

Перевод фрагментов книги Judith Bishop. С# 3.0 Design Patterns.....	2
Структурные паттерны. Паттерн Декоратор (Decorator).....	2
Структурные паттерны. Паттерн Прокси (Proxy). Паттерн Мост (Bridge).....	14
Структурные паттерны. Паттерн Композит (Composite). Паттерн Приспособленец (Flyweight) .....	33
Структурные паттерны. Паттерн Адаптер (Adapter) .....	51
Порождающие паттерны. Абстрактная фабрика (Abstract Factory). Строитель (Builder)....	62
Структурные паттерны. Паттерн Фасад (Façade).....	75
Порождающие паттерны. Шаблон Одиночка (Singleton).....	76
Порождающие паттерны. Шаблон Фабричный метод (Factory Method) .....	77
Порождающие паттерны. Шаблон Прототип (Prototype).....	78
Поведенческие паттерны. Паттерн Стратегия (Strategy).....	80
Поведенческие паттерны. Паттерн Состояние (State) .....	80
Поведенческие паттерны. Паттерн Шаблонный метод (Template Method).....	81
Поведенческие паттерны. Паттерн Цепочка обязанностей (Chain of Responsibility) .....	82
Поведенческие паттерны. Паттерн Команда (Command).....	82
Поведенческие паттерны. Паттерн Итератор (Iterator).....	82
Поведенческие паттерны. Паттерн Медиатор (Mediator).....	82
Поведенческие паттерны. Паттерн Наблюдатель (Observer) .....	83
Поведенческие паттерны. Паттерн Посетитель (Visitor).....	83
Поведенческие паттерны. Паттерн Интерпретатор (Interpreter).....	84
Поведенческие паттерны. Паттерн Хранитель (Memento).....	84

# Перевод фрагментов книги Judith Bishop. C# 3.0 Design Patterns<sup>1</sup>

## Структурные паттерны. Паттерн Декоратор (Decorator)

### *Шаблон Декоратор (Decorator)*

#### Назначение

Обеспечить возможность динамического добавления к объекту нового состояния и поведения. Декорируемый объект не должен знать, что его декорируют. Это полезно в случае добавления возможностей к уже работающим системам. Ключевая особенность шаблона в том, что декоратор одновременно наследует от класса декорируемого объекта и содержит объект этого класса.

#### Пример

Пусть к фотографии нужно добавить рамку и теги, описывающие изображенные на фотографии объекты.



<sup>1</sup> Перевод И.И. Белялетдинова. Сайт оригинальной книги: <http://patterns.cs.up.ac.za>. Примеры <http://patterns.cs.up.ac.za/examples/Csharp%20-%20Design%20Patterns%20Programs-VS.zip>

На второй картинке находятся 4 объекта: исходная фотография, рамка, две надписи с различным содержанием. Каждый из них – это объект декоратор. Мы можем создать сколько угодно таких дополнений.

## Другие примеры

Солнцезащитные очки. Трансформируют солнечный свет. До глаз доходит тоже свет, но измененный. На входе и выходе декоратора всегда нечто одного типа, но возможно с измененными свойствами. Т.е. если бы вместо света до человека от очков доходил звук или запах, то очки были бы не декоратором. Свойство прозрачности декораторов позволяют их объединять друг с другом в цепочки.

## Преимущества паттерна:

- Исходный объект не знает о декорациях.
- Не создается один гигантский класс, включающий все возможные разнообразные вариации объекты.
- Каждый декоратор не зависит от другого.
- Декораторы могут быть скомбинированы друг с другом.

## Архитектура шаблона

В шаблоне задействованы:

### Компонент

Исходный класс, к объектам которого мы хотим добавить новое поведение или модифицировать существующее.

### Операция

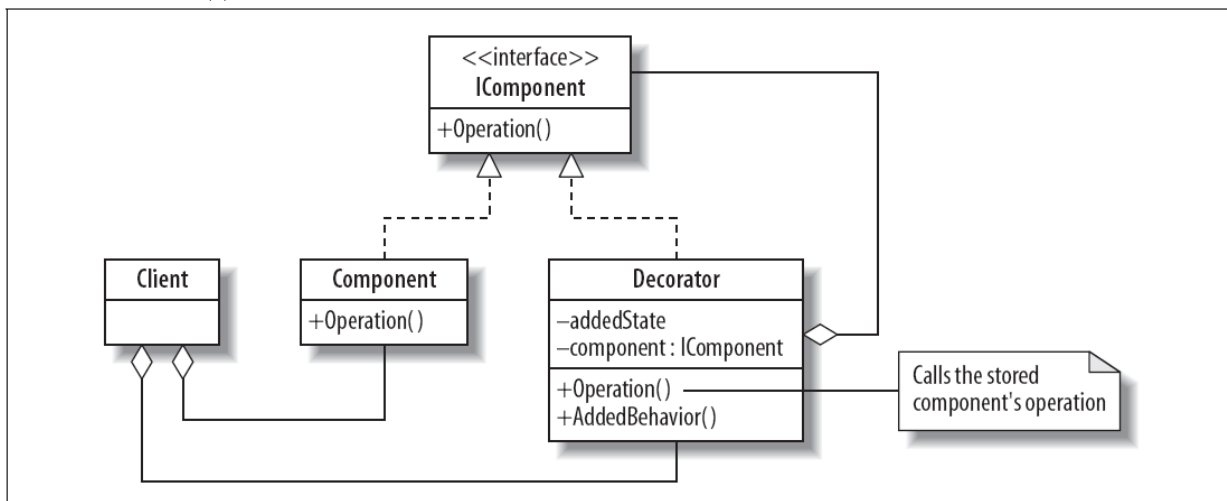
Метод(ы) интерфейса IComponent, который будет переопределен.

### IComponent

Интерфейс, который соответствует типу декорируемых объектов (Компонент один из них)

### Декоратор

Класс, который реализует интерфейс IComponent и добавляет к нему новое состояние/поведение.



Центральным классом является класс Декоратор. Он состоит в двух типах отношений с интерфейсом IComponent:

- 1) отношение «является разновидностью» (Is-a).

Показано пунктирной линией от Декоратора к IComponent. Это отношение дает нам возможность везде, где нужен IComponent, использовать Декоратор. То, что

Компонент также наследует от IComponent, позволяет взаимозаменять объекты Component и Decorator. Это характерная часть этого шаблона.

2) отношение «часть-целое» (Has-a).

Показано линией с ромбом, пристыкованным к IComponent. Это отношение означает, что Декоратор может создавать один или несколько объектов типа IComponent и тем самым их обрамлять. У Декоратора есть атрибут (типа IComponent), через который он может вызвать метод Operation интерфейса IComponent. В самом декораторе этот метод может быть переопределен. Именно благодаря этому оправдывается название Декоратора.

Метод addedBehavior и атрибут addedState – это способ расширить исходный класс Component.

**Задача.**

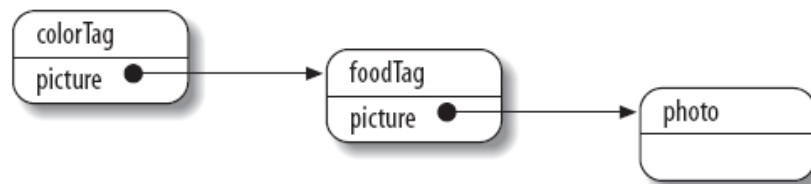
Сопоставить роли объектов и элементы шаблона

IComponent	Любая фотография
Component	Простая фотография
Operation	Изобразить фотографию
Decorator	Помеченная фотография
Client	Автор фотографии и помеченной фотографии

Исходя из этих ролей, пользователь может выполнить одну из следующих команд:

```
Photo photo = new Photo( );
Tag foodTag = new Tag (photo, "Food",1);
Tag colorTag = new Tag (foodTag, "Yellow",2);
```

Как следует из отношения «является разновидностью» photo, foodTag и colorTag – это объекты типа IComponent. Каждая пометка (тег), т.е. декоратор, содержит Component, который может быть фотографией или уже помеченной фотографией. Результат показан на рисунке ниже. В итоге мы имеем три объекта фотографий.



**Замечания к шаблону**

**Различные Компоненты**

Декорированы могут быть различные Компоненты, которые удовлетворяют общему интерфейсу. Поэтому важно, что в шаблоне предусмотрен интерфейс, даже если в нем нет методов. Если мы уверены, что у нас не будет больше одной разновидности Компонентов, то можно отказаться от реализации интерфейса и наследовать Декоратор напрямую от Компонента.

**Различные декораторы**

Мы можем создавать различные разновидности Тегов в качестве декораторов. Например, может быть Декоратор типа Рамка или даже Декоратор, который делает фотографию невидимой. Неважно, какой именно у нас Декоратор. Каждый содержит Компонент, который сам может быть Декоратором.

**Различные операции**

Клиент может вызывать как операции исходного Компонента, так и добавленные Декоратором операции.

## Реализация

Ключевая особенность паттерна в том, что он не полагается на наследование для расширения поведения. Если класс Тегов обязать наследовать от класса Фотография для того, чтобы добавить один или два метода, то класс Тег соберет в себе всё, что касается фотографий и станет чрезмерно перегруженным. Вместо этого, можно сделать так, чтобы класс Тег реализовывал интерфейс Фотографии и добавлял необходимое поведение, не перегружая себя. Следовательно, в классе Тег можно:

- реализовать любые методы интерфейса, меняя при этом исходное поведение компонента,
- добавить новое состояние и поведение,
- добраться до любых доступных методов компонента через включенный объект.

Рассмотрим теоретический пример Декоратора. Такие примеры полезны, т.к. они максимально соответствуют теоретической диаграмме UML для данного шаблона. Примеры из реальной жизни сложнее изобразить на неперегруженной деталями диаграмме.

```
1    using System;
2
3    class DecoratorPattern
4    {
5
6        // Decorator Pattern
7        // Shows two decorators and the output of various
8        // combinations of the decorators on the basic component
9
10       interface IComponent
11       {
12           string Operation();
13       }
14
15       class Component : IComponent
16       {
17           public string Operation()
18           {
19               return "I am walking ";
20           }
21       }
22
23       class DecoratorA : IComponent
24       {
25           IComponent component;
26
27           public DecoratorA(IComponent c)
28           {
29               component = c;
30           }
31
32           public string Operation()
33           {
34               string s = component.Operation();
```

```

35         s += "and listening to Classic FM ";
36         return s;
37     }
38 }
39
40 class DecoratorB : IComponent
41 {
42     IComponent component;
43     public string addedState = "past the Coffee Shop ";
44
45     public DecoratorB(IComponent c)
46     {
47         component = c;
48     }
49
50     public string Operation()
51     {
52         string s = component.Operation();
53         s += "to school ";
54         return s;
55     }
56
57     public string AddedBehavior()
58     {
59         return "and I bought a cappuccino ";
60     }
61 }
62
63 class Client
64 {
65
66     static void Display(string s, IComponent c)
67     {
68         Console.WriteLine(s + c.Operation());
69     }
70
71     static void Main()
72     {
73         Console.WriteLine("Decorator Pattern\n");
74
75         IComponent component = new Component();
76         Display("1. Basic component: ", component);
77         Display("2. A-decorated : ", new DecoratorA(component));
78         Display("3. B-decorated : ", new DecoratorB(component));
79         Display("4. B-A-decorated : ", new DecoratorB(
80             new DecoratorA(component)));
81         // Explicit DecoratorB
82         DecoratorB b = new DecoratorB(new Component());
83         Display("5. A-B-decorated : ", new DecoratorA(b));
84         // Invoking its added state and added behavior
85         Console.WriteLine("\t\t" + b.addedState + b.AddedBehavior());
86     }

```

```

87     }
88     }
89     /* Output
90     Decorator Pattern
91
92     1. Basic component: I am walking
93     2. A-decorated : I am walking and listening to Classic FM
94     3. B-decorated : I am walking to school
95     4. B-A-decorated : I am walking and listening to Classic FM to school
96     5. A-B-decorated : I am walking to school and listening to Classic FM
97     past the Coffee Shop and I bought a cappuccino
98     */

```

Текст программы:  
using System;

```

class DecoratorPattern
{
    // Decorator Pattern
    // Shows two decorators and the output of various
    // combinations of the decorators on the basic component

    interface IComponent
    {
        string Operation();
    }

    class Component : IComponent
    {
        public string Operation()
        {
            return "I am walking ";
        }
    }

    class DecoratorA : IComponent
    {
        IComponent component;

        public DecoratorA(IComponent c)
        {
            component = c;
        }

        public string Operation()
        {
            string s = component.Operation();
            s += "and listening to Classic FM ";
            return s;
        }
    }
}

```

```

class DecoratorB : IComponent
{
    IComponent component;
    public string addedState = "past the Coffee Shop ";

    public DecoratorB(IComponent c)
    {
        component = c;
    }

    public string Operation()
    {
        string s = component.Operation();
        s += "to school ";
        return s;
    }

    public string AddedBehavior()
    {
        return "and I bought a cappuccino ";
    }
}

class Client
{
    static void Display(string s, IComponent c)
    {
        Console.WriteLine(s + c.Operation());
    }

    static void Main()
    {
        Console.WriteLine("Decorator Pattern\n");

        IComponent component = new Component();
        Display("1. Basic component: ", component);
        Display("2. A-decorated : ", new DecoratorA(component));
        Display("3. B-decorated : ", new DecoratorB(component));
        Display("4. B-A-decorated : ", new DecoratorB(
            new DecoratorA(component)));
        // Explicit DecoratorB
        DecoratorB b = new DecoratorB(new Component());
        Display("5. A-B-decorated : ", new DecoratorA(b));
        // Invoking its added state and added behavior
        Console.WriteLine("\t\t" + b.addedState + b.AddedBehavior());
    }
}
}
/* Output
Decorator Pattern

```

1. Basic component: I am walking
  2. A-decorated : I am walking and listening to Classic FM
  3. B-decorated : I am walking to school
  4. B-A-decorated : I am walking and listening to Classic FM to school
  5. A-B-decorated : I am walking to school and listening to Classic FM  
past the Coffee Shop and I bought a cappuccino
- \*/

В строках 10-21 создается интерфейс IComponent и класс Component, в котором он реализуется.

