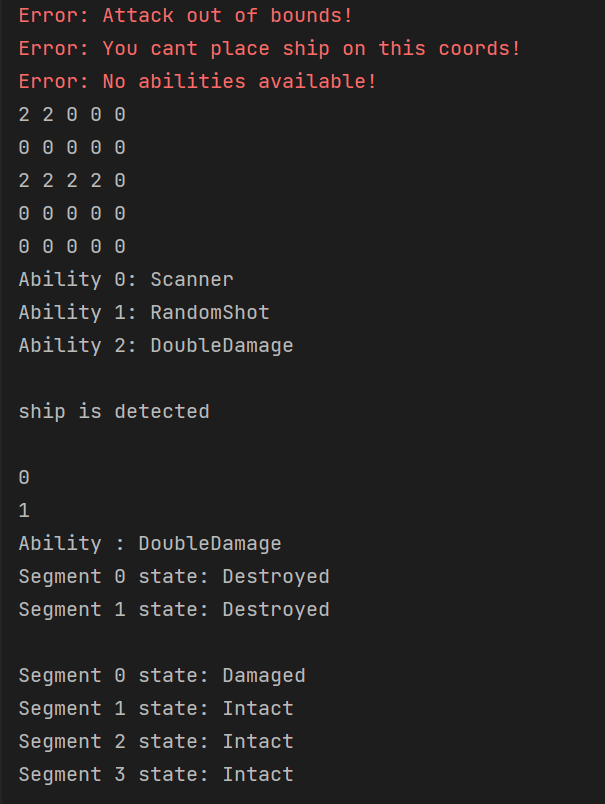
Разработанный программный код см. в приложении А.

