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

ОТЧЕТ

по лабораторной работе № 2

по дисциплине «Объектно-ориентированное программирование»

Тема: «Полиморфизм»

Студент гр. 3343 _____ Бубякина Ю.В.

Преподаватель _____ Жангиров Т. Р.

Санкт-Петербург
2024

Цель работы

Изучить работу классов-интерфейсов, путём усовершенствования программы из предыдущей лабораторной работы. Необходимо создать: класс-интерфейс способности, класс менеджера-способностей и набор классов-исключений для обработки исключительных ситуаций.

Задание

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

- Двойной урон – следующая атака при попадании по кораблю нанесёт 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 кораблём;
- 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

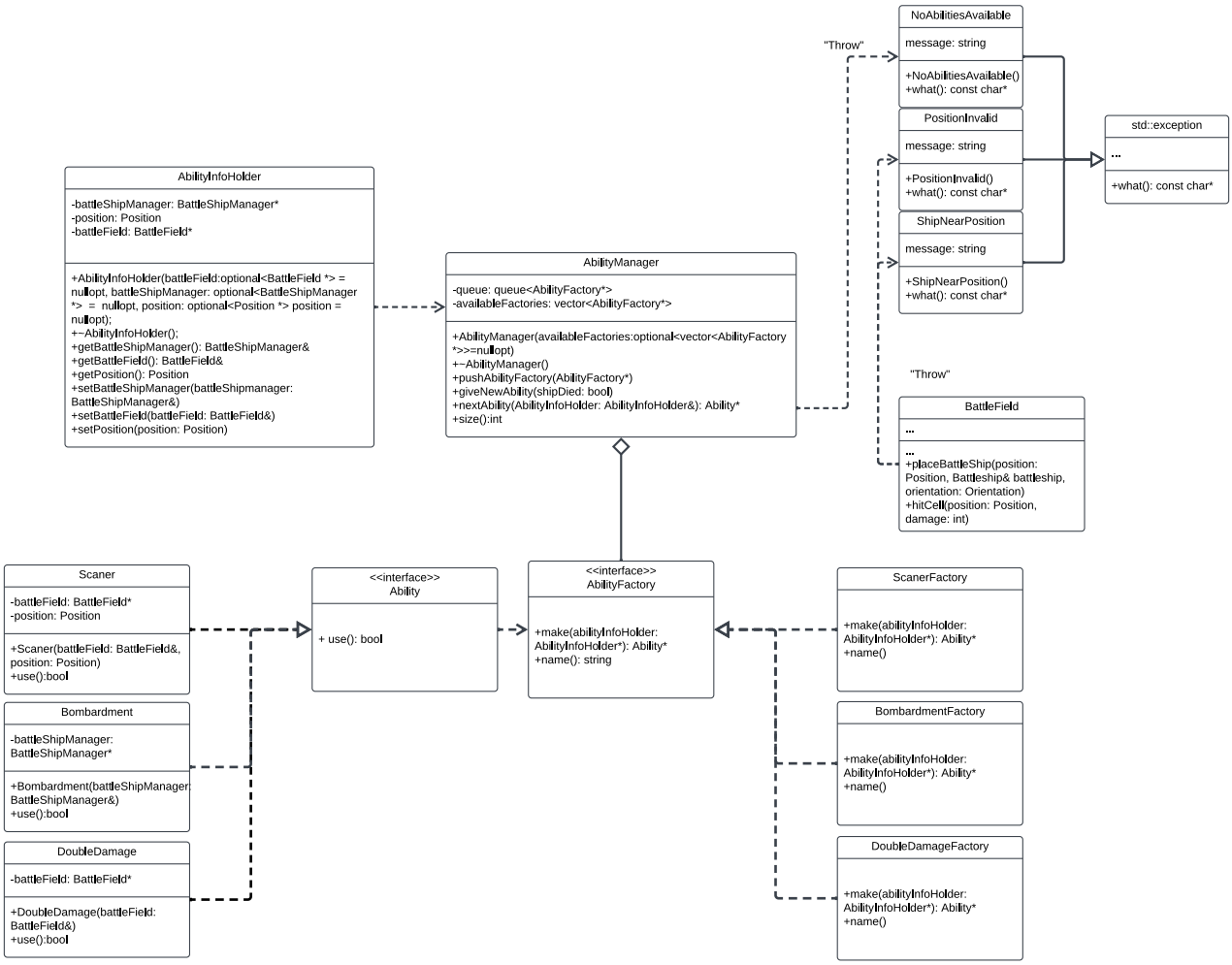

Рисунок 1 – UML-диаграмма классов

Код программы содержит реализацию классов: *Ability, Scaner, DoubleDamage, Bombardment, AbilityFactory, ScanerFactory, DoubleDamageFactory, BombardmentFactory, AbilityManager, AbilityInfoHolder,* а также классы исключений *NoAbilitiesAvailable, PositionInvalid, ShipNearPosition.*

Классы *Ability*, *DoubleDamage*, *Scanner*, *Bombardment* и *AbilityManager* были добавлены согласно заданию.

Классы *AbilityFactory, ScanerFactory, DoubleDamageFactory, BombardmentFactory, AbilityManager* были созданы, чтобы реализовать фабричный метод – паттерн, определяющий класс-интерфейс для создания

объектов, при этом оставляющий своим подклассам решение, какой класс создавать.

Помимо обозначенных классов, реализованы и интегрированы в код 3 классов-исключений для обработки различных исключительных случаев (использование способности когда нет доступных способностей, выход за границы поля, попытка установки корабля рядом с другим).

Ability — класс-интерфейс для способностей. Он имеет следующие виртуальные методы:

- *virtual bool use() = 0* – виртуальный метод для применения способности.
- *virtual ~Ability() {}* – виртуальный деструктор класса.

