Информатика

Принципы ООП

Электронные устройства

- Device – электронное устройство (абстрактное)
  - Можно включать и выключать (но КАК?)
  - Можно посмотреть информацию о работе (но КАКУЮ именно?)
- Phone и Camera – устройства
  - Понятно, как включается
  - Понятно, какая информация

Device

```
public abstract class Device
{
    protected Device(string name) { Name = name;}
    public string Name { get; private set; }
    public abstract void On();
    public abstract void Off();
    public abstract string GetInfo();
}
```

Абстрактный класс с тремя абстрактными методами.
Абстрактный метод не имеет реализации.
Чего не может быть у абстрактного класса?

Phone

```
public class Phone:Device
{
    public Phone(string name) : base(name)
    {…}
    public override void On() {…}
    public override void Off() {…}
    public override string GetInfo()
    {
        return "Можно звонить, хранить номера, принимать звонки";
    }
    public void Call() {…}
}
```

Должны быть реализованы все абстрактные методы или сам класс – потомок должен быть абстрактным

Camera

```
public class Phone:Device
{
    public Camera(string name) : base(name)
    {…}
    public override void On() {…}
    public override void Off() {…}
    public override string GetInfo()
    {
        return "Можно снимать фотографии и просматривать их";
    }
    public void TakePhoto() {…}
}
```

Полиморфизм

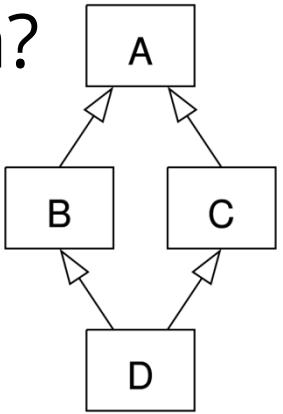
```
Device[] devices = new Device[10];

for(int i=0; i<devices.Length; i+=2)
{
    devices[i] = new Phone($"Phone {i}");
    devices[i + 1] = new Camera($"Camera {i+1}");
}

foreach (Device device in devices)
    Console.WriteLine(device.GetInfo());
```

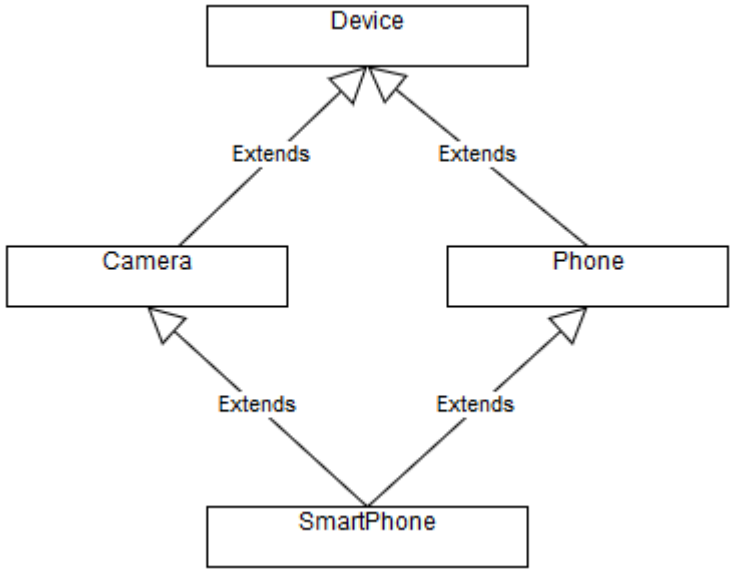
Смартфон

- В нём есть и камера и телефон
- Что делать?
  - Унаследовать от камеры и телефона?
  - В чём здесь проблема?
    - В ромбе!



Diamond problem

- Какой класс – базовый для SmartPhone?
- Чью реализацию брать?
  - GetInfo()?
  - On()? Off()?



Мешает реализация

- Определения (заголовки) то одинаковые…
- … а это идея!