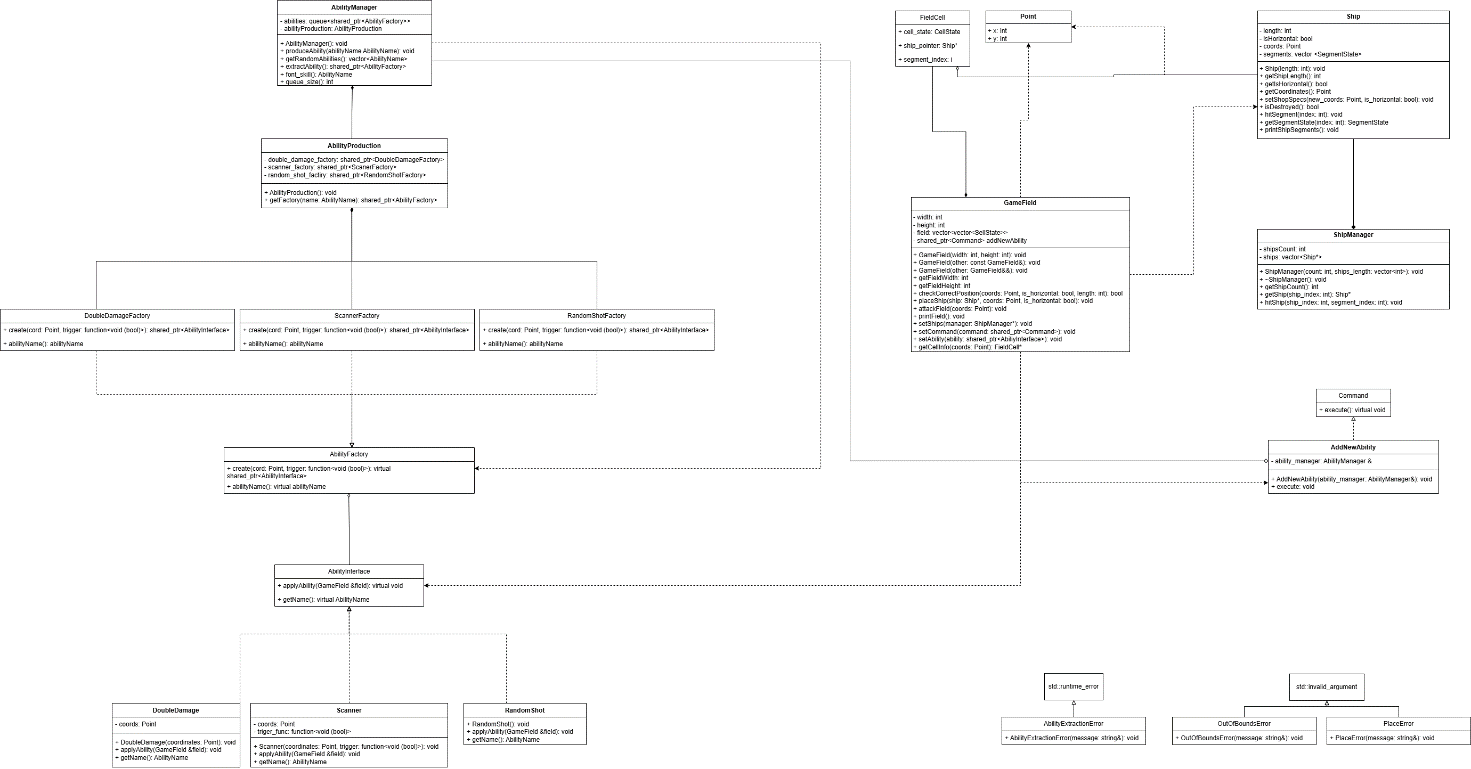
## **Тестирование**

Для проверки правильности работы менеджера способностей, способностей и их взаимодействия друг с другом было проведено тестирование. В ходе тестирования были созданы и протестированы все способности. В некоторых случаях были поданы изначально неправильные данные, чтобы убедится в работоспособности созданных классов исключений.









## **Выводы**

В процессе выполнения задания было изучено функционирование динамического полиморфизма, созданы различные интерфейсы и абстрактные классы, а также их конкретные реализации.

Для лучшего понимания структуры игры была создана UML-диаграмма классов.

# **Приложение А Исходный код программы**

Название файла: main.cpp

#include <iostream>

#include "seagame/game\_field.h"

#include "seagame/ship\_manager.h"

#include "seagame/place\_error.h"

#include "seagame/out\_of\_bounds\_error.h"

#include "seagame/ability\_extract\_error.h"

#include "seagame/ability\_manager.h"

#include "seagame/add\_new\_ability.h"

int main() {

std::vector<int> ship\_length = {2, 4, 3};

GameField game\_field = GameField(5, 5);

ShipManager ship\_manager = ShipManager(3, ship\_length);

Ship\* first = ship\_manager.getShip(0);

Ship\* second = ship\_manager.getShip(1);

Ship\* third = ship\_manager.getShip(2);

try{

game\_field.attackField({100, 100});

}catch (OutOfBoundsError &err){

std::cerr << "Error: " << err.what() << "\n";

}

try{

game\_field.placeShip(first, {0, 0}, true);

game\_field.placeShip(second, {0, 2}, true);

game\_field.placeShip(third, {0, 3}, true);

}catch (PlaceError &err){

std::cerr << "Error: " << err.what() << "\n";

}

game\_field.printField();

AbilityManager a\_manager;

std::shared\_ptr<AddNewAbility> reaction = std::make\_shared<AddNewAbility>(a\_manager);

game\_field.setCommand(reaction);

for (int i = 0; i < 4; ++i) {

std::shared\_ptr<AbilityFactory> ab;

try{

ab = a\_manager.extractAbility();

std::cout << "Ability " << i << ": " << (ab->abilityName() == AbilityName::DoubleDamage ? "DoubleDamage" : ab->abilityName() == AbilityName::Scanner ? "Scanner" : "RandomShot") << "\n";

}catch(AbilityExtractError(& err)){

std::cerr<<"Error: " << err.what() << "\n";

}

}

std::cout << "\n";

a\_manager.produceAbility(AbilityName::Scanner);

bool flag = false;

game\_field.setAbility(a\_manager.extractAbility()->create({3, 2}, [&flag](bool is\_there\_ship){flag = is\_there\_ship;}));

if(flag){

std::cout << "ship is detected" << "\n";

}else{

std::cout << "ship is not detected" << "\n";

}

std::cout << "\n";

a\_manager.produceAbility(AbilityName::DoubleDamage);

game\_field.setAbility(a\_manager.extractAbility()->create({0, 0}));

a\_manager.produceAbility(AbilityName::RandomShot);

game\_field.setAbility(a\_manager.extractAbility()->create());

std::cout << a\_manager.queue\_size() << "\n";

game\_field.attackField({1, 0});

game\_field.attackField({1, 0});

std::cout << a\_manager.queue\_size() << "\n";

