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

Студентка гр. 3383 _____ Земерова С.Н.

Преподаватель _____ Жангиров Т.Р.

Санкт-Петербург

2025

Оглавление

1. Цель работы

Разработать классы способностей для игры «Морской бой» в соответствии с техническим заданием.

2. Задание

a) Создать класс-интерфейс способности, которую игрок может применять. Через наследование создать 3 разные способности:
   i. Двойной урон - следующая атак при попадании по кораблю нанесет сразу 2 урона (уничтожит сегмент).
   ii. Сканер - позволяет проверить участок поля 2x2 клетки и узнать, есть ли там сегмент корабля. Клетки не меняют свой статус.
   iii. Обстрел - наносит 1 урон случайному сегменту случайного корабля. Клетки не меняют свой статус.

b) Создать класс менеджер-способностей. Который хранит очередь способностей, изначально игроку доступно по 1 способности в случайном порядке. Реализовать метод применения способности.

c) Реализовать функциональность получения одной случайной способности при уничтожении вражеского корабля.

d) Реализуйте набор классов-исключений и их обработку для следующих ситуаций (можно добавить собственные):
   i. Попытка применить способность, когда их нет
   ii. Размещение корабля вплотную или на пересечении с другим кораблем
   iii. Атака за границы поля

3. Модель предметной области и инварианты

3.1. Сущности

- Базовая способность (Ability): класс-интерфейс способности, которую игрок может применять, с виртуальным методом use().
- Менеджер способностей (AbilityManager): очередь случайных способностей для использования, связь с игровым полем и полем противника
- Конкретные способности:
  - DoubleDamage: наносит двойной урон по выбранной координате
  - Scanner: сканирует область 2×2 клетки, показывая наличие кораблей
  - Shelling: наносит 1 урон случайному живому сегменту случайного корабля
- Исключения способностей (AbilityException)
  - AbilityException: базовое исключение для ошибок способностей
  - EmptyQueueException: очередь способностей пуста
  - AbilityApplicationException: ошибка применения конкретной способности

3.2. Инварианты системы способностей

- Способности генерируются случайно из трех доступных типов
- Начальная инициализация - три уникальные способности в случайном порядке
- Способности применяются строго в порядке очереди (FIFO)
- После применения способности удаляется из очереди
- Новая способность добавляется в конец очереди при уничтожении корабля
- DoubleDamage: координаты должны быть в пределах поля противника

- Scanner: сканирует область 2x2 вокруг выбранной точки (частично, если у края)
- Shelling: применяется только если есть живые цели (неуничтоженные сегменты)
- Способность нельзя применить, если очередь пуста
- Способности работают только с полем противника
- Scanner не меняет статус клеток, только делает их видимыми
- Shelling не меняет статус клеток, но наносит урон соответствующему сегменту.

4. Детальное описание классов

4.1. Базовый класс способности Ability

Общее назначение

Класс Ability представляет абстрактную базовую сущность для всех специальных способностей. Основное назначение - предоставить единый интерфейс для реализации различных специальных возможностей, которые игрок может применять во время боя, расширяя стандартную механику выстрелов.

Структура данных

Приватные/защищенные поля:

- name - строковое название способности для идентификации

- description - текстовое описание функциональности способности

- coord - пара целых чисел (x, y) для хранения координат применения способности

Описание методов

Конструкторы:

Ability(const std::string& abilityName)

- Создает новую способность с указанным именем

- Инициализирует описание как "Способность: " + переданное имя

- Инициализирует координаты значением (-1, -1) - неопределенное положение

Виртуальные методы:

virtual void use(AbilityManager& manager) = 0

- Чисто виртуальный метод для применения способности

- Должен быть реализован в производных классах

- Принимает ссылку на менеджер способностей для взаимодействия с игровой логикой

- Определяет основную функциональность конкретной способности

Методы доступа к свойствам:

std::string name() const

- Возвращает название способности

std::string description() const

- Возвращает текстовое описание способности

std::pair<int, int> coordinates() const

- Возвращает координаты последнего применения способности

- Значение (-1, -1) указывает на то, что способность еще не применялась

Деструктор:

virtual ~Ability() = default

- Виртуальный деструктор для корректного удаления объектов производных классов

- Реализация по умолчанию

Особенности реализации

Применение шаблонного метода:

- Базовый класс определяет общую структуру, производные классы реализуют конкретное поведение

Координатная система:

- Координаты хранятся в формате std::pair<int, int>

- Инициализируются как (-1, -1) - признак неопределенности

- Заполняются конкретными значениями при применении способности

Расширяемость:

- Абстрактная природа класса позволяет добавлять новые типы способностей

- Все производные классы должны реализовать метод use()

- Совместимость с исключениями AbilityException для обработки ошибок

Взаимодействие с другими компонентами

С менеджером способностей (AbilityManager):

- Получает доступ к игровым полям через AbilityManager

- Использует методы менеджера для воздействия на игровое поле противника

- Взаимодействует с очередью способностей через менеджер

С конкретными способностями:

- Классы DoubleDamage, Scanner, Shelling наследуют от Ability

- Каждая производная способность определяет свою уникальную логику в методе use()

- Сохраняет единый интерфейс для всех типов способностей

4.2. Исключения от базового класса AbilityException

Общее назначение

Иерархия классов AbilityException предоставляет специализированную систему обработки ошибок для модуля способностей. Основное назначение - детализированное описание и классификация ошибок, возникающих при работе со способностями, с возможностью точной идентификации типа сбоя и контекста возникновения.

Структура иерархии (наследование)

```
std::runtime_error
│
└── AbilityException (базовый класс)
    │
    ├── EmptyQueueException (очередь пуста)
    │
    └── AbilityApplicationException (ошибка применения)
```

Описание классов

Базовый класс AbilityException

Конструктор:

explicit AbilityException(const std::string& message)

- Создает исключение с префиксом "Ошибка способности: " + пользовательское сообщение

- Использует explicit для предотвращения неявных преобразований

Деструктор:

virtual ~AbilityException() = default

- Виртуальный деструктор для корректного удаления объектов производных классов

- Реализация по умолчанию

Производный класс: EmptyQueueException

Назначение:

- Специализированное исключение для случаев попытки применения способности из пустой очереди

Конструктор:

EmptyQueueException()

- Создает исключение с фиксированным сообщением: "Нельзя применить способность — у Вас нет способностей"

Производный класс: AbilityApplicationException

Назначение:

- Специализированное исключение для ошибок во время применения конкретной способности

- Содержит информацию о имени способности и причине сбоя

Конструктор:

AbilityApplicationException(const std::string& abilityName, const std::string& reason)

- Принимает имя способности и причину ошибки

- Формирует сообщение: "Не удалось применить способность [abilityName]: [reason]"

- Сохраняет имя способности для последующего анализа

4.3. Класс способности Double Damage

Общее назначение

Способность DoubleDamage представляет специальную атаку. Назначение - нанесение двойного урона по выбранной координате, что позволяет мгновенно уничтожать сегменты кораблей противника.

Структура данных

Наследуемые поля:

- name - "Double Damage" (фиксированное название)

- description - текстовое описание механики способности

- coord - пара целых чисел (x, y) для хранения координат применения

Описание методов

Конструктор:

DoubleDamage()

- Создает способность "Двойной урон" с фиксированным названием

- Устанавливает описание: "Атака по кораблю нанесёт 2 урона (уничтожит сегмент)."

- Наследует базовую инициализацию координат (-1, -1)

Основной метод:

void use(AbilityManager& manager, int x, int y) override

Реализует логику применения способности двойного урона

- Применяет двойной урон по указанным координатам через функцию менеджера способностей - DamageEnemyField

- Обрабатывает исключения, преобразуя их в AbilityApplicationException

Условия корректной работы:

- Поле противника должно быть инициализировано в AbilityManager

- Координаты должны находиться в пределах игрового поля

- Способность должна находиться в очереди применения

Взаимодействие с *системой урона:*

- Интегрируется с трехуровневой системой состояний сегментов, но нарушает стандартную последовательность (сразу DESTROYED)

- Совместима с механикой подсчета уничтоженных кораблей

4.4. Класс способности Scanner

Общее назначение

Способность Scanner представляет собой подсказку для пользователя. Основное назначение - неразрушающее сканирование области 2×2 клетки для обнаружения кораблей противника без нанесения урона и изменения игрового состояния.