Класс *DoubleDamage* — реализация способности двойного урона. Он имеет следующие поля и методы:

- *BattleField* battleField* – указатель на поле.
- *DoubleDamage(BattleField& battleField)* – конструктор класса.
- *bool use() override* – устанавливает флаг *doubleDamageFlag* в объекте поля в положение *true*, после чего следующая атака через метод *hitCell* устанавливает урон = 2 и переключает флаг обратно в *false.*

Класс *Scaner* — реализация способности сканера, сообщающего о наличии/отсутствии кораблей в квадрате 2x2. Он имеет следующие поля и методы:

- *BattleField* battleField* – указатель на поле.
- *Position position* – координаты левого верхнего угла области применения способности.
- *Scaner(BattleField& battleField, Position& position)* – конструктор класса.
- *bool use() override* – сканирует область 2x2 в поле по координатам *position* (левый верхний угол) и возвращает *true/false*.

Класс *Bombardment* является реализацией способности выстрела по случайному сегменту случайного корабля. Он имеет следующие поля и методы:

- *BattleShipManager* battleShipManager* – указатель на менеджер кораблей.
- *Bombardment(BattleShipManager& battleShipManager)* – конструктор класса.
- *bool use() override* – наносится урон случайному сегменту случайного корабля не изменяя состояние клетки поля (способность обязательно попадает в не уничтоженный корабль в его не уничтоженный сегмент) и возвращает true в случае, если в результате попадания корабль был уничтожен.

Класс *AbilityFactory* — класс-интерфейс для классов-создателей способностей. Он имеет следующие виртуальные методы:

- *virtual Ability* make(AbilityInfoHolder& abilityInfoHolder) = 0* – виртуальный метод, необходимый для создание объекта способности.

Класс *DoubleDamageFactory* является реализацией создателя способности двойного урона. Он имеет следующие методы:

- *DoubleDamageFactory()* – конструктор класса.
- *Ability* make(AbilityInfoHolder& abilityInfoHolder) override* – создаёт объект класса *DoubleDamage* и возвращает указатель на него.

