**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 |  | Иванов С.С. |
| Преподаватель |  | Жангиров Т.Р. |

Санкт-Петербург

2024

## **Цель работы**

Научиться использовать динамический полиморфизм путём создания класса-интерфейса способности, через наследование создать 3 разные способности, интерфейс события должен быть унифицирован, чтобы их можно было единообразно использовать через интерфейс.

## **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 менеджер-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

## **Выполнение работы**

Класс Visitor:

Абстрактный класс посетителя. Имплементации этого класса будут посещать имплементации класса Owner (владельцев) и дополнять их функционал.

Класс содержит следующие публичные виртуальные методы:

* virtual void operator()(void \*\_obj) = 0; - принимает владельца и определяет дополняющий его функционал;
* virtual ~Visitor() = 0; - виртуальный деструктор;

Класс Owner:

Абстрактный класс владельца. Имплементации этого класса будут принимать к себе имплементации класса Visitor (посетителей), таким образом расщиряя свой функционал.

Класс содержит следующие публичные методы:

* template <class T, class ...Args\_T>

std::shared\_ptr<Visitor>

accept(Args\_T&& ...\_args) – метод, принимающий в шаблон класс посетителя, создающий его внутри себя и выдающий std::shared\_ptr на него. Тип параметра Args\_T&& ради идеальной передачи аргументов;

Класс содержит следующие публичные виртуальные методы:

* virtual ~Owner() = 0; - виртуальный деструктор;

Класс Field:

Поля начинает имплементить Owner и становится владельцем.

Новые приватные поля:

* std::shared\_ptr<Command> \_react\_of\_destroyed\_ship; - команда, выдающая рандомную способность;

Новые публичные методы:

* void accept\_skill(std::shared\_ptr<iSkill> \_skill); - метод, которые принимает способность-посетителя;

Класс iSkill:

Интерфейс способности. Наследуется от Visitor. Имплементации этого класса будут посещать объекты класса Field и выполнять логику способности.

Класс содержит следующие публичные виртуальные методы:

* virtual inline void use(Field &\_fd) = 0; - принимает поле и использует способность;
* virtual SkillName classname() const noexcept = 0; - возвращает член перечисления, идентифицирующий способность;

Класс DoubleHit:

Класс имплементит iSkill. Реализует двойной урон по определённой клетке на указанном поле.

Класс содержит следующие приватные поля:

* Unit \_unit; - точка, по которой будет совершён выстрел;

Класс содержит следующие публичные методы:

* explicit DoubleHit(const Unit &\_unit); - конструктор, принимающий точку, по которой будет совершён выстрел;
* DoubleHit(std::uint64\_t \_x, std::uint64\_t \_y); - конструктор, принимающий координаты точки, по которой будет совершён выстрел;
* DoubleHit(); - конструктор по умолчанию;
* void operator()(void \*\_obj) override; - метод, содержащий основную логику способности;
* SkillName classname() const noexcept override; - возвращает член перечисления, идентифицирующий способность;
* ~DoubleHit() override; - переопределение деструктора;

Класс RocketAttack:

Класс имплементит iSkill. Реализует удар по случайной клетке на указанном поле.

Класс содержит следующие приватные поля:

* std::random\_device rd; - функтор, выдающий случайные числа;

Класс содержит следующие публичные методы;

* RocketAttack(); - конструктор по умолчанию;
* void operator()(void \*\_obj) override; - метод, содержащий основную логику способности;
* SkillName classname() const noexcept override; - возвращает член перечисления, идентифицирующий способность;
* ~RocketAttack() override; - переопределение деструктора;

Класс содержит следующие приватные методы:

* Unit \_\_get\_random\_unit\_of\_ship(const Field &\_sm) noexcept; - возвращает клетку, на которой расположен верхний левый сегмент рандомного корабля;
* std::uint8\_t \_\_get\_random\_index\_of\_segments(const Ship &\_ship) noexcept; - возвращает индекс рандомного сегмента корабля;

Класс Scanner:

Класс имплементит iSkill. Реализует поиск кораблей на указанном поле.

Класс содержит следующие приватные поля:

* Scanner(const Unit &\_unit, const std::function<void(const Unit&)> &\_funct); - конструктор, принимающий точку и реакцию на обнаружение сегмента корабля;
* Scanner(std::uint64\_t \_x, std::uint64\_t \_y, const std::function<void(const Unit&)> &\_funct); - конструктор, принимающий координаты точки и реакцию на обнаружение сегмента корабля;
* Scanner(); - конструктор по умолчанию;
* void operator()(void \*\_obj) override; - метод, содержащий основную логику способности;
* SkillName classname() const noexcept override; - возвращает член перечисления, идентифицирующий способность;
* ~Scanner() override; - переопределение деструктора;

Класс iSkillFactory:

Интерфейс фабрики способности. Имплементации этого класса будут создавать имплементации класса iSkill.