Структура данных

Наследуемые поля:

- name - "Scanner" (фиксированное название)

- description - текстовое описание механики сканирования

- coord - пара целых чисел (x, y) для хранения центра сканирования

Приватные поля:

- scanRange - радиус сканирования (фиксированное значение 1)

Описание методов

Конструктор:

Scanner()

- Создает способность "Сканер" с фиксированным названием

- Устанавливает описание: "Показывает содержимое участка поля размером 2x2 клетки."

- Инициализирует радиус сканирования значением 1

- Наследует базовую инициализацию координат (-1, -1)

Основной метод:

void use(AbilityManager& manager, int x, int y) override

Реализует основную логику работы сканирующей способности.

Получает от игрока координаты центральной точки сканирования.

Выполняет проверку области заданного радиуса вокруг этой точки (обычно 3×3 при scanRange = 1).

Показывает результаты проверки в консоли или в игровой системе отображения.

Помечает просканированные клетки как видимые, не изменяя их игрового состояния.

Перехватывает возможные исключения и преобразует их в AbilityApplicationException.

Вспомогательный метод:

std::vector<std::pair<Position, bool>> scanArea(AbilityManager& manager, Position center) const

Выполняет непосредственное сканирование выбранной области вокруг центральной клетки.

Формирует и возвращает набор пар: координата клетки + признак наличия корабля (true/false).

Корректно обрабатывает выход за границы поля, исключая недопустимые позиции.

Особенности сканирования:

- Область сканирования - 3×3 клетки (при scanRange=1)

- Граничные условия - у краев поля сканируется только доступная область

- Тип обнаружения - показывает наличие ЛЮБОГО сегмента корабля (независимо от состояния)

- Неразрушающий характер - не наносит урон и не меняет состояние клеток

Условия корректной работы:

- Поле противника должно быть инициализировано в AbilityManager

- Координаты центра должны находиться в пределах игрового поля

- Доступ к методам получения состояния клеток должен быть корректным

Взаимодействие с игровой механикой

- Использует set_enemy_cell_visible() для отображения scanned клеток

- Не влияет на постоянное состояние клеток (UNKNOWN/EMPTY/SHIP)

- Совместима с основной системой отображения игрового поля

4.5. Класс способности Shelling

Общее назначение

Способность Shelling представляет автоматизированную систему залпового огня. Назначение - нанесение гарантированного урона случайному живому сегменту корабля противника без необходимости ручного выбора цели.

Структура данных

Наследуемые поля:

- name - "Shelling" (фиксированное название)

- description - текстовое описание механики обстрела

- coord - пара целых чисел (x, y) для хранения координат пораженной цели

Статические методы:

- rndIndex() - генератор случайных индексов для выбора цели

Описание методов

Конструктор:

Shelling()

- Создает способность "Обстрел" с фиксированным названием

- Устанавливает описание: "Наносит 1 урон случайному живому сегменту."

- Наследует базовую инициализацию координат (-1, -1)

Основной метод:

void use(AbilityManager& manager) override

- Реализует логику автоматического обстрела случайной цели

- Собирает список всех доступных живых сегментов

- Выполняет несколько попыток нанесения урона при необходимости

- Сохраняет координаты успешно пораженной цели

- Обрабатывает различные сценарии ошибок через исключения

Приватные методы:

std::vector<std::pair<int,int>> collectAliveTargets(AbilityManager& manager) const

- Анализирует все корабли противника для составления списка целей

- Исключает уничтоженные корабли и уже уничтоженные сегменты

- Возвращает вектор координат доступных для атаки сегментов

static int rndIndex(int n)

- Генерирует случайный индекс в диапазоне [0, n-1]

- Использует std::random_device

- Обеспечивает равномерное распределение для случайного выбора целей

Механика применения

Критерии выбора целей:

- Только живые корабли - исключаются полностью уничтоженные корабли

- Только неповрежденные сегменты - INTACT или DAMAGED состояния

Особенности реализации

Система повторов

- Исключение неудачных целей - предотвращение повторных атак на одну цель

Взаимодействие с игровой механикой

С системой урона:

- Интегрируется со стандартной системой повреждений через damageEnemyField()

- Совместима с трехуровневой системой состояний сегментов

- Учитывает текущее состояние кораблей для корректного выбора целей

4.6. Класс менеджера способностей AbilityManager

Общее назначение

Класс **AbilityManager** управляет специальными способностями игрока: их созданием, хранением, применением, а также взаимодействием этих способностей с игровыми полями. Он поддерживает очередь способностей, предоставляет доступ к информации о поле противника и отвечает за корректное обновление состояния игры после применения способностей.