Класс *ScannerFactory* является реализацией создателя способности сканера. Он имеет следующие методы:

- *ScannerFactory()* – конструктор класса.
- *Ability* make(AbilityInfoHolder& abilityInfoHolder) override* – создаёт объект класса *Scanner* и возвращает указатель на него.

Класс *BombardmentFactory* является реализацией создателя способности сканера. Он имеет следующие методы:

- *BombardmentFactory()* – конструктор класса.

- *Ability* make(AbilityInfoHolder& abilityInfoHolder) override* – создаёт объект класса *Bombardment* и возвращает указатель на него.

Класс *AbilityManager* — отвечает за контроль над способностями, он хранит в очереди названия способностей, которые используются для создателей способностей. Он имеет следующие поля и методы:

- *std::queue<AbilityFactory*> queue* — очередь доступных для использования способностей.
- *std::vector<AbilityFactory*> availableFactories* — вектор возможных для создания способностей (при создании новой помещаются в *queue*)
- *AbilityManager(std::optional<std::vector<AbilityFactory*>> availableFactories = std::nullopt))* – конструктор класса.
- *void pushAbilityFactory(AbilityFactory *)* — добавляет новую способность в очередь.
- *void giveNewAbility(bool shipDied)* — в случае, если в *shipDied == true,* добавляет новую рандомную способность в очередь доступных способностей.
- *Ability *nextAbility(AbilityInfoHolder &abilityInfoHolder)* — возвращает способность находящуюся первой в очереди и удаляет её. Если нет доступных способностей в очереди, то бросает ошибку *NoAbilitiesAvailable*.
- *int size()* – возвращает размер очереди доступных способностей.

Тестирование:

```
try
{
    std::cout << "Попытка поставить корабль вне поля\n";
    battleField.placeBattleShip({1, 3}, battleShipManager[9], Orientation::Left);
    std::cout << "\n";
}
catch (PositionInvalid &e)
{
    std::cout << e.what() << '\n';
}
std::cout << "\n";

std::cout << "После расстановки кораблей" << battleField;

// Удар по кораблям
battleField.hitCell({5, 6});
battleField.hitCell({4, 3});
battleField.hitCell({4, 4});
battleField.hitCell({2, 1}, 2);

std::cout << "Попытка ударить вне поля\n";
try
{
    battleField.hitCell({100, 100});
}
catch (PositionInvalid &e)
{
    std::cout << e.what() << '\n';
}
std::cout << "\n";
std::cout << "Поле после ударов через поле" << battleField;

auto abilityInfoHolder = AbilityInfoHolder(&battleShipManager, &battleField);

auto emptyAbilityManager = AbilityManager(std::vector<AbilityFactory *>());
std::cout << "Взятие способности из пустого менеджера\n";
try
{
    emptyAbilityManager.nextAbility(abilityInfoHolder);
}
catch (NoAbilitiesAvailable &e)
{
    std::cout << e.what() << '\n';
}
std::cout << "\n";

std::cout << "Взятие способности из заполненного менеджера\n";
auto abilityManager = AbilityManager();
auto ability = abilityManager.nextAbility(abilityInfoHolder);
abilityManager.giveNewAbility(ability->use());
delete ability;
std::cout << "Способности после использования бомбардировки\n"
          << abilityManager << "\n";
std::cout << "Поле после бомбардировки" << battleField;

ability = abilityManager.nextAbility(abilityInfoHolder);
ability->use();
battleField.hitCell({4, 1});
delete ability;
std::cout << "Способности после использования двойного удара\n"
          << abilityManager << "\n";
std::cout << "Поле после двойного удара" << battleField;

abilityInfoHolder.setPosition({1, 1});
ability = abilityManager.nextAbility(abilityInfoHolder);
std::cout << (ability->use() ? "Ship found" : "No ship found") << '\n';
delete ability;
std::cout << "Способности после использования сканера удара\n"
          << abilityManager << "\n";
```

Рисунок 2 — Код тестирования программы.