AbilityName ab = a\_manager.font\_skill();

std::cout << "Ability "<< ": " << (ab == AbilityName::DoubleDamage ? "DoubleDamage" : ab == AbilityName::Scanner ? "Scanner" : "RandomShot") << "\n";

first->printShipSegments();

std::cout << "\n";

second->printShipSegments();

return 0;

}

Название файла: ability\_interface.h

#ifndef OOP\_ABILITY\_INTERFACE\_H

#define OOP\_ABILITY\_INTERFACE\_H

#include "iostream"

#include "ability\_name.h"

class GameField;

class AbilityInterface {

public:

virtual void applyAbility(GameField &field) = 0;

virtual AbilityName getName() = 0;

};

#endif

Название файла: double\_damage.h

#ifndef OOP\_DOUBLE\_DAMAGE\_H

#define OOP\_DOUBLE\_DAMAGE\_H

#include "ability\_interface.h"

#include "game\_field.h"

#include "point.h"

class DoubleDamage: public AbilityInterface{

private:

Point coords;

public:

DoubleDamage(Point coordinates);

void applyAbility(GameField &field) override;

AbilityName getName() override;

};

#endif

Название файла: double\_damage.cpp

#include "../double\_damage.h"

DoubleDamage::DoubleDamage(Point coordinates) : coords(coordinates) {}

void DoubleDamage::applyAbility(GameField &field) {

field.attackField(coords);

field.attackField(coords);

}

AbilityName DoubleDamage::getName() {

return AbilityName::DoubleDamage;

}

Название файла: scanner.h

#ifndef OOP\_SCANNER\_H

#define OOP\_SCANNER\_H

#include "functional"

#include "ability\_interface.h"

#include "game\_field.h"

#include "point.h"

class Scanner:public AbilityInterface{

private:

Point coords;

std::function<void (bool)> trigger\_func;

public:

Scanner(Point coordinates, std::function<void (bool)> trigger);

void applyAbility(GameField &field) override;

AbilityName getName() override;

};

#endif

Название файла: scanner.cpp

#include "../scanner.h"

Scanner::Scanner(Point coordinates, std::function<void (bool)> trigger) : coords(coordinates), trigger\_func(trigger) {}

void Scanner::applyAbility(GameField &field) {

bool is\_there\_ship = false;

std::vector<Point> sector {coords, {coords.x + 1, coords.y}, {coords.x, coords.y}, {coords.x + 1, coords.y + 1}};

for(Point point : sector){

if(field.getSellState(point) == CellState::Ship){

is\_there\_ship = true;

break;

}

}

trigger\_func(is\_there\_ship);

}

AbilityName Scanner::getName() {

return AbilityName::Scanner;

}

Название файла: random\_shot.h

#ifndef OOP\_RANDOM\_SHOT\_H

#define OOP\_RANDOM\_SHOT\_H

#include "ability\_interface.h"

#include "game\_field.h"

#include "random"

class RandomShot: public AbilityInterface{

public:

RandomShot() = default;

void applyAbility(GameField &field) override;

AbilityName getName() override;

};

#endif

Название файла: random\_shot.cpp

#include "../random\_shot.h"

void RandomShot::applyAbility(GameField &field) {

std::random\_device rd;

std::mt19937 gen(rd());

std::vector<Point> ship\_cells;

for (int i = 0; i < field.getFieldHeight(); ++i) {

for (int j = 0; j < field.getFieldWidth(); ++j) {

if(field.getSellState({j, i}) == CellState::Ship){

ship\_cells.push\_back({j, i});

}

}

}

std::uniform\_int\_distribution<> random\_index(0, ship\_cells.size() - 1);

Point coords\_for\_shot = ship\_cells[random\_index(gen)];

FieldCell\* chosen\_segment = field.getCellInfo(coords\_for\_shot);

field.attackField(coords\_for\_shot);

chosen\_segment->cell\_state = CellState::Unknown;

}

AbilityName RandomShot::getName() {

return AbilityName::RandomShot;

}

Название файла: ability\_factory.h

#ifndef OOP\_ABILITY\_FACTORY\_H

#define OOP\_ABILITY\_FACTORY\_H

#include "memory"