Интерфейсы

- Вспоминаем понятие публичного интерфейса класса
  - Публичные методы
  - Публичные свойства
- Берём всё это в абстрактном виде (без реализации)

Интерфейс IDevice

```
public interface IDevice
{
    void On();
    void Off();
    string GetInfo();
}
```

Все методы – публичные и абстрактные «из коробки», модификаторов не пишем

Имя начинается с «I»

ICall и ITakePhoto

```
public interface ICall
{
    void Call();
}

public interface ITakePhoto
{
    void TakePhoto();
}
```

Наследование интерфейсов

Допустимо множественное наследование интерфейсов

Нет тела – нет проблем!

```
public interface IPhone : IDevice, ICall
{

}
```

Реализация интерфейса

- Классы – реализуют интерфейс
  - а не наследуют
  - только интерфейсы наследуют друг друга
- Класс обязан переопределить все методы интерфейса или остаться абстрактным

```
public class Phone : IPhone
{
    public void On() {…}
    public void Off() {…}
    public string GetInfo()
    {
        return "Можно звонить, хранить номера, принимать звонки";
    }
    public void Call() {…}
}
```

Решение проблемы множественного наследования

- Решение №1:
  - Унаследоваться от Phone и реализовать интерфейс TakePhoto
- Решение №2:
  - Реализовать два интерфейса: ICall, ITakePhoto

Решение №1

Наследуемся от Phone,
реализуем ITakePhoto

```
public class SmartPhone : Phone, ITakePhoto
{
    public void TakePhoto(){…}
}
```

Решение №2

```
public class SmartPhone : IPhone, ITakePhoto
{
    public void On(){…}
    public void Off(){…}
    public string GetInfo(){…}
    public void Call(){…}
    public void TakePhoto(){…}
}
```

Полиморфизм уровня интерфейсов

Не забываем про восходящее преобразование:

```
IDevice phone1 = new SmartPhone();
IPhone phone2 = new SmartPhone();
IPhone phone3 = new SmartPhone();
```

Абстрактный класс VS интерфейс

- Абстрактный класс может иметь реализованные методы
- Наследование от класса – единственное
- Реализовывать интерфейсы можно много раз
- Есть поля у абстрактного класса
- У интерфейса только методы и свойства
- У абстрактного класса могут быть private, protected члены

Когда что использовать?

А что, если...

... реализуем два интерфейса, содержащих методы с одинаковым именем

Это плохо, но предположим, что по-другому никак

```
public interface IDoItFast
{
    void Go();
}
public interface IDoItSlow
{
    void Go();
}
public class JustDoIt : IDoItFast, IDoItSlow
{
    void Go()=>Console.WriteLine("I just do it");
}
```

Но тогда...

... у нас будет общая реализация метода для обоих интерфейсов

```
JustDoIt simple= new JustDoIt();
simple.Go();                                // I just do it
IDoItFast fast = new JustDoIt();
fast.Go();                                  // I just do it
IDoItSlow slow = new JustDoIt();
slow.Go();                                  // I just do it
```

Если нужны разные реализации

Если нас не устраивает общая реализация

Если нам нужна специфика

- Что делать?
  - Реализовать несколько «одинаковых» методов?
- Как это провернуть?

Явная реализация интерфейсов

Опишем реализацию для каждого интерфейса

```
public interface IDoItFast
{
    void Go();
}
public interface IDoItSlow
{
    void Go();
}
public class JustDoIt : IDoItFast, IDoItSlow
{
    public void Go()     => Console.WriteLine("I just do it");
    void IDoItFast.Go() => Console.WriteLine("I do it fast");
    void IDoItSlow.Go() => Console.WriteLine("I do it slow");
}
```

Что это нам даст?

Разное поведение для каждого случая

```
JustDoIt simple= new JustDoIt();
simple.Go();                          // I just do it
IDoItFast fast = new JustDoIt();
fast.Go();                            // I do it fast
IDoItSlow slow = new JustDoIt();
slow.Go();                            // I do it slow
```

Более того...

... мы можем спрятать метод от беды подальше (вдруг подходящего общего поведения нет)

```
public interface IDoItFast
{
    void Go();
}
public interface IDoItSlow
{
    void Go();
}
public class JustDoIt : IDoItFast, IDoItSlow
{
    void IDoItFast.Go() => Console.WriteLine("I do it fast");
    void IDoItSlow.Go() => Console.WriteLine("I do it slow");
}
```

Сокрытие метода

- Выглядит странно (лучше не использовать)
- Делает метод невидимым извне при обращении через ссылку на класс (экземпляр класса), а не один из интерфейсов (интерфейсную ссылку)

```
JustDoIt simple= new JustDoIt();
//simple.Go();                          // Compilation error
IDoItFast fast = new JustDoIt();
fast.Go();                              // I do it fast
IDoItSlow slow = new JustDoIt();
slow.Go();                              // I do it slow
```

А вдруг...