В строках 23-38 создается один декоратор, реализующий IComponent.

Декоратор DecoratorB реализует Operation по-своему, добавляя addedState (строка 43) и addedBehavior (57–60). Клиент создает компоненты и декораторы в различных комбинациях.

Комбинации 2,3 (строки 77–78) декорируют исходный компонент по-разному.

В комбинациях 4,5 применяются два декоратора B и A в различном порядке. В комбинации 2-4 объекты декораторов создаются и используются сразу. В комбинации 5 мы сначала создаем объект DecoratorB и сохраняем его в переменной того же типа. Поэтому мы можем обратиться к добавленному методу.

```
DecoratorB b = new DecoratorB(new Component());
Display("5. A-B-decorated : ", new DecoratorA(b));
// Invoking its added state and added behavior
Console.WriteLine("\t\t"+b.addedState + b.AddedBehavior());
5. A-B-decorated : I am walking to school and listening to Classic FM
past the Coffee Shop and I bought a cappuccino
```

Здесь три объекта: явно заданный DecoratorB, неявно заданный DecoratorA и объект Component. Комбинируя их и вызывая метод каждого из них, мы получаем результат комбинации 5.

## Пример с фотографией

```
using System.Windows.Forms;
namespace Given
{
    // The original Photo class
    public class Photo : Form
    {
        Image image;
        public Photo ( )
        {
            image = new Bitmap("jug.jpg");
            this.Text = "Lemonade";
            this.Paint += new PaintEventHandler(Drawer);
        }
        public virtual void Drawer(Object source, PaintEventArgs e)
        {
            e.Graphics.DrawImage(image, 30, 20);
        }
    }
}
```

Наследуем класс фотографии напрямую от Form, чтобы мы могли ее изобразить. В конструкторе установим обработчик перерисовки формы. Он будет срабатывать в тот момент, когда мы создадим объект. Поэтому клиент может сделать следующее:

```
static void Main ( ) {
// Application.Run играет роль простого клиента
Application.Run(new Photo ( ));
}
```

Теперь без внесения каких-либо изменений в класс Photo, начнем создавать декораторы. Первый рисует рамку вокруг фотографии.

```
// Просто добавляем рамку
```

```

class BorderedPhoto : Photo
{
    Photo photo;
    Color color;
    public BorderedPhoto (Photo p, Color c)
    {
        photo = p;
        color=c;
    }
    public override void Drawer(Object source, PaintEventArgs e)
    {
        photo.Drawer(source, e);
        e.Graphics.DrawRectangle(new Pen(color, 10),25,15,215,225);
    }
}

```

Обратите внимание, что в данном случае декоратор наследует от класса Компонента (Photo), а не реализует интерфейс. Это допустимо. Кроме того, в данном случае мы можем переопределить метод Drawer, потому что он объявлен как виртуальный в базовом классе. Если же мы не можем или не хотим изменить нужным нам образом базовый класс, то необходимо вводить интерфейсы.

Декоратор Тег строится похожим образом:

```

// Фото с двумя тегами и рамкой
foodTag = new Tag (photo, "Food", 1);
colorTag = new Tag (foodTag, "Yellow", 2);
composition = new BorderedPhoto(colorTag, Color.Blue);
Application.Run(composition);

```

Теги для разных объектов создаются с параметрами, соответствующими декорируемому объекту.

Наконец, добавим состояние и поведение к Тегам.

```

using System;
using System.Drawing;
using System.Drawing.Drawing2D;
using System.Windows.Forms;
using System.Collections.Generic;
using Given;

```

```

namespace Given
{
    // The original Photo class
    public class Photo : Form
    {
        Image image;
        public Photo ()
        {
            image = new Bitmap("jug.jpg");
            this.Text = "Lemonade";
            this.Paint += new PaintEventHandler(Drawer);
        }
        public virtual void Drawer(Object source, PaintEventArgs e)
        {
            e.Graphics.DrawImage(image,30,20);
        }
    }
}

```

```

class DecoratorPatternExample
{
    // Декоратор - рамка
    class BorderedPhoto : Photo
    {
        Photo photo;
        Color color;

        public BorderedPhoto (Photo p, Color c)
        {
            photo = p;
            color=c;
        }
    }
}

```

```

        public override void Drawer(Object source, PaintEventArgs e)
        {
            photo.Drawer(source, e);
            e.Graphics.DrawRectangle(new Pen(color, 10), 25, 15, 215, 225);
        }
    }

    // TaggedPhoto - декоратор, хранящий все добавленные к нему теги.

    class TaggedPhoto : Photo
    {
        Photo photo;
        string tag;
        int number;
        static int count;
        List <string> tags = new List <string> ();

        public TaggedPhoto(Photo p, string t)
        {
            photo = p;
            tag = t;
            tags.Add(t);
            number = ++count;
        }

        public override void Drawer(Object source, PaintEventArgs e)
        {
            photo.Drawer(source, e);
            e.Graphics.DrawString(tag,
                new Font("Arial", 16),
                new SolidBrush(Color.Black),
                new PointF(80, 100+number*20));
        }

        public string ListTaggedPhotos()
        {
            string s = "Tags are: ";
            foreach (string t in tags) s +=t+" ";
            return s;
        }
    }

    static void Main ()
    {
        // Application.Run выступает как клиент
        Photo photo;
        TaggedPhoto foodTaggedPhoto, colorTaggedPhoto, tag;
        BorderedPhoto composition;

        // Составим фотографию из двух TaggedPhotos и BorderedPhoto голубого цвета
        photo = new Photo();
        Application.Run(photo);
        foodTaggedPhoto = new TaggedPhoto (photo, "Food");
        colorTaggedPhoto = new TaggedPhoto (foodTaggedPhoto, "Yellow");
        composition = new BorderedPhoto(colorTaggedPhoto, Color.Blue);
        Application.Run(composition);
        Console.WriteLine(colorTaggedPhoto.ListTaggedPhotos());

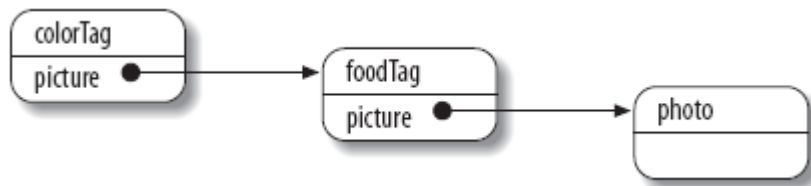
        // Составим фотографию с одним TaggedPhoto и BorderedPhoto желтого цвета
        photo = new Photo();
        tag = new TaggedPhoto (photo, "Jug");
        composition = new BorderedPhoto(tag, Color.Yellow);
        Application.Run(composition);
        Console.WriteLine(tag.ListTaggedPhotos());
    }
}

/* На выходе
TaggedPhotos are: Food Yellow
TaggedPhotos are: Food Yellow Jug
*/

```

Важно отметить, что Декоратор создается на основе некоторых объектов, у которых уже есть свои методы. Некоторые из них могут быть унаследованы, но только на один уровень. Например, когда в программе выше мы могли изменить внутри объекта decorator

такие свойства класса Form как Height и Width. Для самого декоратора первого уровня и для компонента это будет работать. Но если мы введем декоратор второго уровня, то эти изменения мы уже не увидим. Почему это происходит, видно из рисунка:



Первый декоратор содержит ссылку на объект типа Form, а второй уже нет.

### Примеры применения шаблона:

1. В самой .NET декораторы широко применяются. Например,

```
System.IO.Stream
System.IO.BufferedStream
System.IO.FileStream
System.IO.MemoryStream
System.Net.Sockets.NetworkStream
System.Security.Cryptography.CryptoStream
```

В этой иерархии подклассы декорируют Stream, т.к. они не только наследуют от класса Stream, но и содержат объект этого типа.

2. В программах для мобильных устройств часто используется этот прием, когда нужно исключить баннеры и добавить полосы прокрутки в случае работы с маленьким экраном.

Итак, этот шаблон следует использовать когда:

- есть класс компонента, от которого невозможно или не нужно создавать подкласс.

И при этом нужно:

- Динамически добавить к объекту дополнительное состояние или поведение.
- Внести изменения в некоторые объекты, не затрагивая другие объекты.
- Избежать создания подклассов, т.к. их получится слишком много.

### Задачи

1. Предположим, класс Photo был написан с неvirtуальным методом Drawer и поэтому не может быть переопределен. Переделать пример, приведенный выше так, чтобы работоспособность сохранилась (подсказка: задействовать интерфейсы, как в теоретическом примере).
2. Добавить в систему с декораторами фотографий второй тип компонентов – Человек. Пусть у него будут методы для рисования овалов и линий, из которых можно получить примерное изображение человека. Затем, используя те же два декоратора Tag и Border – декорировать объекты класса Человек.
3. Добавить другие обработчики событий в конструкторы декораторов. Например, добавить обработчик события OnClick, по которому тег будет появляться на изображении.
4. Декорировать системный класс Console так, чтобы методы Write и WriteLine перехватывались и вывод переформатировался для строк заданной длины. Например, если длина строки равна 10, то после каждого десятого символа должен выводиться символ перевода строки.
5. Декорировать системный класс Stream так, чтобы при чтении файла показывался индикатор процесса загрузки, т.е. сколько байт уже загружено.

6. Декорировать системный класс `Stream` так, чтобы запрашивался пароль перед тем, как начиналось чтение данных.
7. Написать программу, которая декорирует текстовую строку простым шифрованием. Добавить в цепочку второй декоратор, который расшифровывает сообщение.

# Структурные паттерны. Паттерн Прокси (Proxy). Паттерн Мост (Bridge)

## Шаблон Прокси (Proxy)

### Назначение

Прокси описывает объект, который управляет созданием других объектов и доступом к ним. Прокси – это часто небольшой общедоступный объект, который дублирует более сложный (закрытый) объект, который активизируется только при выполнении некоторых условий.

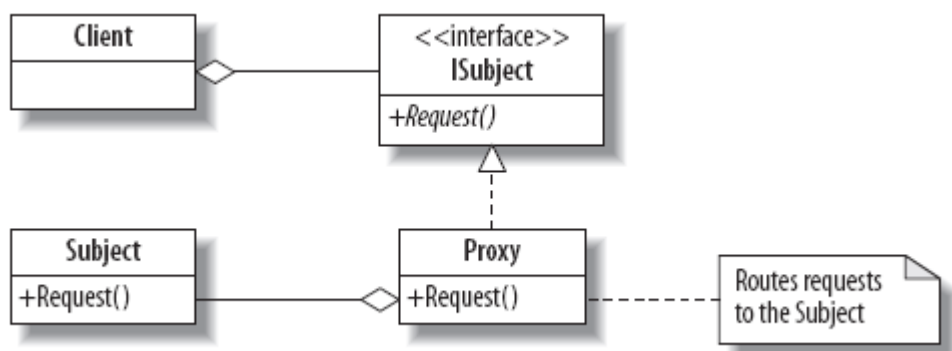
### Пример

В настоящее время очень популярны социальные сети, например, «В контакте». Особенность таких систем в том, что пользователи заходят на свою страницу только для того, чтобы увидеть, что другие пользователи «он-лайн», а не для того, чтобы добавлять какой-то контент. Поэтому имеет смысл сначала пускать пользователей на некую простую страницу и не выделять им весомых ресурсов пока они им реально не понадобятся. Другая особенность, что пользователям нужно пройти процедуру регистрации и авторизации, прежде чем они попадут в систему. После этого все действия сначала происходят в браузере, а потом отправляются на ближайший сервер. В этой схеме прокси – это механизм регистрации и авторизации, а объекты, к которым осуществляется доступ – это веб-страницы системы.

### Другие примеры

Секретарь у президента компании во время обсуждения времени визита клиента. Секретарь имеет тот же интерфейс, что и президент (может обсуждать время визита по телефону) и может переадресовывать или неадресовывать некоторые вопросы своему начальнику. Но всякий, кто ищет встречи с президентом, должен сначала пообщаться с секретарем и никогда с его шефом.

### Архитектура



### Участники

ISubject

Общий интерфейс субъектов, к которым осуществляется доступ, и прокси-объектов. Это дает возможность использовать их взаимозаменяемым образом.

**Subject**

Субъект (предмет), к которому осуществляется доступ

**Proxy**

Класс, который создает, управляет и контролирует доступ к субъекту

**Request**

Операция субъекта, которая доступна клиенту через прокси

Центральный класс Прокси реализует интерфейс ISubject. Клиент должен использовать ISubject, если не хочет завязываться на конкретные разновидности прокси-объектов. Однако иногда можно пренебречь этим интерфейсом. Каждый прокси-объект управляет по ссылке субъектом, в котором и происходит что-то полезное. Клиент общается с прокси-объектом. Ценность прокси может быть повышена, если класс субъекта станет закрытым классом и клиент не сможет добраться до субъекта никак кроме как через прокси.

## Задача

ISubject	Действия, которые можно сделать на веб-странице
Subject	Личные веб-страницы пользователя
Subject.Request	Действие, изменяющее веб-страницу
Proxy	Клиентская сторона для веб-страниц
Proxy.Request	Выполняет некоторое действие, прежде чем передать Запрос дальше
Client	Пользователь системы

Существуют некоторые разновидности прокси:

**Виртуальные прокси**

Управляют созданием субъекта (бывает полезны, если процесс создания объекта медленный)

**Прокси авторизации**

Проверяют права на запрос к субъекту

**Удаленные прокси**

Кодируют запрос и отправляют его по сети

**Умные прокси**

Что-то добавляют или как-то меняют запрос прежде, чем его переслать

В примере с социальными сетями можно выделить такие прокси:

- Виртуальные прокси, которые откладывают создание дорогих с точки зрения ресурсов страниц
- Прокси авторизации, которые пускают или не пускают пользователей
- Отправка запросов по сети (удаленные прокси)
- Умные прокси, которые что-то делают со страницами друзей.

## Реализация

При реализации этого шаблона можно широко применять спецификаторы доступа C#: private, public, protected, а также

internal – доступ разрешен внутри сборки

protected internal – доступ разрешен внутри класса, порожденного класса или внутри сборки

Рассмотрим теоретический пример, в котором выясним

- как можно отделить Клиента от Субъекта с использованием спецификаторов доступа,
- как через виртуальный прокси и прокси авторизации проходит запрос к Субъекту.

Виртуальный прокси можно создать так:

```
public class Proxy : ISubject
{
    Subject subject;
    public string Request( )
    {
        // Виртуальный прокси создает объект только, когда первый раз вызывается его объект
        if (subject == null)
            subject = new Subject( );
        return subject.Request();
    }
}
```

Клиент может создать Прокси в любой момент обычным способом. Поскольку используется конструктор по умолчанию, ссылка на новый субъект в этот момент не создается. Это происходит только когда вызывается метод Request, в котором проверяется, создан субъект ранее или нет.

## Примечание

На физическом уровне программа на C# состоит из нескольких единиц трансляции, каждая из которых содержится в отдельном файле. Когда программа компилируется, все единицы трансляции обрабатываются совместно. Эти единицы могут зависеть друг от друга, возможно, циклически. Сборка – это физический контейнер для типов и связанных с ними ресурсов.

На логическом уровне программы на C# организуются с использованием пространств имен. Пространства имен дают возможность использовать элементы программы другим программам в этой же или других сборках. Модификаторы доступа – это способ определить кому, что и до какой степени доступно в данном пространстве имен.

В C# конкретные модификаторы доступа к элементам могут изменить уровень доступа, данный им по умолчанию.

Если говорить о классах, то правила такие:

- Не вложенный класс по умолчанию виден другим классам в этой же сборке.
- Вложенный класс по умолчанию закрытый и не может иметь уровень доступа выше, чем у внешнего класса.
- Член класса (поле, метод, свойство) по умолчанию закрыты.
- Класс-наследник может иметь доступ к элементам родительского класса, если они помечены как защищенные.

Если не использовать сборки, то модификатор public не обязательно явно указывать.

Однако, если элементы, объявленные вне классов, могут понадобиться в других классах (или сборках), то их лучше сделать явно public.

Есть несколько ограничений насчет того, где могут находиться модификаторы доступа.

Члены интерфейсов всегда открыты и модификаторы не допускаются. Пространства имен не могут иметь закрытых или защищенных членов. Модификаторы могут быть добавлены к типам или членам.

Полный код примера:

```
1     using System;
2
3
4     // Shows virtual and protection proxies
5
6     class SubjectAccessor
7     {
8         public interface ISubject
```

```

9      {
10     string Request();
11     }
12
13     private class Subject
14     {
15     public string Request()
16     {
17         return "Subject Request " + "Choose left door\n";
18     }
19     }
20
21     public class Proxy : ISubject
22     {
23     Subject subject;
24
25     public string Request()
26     {
27         // A virtual proxy creates the object only on its first method call
28         if (subject == null)
29         {
30             Console.WriteLine("Subject inactive");
31             subject = new Subject();
32         }
33         Console.WriteLine("Subject active");
34         return "Proxy: Call to " + subject.Request();
35     }
36     }
37
38     public class ProtectionProxy : ISubject
39     {
40     // An authentication proxy first asks for a password
41     Subject subject;
42     string password = "Abracadabra";
43
44     public string Authenticate(string supplied)
45     {
46         if (supplied != password)
47             return "Protection Proxy: No access";
48         else
49             subject = new Subject();
50         return "Protection Proxy: Authenticated";
51     }
52
53     public string Request()
54     {
55         if (subject == null)
56             return "Protection Proxy: Authenticate first";
57         else return "Protection Proxy: Call to " +
58             subject.Request();
59     }
60     }
61 }
62
63 class Client : SubjectAccessor
64 {
65     static void Main()
66     {
67         Console.WriteLine("Proxy Pattern\n");
68
69         ISubject subject = new Proxy();
70         Console.WriteLine(subject.Request());
71         Console.WriteLine(subject.Request());
72
73         ProtectionProxy subject = new ProtectionProxy();
74         Console.WriteLine(subject.Request());
75         Console.WriteLine((subject as ProtectionProxy).Authenticate("Secret"));
76         Console.WriteLine((subject as ProtectionProxy).Authenticate("Abracadabra"));
77         Console.WriteLine(subject.Request());
78     }
79 }
80
81 /* Output
82
83 Proxy Pattern
84
85 Subject inactive
86 Subject active
87 Proxy: Call to Subject Request Choose left door

```

```

88
89 Subject active
90 Proxy: Call to Subject Request Choose left door
91
92 Protection Proxy: Authenticate first
93 Protection Proxy: No access
94 Protection Proxy: Authenticated
95 Protection Proxy: Call to Subject Request Choose left door
96 */

```

Обратите внимание, что клиент работает с интерфейсом ISubject, а не Subject. Это приводит к тому, что если понадобятся дополнительные возможности Прокси, такие как Authenticate, то потребуется приводить ISubject к нужному типу, что и происходит в строках 75 и 76. В примере вводится еще и Защищенный прокси (38-61). Прежде, чем отправлять запросы субъекту, нужно авторизоваться (Authenticate).

Программа построена следующим образом. Вводится класс SubjectAccessor, группирующий типы Прокси и Субъект. Интерфейс и прокси объявлены как public (строки 8, 21 и 38), поэтому клиент имеет доступ к ним и к их открытым членам. Виртуальный и защищающий прокси нужны, чтобы предоставить Клиенту доступ к Субъекту. Поэтому Субъект объявлен как закрытый член. (13). Метод Request (Запрос) открытый, но видим только классами, которые могут видеть сам класс с этим методом, т.е. теми классам, которые находятся внутри SubjectAccessor.

Мы можем проверить, что уровни доступа расставлены правильно:

1) более логичным было бы разделить типы Прокси и Субъект по разным пространствам имен, но классы в пространствах имен не могут быть закрытыми. Проверим, что мы все сделали правильно. Для этого изменим строку 6, чтобы она содержала namespace SubjectAccessor и добавим using SubjectAccessor после строки 1. Также объявление класса SubjectAccessor можно убрать совсем и все типы станут принадлежать пространству имен по умолчанию.

2) В методе Main добавим на строке 68

```
ISubject test = new Subject( );
```

Этот код не будет компилироваться, потому что программа устроена так, что можно создавать только прокси-объекты, а не субъектов. Однако если убрать модификатор private из строки 13, то экземпляр Subject будет создан. Это происходит потому, что клиент наследует от SubjectAccessor, получая доступ к его внутренним членам.

Итак, этот пример показывает, что можно создавать прокси и субъекты, связанные средствами языка ровно так, как нужно. В данном случае они не должны быть независимыми друг от друга, поэтому мы их группируем в класс. Прокси может как угодно агрегировать субъекты.

## Пример («В контакте» ;))

Сделаем простой аналог механизма социальной сети. «В контакте» хранит страницы пользователей, к которым они получают доступ после ввода логина и пароля, поэтому нам нужна и авторизация и «ленивое» создание страниц. Как было показано выше, простая регистрация не должна создавать «дорогие» объекты. Они их получают, если вначале сами добавят какой-то контент. Поэтому для начала нужно дать возможность людям оставлять записи на страницах других пользователей (эти записи тоже будут считаться контентом). Основной класс может выглядеть так:

```

// The Subject
private class VKontakte
{
    static SortedList <string,RealVKontakte> community =
    new SortedList <string,RealVKontakte> (100);
    string pages;

```

```

string name;
string gap = "\n\t\t\t\t\t";
static public bool IsUnique (string name)
{
    return community.ContainsKey(name);
}
internal VKontakte (string n)
{
    name = n;
    community [n] = this;
}
internal void Add(string s)
{
    pages += gap+s;
    Console.WriteLine(gap+"===== "+name+"'s VKontakte =====");
    Console.WriteLine(pages);
    Console.WriteLine(gap+"=====");
}
internal void Add(string friend, string message)
{
    community[friend].Add(message);
}
}

```

Класс содержит статический список всех текущих пользователей. Мы используем коллекцию System.Collections.Generic с ключем-строкой, которым будет имя пользователя.

Введем два метода Add, один для добавления пользователя и один для добавления сообщения другому пользователю. Также есть метод проверки уникальности имени пользователя.

Рассмотрим клиента системы.

```

// The Client
class ProxyPattern : VKontakteSystem
{
    static void Main ( )
    {
        MyVKontakte me = new MyVKontakte ( );
        me.Add("Hello world");
        me.Add("Today I worked 18 hours");
        MyVKontakte tom = new MyVKontakte ( );
        tom.Add("Igor", "Hello");
        tom.Add("Off to see the Lion King tonight");
    }
}

```

В этом коде нет и намека на авторизацию пользователя. Она реализована в прокси MyVKontakte.

**Вывод для кода выше:**

```

Let's register you for VKontakte
All VKontaktenames must be unique
Type in a user name: Igor
Type in a password: haha
Thanks for registering with VKontakte
Welcome Igor. Please type in your password: haha
Logged into VKontakte
===== Igor's VKontakte=====
Hello world
=====
===== Igor's VKontakte=====
Hello world
Today I worked 18 hours
=====

```

```

Let's register you for VKontakte
All VKontakte names must be unique
Type in a user name: Tom
Type in a password: yey
Thanks for registering with VKontakte
Welcome Tom. Please type in your password: yey
Logged into VKontakte
===== Igor's VKontakte =====
Hello world

```

```

Today I worked 18 hours
Tom said: Hello
=====
===== Tom's VKontakte =====
Off to see the Lion King tonight
=====

```

Вывод с отступами – это функционирование ядра VKontakte, а без – прокси.

Класс VKontakte не имеет конструктора, поэтому ничего не произойдет, если клиент создаст экземпляр этого класса. Ключевой метод прокси MyVKontakte - Add. В нем происходят все проверки.

```

public void Add(string message)
{
    Check( );
    if (loggedIn) myVKontakte.Add(message);
}
void Check( )
{
    if (!loggedIn)
    {
        if (password==null)
            Register( );
        if (myVKontakte == null)
            Authenticate( );
    }
}

```

Полный код:

```

using System;
using System.Collections.Generic;

class VKontakteSystem
{
    // The Subject
    private class VKontakte
    {
        static SortedList <string,VKontakte> community =
        new SortedList <string,VKontakte> (100);
        string pages;
        string name;
        string gap = "\n\t\t\t\t\t";

        static public bool IsUnique (string name)
        {
            return community.ContainsKey(name);
        }

        internal VKontakte (string n)
        {
            name = n;
            community [n] = this;
        }

        internal void Add(string s)
        {
            pages += gap+s;
            Console.Write(gap+"===== "+name+"'s VKontakte =====");
            Console.Write(pages);
            Console.WriteLine(gap+"=====");
        }

        internal void Add(string friend, string message)
        {
            community[friend].Add(message);
        }
    }

    // The Proxy
    public class MyVKontakte
    {
        // Комбинация виртуального и защищающего прокси

```

```

VKontakte myVKontakte;
string password;
string name;
bool loggedIn = false;

void Register ()
{
    Console.WriteLine("Let's register you for VKontakte");
    do
    {
        Console.WriteLine("All VKontakte names must be unique");
        Console.Write("Type in a user name: ");
        name = Console.ReadLine();
    } while (VKontakte.IsUnique(name));
    Console.Write("Type in a password: ");
    password = Console.ReadLine();
    Console.WriteLine("Thanks for registering with VKontakte");
}

bool Authenticate ()
{
    Console.Write("Welcome "+name+". Please type in your password: ");
    string supplied = Console.ReadLine();
    if (supplied==password)
    {
        loggedIn = true;
        Console.WriteLine("Logged into VKontakte");
        if (myVKontakte == null)
            myVKontakte = new VKontakte(name);
        return true;
    }
    Console.WriteLine("Incorrect password");
    return false;
}

public void Add(string message)
{
    Check();
    if (loggedIn) myVKontakte.Add(message);
}

public void Add(string friend, string message)
{
    Check();
    if (loggedIn)
        myVKontakte.Add(friend, name + " said: "+message);
}

void Check()
{
    if (!loggedIn)
        if (password==null)
            Register();
    if (myVKontakte == null)
        Authenticate();
}
}

// The Client
class ProxyPattern : VKontakteSystem
{
    static void Main ()
    {
        MyVKontakte me = new MyVKontakte();
        me.Add("Hello world");
        me.Add("Today I worked 18 hours");

        MyVKontakte tom = new MyVKontakte();
        tom.Add("Igor", "Hello");
        tom.Add("Off to see the Lion King tonight");
    }
}

```

В данном примере интерфейс ISubject не понадобился. VKontakte и MyVKontakte находятся внутри VKontakteSystem и связаны через отношение агрегации.

## Использование

Прокси – это способ организовать доступ клиентов к объектам классов, которые владеют существенными ресурсами или медленными операциями. Они часто используются в системах, работающих с изображениями, когда прокси предоставляет место для вывода изображения на экране и затем запускает отрисовку картинку. Прокси, как и декоратор, перенаправляет запросы другому объекту. Отличие в том, что отношения для прокси устанавливаются на этапе проектирования и известны заранее, а декораторы могут быть добавлены динамически.

Итак, этот шаблон следует использовать когда:

- есть объекты, создание которых обходится дорого,
- нужен контроль доступа,
- нужен доступ к удаленным сайтам,
- нужно выполнить некоторые дополнительные действия в тот момент, когда осуществляется запуск основной функциональности.

И при этом нужно:

- Создавать объекты, только когда понадобились их операции.
- Выполнять вспомогательные действия при доступе к каким-то объектам.
- Иметь локальный объект, который должен ссылаться на удаленный объект.
- Проверять права на доступ к объекту.

## Задачи

1. Преобразовать теоретический пример выше так, чтобы Client не наследовал от SubjectAccessor, а создавал ее экземпляр.
2. Добавить к примеру с VKontakteSystem графический интерфейс пользователя. Можно либо заменить соответствующие методы в VKontakte, либо добавить декораторы, оставив существующий код как есть.
3. Тот пример, который приведен выше (VKontakteSystem), допускает добавление в качестве друга любого пользователя. Улучшить пример так, чтобы сообщение можно было отправлять только пользователям, которые дали на это согласие. Продумать, куда именно нужно внести изменения в VKontakte (субъект) или MyVKontakte (прокси).
4. Другой тип социальных сетей – это сеть, хранящая фоторафии (например, Flickr). Пользователи загружают фотографии, а другие пользователи могут их скачивать и комментировать. Сделать набросок того, как можно использовать паттерн Прокси для такой системы.
5. Пусть требуется следить за использованием некоторой библиотеки. Например, есть некая библиотека сторонних разработчиков, для работы которой нужен класс Stream. Мы же должны подсунуть ей свой класс MyStream, который будет перехватывать обращение к методам библиотеки и вести лог этих обращений. При этом сама библиотека подмены не должна заметить и продолжать функционировать с нашим классом MyStream как она бы функционировала с классом Stream.

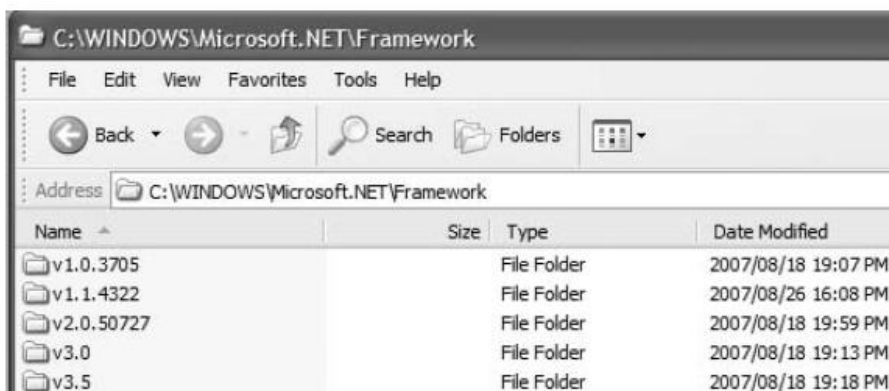
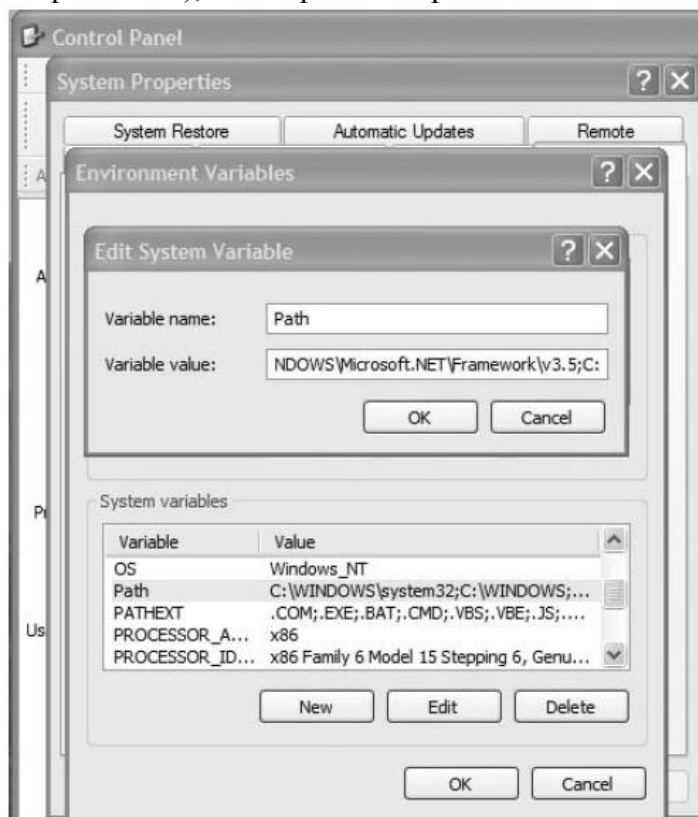
## Шаблон Мост (Bridge)

## Назначение

Шаблон Мост описывает ситуацию, когда нужно отделить абстракцию от реализации, позволив им изменяться независимо. Мост полезен, когда появляется новая версия некоторой программы, но старые версии всё так же должны продолжать использоваться. В случае, если это клиент-серверная программа, то клиентский код не должен меняться, поскольку рассчитан на определенную серверную программу, но клиент должен указать какую версию серверной программы он хочет использовать.

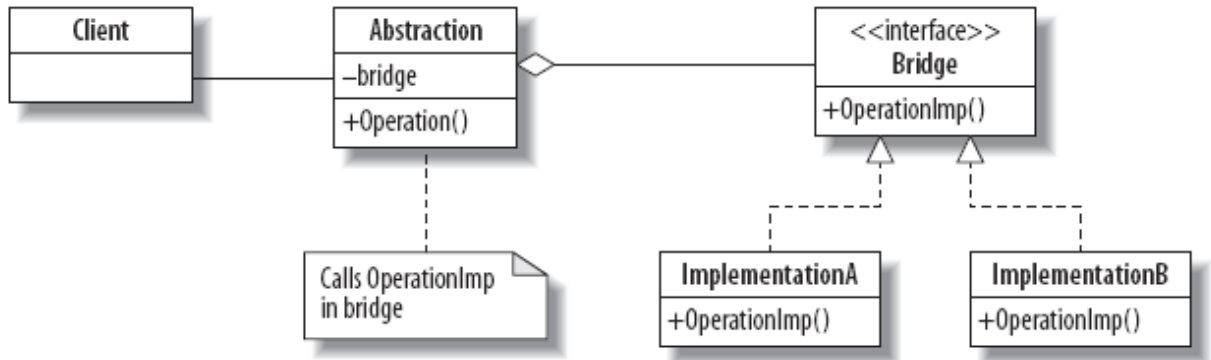
## Пример

Вот, например, появляется новая версия .NET Framework, используемая для компиляции программ на C#, например, версия 3.0. При этом у вас могут быть установлены и все предыдущие версии: например, 1.1, 2.0 и т.д. Пользователь может указать, какую версию он хочет использовать, поменяв путь к .NET Framework в настройках Windows. Установка пути и есть мост между приложениями, которые используют .NET Framework (назовем их абстракциями), и конкретной версией .NET Framework (это будут реализации).



## Другие примеры

Государство устанавливает нормы и стандарты, чтобы гарантировать Качество Обслуживания (Абстракция) потребителей. Различные фирмы по-разному создают товары и услуги (Реализации), но они должны удовлетворять установленному интерфейсу (Мост).



## Участники этого шаблона:

### Абстракция (Abstraction)

Интерфейс, который видит клиент

### Операция (Operation)

Метод, который вызывается клиентом

### Мост (Bridge)

Интерфейс, определяющий те части Абстракции, которые могут меняться.

### РеализацияА (ImplementationA) и РеализацияБ (ImplementationB)

Реализации интерфейса Мост

### OperationImp

Метод интерфейса Мост, который вызывается Операцией в Абстракции. Как мы уже говорили, у Абстракции может быть несколько Реализаций. Вводить и реализовывать интерфейс Мост нужно только в том случае, когда и старые и новые реализации должны работать совместно и взаимно заменять друг друга.

## Реализация

И снова начнем с теоретического примера.

В клиентском коде каждый экземпляр Абстракции получает свой экземпляр Реализации (строки 46-47), а затем вызывается соответствующая Операция. Каждый вызов перенаправляется в различные методы OperationImp. (29 и 37). Результат вывода показан в строках 53-54. Заметьте, что Реализации реализуют интерфейс Мост, а Абстракция их агрегирует.

```
1      using System;
2
3      class BridgePattern
4      {
5
6
7          // Shows an abstraction and two implementations proceeding independently
8
9          class Abstraction
10         {
11             Bridge bridge;
12             public Abstraction(Bridge implementation)
13             {
14                 bridge = implementation;
```

```

15     }
16     public string Operation()
17     {
18         return "Abstraction" + " <<< BRIDGE >>>> " + bridge.OperationImp();
19     }
20 }
21
22 interface Bridge
23 {
24     string OperationImp();
25 }
26
27 class ImplementationA : Bridge
28 {
29     public string OperationImp()
30     {
31         return "ImplementationA";
32     }
33 }
34
35 class ImplementationB : Bridge
36 {
37     public string OperationImp()
38     {
39         return "ImplementationB";
40     }
41 }
42
43 static void Main()
44 {
45     Console.WriteLine("Bridge Pattern\n");
46     Console.WriteLine(new Abstraction(new ImplementationA()).Operation());
47     Console.WriteLine(new Abstraction(new ImplementationB()).Operation());
48 }
49 }
50 /* Output
51 Bridge Pattern
52
53 Abstraction <<< BRIDGE >>>> ImplementationA
54 Abstraction <<< BRIDGE >>>> ImplementationB
55 */

```

## Пример

Расширим пример VKontakte с использованием шаблона Мост. Заодно посмотрим, как можно объединить несколько шаблонов.

Предположим, разработчики выпустили новую версию VKontakte – VKontakte2. Главная особенность VKontakte2 в том, что она не требует авторизации с помощью пароля, пользователи сразу попадают в систему, если у них на компьютере есть специальный паспорт. Поэтому в VKontakte2 вообще теперь нет прокси слоя, отвечающего за проверку логина и пароля.

## Задача

Abstraction	NET Framework
Operation	C# compiler
Bridge	Переменная окружения windows Path
ImplementationA	Версия 3.5 Framework
ImplementationB	Версия 2.0.50377 Framework
A's OperationImp	Компилятор 3.0
B's OperationImp	Компилятор 2.0

Исходя из новых требований, нам теперь нужно, чтобы MyVKontakte и MyVKontakte2 реализовали общий интерфейс. Этот интерфейс мы будем называть Мостом. Абстракцией в данном случае будет то, что обычно называют Портал. В нем хранятся ссылки на копии систем двух типов (MyVKontakte и MyVKontakte2). Портал будет активизировать нужную версию системы и отправлять к ней запросы клиента.

```

class BridgePattern : VKontakteSystem
{
    static void Main ( )
    {
        MyVKontakte me = new MyVKontakte ( );
        me.Add("Hello world");
        me.Add("Today I worked 18 hours");
        Portal tom = new Portal(new MyVKontakte2("Tom"));
        tom.Add("Igor","Hello");
        tom.Add("Hey, I'm also on VKontakte2 - it was so easy!");
    }
}

```

Пользователи (например, Igor) продолжают использовать первую версию VKontakte как ни в чем ни бывало, а Том теперь может попасть в систему через Портал (Portal) (Абстракция), который получает ссылку на MyVKontakte2 (Реализация). Вывод на консоль будет такой же, как и прежде, но Тому теперь не нужно авторизовываться.

```

Let's register you for VKontakte
All VKontaktenames must be unique
Type in a user name: Igor
Type in a password: yey
Thanks for registering with VKontakte
Welcome Igor. Please type in your password: yey
Logged into VKontakte
===== Igor's VKontakte=====
Hello world
Today I worked 18 hours
Tom : Hello
=====
===== Igor's VKontakte=====
Hello world
Today I worked 18 hours
Tom : Hello
=====
===== Tom-1's VKontakte=====
Hey, I'm on VKontakte2- it was so easy!
=====

```

Предположим теперь, что разработчики решили избежать конфликта имен, добавив уникальный номер каждому пользователю. Если имя очередного пользователя Igor, то ему будет присвоено имя Igor-2. Сейчас вывод для Тома все еще опирается на возможности VKontakte. Это видно по заголовку:

```

===== Tom-1's VKontakte=====

```

Устраним этот недостаток.