Класс содержит следующие публичные виртуальные методы:

* virtual std::shared\_ptr<iSkill> create(const Unit &\_unit = Unit(), const std::function<void(const Unit&)> &\_funct = \_\_plug\_for\_react()) = 0; - метод-фабрика способностей;
* virtual SkillName skillname() const noexcept = 0; - возвращает член перечисления, идентифицирующий способность;
* virtual ~iSkillFactory() = 0; - виртуальный деструктор;

Класс DoubleHitFactory:

Класс имплементит iSkillFactory. Создаёт DoubleHit.

Класс содержит следующие публичные методы:

* std::shared\_ptr<iSkill> create(const Unit &\_unit, const std::function<void(const Unit&)> &\_funct = \_\_plug\_for\_react()) override; - метод-фабрика способностей;
* SkillName skillname() const noexcept override; - возвращает член перечисления, идентифицирующий способность;
* ~DoubleHitFactory() override; - переопределение деструктора;

Класс RocketAttackFactory:

Класс имплементит iSkillFactory. Создаёт RocketAttack.

Класс содержит следующие публичные методы:

* std::shared\_ptr<iSkill> create(const Unit &\_unit = Unit(), const std::function<void(const Unit&)> &\_funct = \_\_plug\_for\_react()) override; - метод-фабрика способностей;
* SkillName skillname() const noexcept override; - возвращает член перечисления, идентифицирующий способность;
* ~RocketAttackFactory() override; - переопределение деструктора;

Класс ScannerFactory:

Класс имплементит iSkillFactory. Создаёт Scanner.

Класс содержит следующие публичные методы:

* std::shared\_ptr<iSkill> create(const Unit &\_unit, const std::function<void(const Unit&)> &\_funct) override; - метод-фабрика способностей;
* SkillName skillname() const noexcept override; - возвращает член перечисления, идентифицирующий способность;
* ~ScannerFactory() override; - переопределение деструктора;

Класс SetupReactOfDestroyedShip:

Класс установки реакции на уничтожение корабля у поля. Наследуется от Visitor.

Класс содержит следующие приватные поля:

* std::shared\_ptr<Command> \_cmd; - команда, испоняемая при уничтожении корабля;

Класс содержит следующие публичные методы:

* explicit SetupReactOfDestroyedShip(std::shared\_ptr<Command> \_cmd); - конструктор, принимающий команду;
* void operator()(void \*\_obj) override; - метод, содержащий основную добавления полю реакции на уничтожение корабля;
* ~SetupReactOfDestroyedShip() override; - переопределение деструктора;

Класс Command:

Абстрактный класс команды. Имплементации этого класса будут превращать запросы/действия/реакции в объекты.

Класс содержит следующие публичные виртуальные методы:

* virtual void exec() = 0; - логика запроса/действия/реакции;
* virtual ~Command() = 0; - виртуальный деструктор;

Класс AddRandomSkill:

Класс имплементит Command. Реализует получение рандомной способности при уничтожении корабля.

Класс содержит следующие приватные поля:

* SkillManager &\_skill\_manager; - ссылка на менеджер скилов;

Класс содержит следующие публичные методы:

* explicit AddRandomSkill(SkillManager &\_sm); - конструктор, принимающий ссылку на менеджер скилов;
* void exec() override; - метод, содержащий основную логику получения рандомной способности;
* ~AddRandomSkill() override; - переопределение деструктора;

Класс Skill:

Абстрактный класс способности, наследующийся от iSkill и содержащий методы, которые необходимы всем способностям:

* inline void use(Field &\_fd) override; - метод, посещающий владельца (в слечае способностей, владелец – это поле);

Класс перечисления SkillName:

Класс перечисления, содержащий в себе константы, идентифицирующие способности:

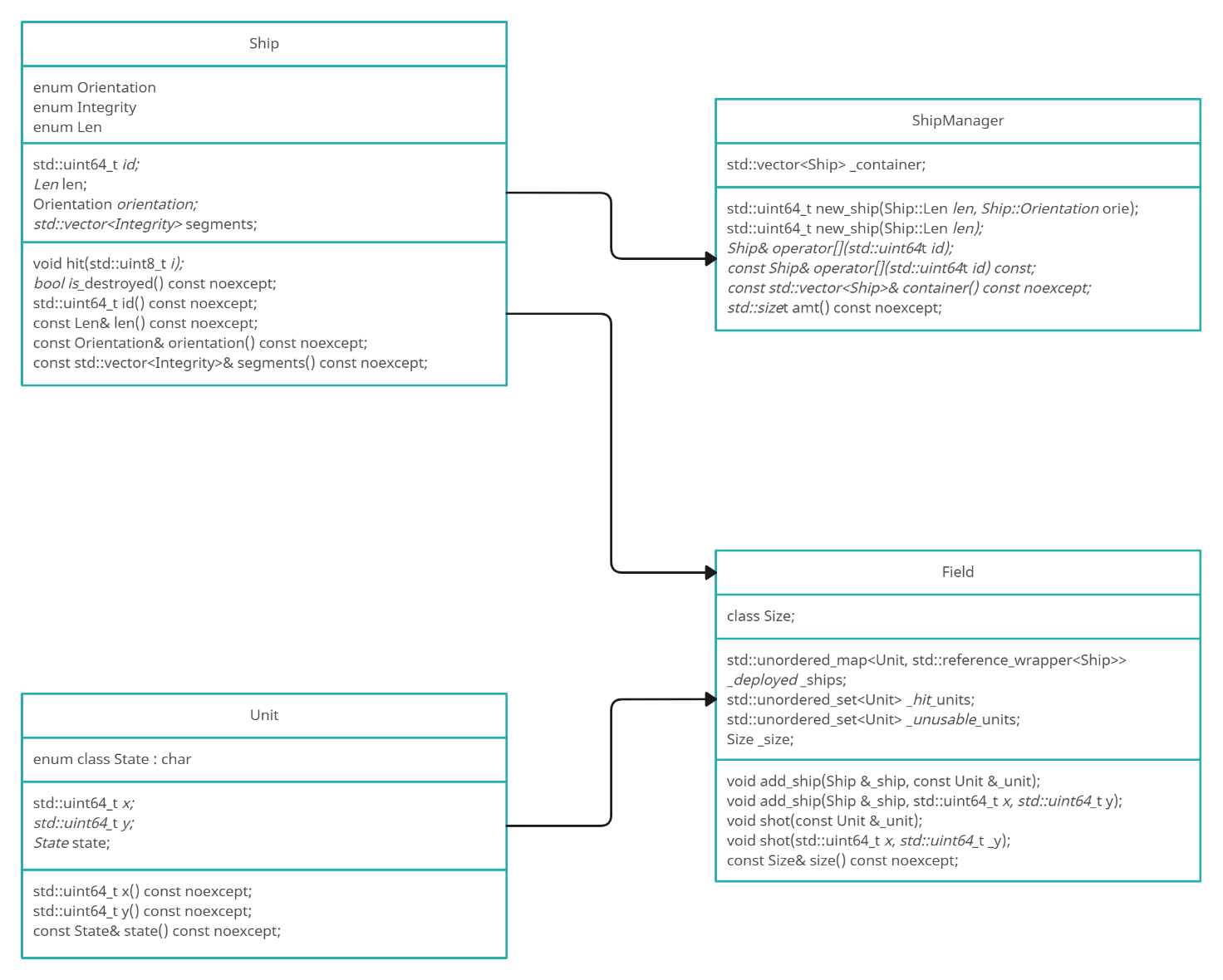
* DOUBLEHIT = 'D' – идентификатор DoubleHit:
* ROCKETATTACK = 'R' – идентификатор RocketAttack;
* SCANNER = 'S' – идентификатор Scanner;

Архитектурные решения:

Ship c ShipManager связан как композиция, объекты Ship не могут быть созданы вне класса ShipManager.

Field хранит информацию о расположении кораблей на поле в виде хеш-таблицы. Ключом является Unit (левый верхний сегмент корабля), а значением - ссылка на корабль. Также есть вспомогательные хеш-таблицы, хранящие Unit’ы с клетками, в которые больше нельзя попадать, и клетки, на которые нельзя ставить новые корабли.

Проверка работоспособности написанного кода:



Разработанный программный код см. в приложении А.

## **Выводы**

В ходе разработки были созданы 4 класса: клетки, корабля, менеджера кораблей, поля. Созданы методы по взаимодействию с этими классами.

# **Приложение А Исходный код программы**

Название файла: unit.h

#pragma once

#ifndef \_UNIT\_H

#define \_UNIT\_H

#include <cstdint>

#include <functional>

#include <string>

namespace seagame

{

class Unit

{

public:

enum class State : char

{

UNDEFINED = '?',

EMPTY = '-',

SHIP = '\*'

};

private:

std::uint64\_t \_x;

std::uint64\_t \_y;

State \_state;

public:

Unit(std::uint64\_t x, std::uint64\_t y, State state);

Unit(std::uint64\_t x, std::uint64\_t y);

Unit();

Unit(const Unit &other);

Unit(Unit &&other) noexcept;

~Unit() = default;

Unit& operator=(const Unit &other);

