**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. 3341 |  | Бойцов В.А. |
| Преподаватель |  | Жангиров Т.Р. |

Санкт-Петербург

2024

## Цель работы.

Целью работы является изучение основ объектно-ориентированного программирования, наследования и полиморфизма, создание архитектуры способностей для написания игры «Морской бой».

Для выполнения поставленной цели требуется:

* Изучить основные принципы наследования и полиморфизма;
* Разработать архитектуру классов способностей, интерфейс способности;
* Реализовать эти классы и связь между ними.

## Задание.

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

Перед разработкой архитектуры способностей к ней был выдвинут ряд дополнительных требований, определивший вид разработанной структуры классов:

1. Архитектура должна быть масштабируемой для любых способностей;
2. Способности должны применяться единообразно через один интерфейс;
3. Для работы способностям могут потребоваться кардинально разные данные, не поддающиеся унификации;
4. Какие-то способности могут возвращать какой-то результат, который необходимо обрабатывать по-разному в зависимости от типа способности;
5. Способности и классы, связанные с ними, не должны принимать лишних данных или выполнять лишние действия в угоду унификации.

Начать описание разработанной архитектуры следует с непосредственно классов-способностей.

В качестве интерфейса способности используется класс IAbility, который определяет интерфейс применения способности, а именно – метод virtual void cast(), который необходимо переопределять всем способностям. Метод ничего не принимает: подразумевается, что в нём способности будут производить операции над полученными через конструктор данными, а также вызывать классы, связанные с обработкой результатов применения способности.

Конкретные способности реализуют интерфейc IAbility и представлены следующими классами:

* DoubleDamage – способность «Двойной урон». В качестве полей хранит в себе (и принимает в конструкторе, соответственно) ссылку на множитель урона int& multiplier. Подразумевается, что эта переменная будет храниться в некоем классе PlayerStats, отвечающим за параметры игрока, и будет использоваться при нанесении урона кораблям противника, однако разработка этого класса не входит в задачи данной лабораторной работы. Способность переопределяет метод cast(), в котором множитель урона устанавливается в два.
* Shelling – способность «Обстрел». В качестве полей хранит в себе (и принимает в конструкторе, соответственно) ссылку на менеджер кораблей ShipManager& manager. Переопределяет метод cast(), нанося урон случайному кораблю из списка активных в менеджере кораблей. Менеджер кораблей используется, так как из него проще всего (и логичнее всего) получать доступ к случайному кораблю для нанесения урона.
* Scanner – способность «Сканер». В качестве полей хранит в себе (и принимает в конструкторе, соответственно) ссылку на игровое поле Battlefield& field, игровые координаты на поле Coords cords, а также ссылку на класс обработчик результата AbilityResultHandler& handler, который будет описан немного ниже. В методе cast() способность проверяет каждую клетку игрового поля из обозначенного координатами сектора, считает количество сегментов кораблей, которые там содержатся. Затем способность записывает эту информацию в обработчик результата с помощью его метода setResult(AbilityResult\*).

Отдельно необходимо описать структуру возврата результата способности. Согласно п.4 и п.1, необходима такая структура, что способности могли бы возвращать разные результаты, но одним и тем же образом. Т.к. непосредственная обработка результатов способности может кардинально различаться и зависит исключительно от конкретной способности, необходимо передавать в клиент (пользователь способности) лишь общий результат, который клиент будет сам обрабатывать.

Для этого используется абстрактный класс AbilityResult, определяющий лишь тип результата возвращаемой способности через виртуальный метод AbilityType getType(). От него должны наследоваться все классы, отвечающие за представление результатов способностей – это позволяет унифицированно передавать их в клиент, который сам должен определять тип способности и что делать с ее результатом.

В текущей реализации это класс ScannerResult, который хранит в себе количество обнаруженных сегментов кораблей int mSegNum, а также может выдавать его с помощью метода int getResult().

Для передачи результата от способности к клиенту используется класс AbilityResultHandler, который хранит в себе указатель на базовый класс результата способности AbilityResult\* mResult. С помощью геттера и сеттера класс может запоминать результат способности и выдавать его.

Данная реализация использует некоторые элементы паттерна «Команда», упаковывая некоторое действие и его результаты в отдельный объект. Преимущество данной реализации в том, что для возврата любого результата из любой способности необходимо лишь создать класс результат способности и передавать его через AbilityResultHandler. Недостатки – клиенту необходимо самому приводить тип результата к желаемому, а также передавать в способность ссылку на AbilityResultHandler.

Теперь необходимо описать процесс создания способностей, для чего была создана достаточно сложная архитектура. Для каждой способности необходимо определить ряд аргументов и параметров, которые она принимает. Для этого создан общий интерфейс настроек способности IAbilitySettings, который определяет виртуальные методы AbilityType getType() и void acceptVisitor(IVisitor& visitor). О классе IVisitor и его роли в данной архитектуре будет сказано несколько позже. Для каждой способности были созданы реализующие IAbilitySettings классы DoubleDamageSettings, ShellingSettings и ScannerSettings, поля которых идентичны принимаемым аргументам соответствующей способности. Переопределенный метод acceptVisitor(IVisitor& visitor) принимает ссылку на класс-посетитель и вызывает его метод visit(), о чем также будет оговорено несколько позже.