Клиент теперь взаимодействует с системой через интерфейс Мост:

```

interface Bridge
{
    void Add(string message);
    void Add(string friend, string message);
}

```

Портал (Portal) очень прост и просто переадресует все запросы.

```

class Portal
{
    Bridge bridge;
    public Portal (Bridge aVKontakte)
    {
        bridge = aVKontakte;
    }
    public void Add(string message)
    {
        bridge.Add(message);
    }
    public void Add(string friend, string message)
    {
        bridge.Add(friend,message);
    }
}

```

Вторая реализация интерфейса Мост тоже является прокси к VKontakte. Но из нее выброшено часть методов, которые теперь не нужны.

```
public class MyVKontakte2 : Bridge
{
    // Комбинация виртуального и защищающего прокси
    VKontakte myVKontakte;
    string name;
    static int users;
    public MyVKontakte2 (string n)
    {
        name = n;
        users++;
        myVKontakte = new VKontakte(name+"-"+users);
    }
    public void Add(string message)
    {
        myVKontakte.Add(message);
    }
    public void Add(string friend, string message)
    {
        myVKontakte.Add(friend, name + " said: "+message);
    }
}
```

Обратите внимание, что интерфейс Мост должен содержать все операции, которые есть в обеих версиях системы.

Теперь предположим, что в VKontakte2 добавилась новая возможность: узнать, кто заходил к вам на страницу.

```
public void Spy (string who, string spyName)
{
    myVKontakte.Add(who, spyName + " spies on you");
}
```

Таким способом Spy просто реализуется через функциональность VKontakte. Если мы поместим Spy в VKontakte2, компилятор примет эти изменения, но мы не сможем вызвать его, поскольку в главной программе Tom ссылается на объект Portal, а Spy не часть Portal. Мы можем решить эту проблему двумя способами:

- Добавить новую операцию к Portal, а не в Мост и таким образом это не окажет влияния на первую версию VKontakte.
- Если мы не можем изменять Portal, мы можем создать расширяющий метод (см. примечание)

```
static class VKontakte2Extensions
{
    public static void Spy (this Portal me, string who, string spyName)
    {
        me.Add(who, spyName + " spies on you");
    }
}
```

и вызвать его как любой другой метод:

```
tom.Spy("Igor-1", "Tom");
```

Расширяющие методы – новая возможность C# 3.0

## Примечание

**Расширяющие** методы позволяют добавлять новые методы к существующему классу без необходимости создавать порожденный класс или перекомпилировать исходный. Поэтому можно добавить методы даже к такому классу, исходников от которого нет (например, System.String). Расширяющие методы определяются так же, как и другие с двумя оговорками:

- Они объявляются как static методы нешаблонного класса.
- Тип, который они расширяют, объявляется как первый параметр с ключевым словом this.

Метод может быть вызван как родной метод объекта.

Полная программа приведена ниже. VKontakte и MyVKontakte не изменены совсем. Из вывода на консоль видно, что Igor переключился на VKontakte2 и поэтому его имя теперь Igor-1. Tom – второй пользователь VKontakte2 и получил имя Tom-2.

```
using System;
using System.Collections.Generic;

// Расширение VKontakte второй реализацией через Portal

// Abstraction
class Portal {
    Bridge bridge;
    public Portal (Bridge aVKontakte) {
        bridge = aVKontakte;
    }
    public void Add(string message)
        {bridge.Add(message);}
    public void Add(string friend, string message)
        {bridge.Add(friend,message);}
}

//Bridge
interface Bridge {
    void Add(string message);
    void Add(string friend, string message);
}

class VKontakteSystem {

    // The Subject
    private class VKontakte {
        static SortedList <string,VKontakte> community =
            new SortedList <string,VKontakte> (100);
        string pages;
        string name;
        string gap = "\n\t\t\t\t";

        static public bool IsUnique (string name) {
            return community.ContainsKey(name);
        }

        internal VKontakte (string n) {
            name = n;
            community [n] = this;
        }

        internal string Add(string s) {
            pages += gap+s;
            return gap+"=====" +name+"'s VKontakte ====="+
                pages+"\n"+
                gap+"=====";
        }

        internal string Add(string friend, string message) {
            return community[friend].Add(message);
        }
    }
}

// The Proxy (МЕНЯТЬ НЕЛЬЗЯ)
public class MyVKontakte {
    // Combination of a virtual and authentication proxy
    VKontakte myVKontakte;
    string password;
    string name;
    bool loggedIn = false;

    void Register () {
        Console.WriteLine("Let's register you for VKontakte");
        do {
            Console.WriteLine("All VKontakte names must be unique");
            Console.Write("Type in a user name: ");
            name = Console.ReadLine();
        } while (VKontakte.IsUnique(name));
        Console.Write("Type in a password: ");
        password = Console.ReadLine();
        Console.WriteLine("Thanks for registering with VKontakte");
    }
}
```

```

    }

    bool Authenticate () {
        Console.WriteLine("Welcome "+name+". Please type in your password: ");
        string supplied = Console.ReadLine();
        if (supplied==password) {
            loggedIn = true;
            Console.WriteLine( "Logged into VKontakte");
            if (myVKontakte == null)
                myVKontakte = new VKontakte(name);
            return true;
        }
        Console.WriteLine("Incorrect password");
        return false;
    }

    public void Add(string message) {
        Check();
        if (loggedIn) myVKontakte.Add(message);
    }

    public void Add(string friend, string message) {
        Check();
        if (loggedIn)
            Console.WriteLine(myVKontakte.Add(friend, name + " said: "+message));
    }

    void Check() {
        if (!loggedIn) {
            if (password==null)
                Register();
            if (myVKontakte == null)
                Authenticate();
        }
    }
}

// Этот прокси делает совсем немного, просто иллюстрирует
// альтернативную реализацию шаблона Мост
public class MyVKontakte2 : Bridge {
    // Combination of a virtual and authentication proxy
    VKontakte myVKontakte2;
    string name;
    static int users;

    public MyVKontakte2 (string n) {
        name = n;
        users++;
        myVKontakte2 = new VKontakte(name+"-"+users);
    }

    public void Add(string message) {
        Console.WriteLine(myVKontakte2.Add(message));
    }

    public void Add(string friend, string message) {
        Console.WriteLine(myVKontakte2.Add(friend, name + " : "+message));
    }

}

} // end class VKontakteSystem

static class VKontakte2Extensions {
    public static void Spy (this Portal me, string who, string spyName) {
        me.Add(who, spyName + " spies on you");
    }
}

// The Client
class BridgePattern : VKontakteSystem {
    static void Main () {
        Portal me = new Portal(new MyVKontakte2("Igor"));
        me.Add("Hello world");
        me.Add("Today I worked 18 hours");

        Portal tom = new Portal(new MyVKontakte2("Tom"));
        tom.Spy("Igor-1", "Tom");
        tom.Add("Igor-1","Hello");
    }
}

```

```

        tom.Add("Hey, I'm on VKontakte2 - it was so easy!");
        //Added Console.ReadKey to stop Console Window from closing.
        Console.ReadKey();
    }
}
/* Output
Let's register you for VKontakte
All VKontakte names must be unique
Type in a user name: Igor
Type in a password: yey
Thanks for registering with VKontakte
Welcome Igor. Please type in your password: yey
Logged into VKontakte

===== Igor's VKontakte =====
Hello world
Today I worked 18 hours
Tom : Tom spies on you

=====

===== Igor's VKontakte =====
Hello world
Today I worked 18 hours
Tom : Tom spies on you

=====

===== Tom-1's VKontakte =====
Hey, I'm on VKontakte2 - it was so easy!

=====
*/

```

## **Использование**

Мост простой, но очень мощный шаблон. Исходя из существующей реализации (версии) чего-либо, мы добавляем вторую через Мост и Абстракцию и получаем значительно более обобщенное решение. Хорошим примером использования Моста являются графические программы, где различные мониторы могут иметь различные возможности и соответственно драйверы. Они будут реализациями в терминах паттерна Мост, а Мост будет интерфейсом к их особенным возможностям. Клиент работает с некоей Абстракцией – виртуальным монитором, а Абстракция может добывать конкретные сведения о свойствах одного или нескольких объектов, реализующих интерфейс Мост (ими могут быть как раз драйверы) и выбрать подходящий.

Используя шаблон Мост, вы можете:

- Выделить операции, которые не всегда реализуются одинаково.

При этом:

- Полностью скрываются детали реализации от клиентов.
- Избегается привязка реализации напрямую к абстракции.
- Изменение реализации возможно даже без перекомпиляции абстракции.
- Комбинирование различных частей системы даже во время выполнения.

## **Задача**

1. Хотя некоторые новые операции были добавлены в VKontakte2, но некоторые базовые операции, такие как вывод шапки, остались встроенными в закрытый класс VKontakte. Поэтому внесение изменений по-прежнему затруднительно. Предложите с использованием UML диаграмм как все же построить расширяемую систему на основе первой версии.

## Сравнение шаблонов

Все три шаблона похожи. Грубо говоря, все они помогают расширить классы нестандартным способом и все они предоставляют альтернативу наследованию. Сравним эти шаблоны.

Сначала замечание по поводу Моста. С ним можно работать двумя способами. В наших примерах у нас уже была одна реализация (первая версия системы) и нам нужно было выделить некий общий интерфейс (мы назвали его Мост), который должна была бы реализовать и вторая версия. Для этого мы ввели Абстракцию (Портал), которая напрямую соединена с интерфейсом Мост (это подход называется «к Мосту» (Bridge-up)). Однако можно сделать по-другому, т.е. начать с Абстракции и создавать параллельно с ней реализации (подход - «от Моста» (Bridge-down)).

Сравнение шаблонов приведено в таблице:

Название	Декоратор Component	Прокси Subject	«от Моста» Abstraction	«к Мосту» Implementation
<b>исходного объекта</b> (до применения шаблона)				
Вводится <b>интерфейс</b>	IComponent	ISubject	Bridge	Bridge
Создается <b>новый объект</b>	Decorator	Proxy	Implementation	Abstraction
Клиент должен <b>агрегировать</b>	Новый объект с интерфейсом	Новый объект	Исходный объект с новым объектом	Новый объект с исходным объектом
Клиент <b>может общаться</b>	С исходным объектом и новым объектом	С новым объектом	С исходным объектом	С новым объектом
<b>Исходный объект</b> должен быть изменен с помощью	Реализации интерфейса	Не меняется	Агрегирования интерфейса	Реализации интерфейса
Нужно добавить <b>новые классы</b> , которые	агрегируют интерфейс и реализуют интерфейс	агрегируют исходный объект и реализуют интерфейс	реализуют интерфейс	агрегируют интерфейс
<b>Операция</b> перенаправляется	от нового объекта к исходному	от нового объекта к исходному	от исходного объекта к новому	от нового объекта к исходному

Декоратор агрегирует интерфейс, который реализует исходный объект. То же делает и Мост.

Например:

```
// Decorator
Display(new DecoratorA(new Component ( ));
// К Мосту
Console.WriteLine(new Abstraction(new ImplementationA ( )).Operation ( ));
```

В пятой строке указаны объекты, с которыми клиент может общаться. Если клиент использует шаблон Декоратор, то он может общаться и с исходным объектом и с новым объектом (декоратором), но если использовать варианты шаблона Мост, то только с одним из них. Если же задействуется шаблон Прокси, то клиенту остается только работать с новым объектом (прокси-объектом).

Следующая строка показывает, как исходный объект изменяется после применения шаблона. Только Прокси не меняет исходный объект. Декоратор рассчитывает на то, что в исходном объекте будет реализован требуемый интерфейс, т.е. сначала разрабатывается класс исходного объекта, а потом к нему добавляются новые обязанности через интерфейс. В шаблоне Мост связанность более высокая и нужно понимать, что после применения этого шаблона исходный объект начинает знать слишком много о других частях системы.

Во всех шаблонах происходит перенаправление вызова некоторой операции, которой владеет исходный объект. Операция переадресовывается от нового объекта к исходному, а в шаблоне Мост все зависит от того, какой объект назвать новым, а какой исходным. Следует отметить, что для реальных приложений затрачиваемое время на переадресацию может иметь значение и стать слишком большим (особенно в случае распределенных систем).

## Структурные паттерны. Паттерн Композит (Composite). Паттерн Приспособленец (Flyweight)

### **Шаблон Композит (Компоновщик) (Composite)**

#### **Назначение**

Этот шаблон проектирования объединяет объекты в древовидную структуру для представления иерархии от частного к целому. Компоновщик позволяет клиентам обращаться к отдельным объектам и к группам объектов одинаково.

Предназначен для управления иерархиями, когда с одиночным компонентом иерархии (лист) и составными компонентами (композициями) можно работать одинаковым образом. Типовые операции у компонентов – это операции добавления, удаления, отображения, поиска и группировки.

#### **Пример**

Современные программы, как правило, хранят те или иные данные. Например, музыкальный плеер или альбом цифровых фотографий, такой как Flickr. В них элементы собираются в большой список, а каждый из элементов составляется из отдельных подэлементов.