Unit& operator=(Unit &&other) noexcept;

std::uint64\_t x() const noexcept;

std::uint64\_t y() const noexcept;

const State& state() const noexcept;

};

} // seagame

namespace std

{

template <>

struct hash<seagame::Unit>

{

std::size\_t operator()(const seagame::Unit& u) const;

};

template <>

struct equal\_to<seagame::Unit>

{

bool operator()(const seagame::Unit& lhs, const seagame::Unit& rhs) const;

};

template <>

struct hash<std::reference\_wrapper<seagame::Unit>>

{

std::size\_t operator()(const std::reference\_wrapper<seagame::Unit>& u) const;

};

template <>

struct equal\_to<std::reference\_wrapper<seagame::Unit>>

{

bool operator()(const std::reference\_wrapper<seagame::Unit>& lhs, const std::reference\_wrapper<seagame::Unit>& rhs) const;

};

} // std

#endif // \_UNIT\_H

Название файла: ship.h

#pragma once

#ifndef \_SHIP\_H

#define \_SHIP\_H

#include <cstdint>

#include <vector>

#include <string>

namespace seagame

{

class ShipManager;

class Ship

{

friend ShipManager;

public:

enum Orientation

{

HORIZONTAL,

VERTICAL

};

enum Integrity

{

WHOLE = 2,

HALF\_DESTROYED = 1,

DESTROYED = 0

};

enum Len

{

ONE = 1,

TWO = 2,

THREE = 3,

FOUR = 4

};

private:

std::uint64\_t \_id;

Len \_len;

Orientation \_orientation;

std::vector<Integrity> \_segments;

Ship(Len \_len, Orientation \_orie);

explicit Ship(Len \_len);

Ship();

public:

Ship(const Ship &other);

Ship(Ship &&other) noexcept;

~Ship() = default;

Ship& operator=(const Ship &other);

Ship& operator=(Ship &&other) noexcept;

void hit(std::uint8\_t \_i);

bool is\_destroyed() const noexcept;

std::uint64\_t id() const noexcept;

const Len& len() const noexcept;

const Orientation& orientation() const noexcept;

const std::vector<Integrity>& segments() const noexcept;

};

} // seagame

namespace std

{

template <>

struct hash<seagame::Ship>

{

std::size\_t operator()(const seagame::Ship& u) const;

};

template <>

struct equal\_to<seagame::Ship>

{

bool operator()(const seagame::Ship& lhs, const seagame::Ship& rhs) const;

};

template <>

struct hash<std::reference\_wrapper<seagame::Ship>>

{

std::size\_t operator()(const std::reference\_wrapper<seagame::Ship>& u) const;

};

template <>

struct equal\_to<std::reference\_wrapper<seagame::Ship>>

{

bool operator()(const std::reference\_wrapper<seagame::Ship>& lhs, const std::reference\_wrapper<seagame::Ship>& rhs) const;

};

} // std

#endif // \_SHIP\_H

Название файла: ship\_manager.h

#pragma once

#ifndef \_SHIP\_MANAGER\_H

#define \_SHIP\_MANAGER\_H

#include <vector>

#include <cstdint>

#include <type\_traits>

#include "ship.h"

namespace seagame

{

class ShipManager

{

public:

using ArrayOfLens = std::initializer\_list<Ship::Len>;

private:

std::vector<Ship> \_container;

// если кораблям понадобятся идентификаторы, то взять за таковые индексы в векторе

public:

template <class T,

typename std::void\_t<

typename std::remove\_reference<T>::type::value\_type,

typename std::remove\_reference<T>::type::iterator,

typename std::remove\_reference<T>::type::const\_iterator

>\* = nullptr>

ShipManager(const T &\_lens)

{

static\_assert(std::is\_same<

typename std::remove\_reference<T>::type::value\_type,

Ship::Len

>::value, "Container must contain seagame::Ship::Len");

this->\_\_construct(\_lens);

}

ShipManager(const ArrayOfLens &\_lens);

ShipManager();

ShipManager(const ShipManager &other);

ShipManager(ShipManager &&other) noexcept;

~ShipManager() = default;

ShipManager& operator=(const ShipManager &other);

ShipManager& operator=(ShipManager &&other) noexcept;

std::uint64\_t new\_ship(Ship::Len \_len, Ship::Orientation \_orie);

std::uint64\_t new\_ship(Ship::Len \_len);

Ship& operator[](std::uint64\_t \_id);

const Ship& operator[](std::uint64\_t \_id) const;

const std::vector<Ship>& container() const noexcept;

std::size\_t amt() const noexcept;

private:

template <class T>

inline void \_\_construct(const T &\_lens)

{

for (const auto &\_len : \_lens)

this->new\_ship(\_len);

}

};

