

Интерфейсы и абстракции в играх: Волшебный мир закулисъя

Интерфейс = Универсальный разъем для способностей

Представь, что в игре есть **универсальные разъемы для способностей** (как USB-порты). Любой предмет, заклинание или умение, которое имеет такой “разъем”, может подключиться к персонажу:

```
// ИНТЕРФЕС - это "контракт" или "правила подключения"
class IMожетАтаковать { // Буква I в начале - обозначение интерфейса
public:
    virtual void Атаковать() = 0; // Чисто виртуальный метод
};
```

```
class IMожетКолдовать {
public:
    virtual void ПроизнестиЗаклинание() = 0;
    virtual int ПолучитьМану() = 0;
};
```

```
class IMожетЛечить {
public:
    virtual void Лечить(Персонаж& цель) = 0;
};
```

Аналогия из игр:

- **Skyrim**: Слоты для рук (правая/левая) - интерфейсы
- Любой предмет с интерфейсом IMожноДержатьВРуке можно экипировать
- **World of Warcraft**: Слоты для талантов - каждый талант реализует интерфейс IУмение

Зачем нужны интерфейсы? Представь магазин в RPG:

// Без интерфейсов - кошмар!

```
void КупитьПредмет(Предмет предмет) {
    if (предмет.тип == "оружие") {
        персонаж.оружие = предмет;
    } else if (предмет.тип == "броня") {
        персонаж.броня = предмет;
    } else if (предмет.тип == "зелье") {
        персонаж.инвентарь.добавить(предмет);
    } else if (предмет.тип == "свиток") {
        // и так 100 раз...
```

```

        }
    }

// С интерфейсами - элегантно!
class IMожноКупить {
public:
    virtual void ДобавитьВИнвентарь(Персонаж& покупатель) = 0;
    virtual int ПолучитьЦену() = 0;
};

void КупитьПредмет(IMожноКупить& предмет, Персонаж& покупатель) {
    if (покупатель.золото >= предмет.ПолучитьЦену()) {
        покупатель.золото -= предмет.ПолучитьЦену();
        предмет.ДобавитьВИнвентарь(покупатель);
    }
}
}

// Теперь ЛЮБОЙ предмет может быть куплен, если реализует интерфейс
class Меч : public IMожноКупить {
    void ДобавитьВИнвентарь(Персонаж& п) override {
        п.ЭкипироватьОружие(this);
    }
};

class Зелье : public IMожноКупить {
    void ДобавитьВИнвентарь(Персонаж& п) override {
        п.инвентарь.добавить(this);
    }
};

```

Абстрактный класс = Полуфабрикат персонажа

Абстрактный класс - это как **шаблон персонажа в редакторе** (уже есть скелет, но нужно дорисовать детали):

// АБСТРАКТНЫЙ КЛАСС - нельзя создать объект

class Герой {

protected:

 string имя;

 int здоровье;

public:

// Виртуальный метод - можно переопределить

virtual void Представиться() {

```

        cout << "Я герой!" << endl;
    }

// ЧИСТО виртуальный метод - ДОЛЖЕН быть переопределен
virtual void ИспользоватьУльтимейт() = 0; // = 0 делает метод чисто
виртуальным

// Невиртуальный метод - нельзя переопределить
void ПолучитьУрон(int урон) {
    здоровье -= урон;
}
};

// НЕЛЬЗЯ:
// Герой неизвестный; // Ошибка! Абстрактный класс

// НАДО создать конкретного наследника:
class Паладин : public Герой {
public:
    void ИспользоватьУльтимейт() override {
        cout << "СВЯЩЕННЫЙ УДАР!" << endl;
        // Реализация для паладина
    }
};
};

Паладин утер; // Теперь можно!

```

Виртуальные методы = Кнопки с разными эффектами

Виртуальный метод - это как **кнопка на джойстике, которая делает разное в зависимости от игры:**

```

class УстройствоВвода {
public:
    // ОБЫЧНЫЙ метод - одинаковая реакция всегда
    void КнопкаДомой() {
        ВыключитьКонсоль(); // Всегда одно и то же
    }

    // ВИРТУАЛЬНЫЙ метод - разная реакция у наследников
    virtual void КнопкаA() {
        cout << "Нажата A" << endl; // Базовая реализация
    }
};

```

```

class ДжойстикPS5 : public УстройствоВвода {
public:
    void КнопкаA() override {
        Прыжок(); // В играх для PS5 - прыжок
    }
};

class ДжойстикXbox : public УстройствоВвода {
public:
    void КнопкаA() override {
        Ускорение(); // В играх для Xbox - ускорение
    }
};

class СломанныйДжойстик : public УстройствоВвода {
    // Не переопределяет КнопкаA() - использует базовую реализацию
};