... после сборки нам всё-таки нужно добавить общее поведение

... не затрагивая при этом описанный класс

... расширить его поведение

... извне

Метод расширения

- В статическом классе описывается статический метод, принимающий аргумент расширяемого типа как this
  - имитируем ссылку на текущий экземпляр

```
public static class JustDoItExtension
{
    public static void Go(this JustDoIt _)
        => Console.WriteLine("I just do it");
}
```

Всё вместе

```
public interface IDoItFast
{
    void Go();
}
public interface IDoItSlow
{
    void Go();
}
public class JustDoIt : IDoItFast, IDoItSlow
{
    void IDoItFast.Go() => Console.WriteLine("I do it fast");
    void IDoItSlow.Go() => Console.WriteLine("I do it slow");
}
public static class JustDoItExtension
{
    public static void Go(this JustDoIt _) => Console.WriteLine("I just do it");
}
```

Всё вместе

Теперь снова можно вызывать метод Go() по ссылке на экземпляр класса JustDoIt

```
JustDoIt simple= new JustDoIt();
simple.Go();                          // I just do it
IDoItFast fast = new JustDoIt();
fast.Go();                            // I do it fast
IDoItSlow slow = new JustDoIt();
slow.Go();                            // I do it slow
```

Что ещё позволяют методы расширения?

Расширять/дополнять стандартные классы

```
public static class IntExtension
{
    public static bool IsPrime(this int number)
    {
        if (number < 2) return false;
        for (int i = 2; i * i <= number; i++)
            if (number%i == 0)
                return false;
        return true;
    }
}
...
int number;
int.TryParse(Console.ReadLine(), out number);
Console.WriteLine(number.IsPrime());
```

Что ещё позволяют методы расширения?

- Добавлять типовые реализации интерфейсам, описывая методы расширения для них
  - Но у обычных методов и явных реализаций интерфейса будет больший приоритет
- Так работают методы LINQ (интегрированного языка запросов)

```
public static IEnumerable<TSource> Where<TSource>(
    this IEnumerable<TSource> source,
    Func<TSource, bool> predicate)
{
    ...
}
```

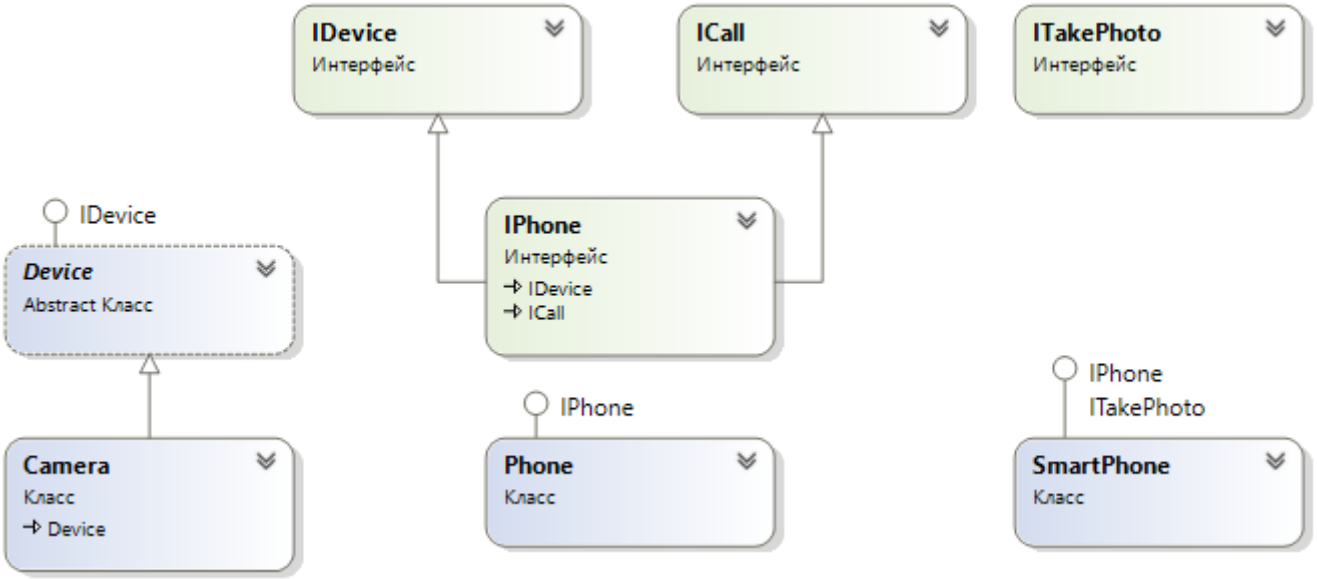
Принципы ООП

ООП:

- Объектно-Ориентированное *Программирование*
- Объектно-Ориентированное *Проектирование*
  - Object-Oriented Design

Design

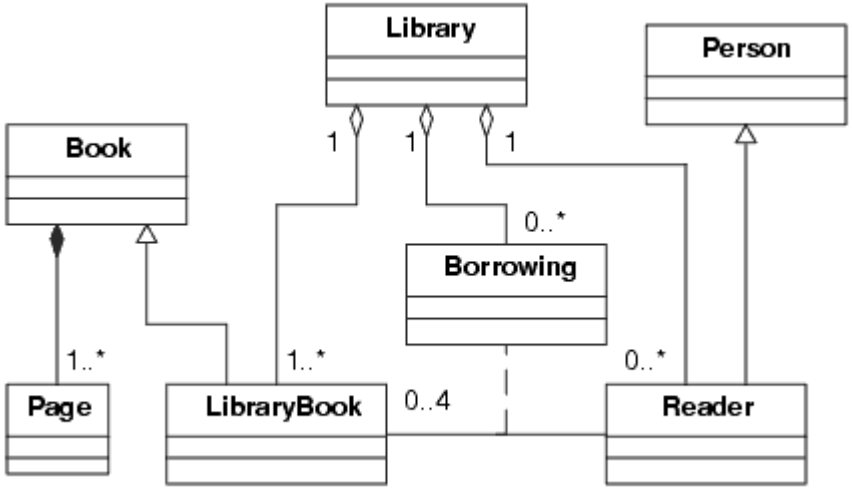
- Диаграмма классов
  - Одна из диаграмм визуального проектирования



на слайде не классическая диаграмма классов, а схема классов Visual Studio

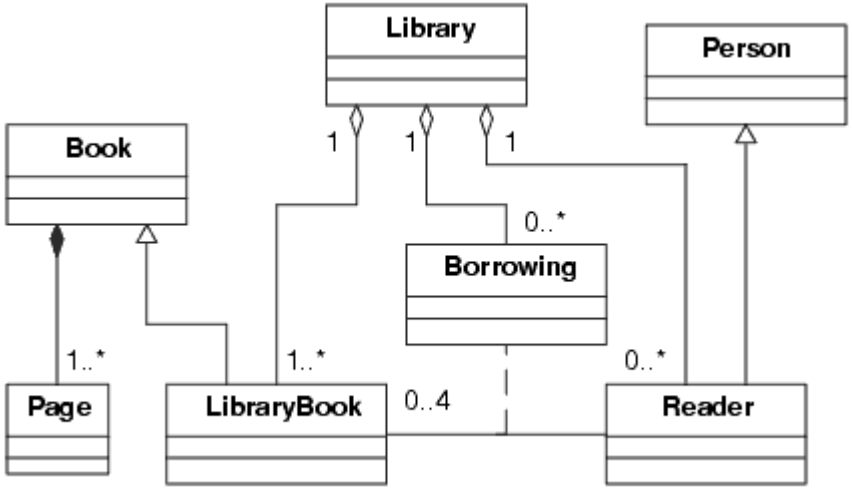
Виды связей между классами

- Наследование
- Ассоциация
- Агрегация
- Композиция



Виды связей между классами

- Страницы – неотъемлемая часть книги (*композиция*)
- Библиотечные книги – частный случай книги (*наследование*)
- Библиотечные книги содержатся в библиотеке (*агрегация*)
- Выписки – каталог *ассоциаций* между библиотечными книгами и читателями



Принципы проектирования

Пять основных принципов объектно-ориентированного программирования и проектирования

S.O.L.I.D.