} // seagame

#endif // \_SHIP\_MANAGER\_H

Название файла: field.h

#pragma once

#ifndef \_FIELD\_H

#define \_FIELD\_H

#include <unordered\_map>

#include <unordered\_set>

#include <functional>

#include <stdexcept>

#include "unit.h"

#include "ship.h"

namespace seagame

{

class Field

{

public:

class Size

{

public:

Size(std::uint64\_t m, std::uint64\_t n);

Size(const Size &other);

Size(Size &&other) noexcept;

~Size() = default;

Size& operator=(const Size &other);

Size& operator=(Size &&other) noexcept;

std::uint64\_t m() const noexcept;

std::uint64\_t n() const noexcept;

private:

std::uint64\_t \_m;

std::uint64\_t \_n;

};

private:

std::unordered\_map<Unit, std::reference\_wrapper<Ship>> \_deployed\_ships;

std::unordered\_set<Unit> \_hit\_units;

std::unordered\_set<Unit> \_unusable\_units;

Size \_size;

public:

Field(std::uint64\_t \_m, std::uint64\_t \_n);

explicit Field(Size \_size);

Field(const Field &other);

Field(Field &&other) noexcept;

~Field() = default;

Field& operator=(const Field &other);

Field& operator=(Field &&other) noexcept;

void add\_ship(Ship &\_ship, const Unit &\_unit);

void add\_ship(Ship &\_ship, std::uint64\_t \_x, std::uint64\_t y);

void shot(const Unit &\_unit);

void shot(std::uint64\_t \_x, std::uint64\_t \_y);

const Size& size() const noexcept;

private:

void \_\_block\_unit(const Unit &\_unit, const Ship::Orientation &\_orie, bool \_flag);

void \_\_block\_units(const Unit &\_unit, const Ship &\_ship, bool \_flag);

void \_\_block\_units\_for\_add(const Unit &\_unit, const Ship &\_ship);

void \_\_block\_units\_for\_hit(const Unit &\_unit, const Ship &\_ship);

bool \_\_is\_valid\_unit(const Unit &\_u) const noexcept;

bool \_\_is\_valid\_unit(std::uint64\_t \_x, std::uint64\_t \_y) const noexcept;

bool \_\_is\_same\_ship(const Unit &\_lu, const Unit &\_unit, std::uint8\_t \_offset);

Unit \_\_get\_lu\_seg\_of\_ship(const Unit &\_unit, const Ship::Orientation &\_orie, std::uint8\_t &i);

};

} // seagame

#endif // \_FIELD\_H

Название файла: unit.cpp

#include "../unit.h"

namespace seagame

{

Unit::Unit(std::uint64\_t \_x, std::uint64\_t \_y, State \_state)

: \_x(\_x), \_y(\_y), \_state(\_state)

{ }

Unit::Unit(std::uint64\_t \_x, std::uint64\_t \_y)

: Unit(\_x, \_y, State::UNDEFINED)

{ }

Unit::Unit()

: \_x(0), \_y(0)

{ }

Unit::Unit(const Unit &other)

: \_x(other.\_x), \_y(other.\_y), \_state(other.\_state)

{ }

Unit::Unit(Unit &&other) noexcept

: \_x(other.\_x), \_y(other.\_y), \_state(other.\_state)

{ }

Unit&

Unit::operator=(const Unit &other)

{

if (this != &other)

{

this->\_x = other.x();

this->\_y = other.y();

this->\_state = other.state();

}

return \*this;

}

Unit&

Unit::operator=(Unit &&other) noexcept

{

if (this != &other)

{

this->\_x = other.\_x;

this->\_y = other.\_y;

this->\_state = other.\_state;

}

return \*this;

}

std::uint64\_t Unit::x() const noexcept

{ return this->\_x; }

std::uint64\_t Unit::y() const noexcept

{ return this->\_y; }

const Unit::State& Unit::state() const noexcept

{ return this->\_state; }

} // seagame

namespace std

{

std::size\_t

hash<seagame::Unit>::operator()(const seagame::Unit& u) const

{

return hash<string>()(to\_string(u.x()) + to\_string(u.y()));

}

bool

equal\_to<seagame::Unit>::operator()(const seagame::Unit& lhs, const seagame::Unit& rhs) const

{

return lhs.x() == rhs.x() && lhs.y() == rhs.y();

}

std::size\_t

hash<std::reference\_wrapper<seagame::Unit>>::operator()(const std::reference\_wrapper<seagame::Unit>& u) const

{

return hash<seagame::Unit>()(u.get());

}

bool

equal\_to<std::reference\_wrapper<seagame::Unit>>::operator()(const std::reference\_wrapper<seagame::Unit>& lhs, const std::reference\_wrapper<seagame::Unit>& rhs) const