Для создания конкретной способности был разработан класс AbilityFactory, который имеет полем указатель на базовый класс созданной способности IAbility\* mBuildedAbility, а методами - void buildDoubleDamage(DoubleDamageSettings\*), buildShelling(ShellingSettings\*), buildScanner(ScannerSettings\*) для создания способностей и IAbility\* getAbility() для получения созданной способности. В методах создания способности вызывается конструктор соответствующей способности, способность создаётся по передаваемым их класса параметров способности аргументам и записывается в mBuildedAbility.

По сути, данное решение представляет собой паттерн «Фабричный метод» с некоторыми модификациями для конструирования объектов способности.

Связующим звеном между настройками способности и фабричным методом служит класс, реализованый по принципу «Посетитель» AbilitySettingsVisitor, реализующий интерфейс IVisitor. В нем определены методы virtual void visit(DoubleDamageSettings\*) и др., для каждой способности. AbilitySettingsVisitor хранит в себе ссылку на AbilityFactory и переопределяет методы visit(), в каждом из которых вызывается соответствующий метод фабрики.

Т.к. метод visit() вызывается в классе конкретной настройки способности, у нас есть возможность передать в AbilitySettingsVisitor указатель this, что позволяет определить, какой тип способности необходимо создавать, и передать в точности необходимый тип без его явного приведения. В этом и кроется красота и выгода применения паттерна «Посетитель» в данном контексте.

Остаётся лишь описать класс менеджер способностей AbilityManager. Он имеет полями очередь типов способностей std::queue<AbilityType> mAbilities, фабрику способностей mFactory и посетитель настроек способности mVisitor. Методы класса:

* iAbility\* buildability(IAbilitySettings\*) – вызывает у настроеки способности метод acceptVisitor(), который вызывает по цепочке построение способности, а затем позвращает полученную способность через mFactory.getResult().
* Void castLastAbility(IAbilitySettings& settings) – вызывает метод постройки способности buildAbility, затем использует метод cast() созданной способности.
* AbilityType getFirstAbility() – позволяет узнать тип текущей способности в очереди
* Void addRandomAbility() – добавляет случайную способность в очередь способностей.

Диаграмма классов, разработанных в ходе выполнения лабораторной работы, представлена в общем виде на рис.1

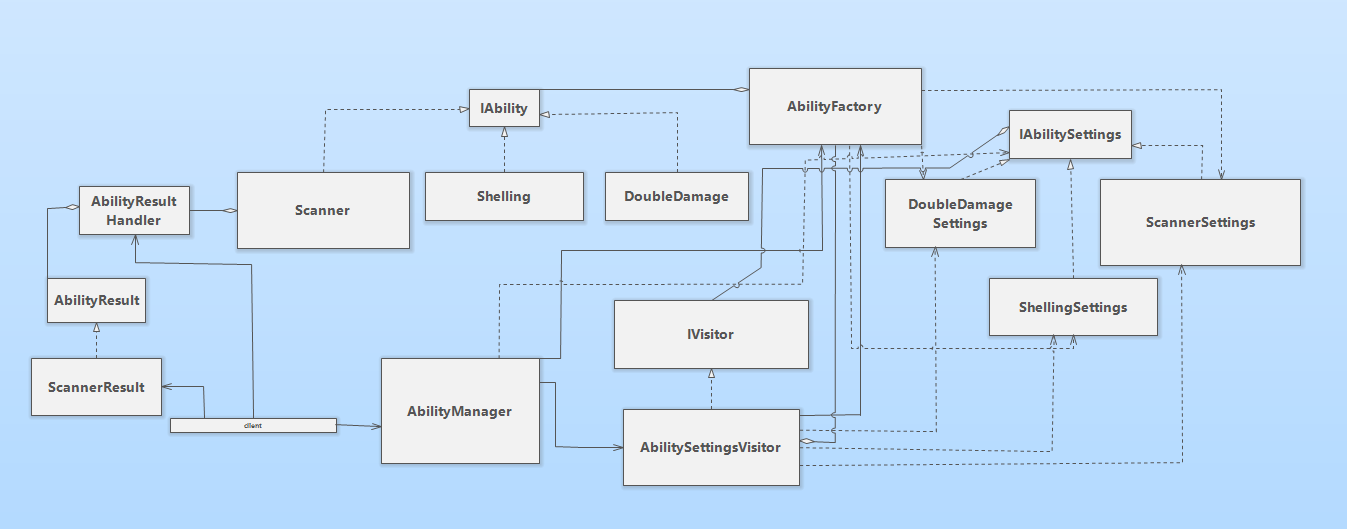


Рисунок 1 – диаграмма классов

Результаты тестирования см. в приложении А.

## Выводы.

В результате выполнения лабораторной работы были изучены основы Объектно-ориентированного программирования на языке С++, базовые принципы наследования и полиморфизма. Были реализованы классы способностей, их создание и хранение, прописаны методы взаимодействия с этими классами и между этими классами. Была написана программа, проверяющая работоспособность разработанных классов.

# Приложение Б Тестирование

Тестирование написанных классов было выполнено в виде небольшой программы, результат работы которой представлен ниже: