**Принцип разделения интерфейсов** (Interface Segregation Principle) относится к тем случаям, когда классы имеют "жирный интерфейс", то есть слишком раздутый интерфейс, не все методы и свойства которого используются и могут быть востребованы. Таким образом, интерфейс получатся слишком избыточен или "жирным".

Принцип разделения интерфейсов можно сформулировать так:

**Клиенты не должны вынужденно зависеть от методов, которыми не пользуются.**

При нарушении этого принципа клиент, использующий некоторый интерфейс со всеми его методами, зависит от методов, которыми не пользуется, и поэтому оказывается восприимчив к изменениям в этих методах. В итоге мы приходим к жесткой зависимости между различными частями интерфейса, которые могут быть не связаны при его реализации.

В этом случае интерфейс класса разделяется на отдельные части, которые составляют раздельные интерфейсы. Затем эти интерфейсы независимо друг от друга могут применяться и изменяться. В итоге применение принципа разделения интерфейсов делает систему слабосвязанной, и тем самым ее легче модифицировать и обновлять.

Рассмотрим на примере. Допустим у нас есть интерфейс отправки сообщения:

interface IMessage

{

    void Send();

    string Text { get; set;}

    string Subject { get; set;}

    string ToAddress { get; set; }

    string FromAddress { get; set; }

}

Интерфейс определяет все основное, что нужно для отправки сообщения: само сообщение, его тему, адрес отправителя и получателя и, конечно, сам метод отправки. И пусть есть класс EmailMessage, который реализует этот интерфейс:

class EmailMessage : IMessage

{

    public string Subject { get; set; }

    public string Text { get; set; }

    public string FromAddress { get; set; }

    public string ToAddress { get; set; }

    public void Send()

    {

        Console.WriteLine("Отправляем по Email сообщение: {0}", Text);

    }

}

Надо отметить, что класс EmailMessage выглядит целостно, вполне удовлетворяя принципу единственной ответственности. То есть с точки зрения связанности (cohesion) здесь проблем нет.

Теперь определим класс, который бы отправлял данные по смс:

class SmsMessage : IMessage

{

    public string Text { get; set; }

    public string FromAddress { get; set; }

    public string ToAddress { get; set; }

    public string Subject

    {

        get

        {

            throw new NotImplementedException();

        }

        set

        {

            throw new NotImplementedException();

        }

    }

    public void Send()

    {

        Console.WriteLine("Отправляем по Sms сообщение: {0}", Text);

    }

}

Здесь мы уже сталкиваемся с небольшой проблемой: свойство Subject, которое определяет тему сообщения, при отправке смс не указывается, поэтому оно в данном классе не нужно. Таким образом, в классе SmsMessage появляется избыточная функциональность, от которой класс SmsMessage начинает зависеть.

Это не очень хорошо, но посмотрим дальше. Допустим, нам надо создать класс для отправки голосовых сообщений.

Класс голосовой почты также имеет отправителя и получателя, только само сообщение передается в виде звука, что на уровне C# можно выразить в виде массива байтов. И в этом случае было бы неплохо, если бы интерфейс IMessage включал бы в себя дополнительные свойства и методы для этого, например:

interface IMessage

{

    void Send();

    string Text { get; set;}

    string ToAddress { get; set; }

    string Subject { get; set; }

    string FromAddress { get; set; }

    byte[] Voice { get; set; }

}

Тогда класс голосовой почты VoiceMessage мог бы выглядеть следующим образом:

class VoiceMessage : IMessage

{

    public string ToAddress { get; set; }

    public string FromAddress { get; set; }

    public byte[] Voice { get; set; }

    public string Text

    {

        get

        {

            throw new NotImplementedException();

        }

        set

        {

            throw new NotImplementedException();

        }

    }

    public string Subject

    {

        get

        {

            throw new NotImplementedException();

        }

        set

        {

            throw new NotImplementedException();

        }

    }

    public void Send()

    {

        Console.WriteLine("Передача голосовой почты");

    }

}

И здесь опять же мы сталкиваемся с ненужными свойствами. Плюс нам надо добавить новое свойство в предыдущие классы SmsMessage и EmailMessage, причем этим классам свойство Voice в принципе то не нужно. В итоге здесь мы сталкиваемся с явным нарушением принципа разделения интерфейсов.

Для решения возникшей проблемы нам надо выделить из классов группы связанных методов и свойств и определить для каждой группы свой интерфейс:

interface IMessage

{

    void Send();

    string ToAddress { get; set; }

    string FromAddress { get; set; }

}

interface IVoiceMessage : IMessage

{

    byte[] Voice { get; set; }

}

interface ITextMessage : IMessage

{

    string Text { get; set; }

}

interface IEmailMessage : ITextMessage

{

    string Subject { get; set; }

}

class VoiceMessage : IVoiceMessage

{

    public string ToAddress { get; set; }

    public string FromAddress { get; set; }

    public byte[] Voice { get; set; }

    public void Send()

    {

        Console.WriteLine("Передача голосовой почты");

    }

}

class EmailMessage : IEmailMessage

{

    public string Text { get; set; }

    public string Subject { get; set; }

    public string FromAddress { get; set; }

    public string ToAddress { get; set; }

    public void Send()

    {

        Console.WriteLine("Отправляем по Email сообщение: {0}", Text);

    }

}

class SmsMessage : ITextMessage

{

    public string Text { get; set; }

    public string FromAddress { get; set; }

    public string ToAddress { get; set; }

    public void Send()

    {

        Console.WriteLine("Отправляем по Sms сообщение: {0}", Text);

    }

}

Теперь классы больше не содержат неиспользуемые методы. Чтобы избежать дублирование кода, применяется наследование интерфейсов. В итоге структура классов получается проще, чище и яснее.

Рассмотрим другой пример:

interface IPhone

{

    void Call();

    void TakePhoto();

    void MakeVideo();

    void BrowseInternet();

}

class Phone : IPhone

{

    public void Call()

    {

        Console.WriteLine("Звоним");

    }

    public void TakePhoto()

    {

        Console.WriteLine("Фотографируем");

    }

    public void MakeVideo()

    {

        Console.WriteLine("Снимаем видео");

    }

    public void BrowseInternet()

    {

        Console.WriteLine("Серфим в интернете");

    }

}

В наше время телефоны могут если не все, то очень многое, и представленный выше класс определяет ряд задач, которые выполняются стандартным телефоном: звонки, фото, видео, интернет.

Но пусть у нас есть также класс клиента, который использует объект Phone для фотографирования:

class Photograph

{

    public void TakePhoto(Phone phone)

    {

        phone.TakePhoto();

    }

}

Применим данный класс:

Photograph photograph = new Photograph();

Phone lumia950 = new Phone();

photograph.TakePhoto(lumia950);

Объект Photograph, который представляет фотографа, теперь может фотографировать, используя объект Phone. Однако более для фотосъемки можно использовать также и обычную фотокамеру, которая не обладает множеством возможностей телефона. И мы хотели бы, чтобы фотограф мог бы также использовать и фотокамеру для фотосъемки. В этом случае мы могли взять общий интерфейс IPhone и реализовать его метод TakePhoto в классе фотокамеры:

class Camera : IPhone

{

    public void Call() { }

    public void TakePhoto()

    {

        Console.WriteLine("Фотографируем");

    }

    public void MakeVideo() { }

    public void BrowseInternet() { }

}

Однако здесь мы опять сталкиваемся с тем, что клиент - класс Camera зависит от методов, которые он не использует - то есть методов Call, MakeVideo, BrowseInternet.

Для решения возникшей задачи мы можем воспользоваться принципом разделения интерфейсов:

interface ICall

{

    void Call();

}

interface IPhoto

{

    void TakePhoto();

}

interface IVideo

{

    void MakeVideo();

}

interface IWeb

{

    void BrowseInternet();

}

class Camera : IPhoto

{

    public void TakePhoto()

    {

        Console.WriteLine("Фотографируем с помощью фотокамеры");

    }

}

class Phone : ICall, IPhoto, IVideo, IWeb

{

    public void Call()

    {

        Console.WriteLine("Звоним");

    }

    public void TakePhoto()

    {

        Console.WriteLine("Фотографируем с помощью смартфона");

    }

    public void MakeVideo()

    {

        Console.WriteLine("Снимаем видео");

    }

    public void BrowseInternet()

    {

        Console.WriteLine("Серфим в интернете");

    }

}

Для применения принципа разделения интерфейсов опять же интерфейс класса Phone разделяется на группы связанных методов (в данном случае получается 4 группы, в каждой по одному методу). Затем каждая группа обертывается в отдельный интерфейс и используется самостоятельно.

При необходимости мы можем применить все интерфейсы сразу, как в классе Phone, либо только отдельные интерфейсы, как в классе Camera.

Теперь изменим код клиента - класса фотографа:

class Photograph

{

    public void TakePhoto(IPhoto photoMaker)

    {

        photoMaker.TakePhoto();

    }

}

Теперь фотографу не важно, что передается в метод TakePhoto - фотокамера или телефон, главное, что этот объект обладается только всем необходимым для фотографирования функционалом и больше ничем.