В этой программе можно по-разному упорядочивать фотографии: в хронологическом порядке или по каким-то событиям. Одна и та же фотография может появляться во многих

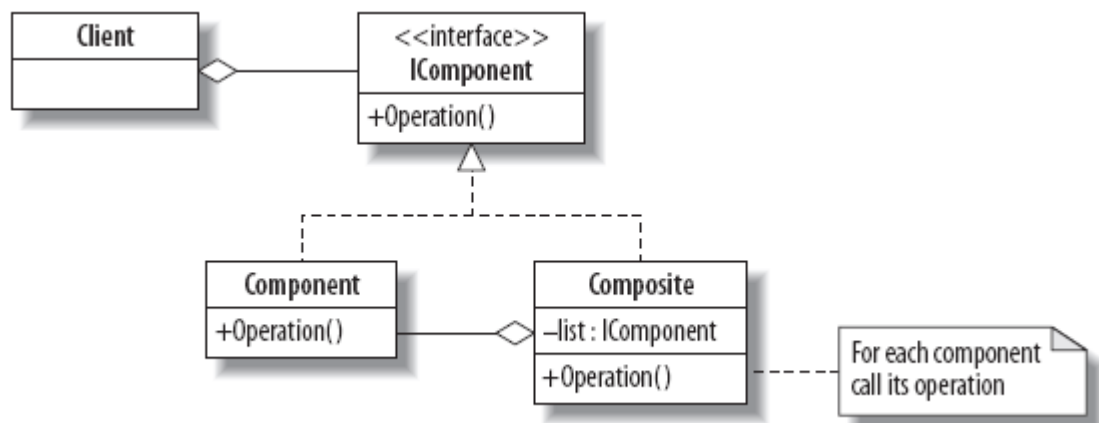
альбомах. Создание альбома – это на самом деле создание объекта Композит, но фотографии в него не обязательно копируются. Здесь важно то, что операции, применимые к фотографиям, должны быть применимы и к альбомам, т.е. должны иметь те же названия и давать один и тот же результат, несмотря на то, что реализация может быть разной. Например, пользователь должен иметь возможность отобразить фотографию или альбом (который тоже содержит фотографии). Или удаление и фотографии и альбома должно приводить к одному результату – их исчезновению.

## Другие примеры

Многие организации построены по иерархическому принципу. Например, когда начальнику нужен отчет он в цикле перебирает подразделения и сотрудников в них, отправляя каждому сообщение «написать свою часть отчета».

## Архитектура

На первый взгляд этот шаблон один из простейших. В нем участвуют два типа: Компоненты (Components) и Композиты (Composites). Оба типа удовлетворяют требованиям общего интерфейса. Композиты состоят из Компонентов и операций, применимые к Композитам, также применимы и к Компонентам.



## Участники шаблона:

### **IComponent**

Определяет операции, которые имеют смысл и для компонентов и для композитов (например, для файлов и для каталогов).

### **Операция (Operation)**

Операция из интерфейса IComponent

### **Компонент (Component)**

Реализует операции, которые применимы к одиночному объекту, т.е. объекту который уже не может быть разбит на составляющие.

### **Композит (Composite)**

Реализует операции, которые применимы как к составным объектам (композитам) так и к их составляющим.

Клиент имеет дело только с интерфейсом IComponent.

## Задача

IComponent	Видимый элемент в iPhoto
Component	Одна фотография
Composite	Альбом фотографий
Operation	Открыть и просмотреть

## Реализация

Реализовывать этот шаблон нужно так, чтобы не было важно, из элементов какого типа будет состоять композит. В нашем примере это альбомы и фотографии, но с таким же успехом могут быть люди и банковские счета. Операции над составными объектами, тем не менее, должны быть применимы к каждому одиночному элементу. Выполняться они должны при переборе всех элементов композита. Чтобы понять гибкость композитов, которые могут содержать объекты любых типов, рассмотрим **параметризованные типы** в C#.

Параметризованными могут быть структуры, классы, интерфейсы, делегаты и методы. Например, List<T> может быть на самом деле List<string> или List<Person>. Все коллекции в .NET могут быть параметризованными. Коллекции – это Dictionary, Stack, Queue и List и их некоторые разновидности. Также есть обобщенные методы такие, как Sort или BinarySearch. Они требуют, чтобы классы, с которыми они работают, реализовывали интерфейс IComparer.

Чтобы задать параметризованный (обобщенный) тип или метод, нужно использовать параметр типа в угловых скобках: <T> или <T, P>. Параметризованные типы могут сами быть параметрами. Чтобы определить конкретный тип по параметризованному, нужно указать вместо параметра имя конкретного типа, например, <string>.

Мы определим IComponent, Component и Composite как параметризованные типы и объявим их в отдельных пространствах имен:

```
public interface IComponent<T>
{
    void Add(IComponent<T> c);
    IComponent<T> Remove(T s);
    IComponent<T> Find(T s);
    string Display(int depth);
    T Item { get; set; }
}

1 using System;
2 using System.Collections.Generic;
3 using System.Text; // for StringBuilder
4
5 namespace CompositePattern
6 {
7
8     // The Interface
9     public interface IComponent<T>
10    {
11        void Add(IComponent<T> c);
12        IComponent<T> Remove(T s);
13        string Display(int depth);
14        IComponent<T> Find(T s);
15        T Name { get; set; }
16    }
17
18    // The Component
19    public class Component<T> : IComponent<T>
20    {
```

```

21     public T Name { get; set;}
22
23     public Component(T name)
24     {
25         Name = name;
26     }
27
28     public void Add(IComponent<T> c)
29     {
30         Console.WriteLine("Cannot add to an item");
31     }
32
33     public IComponent<T> Remove(T s)
34     {
35         Console.WriteLine("Cannot remove directly");
36         return this;
37     }
38
39     public string Display(int depth)
40     {
41         return new String('-', depth) + Name + "\n";
42     }
43
44     public IComponent<T> Find(T s)
45     {
46         if (s.Equals(Name))
47             return this;
48         else
49             return null;
50     }
51 }
52
53 // The Composite
54 public class Composite<T> : IComponent<T>
55 {
56     List<IComponent<T>> list;
57
58     public T Name { get; set;}
59
60     public Composite(T name)
61     {
62         Name = name;
63         list = new List<IComponent<T>>();
64     }
65
66     public void Add(IComponent<T> c)
67     {
68         list.Add(c);
69     }
70
71     IComponent<T> holder = null;
72
73     // Находит элемент, начиная с определенного места в иерархии
74     // и возвращает композит из которого был удален элемент.
75     // Если ничего не найдено, то возвращает исходный элемент.
76     public IComponent<T> Remove(T s)
77     {
78         holder = this;
79         IComponent<T> p = holder.Find(s);
80         if (holder != null)
81         {
82             (holder as Composite<T>).list.Remove(p);
83             return holder;
84         }
85         else
86             return this;
87     }
88
89     // Рекурсивно ищет заданный элемент
90     // Возвращает ссылку на него или null
91     public IComponent<T> Find(T s)
92     {
93         holder = this;
94         if (Name.Equals(s)) return this;
95         IComponent<T> found = null;
96         foreach (IComponent<T> c in list)
97         {
98             found = c.Find(s);
99             if (found != null)

```

```

100         break;
101     }
102     return found;
103 }
104
105 // Отображает иерархию с отступами
106 public string Display(int depth)
107 {
108     StringBuilder s = new StringBuilder(new String('-', depth));
109     s.Append("Set " + Name + " length :" + list.Count + "\n");
110     foreach (IComponent<T> component in list)
111     {
112         s.Append(component.Display(depth + 2));
113     }
114     return s.ToString();
115 }
116 }
117 }

```

Не все методы в интерфейсе IComponent существенны для Компонентов. Добавление и удаление возможно только в Композитах. Поэтому реализация таких методов для Компонентов заключается в выводе сообщения о невозможности выполнения данной операции. Для метода Find (44-50) нужно определить параметр T. Если его не определить, то возникнет ошибка во время компиляции.

Для метода Display (39-42) нужно, чтобы тип T для Name имел метод ToString().

Класс Composite также реализует интерфейс IComponent. Find и Remove устроены сложнее, чем в Компоненте.

Композит хранит свою структуру в списке, содержащем элементы типа Component и Composite (строка 56). Если элемент типа Composite, то создается новый объект, содержащий список. Список задается так:

```
List <IComponent <T>> list;
```

Видно, что элементом списка может быть элемент параметризованного типа.

Поведение метода Remove (76-87) заключается в том, что сначала мы находим нужный элемент и затем удаляем его из списка, хранящегося в Композите (82)

```
(holder as Composite<T>).list.Remove(p);
```

В списке хранятся переменные типа IComponent и поэтому их нужно приводить к типу Composite перед тем как что-то с ним делать.

Элементы обобщенного типа можно перебирать в цикле foreach (см. методы Find и Display (96 и 110))

```
foreach (IComponent <T> c in list) {
found = c.Find(s);
```

Когда мы вызываем Find, то на самом деле будет вызван метод переменной c, а она может иметь во время выполнения программы разный тип. Такое поведение наиболее целесообразно, если мы должны работать с Компонентами и Композитами одинаковым образом.

Итак, мы закончили реализацию шаблона Композит для теоретического примера. Кроме ограничений на метод Display, больше нам ничего не мешает применять три введенных типа при создании структуры данных, в которую могут быть помещены объекты любых типов.

В следующем примере мы построим дерево имен файлов, в которых хранятся фотографии. Файлы объединяются в папки (альбомы). С точки зрения пользователя наша структура должна уметь отвечать на вопрос: в каком месте дерева в данный момент выполняются какие-то действия. В этом нам поможет так называемый курсор. После выполнения некоторых команд курсор будет на компоненте, с которым велась работа, а после некоторых курсор будет перемещаться в начало папки, содержащего файлы.

Таким образом, к структуре применимы следующие команды:

AddSet

Создает новую пустую папку с заданным именем для файлов и оставляет курсор на ней.

AddPhoto

Добавляет новую фотографию в дерево сразу после курсора и перемещает курсор на нее.

Find

Ищет заданный компонент (папку или фотографию) или возвращает null, если ничего не найдено.

Remove

Удаляет заданный компонент (папку или фотографию) и передвигает курсор на папку, в котором содержался удаленный элемент.

Display

Отображает все дерево.

Quit

Выходит из программы.

Как видно, две операции применимы и Компонентам и Композитам одновременно: Find и Remove. Рассмотрим пример работы с системой. Команды приведены слева, названия элементов даны посередине, справа приведены комментарии:

```
AddSet Home
AddPhoto Dinner.jpg
AddSet Pets Вводим новый уровень иерархии
AddPhoto Dog.jpg
AddPhoto Cat.jpg
Find Album Перемещаемся на тот же уровень, на котором находится папка Home
AddSet Garden
AddPhoto Spring.jpg
AddPhoto Summer.jpg
AddPhoto Flowers.jpg
AddPhoto Trees.jpg
Display Возвращаемся к началу альбома и выводим всю структуру
Set Album length :2
--Set Home length :2
----Dinner.jpg
----Set Pets length :2
-----Dog.jpg
-----Cat.jpg
--Set Garden length :4
---Spring.jpg
---Summer.jpg
---Flowers.jpg
---Trees.jpg
    Remove Flowers.jpg Остаемся в Garden
    AddPhoto BetterFlowers.jpg Добавляем в конец Garden
    Display
Set Album length :2
--Set Home length :2
----Dinner.jpg
----Set Pets length :2
-----Dog.jpg
-----Cat.jpg
--Set Garden length :4
---Spring.jpg
---Summer.jpg
---Trees.jpg
---BetterFlowers.jpg
    Find Home
    Remove Pets
    Display
Set Album length :2
--Set Home length :1
----Dinner.jpg
--Set Garden length :4
---Spring.jpg
---Summer.jpg
---Trees.jpg
---BetterFlowers.jpg
Quit
```

Клиентский класс Client с реализацией приведен ниже. Фактически – это некий интерпретатор команд.

```
using System;
using System.Collections.Generic;
using System.IO;
using CompositePattern;
```

```

// Клиент
class CompositePatternExample
{
    static void Main()
    {
        IComponent<string> album = new Composite<string>("Album");
        IComponent<string> point = album;
        string[] s;
        string command, parameter;
        // Создание и управление нашей структурой
        StreamReader instream = new StreamReader("Composite.dat");
        do
        {
            string t = instream.ReadLine();
            Console.WriteLine("\t\t\t\t" + t);
            s = t.Split();
            command = s[0];
            if (s.Length > 1) parameter = s[1]; else parameter = null;
            switch (command)
            {
                case "AddSet":
                    IComponent<string> c = new Composite<string>(parameter);
                    point.Add(c);
                    point = c;
                    break;
                case "AddPhoto":
                    point.Add(new Component<string>(parameter));
                    break;
                case "Remove":
                    point = point.Remove(parameter);
                    break;
                case "Find":
                    point = album.Find(parameter);
                    break;
                case "Display":
                    Console.WriteLine(album.Display(0));
                    break;
                case "Quit":
                    break;
            }
        } while (!command.Equals("Quit"));
    }
}

```

## Использование

Шаблон Композит применяется очень часто и обычно применяется вместе с Декоратором, Итератором и Посетителем. Композит – это фактически иерархическая структура данных, но структура, которая может содержать данные разных типов и позволяющая работать с ними одинаковым образом.

Композит нужно использовать, когда

- есть некоторая структура элементов и объединяющих их контейнеров.

При этом нужно:

- чтобы для клиента не было разницы, с какими элементами он работает: одиночными или составными.

В то же время:

- Можно использовать Декоратор, в который добавить операции Add, Remove и Find.

- Шаблон Flyweight (Приспособленец) похож на Композит, но он не умеет отвечать на вопрос «в каком месте была выполнена операция» и начинает выполнение всех операций с корневого элемента.
- Шаблон Visitor (Посетитель) собирает операции, разделенные в шаблоне Композит по классам Компонент и Композит, в одном месте.