#include "functional"

#include "ability\_name.h"

#include "ability\_interface.h"

#include "Point.h"

class AbilityFactory{

public:

virtual std::shared\_ptr<AbilityInterface> create(Point cord = {0, 0}, std::function<void (bool)> trigger = [](bool) {}) = 0;

virtual AbilityName abilityName() = 0;

};

#endif

Название файла: double\_damage\_factory.h

#ifndef OOP\_DOUBLE\_DAMAGE\_FACTORY\_H

#define OOP\_DOUBLE\_DAMAGE\_FACTORY\_H

#include "ability\_factory.h"

#include "double\_damage.h"

class DoubleDamageFactory: public AbilityFactory{

public:

std::shared\_ptr<AbilityInterface> create(Point cords, std::function<void (bool)> trigger) override;

AbilityName abilityName() override;

};

#endif

Название файла: double\_damage\_factory.cpp

#include "../double\_damage\_factory.h"

std::shared\_ptr<AbilityInterface> DoubleDamageFactory::create(Point cords, std::function<void(bool)> trigger) {

return std::make\_shared<DoubleDamage>(cords);

}

AbilityName DoubleDamageFactory::abilityName() {

return AbilityName::DoubleDamage;

}

Название файла: scanner\_factory.h

#ifndef OOP\_SCANNER\_FACTORY\_H

#define OOP\_SCANNER\_FACTORY\_H

#include "ability\_factory.h"

#include "scanner.h"

class ScannerFactory: public AbilityFactory {

public:

std::shared\_ptr<AbilityInterface> create(Point cords, std::function<void (bool)> trigger) override;

AbilityName abilityName() override;

};

#endif

Название файла: scanner\_factory.cpp

#include "../scanner\_factory.h"

std::shared\_ptr<AbilityInterface> ScannerFactory::create(Point cords, std::function<void(bool)> trigger) {

return std::make\_shared<Scanner>(cords, trigger);

}

AbilityName ScannerFactory::abilityName() {

return AbilityName::Scanner;

}

Название файла: random\_shot\_factory.h

#ifndef OOP\_RANDOM\_SHOT\_FACTORY\_H

#define OOP\_RANDOM\_SHOT\_FACTORY\_H

#include "ability\_factory.h"

#include "random\_shot.h"

class RandomShotFactory: public AbilityFactory {

public:

std::shared\_ptr<AbilityInterface> create(Point cords, std::function<void (bool)> trigger) override;

AbilityName abilityName() override;

};

#endif

Название файла: random\_shot \_factory.cpp

#include "../random\_shot\_factory.h"

std::shared\_ptr<AbilityInterface> RandomShotFactory::create(Point cords, std::function<void(bool)> trigger) {

return std::make\_shared<RandomShot>();

}

AbilityName RandomShotFactory::abilityName() {

return AbilityName::RandomShot;

}

Название файла: ability\_production.h

#ifndef OOP\_ABILITY\_PRODUCTION\_H

#define OOP\_ABILITY\_PRODUCTION\_H

#include "ability\_factory.h"

#include "double\_damage\_factory.h"

#include "scanner\_factory.h"

#include "random\_shot\_factory.h"

class AbilityProduction {

public:

AbilityProduction();

std::shared\_ptr<AbilityFactory> getFactory(AbilityName name);

private:

std::shared\_ptr<DoubleDamageFactory> double\_damage\_factory;

std::shared\_ptr<ScannerFactory> scanner\_factory;

std::shared\_ptr<RandomShotFactory> random\_shot\_factory;

};

#endif

Название файла: ability\_production.cpp

#include "../ability\_production.h"

AbilityProduction::AbilityProduction() {

this->double\_damage\_factory = std::make\_shared<DoubleDamageFactory>();

this->scanner\_factory = std::make\_shared<ScannerFactory>();

this->random\_shot\_factory = std::make\_shared<RandomShotFactory>();

}

std::shared\_ptr<AbilityFactory> AbilityProduction::getFactory(AbilityName name) {

switch (name) {

case AbilityName::DoubleDamage:

return double\_damage\_factory;

case AbilityName::Scanner:

return scanner\_factory;

case AbilityName::RandomShot:

return random\_shot\_factory;