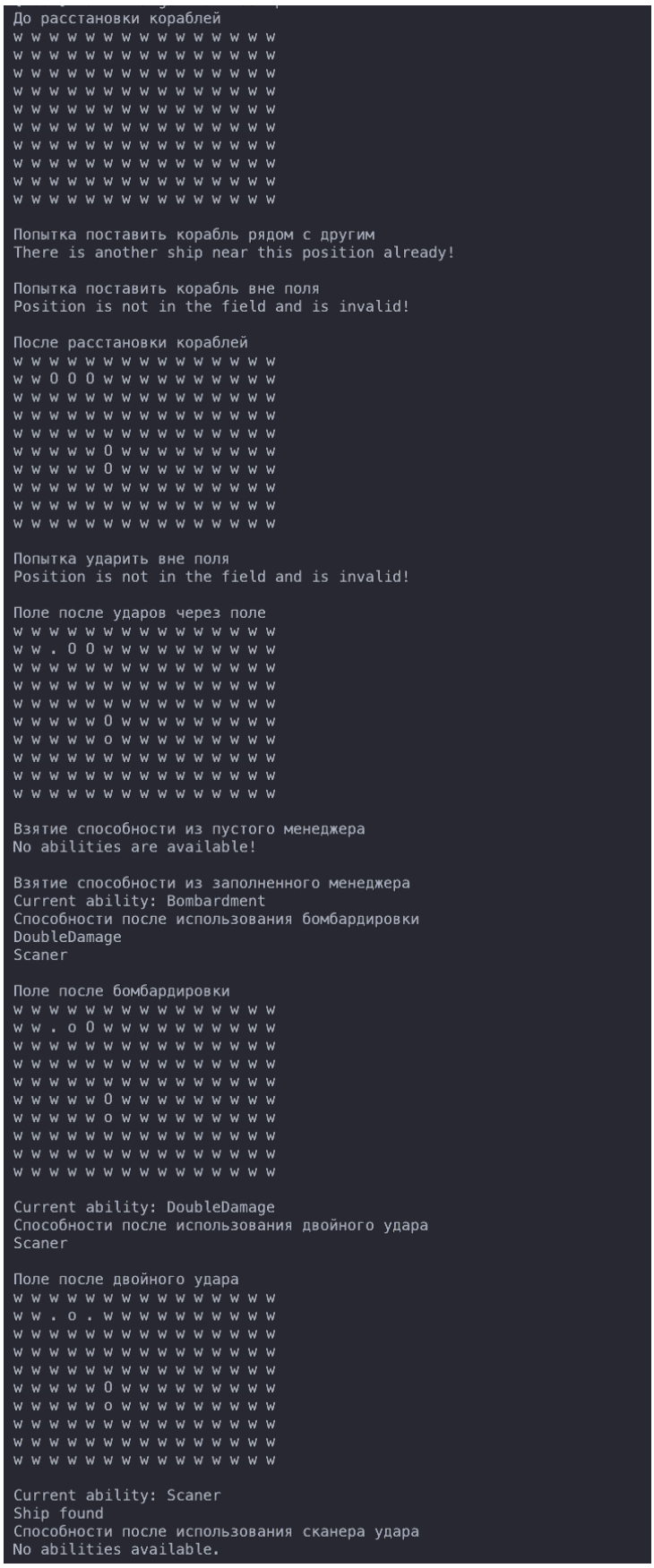

Рисунок 3 — Консольный вывод программы.

Выводы

В ходе лабораторной работы было исследовано использование классов-интерфейсов, а также разработаны: класс-интерфейс для представления способностей, класс менеджера способностей и набор специализированных классов для обработки исключительных ситуаций.