```

Чисто виртуальные методы = Обязательные задания в квесте

Чисто виртуальный метод (= 0) - это обязательный квест, без выполнения которого нельзя двигаться дальше:

```

class Квест {
public:
    string название;

    // Обычный метод - есть реализация по умолчанию
    void Начать() {
        cout << "Квест " << название << " начат!" << endl;
    }

    // Чисто виртуальный метод - БЕЗ реализации
    // Каждый квест ДОЛЖЕН определить свою логику выполнения
    virtual void Выполнить() = 0; // = 0 - это как "TODO: заполнить"

    // Виртуальный метод - есть базовая реализация, но можно переопределить
    virtual void Завершить() {
        cout << "Квест завершен!" << endl;
        ВыдатьНаграду();
    }
};

```

```

// НЕЛЬЗЯ создать объект Квест - он абстрактный
// Квест главный; // Ошибка!

// Нужно создать конкретный квест:
class КвестУбитьГоблинов : public Квест {
public:
    КвестУбитьГоблинов() {
        название = "Охота на гоблинов";
    }

// ОБЯЗАТЕЛЬНО переопределить чисто виртуальный метод
void Выполнить() override {
    int убитоГоблинов = 0;
    while (убитоГоблинов < 10) {
        cout << "Убит гоблин! (" << убитоГоблинов+1 << "/10)" << endl;
        убитоГоблинов++;
    }
}

// МОЖНО переопределить виртуальный метод
void Завершить() override {
    cout << "Все гоблины уничтожены!" << endl;
    ВыдатьНаграду();
    ДатьДостижение("Охотник на гоблинов");
}
};

КвестУбитьГоблинов квест1; // Теперь можно!

```

Реальный пример: Система урона в Dark Souls

// ИНТЕРФЕЙС - всё, что может получать урон

```

class IУязвимый {
public:
    virtual void ПолучитьУрон(int урон, string типУрона) = 0;
    virtual bool Жив() const = 0;
};

// Абстрактный класс для всех существ
class Существо : public IУязвимый {
protected:
    int здоровье;
    int максимальноеЗдоровье;

```

```

public:
    virtual void Атаковать(IУязвимый& цель) = 0; // Чисто виртуальный

    // Реализуем интерфейс IУязвимый
    void ПолучитьУрон(int урон, string типУрона) override {
        // Базовая логика для всех существ
        здоровье -= урон;
        if (здоровье < 0) здоровье = 0;
        cout << "Получено " << урон << " " << типУрона << " урона!" << endl;
    }

    bool Жив() const override {
        return здоровье > 0;
    }

    // Виртуальный метод с реализацией
    virtual void ОтпраздноватьПобеду() {
        cout << "Я победил!" << endl; // Базовая реализация
    }
};

// Конкретные классы
class Игрок : public Существо {
private:
    int эсти;

public:
    void Атаковать(IУязвимый& цель) override {
        cout << "Игрок атакует!" << endl;
        цель.ПолучитьУрон(50, "физический");
    }

    // Переопределяем логику получения урона
    void ПолучитьУрон(int урон, string типУрона) override {
        // Игрок может парировать
        if (МожноПарировать() && типУрона == "физический") {
            cout << "Парирование!" << endl;
            урон = 0;
        }
        Существо::ПолучитьУрон(урон, типУрона); // Вызываем родительский
        метод
    }
};

class Босс : public Существо {

```

public:

```
void Атаковать(IУязвимый& цель) override {
    cout << "Босс использует смертельную атаку!" << endl;
    цель.ПолучитьУрон(100, "тьма");
}

void ОтпраздноватьПобеду() override {
    cout << "МУАХАХА! СМЕРТНЫЕ НИЧТОЖНЫ!" << endl;
    ИспугатьИгрока();
}
};
```

// Даже неживые объекты могут получать урон

```
class Бочка : public IУязвимый {
public:
    void ПолучитьУрон(int урон, string типУрона) override {
        if (типУрона == "огонь") {
            cout << "Бочка взрывается!" << endl;
            НанестиУронВокруг();
        }
    }

    bool Жив() const override {
        return false; // Бочка не живая
    }
};
```