## Задачи

1. Руководитель – это сотрудник, который управляет инженерами, программистами и обслуживающим персоналом, каждый из которых тоже сотрудник фирмы. К каждому из них применима операция «уходить в отпуск». Смоделировать эту ситуацию с помощью шаблона Композит.
2. В теоретическом примере выше в классе Component были введены две функции, выводящие сообщения об ошибках. Переписать этот пример, используя исключения. Проанализировать: должны ли быть исключения частью интерфейса IComponent?
3. Метод Display в классе Component и Composite предполагают, что тип T может быть приведен к строке. Допустим, это не так (например, T это Image). Можно ли переопределить Display так, чтобы он работал и в этом случае (например, с помощью расширяющих методов)?
4. В классе Composite поменяйте класс List на класс Dictionary. Как тогда нужно изменить метод Find?
5. Хотя мы ввели один интерфейс, но можно разделить интерфейс на несколько, один из которых применим, например, только к Композитам. Тогда у нас получится интерфейс из двух уровней:

```
interface IComponent {
// Name, Display
}
interface IComposite<T> IComponent where T : IComponent {
// Add, Remove, Find
}
```

Теперь нам не обязательно делать IComponent параметризованным – Компонент теперь просто нечто, имеющее имя и его можно отобразить. При этом снимается проблема выбора: что использовать T или IComponent<T>. Конструкция из этого примера – это тоже новая возможность C# 3.0, которая называется параметризованные ограничения.

## Шаблон Приспособленец (Flyweight)

### Назначение

Шаблон Приспособленец предлагает эффективный способ разделить общую информацию, находящуюся в небольших объектах, причем число таких объектов в системе велико. Основная проблема - в затратах на хранение таких объектов, причем часто информация, содержащаяся в них, дублируется. В шаблоне заостряется внимание на двух состояниях объекта: внутреннем и внешнем. Наибольший выигрыш от применения шаблона возникает, когда объекты используют оба состояния, причем:

- **внутреннее** состояние может быть разделено между многими объектами с целью минимизировать расходы на хранение,
- **внешнее** состояние может быть вычислено «на лету», т.е. предпочтение отдается вычислению по сравнению с хранением.

## Пример

В нашем примере с библиотекой фотографий мы рассматривали шаблон Композит. Одна из возможностей библиотеки – показывать страницу, заполненную уменьшенными копиями изображений. При этом мы хотим, чтобы эта страница при необходимости могла прокручиваться без существенных временных затрат. Это означает, что как можно больше изображений должны быть предварительно загружены в память и храниться там, пока запущена программа. В нашей программе также есть функция группировки фотографий, причем фотография может принадлежать одновременно нескольким группам, поэтому число preview-изображений может превысить допустимое количество. Из-за того, что все фотографии в память не поместятся и каждая конкретная фотография может принадлежать нескольким группам, возрастает нагрузка на жесткий диск.

Вот, например, пример группировки фотографий:



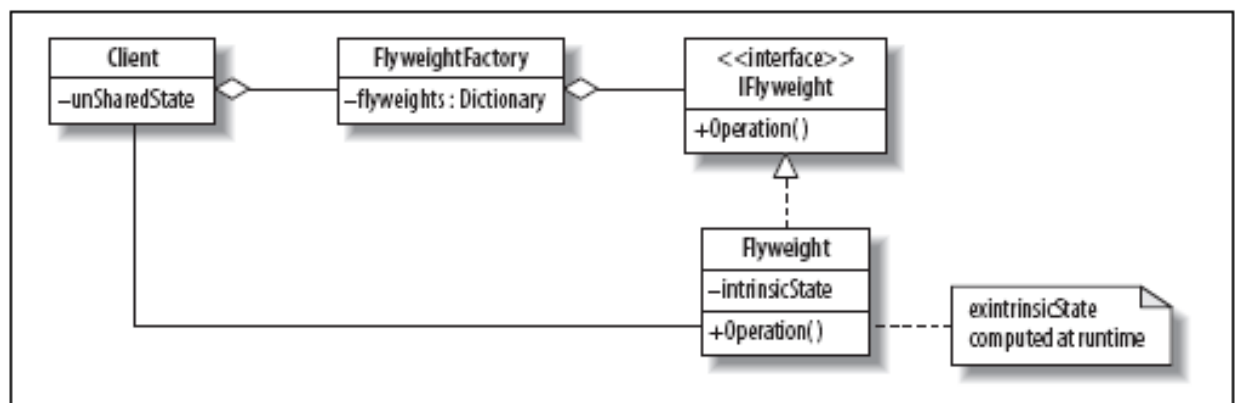
Если это окно уменьшить, то появится полоса прокрутки. Если некоторые изображений окажутся за пределами этого окна, то их потребуются отображать заново при прокрутке, а это ресурсоемкая операция. **Общее (неразделяемое) состояние** объектов в данном случае – это принадлежность к одной или нескольким группам. **Внешнее состояние** изображения – это изображение полного размера, которое может быть довольно большим (например, 2 Мб). Также может быть и **внутреннее состояние** – уменьшенная копия изображения (thumbnail) размером 8 Кб. Объекты такого размера уже можно хранить в памяти во время работы программы. Эти небольшие изображения могут быть перегруппированы в соответствии с принадлежностью к группам. Изображения же полного размера могут быть показаны по требованию.

## Другие примеры

Некоторые фирмы используют пул разработчиков с разными навыками (Приспособленцев) вместо того, чтобы надолго привязывать определенное количество разработчиков к определенным проектам. Когда очередному проекту требуются люди, какой-то относительно свободный разработчик назначается на небольшую работу по проекту. После ее выполнения он возвращается в пул. Преимущество такого подхода в том, что программисты могут быть разделены между различными проектами. Недостаток – возрастают затраты на управление этим процессом, а также возрастает время на разнообразные совещания.

## Архитектура

Рассмотрим диаграмму шаблона.



Суть шаблона в том, что разделить состояние некоторого объекта на состояния трех типов. Внутреннее состояние (intrinsicState) принадлежит самому объекту. Класс Flyweight реализует интерфейс IFlyweight, который определяет операции, в которых заинтересована остальная часть системы. Клиент владеет общим (неразделяемым) состоянием (unSharedState), а также коллекцией Приспособленцев, которых производит Фабрика Приспособленцев (FlyweightFactory). Ее задачей является контроль за тем, что будет создан ровно один объект нужного типа. Наконец, внешнее состояние (extrinsicState) не появляется в системе как таковое. Если оно понадобится, то оно будет вычислено уже во время выполнения программы для каждого внутреннего состояния (intrinsicState).

### Клиент (Client)

Вычисляет и владеет общим (неразделяемым) состоянием объектов.

### IFlyweight

Определяет интерфейс, через который Приспособленцы могут получать сообщения, касающиеся их внутреннего состояния.

### Фабрика Приспособленцев (FlyweightFactory)

Создает и управляет уникальными Приспособленцами.

### Приспособленец (Flyweight)

Хранит внутреннее состояние, различное для различных объектов Приспособленцев.

Возможны варианты. Например, на диаграмме Приспособленец вычисляет свое внешнее состояние (extrinsicState). Однако, это может сделать и Клиент, выполнив все вычисления и передав extrinsicState как параметр метода Operation Приспособленцу. Также unSharedState может иметь более сложную структуру и даже иметь свой собственный класс. В этом случае, он также должен реализовывать IFlyweight.

## Задача

Client	Приложение с функцией группировки фотографий
IFlyweight	Спецификация изображения
FlyweightFactory	Реестр уникальных изображений
Flyweight	Создатель уменьшенной копии изображения
intrinsicState	Уменьшенная копия изображения
extrinsicState	Изображение полного размера
unSharedState	Информация о принадлежности к группам

## Реализация

Наша реализация будет включать возможности языка C# от 1.0 до 3.0:

- структуры;
- индексаторы;
- неявное задание типов для локальных переменных и массивов;
- анонимные типы;
- инициализацию объектов и коллекций.

Также будем использовать параметризованные (обобщенные) коллекции, появившиеся в C# 2.0.

В шаблоне задействованы три типа: IFlyweight, Flyweight и FlyweightFactory. Как и в примере с Композитом, мы поместим их в пространства имен.

Начнем с интерфейса:

```
public interface IFlyweight
{
    void Load(string filename);
    void Display(PaintEventArgs e, int row, int col);
}
```

Метод Load загружает в память уменьшенную копию изображения заданной фотографии по ее имени файла. Метод Display отображает уменьшенную копию изображения. Чуть позже нам понадобится дополнительные параметры метода Display для отображения полноразмерной фотографии.

Рассмотрим такое понятие C# как **структуры**.

В C# есть две конструкции для хранения атрибутов и методов: классы и структуры.

Структуры похожи на классы в том, что они тоже представляют типы. Однако, переменная типа структуры содержит соответствующий ей объект целиком (по значению), в то время как объект класса содержит только ссылку. Экземпляры и классов и структур – объекты. Однако, структуры имеют семантику значений, т.е. при присваивании объекта типа структура объект передается целиком, а не по ссылке, как в случае с экземплярами классов. Структуры обычно используются для хранения небольшого набора данных. Ограничение в том, что структуры не могут быть унаследованы от классов, но могут реализовывать интерфейсы.

Flyweight – идеальный кандидат на структуру. Flyweight – небольшой и он ни от чего не наследует (хотя он реализует интерфейс).

Поэтому введем структуру:

```

public struct Flyweight : IFlyweight
{
    // Внутренне состояние
    Image pThumbnail;
    public void Load(string filename)
    {
        pThumbnail = new Bitmap("images/" + filename).
            GetThumbnailImage(100, 100, null, new IntPtr());
    }
    public void Display(PaintEventArgs e, int row, int col)
    {
        e.Graphics.DrawImage(pThumbnail, col * 100 + 10, row * 130 + 40,
            pThumbnail.Width, pThumbnail.Height);
    }
}

```

Как и требовалось, Load использует изображение с заданным именем, но не оставляет его полноразмерную копию во внутреннем состоянии, он только сохраняет уменьшенную копию. Вычисления нужны для позиционирования уменьшенной копии на поверхности окна. Обратите внимание, что параметры row и col наследуются из unSharedState, который хранится у клиента.

Теперь рассмотрим Фабрику. Обычно фабрики нужны для того, чтобы создавать объекты в соответствии со специальными требованиями. В данном случае, нам нужно проверить, существует ли объект и если не существует, то добавить его в коллекцию.

Покажем, чем могут быть полезны так называемые индексаторы.

```

1     public class FlyweightFactory
2     {
3         // Хранит индексированный список объектов IFlyweight
4         Dictionary<string, IFlyweight> flyweights =
5         new Dictionary<string, IFlyweight>();
6
7         public FlyweightFactory()
8         {
9             flyweights.Clear();
10        }
11
12        public IFlyweight this[string index]
13        {
14            get
15            {
16                if (!flyweights.ContainsKey(index))
17                    flyweights[index] = new Flyweight();
18                return flyweights[index];
19            }
20        }
21    }

```

В строках 4-5 создается словарь (Dictionary), который ставит в соответствие Приспособленцам строки. В строках 17-18 происходит обращение к объекту словаря с помощью индексатора, который является частью Dictionary. В строке 13 используется метод set, а в 14 – get. Все это части метода на строках 12-20, где мы определяем индексатор для нашего класса FlyweightFactory. В строке 12 указаны тип возвращаемого значения и тип ключа, а в 14-19 – тело метода get. В нем сначала проверяется, существует ли объект Flyweight с таким ключом. Если не существует, то он создается, иначе возвращается элемент с соответствующим индексом.

Индексатор – это метод, который позволяет объекту быть проиндексированным таким же способом, как и массив. Индексатор похож на свойство и использует такой же синтаксис для методов `get` и `set`. Общее определение синтаксиса таково:

```
modifiers return-type this[key-type key] {  
get { ... }  
set { ... }  
}
```

Если в классе есть индексатор, то его можно использовать так:

```
obj[index] = x;  
x = obj[index];
```

Индексаторы уже определены для массивов и для коллекций.

Таким образом, если мы создам нашу фабрику так:

```
static FlyweightFactory album = new FlyweightFactory( );
```

то мы можем получить к ней доступ:

```
album[filename].Load(filename);
```

Таким образом, индексатор позволяет работать с объектом, который владеет некоторой коллекцией, как с самой коллекцией.