}

}

Название файла: ability\_name.h

#ifndef OOP\_ABILITY\_NAME\_H

#define OOP\_ABILITY\_NAME\_H

enum class AbilityName{ DoubleDamage, Scanner, RandomShot };

#endif

Название файла: ability\_manager.h

#ifndef OOP\_ABILITY\_MANAGER\_H

#define OOP\_ABILITY\_MANAGER\_H

#include "iostream"

#include "functional"

#include "queue"

#include "memory"

#include "ability\_interface.h"

#include "ability\_production.h"

#include "point.h"

class AbilityManager {

private:

std::queue<std::shared\_ptr<AbilityFactory>> abilities;

AbilityProduction abilityProduction;

public:

AbilityManager();

void produceAbility(AbilityName abilityName);

std::vector<AbilityName> getRandomAbilities();

std::shared\_ptr<AbilityFactory> extractAbility();

AbilityName font\_skill();

int queue\_size();

};

#endif

Название файла: ability\_manager.cpp

#include "../ability\_manager.h"

#include "../ability\_extract\_error.h"

AbilityManager::AbilityManager() {

std::vector<AbilityName> random\_abilities = this->getRandomAbilities();

for (auto name:random\_abilities){

this->produceAbility(name);

}

}

void AbilityManager::produceAbility(AbilityName abilityName) {

this->abilities.push(this->abilityProduction.getFactory(abilityName));

}

std::shared\_ptr<AbilityFactory> AbilityManager::extractAbility() {

if (this->abilities.empty()) throw AbilityExtractError("No abilities available!");

std::shared\_ptr<AbilityFactory> first\_ability = this->abilities.front();

this->abilities.pop();

return first\_ability;

}

std::vector<AbilityName> AbilityManager::getRandomAbilities() {

std::vector<AbilityName> random\_abilities = { AbilityName::DoubleDamage, AbilityName::Scanner, AbilityName::RandomShot};

std::random\_device rd;

std::mt19937 gen(rd());

std::shuffle(random\_abilities.begin(), random\_abilities.end(), gen);

return random\_abilities;

}

AbilityName AbilityManager::font\_skill() {

return this->abilities.front()->abilityName();

}

int AbilityManager::queue\_size() {

return this->abilities.size();

}

Название файла: command.h

#ifndef OOP\_COMMAND\_H

#define OOP\_COMMAND\_H

class Command{

public:

virtual void execute() = 0;

};

#endif

Название файла: add\_new\_ability.h

#ifndef OOP\_ADD\_NEW\_ABILITY\_H

#define OOP\_ADD\_NEW\_ABILITY\_H

#include "command.h"

#include "ability\_manager.h"

class AddNewAbility: public Command{

public:

AddNewAbility(AbilityManager& ability\_manager);

void execute() override;

private:

AbilityManager &ability\_manager;

};

#endif

Название файла: add\_new\_ability.cpp

#include "../add\_new\_ability.h"

AddNewAbility::AddNewAbility(AbilityManager &ability\_manager) : ability\_manager(ability\_manager) {}

void AddNewAbility::execute() {

this->ability\_manager.produceAbility(this->ability\_manager.getRandomAbilities()[0]);

}

Название файла: out\_of\_bounds\_error.h

#ifndef OOP\_OUT\_OF\_BOUNDS\_ERROR\_H

#define OOP\_OUT\_OF\_BOUNDS\_ERROR\_H

#include "stdexcept"

class OutOfBoundsError: public std::invalid\_argument{

public:

OutOfBoundsError(const std::string &message): std::invalid\_argument(message) {};

};

#endif

Название файла: ability\_extract\_error.h

#ifndef OOP\_ABILITY\_EXTRACT\_ERROR\_H

#define OOP\_ABILITY\_EXTRACT\_ERROR\_H

#include "stdexcept"

class AbilityExtractError: public std::runtime\_error{

public:

AbilityExtractError(const std::string &message): std::runtime\_error(message) {};

};

#endif

Название файла: place\_error.h

#ifndef OOP\_PLACE\_ERROR\_H

#define OOP\_PLACE\_ERROR\_H

#include "stdexcept"

class PlaceError: public std::invalid\_argument{

public:

PlaceError(const std::string &message): std::invalid\_argument(message) {};

};

#endif