Структура данных

Приватные поля

ability_queue_ — очередь std::queue<std::shared_ptr<Ability>>, хранящая способности в порядке FIFO.

enemy_field_ — ссылка на поле противника (PlayingField).

field_ — ссылка на собственное поле игрока (PlayingField).

Статические функции

rng() — статический генератор псевдослучайных чисел типа std::mt19937, используется для выбора случайных способностей и перемешивания начального набора.

Описание методов

Конструкторы

AbilityManager() = default

Создает менеджер без инициализированных полей. Поля должны быть заданы позже через set_fields.

AbilityManager(PlayingField& enemy, PlayingField& self)

Инициализирует ссылки на поля противника и игрока.

Автоматически вызывает InitializeThreeUniqueAbilities(), добавляя в очередь по одному экземпляру каждой способности в случайном порядке.

Методы инициализации

void InitializeThreeUniqueAbilities()

Создает три способности: DoubleDamage, Scanner, Shelling.

Перемешивает их с помощью std::shuffle.

Добавляет все три способности в очередь.

std::shared_ptr<Ability> GenerateRandomAbility()

Генерирует одну случайную способность из трёх доступных

Использует std::uniform_int_distribution.

Управление очередью способностей

std::pair<int, int> ApplyNextAbility(int x, int y)

Снимает следующую способность из очереди.

Применяет её: ability->Use(this, x, y).

Возвращает координаты применения (ability → coordinates()).

Выбрасывает EmptyQueueException, если очередь пуста.

void AddNextAbility()

Добавляет новую случайную способность в конец очереди.

void clear()

Полностью очищает очередь способностей.

std::queue<std::shared_ptr<Ability>> ability_queue() const

Возвращает копию текущей очереди способностей.

Информационные методы

bool HasAbilities() const

Проверяет, есть ли способности в очереди.

size_t queue_size() const

Возвращает количество элементов в очереди.

std::string ability_name() const

Возвращает имя следующей способности или "Нет способностей".

std::string PeekNextAbility() const

Возвращает имя и описание следующей способности.

Формат: "Scanner: Показывает содержимое...".

std::string PrintAbilities() const

Форматированный список всех способностей в очереди.

Используется для отображения доступных способностей игроку.

Взаимодействие с игровым полем

int enemy_field_size_x() const

int enemy_field_size_y() const

Возвращают размеры поля противника.

void DamageEnemyField(int x, int y, int dam = 1)

Наносит урон клетке поля противника.

Если функция PlayingField::Damage вернула **2** (полное уничтожение корабля), автоматически добавляет новую способность.

void set_enemy_cell_visible(int x, int y)

Делает клетку противника видимой игроку.

bool enemy_cell_state(int x, int y) const

Возвращает true, если клетка содержит корабль (IsShipCell), иначе false.

Работа с кораблями

const Ship& ship(int index) const

Возвращает корабль по индексу.

int ship_count() const

Возвращает количество кораблей на поле противника.

Управление состоянием менеджера

void set_fields(PlayingField& enemy, PlayingField& self)

Устанавливает ссылки на игровые поля.

void set_ability_queue(std::queue<std::shared_ptr<Ability>> new_queue)

Полностью заменяет очередь способностей новой.

void reset()

Полностью очищает очередь.

Заново создаёт три уникальные способности.

Операторы ввода/вывода

operator<<

Сериализует менеджер: количество способностей + их имена.

operatoОбработка граничных условийr>>

Десериализует менеджер: по имени способности создаёт правильный объект.

Особенности реализации

Механика пополнения очереди

Пополнение происходит:

- после уничтожения корабля (DamageEnemyField → AddNextAbility()),
- после инициализации менеджера,
- при загрузке через оператор >>.