{

return equal\_to<seagame::Unit>()(lhs.get(), rhs.get());

}

} // std

Название файла: ship.cpp

#include "../ship.h"

#include <stdexcept>

namespace seagame

{

Ship::Ship(Len \_len, Orientation \_orie)

: \_len(\_len), \_orientation(\_orie), \_id(0)

{

for (std::uint8\_t i = 0; i < \_len; ++i)

this->\_segments.push\_back(Integrity::WHOLE);

}

Ship::Ship(Len \_len)

: Ship(\_len, Ship::Orientation::HORIZONTAL)

{ }

Ship::Ship()

: Ship(Ship::Len::ONE)

{ }

Ship::Ship(const Ship &other)

: \_id(other.id()), \_len(other.len()), \_orientation(other.orientation()), \_segments(other.segments())

{ }

Ship::Ship(Ship &&other) noexcept

: \_id(other.\_id), \_len(other.\_len), \_orientation(other.\_orientation), \_segments(std::move(other.\_segments))

{ }

Ship&

Ship::operator=(const Ship &other)

{

if (this != &other)

{

this->\_id = other.id();

this->\_len = other.len();

this->\_orientation = other.orientation();

this->\_segments = other.segments();

}

return \*this;

}

Ship&

Ship::operator=(Ship &&other) noexcept

{

if (this != &other)

{

this->\_id = other.\_id;

this->\_len = other.\_len;

this->\_orientation = other.\_orientation;

this->\_segments = std::move(other.\_segments);

}

return \*this;

}

void

Ship::hit(std::uint8\_t \_i)

{

if (\_i >= this->\_len)

throw std::out\_of\_range("bad segment");

switch (this->\_segments[\_i])

{

case Integrity::WHOLE:

this->\_segments[\_i] = Integrity::HALF\_DESTROYED;

break;

case Integrity::HALF\_DESTROYED:

this->\_segments[\_i] = Integrity::DESTROYED;

break;

case Integrity::DESTROYED:

throw std::logic\_error("bad hit");

default:

throw std::logic\_error("bad Ship::Integrity");

}

}

bool

Ship::is\_destroyed() const noexcept

{

for (const auto &\_seg : this->\_segments)

if (\_seg != Integrity::DESTROYED)

return false;

return true;

}

std::uint64\_t

Ship::id() const noexcept

{ return this->\_id; }

const Ship::Len&

Ship::len() const noexcept

{ return this->\_len; }

const Ship::Orientation&

Ship::orientation() const noexcept

{ return this->\_orientation; }

const std::vector<Ship::Integrity>&

Ship::segments() const noexcept

{ return this->\_segments; }

} // seagame

namespace std

{

std::size\_t

hash<seagame::Ship>::operator()(const seagame::Ship& u) const

{

return hash<string>()(to\_string(u.id()) + to\_string(u.len()) + to\_string(u.orientation()));

}

bool

equal\_to<seagame::Ship>::operator()(const seagame::Ship& lhs, const seagame::Ship& rhs) const

{

return lhs.len() == rhs.len() && lhs.orientation() == rhs.orientation();

}

std::size\_t

hash<std::reference\_wrapper<seagame::Ship>>::operator()(const std::reference\_wrapper<seagame::Ship>& u) const

{

return hash<seagame::Ship>()(u.get());

}

bool

equal\_to<std::reference\_wrapper<seagame::Ship>>::operator()(const std::reference\_wrapper<seagame::Ship>& lhs, const std::reference\_wrapper<seagame::Ship>& rhs) const

{

return equal\_to<seagame::Ship>()(lhs.get(), rhs.get());

}

} // std

Название файла: ship\_manager.cpp

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

namespace seagame

{

ShipManager::ShipManager(const ArrayOfLens &\_lens)

{

this->\_\_construct(\_lens);

}

ShipManager::ShipManager() : ShipManager({})

{ }

ShipManager::ShipManager(const ShipManager &other)

: \_container(other.container())

{ }

ShipManager::ShipManager(ShipManager &&other) noexcept

: \_container(std::move(other.\_container))

{ }

ShipManager&

ShipManager::operator=(const ShipManager &other)

{

if (this != &other)

{

this->\_container = other.container();

}

return \*this;

}

ShipManager&

ShipManager::operator=(ShipManager &&other) noexcept

{

if (this != &other)

{

this->\_container = std::move(other.\_container);

}

return \*this;

}

std::uint64\_t

ShipManager::new\_ship(Ship::Len \_len, Ship::Orientation \_orie)

{

Ship \_ship(\_len, \_orie);

\_ship.\_id = this->amt();

this->\_container.push\_back(\_ship);

return \_ship.\_id;