Полиморфизм в действии: Аrena боя

// Функция принимает ЛЮБОГО, кто реализует IУязвимый

```
void НанестиМассовыйУрон(IУязвимый** цели, int количество, int урон) {
    for (int i = 0; i < количество; i++) {
        if (цели[i]->Жив()) {
            цели[i]->ПолучитьУрон(урон, "магический");
        }
    }
}

int main() {
    // Создаём разных существ
    Игрок артас;
    Босс саурфанг;
    Бочка бочка1;
    Бочка бочка2;
```

```
// Все они реализуют IУязвимый, поэтому можем собрать в один массив
IУязвимый* аrena[] = {&артас, &саурфанг, &бочка1, &бочка2};

// Наносим массовый урон ВСЕМ, независимо от их типа
НанестиМассовыйУрон(арена, 4, 30);

// Демонстрация виртуальных методов
Существо* существа[] = {&артас, &саурфанг};

for (auto существо : существа) {
    существо->Атаковать(артас); // Полиморфизм!

    if (!артас.Жив()) {
        существо->ОтпраздноватьПобеду(); // У каждого своя реализация
    }
}
```

Сравнение: Когда что использовать?

| Концепция | Аналогия в игре | Когда использовать | Пример |
|-------------------|----------------------|---|------------------------------|
| Интерфейс | Универсальный слот | Когда нужна совместимость разных классов | IMожноЭкипировать |
| Абстрактный класс | Шаблон персонажа | Когда есть общая логика для наследников | Существо с базовым здоровьем |
| Виртуальный метод | Настраиваемая кнопка | Когда нужно разное поведение у наследников | Атаковать() |
| Чисто виртуальный | Обязательный квест | Когда каждый наследник ДОЛЖЕН реализовать метод | ИспользоватьСпособность() |

Практическое задание: Создай систему умений для RPG

// 1. Создай интерфейс для всех умений

class ИУмение {

public:

virtual void Использовать(Персонаж& щель) = 0;

virtual void Использовать(Персонажа Целевого)

virtual string ПолучитьНазвание() = 0;

```

};

// 2. Создай абстрактный класс для заклинаний
class Заклинание : public IУмение {
protected:
    string название;
    int стоимостьМаны;
    int уровень;

public:
    virtual void ПроизнестиСлова() = 0; // Каждое заклинание имеет свои слова

    void Использовать(Персонаж& цель) override {
        if (цель.ПолучитьМану() >= стоимостьМаны) {
            цель.ИзменитьМану(-стоимостьМаны);
            ПроизнестиСлова();
            ПрименитьЭффект(цель);
        }
    }
};

virtual void ПрименитьЭффект(Персонаж& цель) = 0; // Чисто виртуальный
};

// 3. Реализуй конкретные заклинания
class ОгненныйШар : public Заклинание {
public:
    ОгненныйШар() {
        название = "Огненный шар";
        стоимостьМаны = 30;
        уровень = 1;
    }

    void ПроизнестиСлова() override {
        cout << "Игнис фламма!" << endl;
    }

    void ПрименитьЭффект(Персонаж& цель) override {
        cout << "ШАР ОГНЯ!" << endl;
        цель.ПолучитьУрон(50, "огонь");
        цель.ПрименитьЭффект("горение", 5); // ДоТ урон
    }
};

```

```

class Лечение : public Заклинание {
public:
    void ПроизнестиСлова() override {
        cout << "Санаре корпус!" << endl;
    }

    void ПрименитьЭффект(Персонаж& цель) override {
        cout << "Луч света исцеляет раны!" << endl;
        цель.Лечить(80);
        цель.ПрименитьЭффект("регенерация", 10); // ХоТ лечение
    }
};

// 4. Создай систему горячих клавиши
class ПанельСпособностей {
private:
    IУмение* слоты[10]; // 10 слотов для умений

public:
    void НазначитьУмение(int слот, IУмение* умение) {
        слоты[слот] = умение;
    }

    void ИспользоватьУмение(int слот, Персонаж& цель) {
        if (слоты[слот] != nullptr) {
            cout << "Используется: " << слоты[слот]->ПолучитьНазвание() <<
            endl;
            слоты[слот]->Использовать(цель);
        }
    }
};

```

Челлендж для тебя:

Создай систему, где:

1. Есть интерфейс IVзаимодействие с методами Осмотреть(), Взаимодействовать()

2. Разные объекты реализуют этот интерфейс:

- Сундук: при открытии дает предметы
- NPC: начинает диалог
- Дверь: открывается при наличии ключа

- Книга: показывает текст

3. Сделай так, чтобы игрок мог взаимодействовать с ЛЮБЫМ объектом через одну кнопку E

Помни: Интерфейсы - это как **универсальный язык** в игре. Если предмет “говорит” на языке “можно экипировать”, то его можно надеть. Если враг “говорит” на языке “можно атаковать”, то в него можно стрелять. Это делает код гибким и расширяемым!