Логика очереди:

FIFO: способности применяются строго в порядке добавления.

После применения способность удаляется из очереди.

Новые способности попадают в конец очереди.

4.7. Класс игрока Player

Общее назначение

Класс **Player** интегрируется с системой специальных способностей через компонент **AbilityManager**. Он обеспечивает их корректное применение, управление и начисление в рамках игрового процесса.

Структура данных — добавление для способностей

ability_manager_ — std::shared_ptr<AbilityManager>; менеджер способностей, управляющий созданием и очередью доступных игроку способностей.

Методы для работы со способностями

1. Применение способности

std::pair<int, int> UseAbility(int x, int y)

Активирует следующую способность, вызывая:

ability_manager_->ApplyNextAbility(x, y).

Возвращает координаты применения способности (определяются самой способностью).

Если ability_manager_ == nullptr или очередь пуста, выбрасывает EmptyQueueException.

Используется как специальное действие игрока, не связанное с обычным выстрелом.

2. Использование обычного хода (начисление способностей)

int MakeMove(std::unique_ptr<Player>& opponent, int x, int y)

Добавление в метод:

- При полном уничтожении корабля (res == 2):
    - увеличивает destroyed_ships_
    - если игрок — человек (**PlayerType::HUMAN**) и есть ability_manager, вызывает: ability_manager_->AddNextAbility()

Проверки и доступ

bool HasAbilities() const

- Проверяет, подключён ли менеджер способностей.
- Возвращает true, если ability_manager_ не равен nullptr.
- Не проверяет содержимое очереди — этим занимается AbilityManager.

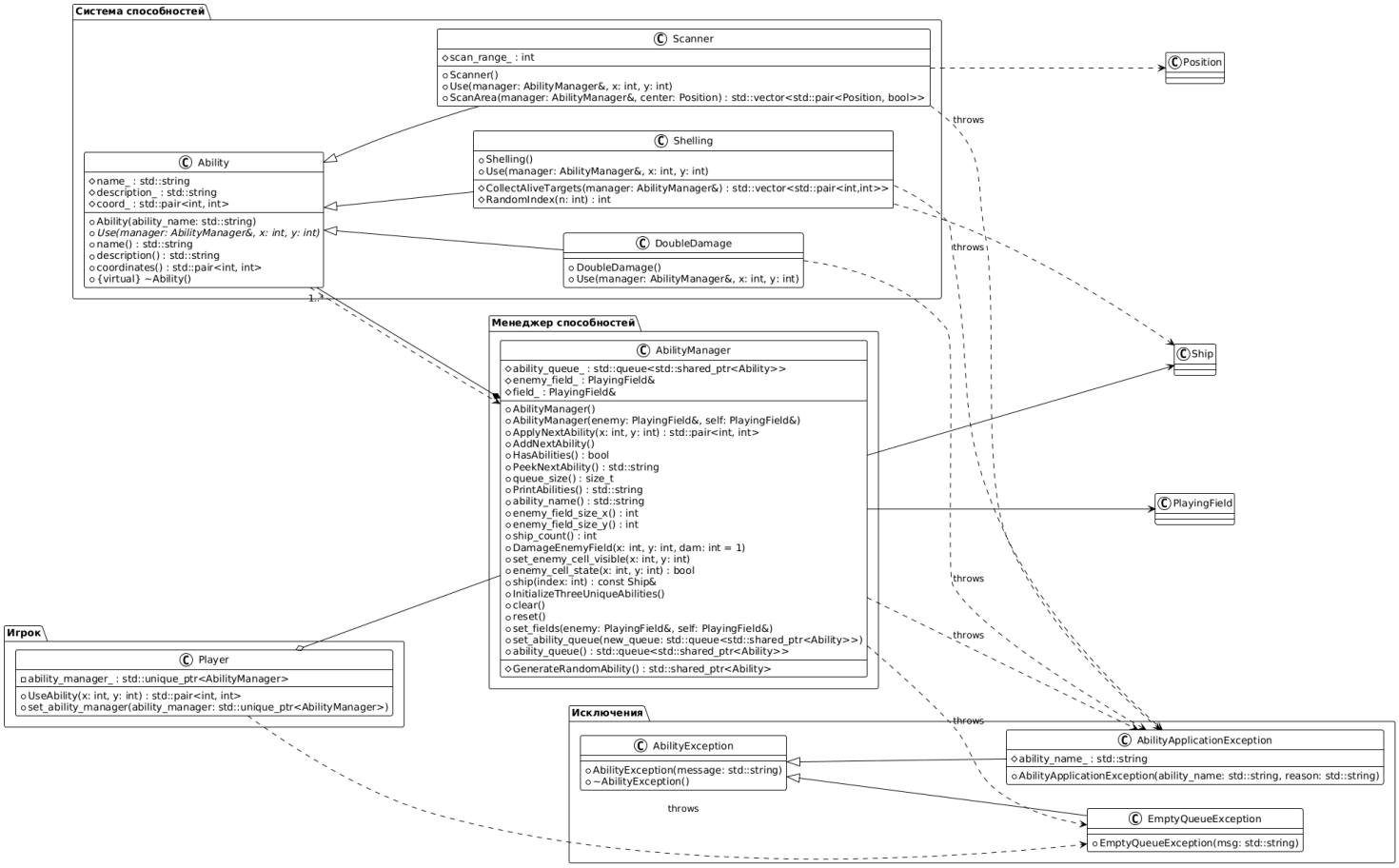
Управление состоянием менеджера способностей