}

std::uint64\_t

ShipManager::new\_ship(Ship::Len \_len)

{ return this->new\_ship(\_len, Ship::Orientation::HORIZONTAL); }

Ship&

ShipManager::operator[](std::uint64\_t \_id)

{ return this->\_container.at(\_id); }

const Ship&

ShipManager::operator[](std::uint64\_t \_id) const

{ return this->\_container.at(\_id); }

const std::vector<Ship>&

ShipManager::container() const noexcept

{ return this->\_container; }

std::size\_t

ShipManager::amt() const noexcept

{ return this->\_container.size(); }

} // seagame

Название файла: field.cpp

#include "../field.h"

namespace seagame

{

Field::Size::Size(std::uint64\_t \_m, std::uint64\_t \_n)

{

if (\_m < 1 || \_n < 1)

throw std::invalid\_argument("component of size must be greater than 0");

this->\_m = \_m;

this->\_n = \_n;

}

Field::Size::Size(const Size &other)

: \_m(other.m()), \_n(other.n())

{ }

Field::Size::Size(Size &&other) noexcept

: \_m(other.\_m), \_n(other.\_n)

{ }

Field::Size&

Field::Size::operator=(const Size &other)

{

if (this != &other)

{

this->\_m = other.m();

this->\_n = other.n();

}

return \*this;

}

Field::Size&

Field::Size::operator=(Size &&other) noexcept

{

if (this != &other)

{

this->\_m = other.\_m;

this->\_n = other.\_n;

}

return \*this;

}

std::uint64\_t Field::Size::m() const noexcept

{ return this->\_m; }

std::uint64\_t Field::Size::n() const noexcept

{ return this->\_n; }

Field::Field(std::uint64\_t \_m, std::uint64\_t \_n)

: \_size(\_m, \_n)

{

if (\_m < 1 || \_n < 1)

throw std::invalid\_argument("minimal field size: 1x1");

}

Field::Field(Size \_size)

: \_size(\_size)

{ }

Field::Field(const Field &other)

: \_size(other.size()),

\_deployed\_ships(other.\_deployed\_ships),

\_hit\_units(other.\_hit\_units),

\_unusable\_units(other.\_unusable\_units)

{ }

Field::Field(Field &&other) noexcept

: \_size(other.\_size),

\_deployed\_ships(std::move(other.\_deployed\_ships)),

\_hit\_units(std::move(other.\_hit\_units)),

\_unusable\_units(std::move(other.\_unusable\_units))

{ }

Field&

Field::operator=(const Field &other)

{

if (this != &other)

{

this->\_size = other.size();

this->\_deployed\_ships = other.\_deployed\_ships;

this->\_hit\_units = other.\_hit\_units;

this->\_unusable\_units = other.\_unusable\_units;

}

return \*this;

}

Field&

Field::operator=(Field &&other) noexcept

{

if (this != &other)

{

this->\_size = other.\_size;

this->\_deployed\_ships = std::move(other.\_deployed\_ships);

this->\_hit\_units = std::move(other.\_hit\_units);

this->\_unusable\_units = std::move(other.\_unusable\_units);

}

return \*this;

}

void

Field::add\_ship(Ship &\_ship, const Unit &\_unit)

{

if (this->\_unusable\_units.find(\_unit) != this->\_unusable\_units.end())

throw std::invalid\_argument("bad unit for add ship");

Ship::Len i = Ship::Len::ONE;

while (i != \_ship.len())

{

if (this->\_unusable\_units.find(Unit(

\_ship.orientation() == Ship::Orientation::HORIZONTAL

? \_unit.x() + i

: \_unit.x(),

\_ship.orientation() == Ship::Orientation::VERTICAL

? \_unit.y() + i

: \_unit.y()

)) != this->\_unusable\_units.end())

throw std::invalid\_argument("bad unit for add ship");

i = Ship::Len(i + 1);

}

switch (\_ship.orientation())

{

case Ship::Orientation::HORIZONTAL:

if (!this->\_\_is\_valid\_unit(\_unit.x() + \_ship.len() - 1, \_unit.y()))

throw std::invalid\_argument("bad unit for add ship");

break;

case Ship::Orientation::VERTICAL:

if (!this->\_\_is\_valid\_unit(\_unit.x(), \_unit.y() + \_ship.len() - 1))

throw std::invalid\_argument("bad unit for add ship");

break;

default:

throw std::logic\_error("bad orientation");

}

this->\_deployed\_ships.emplace(\_unit, \_ship);

this->\_\_block\_units\_for\_add(\_unit, \_ship);

}

void

Field::add\_ship(Ship &\_ship, std::uint64\_t \_x, std::uint64\_t \_y)

{ return this->add\_ship(\_ship, Unit(\_x, \_y)); }

void

Field::shot(const Unit &\_unit)