«...акроним, введённый Майклом Фэзерсом для пяти принципов, предложенных Робертом Мартинов в начале 2000-х» - Wiki

S.O.L.I.D.

- Критерии проверки системы на изменяемость и расширяемость
- Руководства для проведения рефакторинга плохо спроектированного кода

Признаки плохой модели

Читайте про «Код с душком»: Model smell, Code Smell

Несколько примеров:

- Слишком большие/маленькие классы
- Одно изменение затрагивает несколько компонент.
- После изменений перестаёт работать то, что раньше работало
- Трудно выделить компоненты для повторного использования
- Трудно добиться корректного поведения, а выполнить некорректные действия – легко
- Неоправданная сложность, зависимость от модулей, не дающих непосредственной выгоды
- Трудно разобраться в проекте, анализировать его

Расшифровка акронима

- S – single responsibility
- O – open/closed
- L – Liskov substitution
- I – interface segregation
- D – dependency inversion

S: Single responsibility

Принцип единственной ответственности

Не должно быть больше одной причины для изменения класса

Если у класса больше одной ответственности, он будет меняться часто, дизайн будет хрупким, изменения - непредсказуемыми

Пример: валидация данных

Если осуществлять проверку данных в самом классе, то может возникнуть потребность в изменении класса при использовании его экземпляров в разных клиентах

Решение: делегировать валидацию стороннему объекту, чтобы основной объект не зависел от реализации валидатора

Предельные случаи (ошибки)

- Крайний пример несоблюдения принципа – God object
  - – божественный объект – знает и умеет всё
- Крайний пример соблюдения принципа - необоснованная сложность, размазывание логики

O: Open/closed

Программные сущности должны быть открыты для расширения (изменения), но закрыты для изменения

Модель должна быть устойчива к изменениям

Закрытость – стабильность интерфейсов

Открытость – можно изменять поведение без изменения исходного кода класса (пересборки)

Примеры ошибок:

- Использование конкретных объектов без выделения абстракций
- Привязка к абстракции, но выстраивание логики в зависимости от конкретных типов (is, typeof)

Решение: выделять абстракции, привязываться к ним, не требовать дополнительной информации о конкретных типах

Крайние случаи (ошибки)

- Крайний случай несоблюдения – отсутствие абстракций
- Крайний случай соблюдения – чрезмерное число уровней абстракции
  - «фабрика фабрик»

L: Liskov substitution

Принцип подстановки/замещения Барбары Лисков

Вместо базового типа всегда должна быть возможность подставить его подтип

Реализация «правильного» наследования

Требовать меньше, гарантировать больше

Примеры ошибок

- Ошибочное наследование
  - Неочевидное изменение поведения, несогласующееся с иерархией
  - Изменение интерфейса при наследовании
- Анти-пример – непонятное наследование (или слишком много или совсем нет)

I: Interface segregation

Клиенты не должны зависеть от методов, которыми не пользуются

Идея – предоставить удобный интерфейс с точки зрения различных клиентов

Не заставлять клиенты реализовывать методы, которые им не нужны

Примеры ошибок

- Вынуждение наследников знать и делать слишком много
- «Жирный» интерфейс: все функциональные возможности в одном интерфейсе (клиенты разделены, а интерфейс – нет)
- Анти-пример: тысяча интерфейсов (интерфейс на каждый метод)