void set_ability_manager(std::shared_ptr<AbilityManager> ability_manager)

- Назначает или заменяет менеджер способностей игрока.
- Позволяет менять систему способностей динамически (например, при загрузке сохранений или смене режима игры).

UML ДИАГРАММЫ!!!!

5. Диаграмма классов



Обоснование типов связей:

Композиция (Composition):

AbilityManager → Ability: Менеджер полностью владеет очередью способностей через std::queue<std::shared_ptr<Ability>>, управляет их жизненным циклом. Способности не имеют смысла вне менеджера.

Агрегация (Aggregation):

Player → AbilityManager: Игрок использует менеджер способностей через unique_ptr<AbilityManager>, но менеджер является самостоятельной сущностью, которая может быть переиспользована.

Наследование (Inheritance):

Ability → DoubleDamage/Scanner/Shelling: Конкретные способности являются специализациями абстрактной способности и реализуют полиморфный интерфейс Use()

AbilityException → EmptyQueueException/AbilityApplicationException: Специализированные исключения наследуют общую логику обработки ошибок способностей

Ассоциации:

AbilityManager → PlayingField: Менеджер имеет постоянный структурный доступ к игровым полям для применения способностей

AbilityManager → Ship: Менеджер имеет доступ к кораблям для реализации механики способностей (особенно Shelling)

Зависимости:

Ability → AbilityManager: Способности временно используют менеджер в методе Use() для доступа к игровой логике

Scanner → Position: Сканер временно использует позиции для определения области сканирования

Shelling → Ship: Обстрел временно использует корабли для выбора случайных целей

Генерация исключений:

AbilityManager → EmptyQueueException: Бросает при попытке применить способность из пустой очереди

AbilityManager/DoubleDamage/Scanner/Shelling → AbilityApplicationException: Бросают при ошибках применения конкретных способностей

Player → EmptyQueueException: Бросает при вызове UseAbility() без доступных способностей

Принципы проектирования

Разделение ответственности:

- **Abilities:** Система специальных способностей с полиморфным поведением

Полиморфизм и расширяемость:

- Единый интерфейс Ability позволяет добавлять новые типы способностей
- Иерархия исключений обеспечивает детализированную обработку ошибок для разных сценариев
- Возможность легкого расширения системы новыми способностями

Инкапсуляция*:*

- Логика применения способностей инкапсулирована в классах
- Менеджер скрывает сложность управления очередью способностей
- Состояния объектов защищены и изменяются только через определенные методы
- Данные классов объявлены как private/protected с доступом через методы

Правильное направление зависимостей:

- Abilities → Core (специальные возможности зависят от базовой логики

6. Архитектурные решения

6.1. Причины выбора структуры классов и их взаимосвязей

1. Базовый класс Ability - полиморфный интерфейс

- Почему абстрактный класс: Класс Ability служит единым методом для всех специальных действий в игре. Это позволяет управлять разнородными способностями через общий интерфейс, что является прямым требованием задания для реализации полиморфизма.