{

if (!this->\_\_is\_valid\_unit(\_unit)

|| this->\_hit\_units.find(\_unit) != this->\_hit\_units.end())

throw std::invalid\_argument("bad unit for shot");

Unit \_u0;

Unit \_u;

std::equal\_to<Unit> eq;

std::uint8\_t i;

\_u = this->\_\_get\_lu\_seg\_of\_ship(\_unit, Ship::Orientation::HORIZONTAL, i);

if (eq(\_u, \_u0) || !this->\_\_is\_same\_ship(\_u, \_unit, i))

\_u = this->\_\_get\_lu\_seg\_of\_ship(\_unit, Ship::Orientation::VERTICAL, i);

if (eq(\_u, \_u0) || !this->\_\_is\_same\_ship(\_u, \_unit, i))

{

this->\_hit\_units.insert(\_unit);

return;

}

Ship &\_ship = this->\_deployed\_ships.at(\_u);

\_ship.hit(i);

if (\_ship.segments()[i] == Ship::Integrity::DESTROYED)

this->\_hit\_units.insert(\_unit);

if (\_ship.is\_destroyed())

this->\_\_block\_units\_for\_hit(\_u, \_ship);

}

void

Field::shot(std::uint64\_t \_x, std::uint64\_t \_y)

{ return this->shot(Unit(\_x, \_y)); }

const Field::Size&

Field::size() const noexcept

{ return this->\_size; }

void

Field::\_\_block\_unit(const Unit &\_unit, const Ship::Orientation &\_orie, bool \_flag)

{

Unit \_u;

for (std::int8\_t i = -1; i < 2; ++i)

{

\_u = Unit(

\_orie == Ship::Orientation::VERTICAL

? \_unit.x() + i

: \_unit.x(),

\_orie == Ship::Orientation::HORIZONTAL

? \_unit.y() + i

: \_unit.y()

);

if (!this->\_\_is\_valid\_unit(\_u))

continue;

if (\_flag)

this->\_unusable\_units.insert(\_u);

else

this->\_hit\_units.insert(\_u);

}

}

void

Field::\_\_block\_units(const Unit &\_unit, const Ship &\_ship, bool \_flag)

{

for (std::int8\_t i = -1; i <= \_ship.len(); ++i)

this->\_\_block\_unit(Unit(

\_ship.orientation() == Ship::Orientation::HORIZONTAL

? \_unit.x() + i

: \_unit.x(),

\_ship.orientation() == Ship::Orientation::VERTICAL

? \_unit.y() + i

: \_unit.y()

), \_ship.orientation(), \_flag);

}

void

Field::\_\_block\_units\_for\_add(const Unit &\_unit, const Ship &\_ship)

{ return this->\_\_block\_units(\_unit, \_ship, true); }

void

Field::\_\_block\_units\_for\_hit(const Unit &\_unit, const Ship &\_ship)

{ return this->\_\_block\_units(\_unit, \_ship, false); }

bool

Field::\_\_is\_valid\_unit(const Unit &\_u) const noexcept

{

return \_u.x() > 0 && \_u.x() <= this->\_size.m()

&& \_u.y() > 0 && \_u.y() <= this->\_size.n();

}

bool

Field::\_\_is\_valid\_unit(std::uint64\_t \_x, std::uint64\_t \_y) const noexcept

{ return this->\_\_is\_valid\_unit(Unit(\_x, \_y)); }

bool

Field::\_\_is\_same\_ship(const Unit &\_lu, const Unit &\_unit, std::uint8\_t \_offset)

{

Ship &\_ship = this->\_deployed\_ships.at(\_lu);

if (\_ship.len() <= \_offset)

return false;

Unit \_u(

\_ship.orientation() == Ship::Orientation::HORIZONTAL

? \_lu.x() + \_offset

: \_lu.x(),

\_ship.orientation() == Ship::Orientation::VERTICAL

? \_lu.y() + \_offset

: \_lu.y()

);

std::equal\_to<Unit> eq;

return eq(\_unit, \_u);

}

Unit

Field::\_\_get\_lu\_seg\_of\_ship(const Unit &\_unit, const Ship::Orientation &\_orie, std::uint8\_t &i)

{

Unit \_u;

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

{

\_u = Unit(

\_orie == Ship::Orientation::HORIZONTAL

? \_unit.x() - i

: \_unit.x(),

\_orie == Ship::Orientation::VERTICAL

? \_unit.y() - i

: \_unit.y()

);

if (!this->\_\_is\_valid\_unit(\_u))

break;

if (this->\_deployed\_ships.find(\_u) != this->\_deployed\_ships.end())

return \_u;

}

return Unit();

}

} // namespace seagame