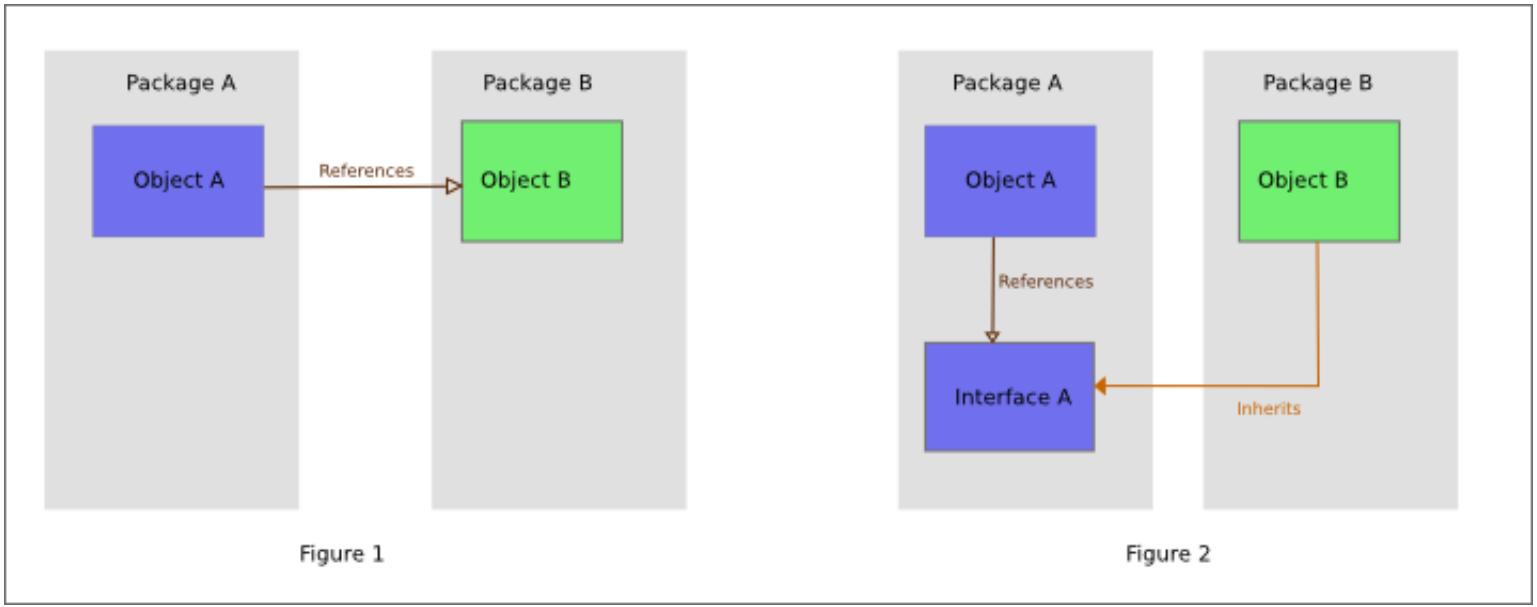
D: Dependency inversion

Принцип инверсии зависимостей

Модули верхнего уровня не должны зависеть от модулей нижнего, должны зависеть от абстракций

Идея – сделать ключевые или изменчивые зависимости явными

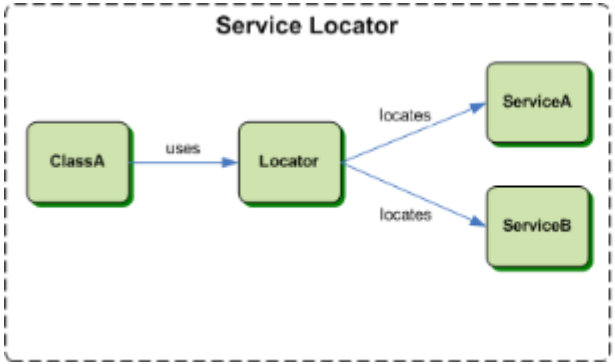
В чём инверсия

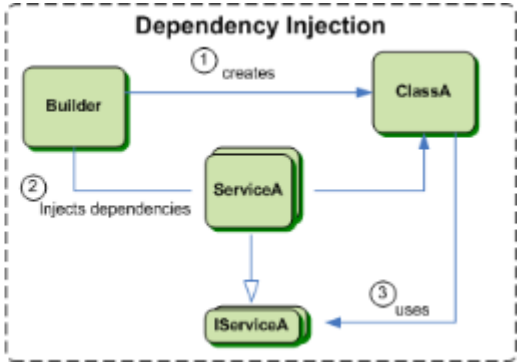


Картинка для Java, но суть - одна

Спойлер

- Инверсия завитимостей (Dependency Inversion)
- Инверсия управления (Inversion Of Control)
  - Service Locator
  - Dependency Injection (через конструктор, метод, интерфейс)





Примеры ошибок

Синглтоны, создание ключевых зависимостей в закрытых методах

Анти-пример: выделение интерфейса для каждого класса, передача всего и вся через конструкторы (невозможно понять где логика)

Спойлер

Обобщение опыта экспертов в решении типовых ситуаций

- Паттерны проектирования (Design Patterns)
  - Gang of Four и не только
- Архитектурные паттерны

Вопросы?

e-mail: marchenko@it.kfu.ru