- Чисто виртуальный метод Use(): Этот метод определяет общий протокол для применения любой способности. Его абстрактность гарантирует, что все конкретные способности будут обязаны предоставить свою реализацию.

- Поля name_ и description_: Наличие этих полей в базовом классе позволяет унифицировано получать информацию о способности для отображения в интерфейсе, не зная ее конкретного типа.

2. Иерархия исключений AbilityException - детализированная обработка ошибок

- Почему собственная иерархия: Стандартных исключений недостаточно для точной передачи доменной логики ошибок, связанных со способностями. Собственная иерархия позволяет перехватывать ошибки на разных уровнях детализации.

- EmptyQueueException: Четко отделяет ошибку "нет способностей" от ошибок их применения, что позволяет, например, предложить игроку выбрать другую опцию вместо аварийной остановки игры.

- AbilityApplicationException: Инкапсулирует контекст ошибки (имя способности и причину), что значительно упрощает формирование сообщений для пользователя.

3. Класс DoubleDamage – 1 тип способности

- Почему наследует Ability: Реализует конкретную механику, определенную в задании, через переопределение метода use().

- Логика в методе Use(): Взаимодействует с AbilityManager для нанесения урона, демонстрируя паттерн "Инверсия зависимостей" — способность зависит от абстрактного менеджера, а не наоборот.

- Обработка исключений: Преобразует стандартные исключения (например, от неправильных координат) в AbilityApplicationException, что поддерживает целостность системы ошибок.

4. Класс Scanner – 2 тип способности

- Почему наследует Ability: Предоставляет игровой механизм, не связанный с нанесением урона, но требующий взаимодействия с игровым полем.

- Метод ScanArea(): Вынесение логики сканирования в отдельный метод улучшает читаемость кода и следует принципу единственной ответственности. Метод Use() отвечает за общую координацию, а ScanArea() — за конкретные вычисления.

- Работа с областью: Алгоритм автоматически учитывает границы поля. Способность не меняет состояние клеток, а лишь делает их видимыми, что соответствует требованию задания.

5. Класс Shelling – 3 тип способности

- Почему наследует Ability: Реализует механику "авто-атаки", которая отличается от действий, требующих выбора цели игроком.

- Метод collectAliveTargets(): Централизует логику выбора целей. Это предотвращает дублирование кода и упрощает потенциальные изменения в правилах выбора (например, приоритет атаки на раненые корабли).

- Система повторов: Несколько попыток применения (do-while цикл) делают способность устойчивой к динамическим изменениям на поле (например, если цель была уничтожена другим игроком между сбором целей и применением урона). Это решение повышает отказоустойчивость.

- Статический метод rndIndex(): Инкапсуляция генерации случайных чисел внутри класса, где она нужна, следует принципу инкапсуляции.

6. Класс AbilityManager – координатор способностей

- Почему отдельный класс: Управление очередью способностей, их генерацией и применением — это сложная логика, которую нецелесообразно помещать в класс Player или PlayingField. Разделение ответственности делает систему более модульной и понятной.

- Использование std::queue: Очередь отражает требование применения способностей в порядке FIFO.

- Метод initializeThreeUniqueAbilities(): Гарантирует выполнение требования о начальном наборе из трех случайных уникальных способностей. Использование std::shuffle обеспечивает случайный порядок без дубликатов.

- Связь с игровыми полями: Менеджер хранит ссылки на поля, так как для применения способностей необходим доступ к игровому состоянию. Это агрегация, так как поля создаются и управляются вне менеджера.

- Интеграция с уничтожением кораблей: Метод DamageEnemyField не только наносит урон, но и проверяет, был ли корабль уничтожен, чтобы добавить новую способность.

7. Интеграция способностей в класс Player

- Поле ability_manager_: Агрегация менеджера через unique_ptr позволяет одному менеджеру использоваться одним объектом (у одного игрока свой список способностей) и корректно управлять временем жизни.

- Метод UseAbility(): Предоставляет простой и безопасный интерфейс для игрового цикла, чтобы активировать следующую способность. Инкапсулирует внутри себя проверку на наличие способностей.

- Начисление способностей в MakeMove(): Логика добавления случайной способности за уничтожение корабля интегрирована в основной метод хода игрока.

6.2. Пакетная структура

```
src/
├── core/         # Базовая логика игры (из ЛР №1)
│   ├── Player      # Класс игрока
│   ├── ...
└── abilities/     # Система способностей (ЛР №2)
    ├── Ability.h/cpp     # Базовый класс способности
    ├── AbilityException.h/cpp      # Исключения способностей
    ├── DoubleDamage.h/cpp      # Способность «Двойной урон»
    ├── Scanner.h/cpp       # Способность «Сканнер»
    ├── Shelling.h/cpp       # Способность «Обстрел»
    └── AbilityManager.h/cpp       # Менеджер способностей
```

6.3. Принципы

Принцип единственной ответственности (SRP)

- Ability: Определяет метод для всех способностей.
- DoubleDamage, Scanner, Shelling: Реализуют одну конкретную игровую механику.
- AbilityManager: Управляет генерацией, очередью и применением способностей.
- AbilityException и наследники: Представляют конкретные типы ошибок.

Принцип расширяемости

- Система открыта для расширения: новая способность создается путем наследования от Ability и реализации метода use().

Принцип разделения интерфейса (ISP)

- Интерфейс Ability содержит минимально необходимый набор методов (Use(), name() и т.д.), что не заставляет реализации зависеть от методов, которые им не нужны.

Принцип инверсии зависимостей (DIP)

- Конкретные способности зависят от абстракции (AbilityManager), так как принимают его в методе Use() для доступа к игровому состоянию.

6.4. Выбор структур данных и алгоритмов

std::queue<std::shared_ptr<Ability>> для очереди способностей

- Выбор очереди: Структура "очередь" (FIFO) соответствует требованию задания — способности применяются в том порядке, в котором были получены.

- Использование shared_ptr: Умные указатели автоматически управляют временем жизни объектов способностей, что предотвращает утечки памяти.

std::vector и std::pair в вспомогательных методах

- В Scanner::ScanArea и Shelling::CollectAliveTargets: Использование вектора пар std::vector<std::pair<Position, bool>> позволяет эффективно собрать и вернуть результаты методов.

Стратегия обработки ошибок через исключения

- Использование иерархии исключений: Позволяет точно идентифицировать тип ошибки и обработать ее соответствующим образом.

- Преобразование исключений: Конкретные способности преобразуют стандартные и доменные исключения из Core в AbilityApplicationException, что поддерживает согласованность на уровне системы способностей.

7. Выводы

В ходе лабораторной работы была успешно реализована система специальных способностей для игры «Морской бой» с использованием механизмов полиморфизма.

Достигнутые результаты:

1. Соответствие техническому заданию

- Создан полиморфный интерфейс Ability с чисто виртуальным методом use()
- Реализованы три конкретные способности: DoubleDamage, Scanner и Shelling, каждая с уникальной игровой механикой
- Разработан класс AbilityManager для управления очередью способностей по принципу FIFO
- Реализована система начисления случайных способностей при уничтожении кораблей противника
- Создана иерархия доменно-специфичных исключений AbilityException для обработки ошибок

2. Применение принципов ООП

- Полиморфизм: Единый интерфейс Ability позволяет работать с разнотипными способностями через указатели на базовый класс
- Инкапсуляция: Логика применения каждой способности инкапсулирована в соответствующих классах, данные защищены
- Наследование: Построена четкая иерархия классов способностей и исключений
- Абстракция: Сложность реализации скрыта за простыми и понятными интерфейсами

3. Реализация стандартов C++ и паттернов проектирования

- Применены умные указатели (std::shared_ptr, std::unique_ptr) для автоматического управления памятью
- Использованы STL-контейнеры (std::queue, std::vector) для эффективного хранения и управления данными
- Реализована сериализация/десериализация через перегрузку операторов ввода/вывода
- Применен паттерн "Шаблонный метод" в иерархии способностей

4**. Обеспечение надежности**

- Комплексная система исключений обеспечивает детализированную обработку ошибок
- Механизм повторов в способности Shelling повышает устойчивость к динамическим изменениям состояния игры
- Валидация входных данных и граничных условий предотвращает некорректные состояния