### Текст программы:

```
using System;  
using System.Collections.Generic;  
using System.Drawing;  
using System.Windows.Forms;  
using FlyweightPattern;  
  
namespace FlyweightPattern  
{  
    // Flyweight Pattern  
    public interface IFlyweight  
    {  
        void Load(string filename);  
        void Display(PaintEventArgs e, int row, int col);  
    }  
    public struct Flyweight : IFlyweight  
    {  
        // Внутреннее состояние  
        Image pThumbnail;  
        public void Load(string filename)  
        {  
            pThumbnail = new Bitmap("images/" + filename).  
                GetThumbnailImage(100, 100, null, new IntPtr());  
        }  
        public void Display(PaintEventArgs e, int row, int col)  
        {  
            e.Graphics.DrawImage(pThumbnail, col * 100 + 10, row * 130 + 40,  
                pThumbnail.Width, pThumbnail.Height);  
        }  
    }  
    public class FlyweightFactory  
    {  
        // Хранит объекты типа IFlyweight  
        Dictionary<string, IFlyweight> flyweights =  
            new Dictionary<string, IFlyweight>();  
        public FlyweightFactory()  
        {  
            flyweights.Clear();  
        }  
    }  
}
```

```

    public IFlyweight this[string index]
    {
        get
        {
            if (!flyweights.ContainsKey(index))
                flyweights[index] = new Flyweight();
            return flyweights[index];
        }
    }
}

//===== End of namespace, start of program

class Client
{
    // Разделяемое состояние - сами изображения
    static FlyweightFactory album = new FlyweightFactory();
    // Неразделяемое состояние - принадлежность к группам
    static Dictionary<string, List<string>> allGroups =
    new Dictionary<string, List<string>>();
    public void LoadGroups()
    {
        var myGroups = new[] {
            new {Name = "Garden",
                Members = new [] {"pot.jpg", "spring.jpg",
                "barbeque.jpg", "flowers.jpg"}},
            new {Name = "Italy",
                Members = new [] {"cappucino.jpg", "pasta.jpg",
                "restaurant.jpg", "church.jpg"}},
            new {Name = "Food",
                Members = new [] {"pasta.jpg", "veggies.jpg",
                "barbeque.jpg", "cappucino.jpg", "lemonade.jpg" }},
            new {Name = "Friends",
                Members = new [] {"restaurant.jpg", "dinner.jpg"}}
        };
        // Загружаем Flyweights, сохраняем разделяемое внутреннее состояние
        foreach (var g in myGroups)
        { // неявное выведение типов
            allGroups.Add(g.Name, new List<string>());
            foreach (string filename in g.Members)
            {
                allGroups[g.Name].Add(filename);
                album[filename].Load(filename);
            }
        }
    }
    public void DisplayGroups(Object source, PaintEventArgs e)
    {
        // Отображаем Flyweights, передавая неразделяемое состояние
        int row;
        foreach (string g in allGroups.Keys)
        {
            int col;
            e.Graphics.DrawString(g,
                new Font("Arial", 16),
                new SolidBrush(Color.Black),
                new PointF(0, row * 130 + 10));
            foreach (string filename in allGroups[g])
            {
                album[filename].Display(e, row, col);
                col++;
            }
            row++;
        }
    }
}

```

```

    }
}

class Window : Form
{
    Window()
    {
        this.Height = 600;
        this.Width = 600;
        this.Text = "Picture Groups";
        Client client = new Client();
        client.LoadGroups();
        this.Paint += new PaintEventHandler(client.DisplayGroups);
    }
    static void Main()
    {
        Application.Run(new Window());
    }
}

```

Приложение работает в два этапа: создание групп и отображение групп. Для простоты, мы будем использовать метод `Application.Run`, чтобы отобразить изображение. Объект `Window` вызывает `LoadGroups` явно, а `DisplayGroups` вызывается неявно через событие `PaintEventHandler`, которое возникает, когда форма (`Form`) отрисовывается. `LoadGroups` иллюстрирует применение неявной типизации, которая появилась в C# 3.0.

Рассмотрим эту возможность неявной типизации.

Переменные могут быть объявлены как поля в классах или как локальные переменные в методах. Тип локальной переменной метода может быть выведен из инициализирующего ее выражения. Это называется неявной типизацией. Для таких переменных вместо типа может быть указано ключевое слово `var`. Таким образом, получается экономия на записи типа переменной. В следующем примере тип указан только один раз, а не два, как нужно было до C# 3.0:

```
var marks = new Dictionary <string, int> ( );
```

Если переменная анонимного типа (о них будет сказано дальше), то такой тип также может быть выведен, как например:

```
foreach (var g in myGroups) { ... }
```

`myGroups` в методе `LoadGroups` – пример анонимного типа, который инициализируется коллекциями или массивами. Этот тип используется для хранения имен файлов изображений в каждой группе. Конечно, непосредственно данные изображения должны быть прочитаны с диска, но интересно отметить как имена файлов могут быть введены в программу через структуру данных анонимного типа.

Мы не указываем явно тип переменной `g`. Однако, компилятор после вывода ее типа допускает использование `g.Name` и `g.Members`, т.е. он знает к этому моменту тип `g`.

```

foreach (var g in myGroups)
{ // неявное выведение типа
    allGroups.Add(g.Name, new List<string>());
    foreach (string filename in g.Members)
    {
        allGroups[g.Name].Add(filename);
        album[filename].Load(filename);
    }
}

```

Обращение к объекту Приспособленец происходит в последней строке. В ней либо возвращается ссылка на существующий объект, либо создается новый. Метод Load вызывается у объекта типа IFlyweight.

Массивы также можно инициализировать, применяя неявную типизацию:

```
var myGroups = new [] {  
    new {Name= "Garden",  
        Members = new [] {"pot.jpg", "spring.jpg"  
            "barbeque.jpg", "flowers.jpg"}},  
    new {Name = "Friends",  
        Members = new [] {"restaurant.jpg", "dinner.jpg"}}  
};
```

В данном случае будет создан массив из двух строк. Каждый элемент – объект с двумя полями: Name и Members. Members является массивом переменной длины. Причем класс для этого объекта определяется автоматически и объект этого класса сразу же инициализируется.

Инициализаторы задают значения для полей или свойств объектов или коллекций.

Примеры инициализаторов:

```
Point p = new Point {X = 0, Y = 1};  
List<int> digits = new List<int> { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
```

Анонимные типы создаются при помощи инициализаторов объектов. Анонимный тип – это класс без имени, который наследует напрямую от базового типа object. Члены анонимного типа – последовательность свойств только для чтения, которые соответствуют тому, что было указано в инициализаторе при создании экземпляра анонимного типа. Инициализаторы могут задавать значения и для коллекций объектов. Например, в данном случае будет создан анонимный тип:

```
var group = new {Name = "Italy",  
    Members = new [] {"cappuccino.jpg", "pasta.jpg",  
        "restaurant.jpg", "church.jpg"}},
```

Этот тип имеет два свойства Name и Members, где Members – массив строк.

Метод get создается автоматически, поэтому мы можем получить значение group.Name, но set метод при этом не создается.

## Применение шаблона

Шаблон Приспособленец следует использовать, если выполняются следующие условия:

- Приходится работать с очень большим количеством объектов, которые могут не поместиться в оперативную память.
- Значительная (по размеру) часть объекта (некоторое его состояние) может храниться на диске или быть вычислена во время выполнения программы.
- Оставшаяся часть состояний может быть разбита на большое количество небольших объектов, которые могут иметь разделяемое состояние.
- Чаще всего этот шаблон применяется при обработке текстов, где Приспособленцами являются буквы.

Итак, шаблон Приспособленец нужно использовать когда:

- имеется настолько большое количество объектов, что их неудобно хранить в памяти,
- можно выделить несколько состояний объектов, которые можно обрабатывать по отдельности,
- есть группы объектов, которые имеют общее состояние,
- есть методы вычисления состояний во время выполнения программы,

- в итоге нужно создать систему, независимую от ограничений на оперативную память.

## Задачи

1. Реализовать окончательно приложение для группировки фотографий, в котором можно просматривать полноразмерные изображения (внешнее состояние) после щелчка по уменьшенной копии изображения (внутреннее состояние)
2. Интегрировать библиотеку фотографий и приложение для группировки фотографий так, чтобы были использованы два шаблона: Композит и Приспособленец. При этом должны выполняться команды и отображаться фотографии.
3. В примере шаблона Композит спецификация библиотеки фотографий читалась из файла. В качестве примера инициализации объектов, определить начальное состояние библиотеки, используя некоторые данные:
 

```
AddSet Home
AddPhoto Dinner.jpg
AddSet Pets Добавляем папку Pets и теперь будем помещать данные в нее
AddPhoto Dog.jpg
AddPhoto Cat.jpg
Find Album Перемещаемся на тот уровень, на котором находится папка Home
AddSet Garden
AddPhoto Spring.jpg
AddPhoto Summer.jpg
AddPhoto Flowers.jpg
AddPhoto Trees.jpg
```

## Сравнение шаблонов

Мы можем сравнить шаблоны Композит и Приспособленец, т.к. они создают структуру для хранения большого числа объектов.

Шаблон Композит удобен, когда нужно выполнять команды над объектами, входящими в иерархическую структуру, причем команды не должны отличаться для контейнеров и для входящих в них элементов. Шаблон Приспособленец нужен тогда, когда нужно экономно распределить место для хранения большого количества однотипных объектов.

В наших примерах при реализации шаблона Композит мы применяли параметризованные (обобщенные) классы. Однако нужно не забывать, что тип, соответствующий Компоненту (Component), должен иметь методы Equals и ToString. Equals нужен, чтобы работал метод Find, а ToString нужен для вывода и упорядочивания информации о данном классе.

В шаблоне Приспособленец задается пространство имен, содержащее интерфейс IFlyweight и класс Flyweight, которые в нашем случае весьма тесно завязаны на фотоизображения.

Шаблон Приспособленец может быть полезен в любых ситуациях, где нужно компактно хранить данные.

Разделение на внутреннее и внешнее состояние в этом шаблоне приводит к мысли об аналогии с шаблоном Прокси (виртуальным Прокси). Изначально при разработке этого шаблона предполагалось, что внешнее состояние должно вычисляться по внутреннему. В нашем примере приложения для группировки фотографий мы проводили не слишком много вычислений для построения внешнего состояния, а просто использовали имя файла фотографии, чтобы ее загрузить. Шаблон Прокси похож тем, что по имени файла происходит обращение к «тяжелому» объекту – жесткому диску, откуда и происходит загрузка. Отличие в том, что в случае Прокси происходит работа с одним объектом, а Приспособленец работает с коллекцией (словарем) объектов.

	Композит	Приспособленец
Задаваемые типы	IComponent <T>	IFlyweight

	Component <T> Composite <T>	Flyweight FlyweightFactory
Используются ли параметризованные типы?	Да	FlyweightFactory не зависит от других типов
Ограничение	T должен реализовывать Equals и ToString	Структура и взаимодействие внутреннего и внешнего состояния встроены в класс Flyweight

## Структурные паттерны. Паттерн Адаптер (Adapter)

Шаблон Адаптер предназначен для обеспечения совместной работы классов, которые изначально не были предназначены для совместного использования. Такие ситуации часто возникают, когда идет работа с библиотеками. Часто они написаны так, что нет возможности их изменить, но при этом изменения нужны. Другими словами, интерфейс библиотеки не отвечает требованиям к интерфейсу со стороны системы.

Требования особенно часто меняются для методов ввод-вывода. Допустим, существует полезная библиотека, которая обладает актуальной функциональностью, но те способы, которыми она вводит/выводит данные, уже устарели. Гораздо проще изменить ввод/вывод, чем переписывать всю библиотеку. Возникает задача адаптации ее интерфейса к современным требованиям.

Рассмотрим реальный пример, в котором был применен подход Адаптер.

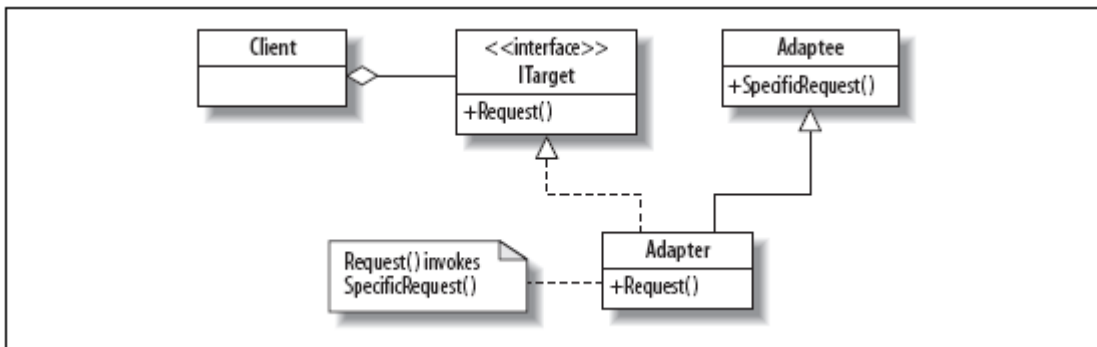
ОС Mac OS X была спроектирована для процессоров PowerPC и поэтому программы использовали библиотеку AltiVec для вычислений с плавающей запятой. Библиотека опиралась на инструкции SIMD процессора PowerPC. Недавно ОС была переведена на использование процессоров Intel с инструкциями SSE. Все программы, которые были написаны с использованием AltiVec могли быть, конечно, переписаны так, чтобы использовать SSE. Однако Apple понимала, что разработчики могут не иметь исходных текстов программ, использующих AltiVec. Поэтому было рекомендовано использовать фреймворк Accelerate. При его использовании разработчик может пользоваться абстракциями высокого уровня без детализации до конкретных команд, которые реально производят вычисления в процессоре и ему не нужно знать архитектуру процессора. Фреймворк автоматически задействует необходимый набор процессорных инструкций (PowerPC или Intel). Получилось, что Accelerate стал адаптером. Ему предоставляется некоторая сущность (набор инструкций), которую он адаптирует к требуемой. С помощью адаптера стало возможно встраивание существующего кода в новую среду. При этом изменения в существовавший код вносить не потребовалось.

### **Структура шаблона**

В Адаптере четко видно преимущество программирования согласно интерфейсам. Client работает в соответствии с требованиями своей предметной области, эти требования отражены в целевом интерфейсе ITarget (интерфейс, в котором заинтересован клиент). Адаптируемый класс Adaptee обладает требуемой функциональностью, но неподходящим интерфейсом. Адаптер Adapter реализует интерфейс ITarget и перенаправляет вызовы от Client к Adaptee, изменяя при необходимости параметры и возвращаемые значения. Также может существовать дополнительный класс Target, который также реализует интерфейс ITarget, но это не обязательно. В любом случае, Client зависит только от интерфейса ITarget.

Существует несколько типов адаптеров.

На следующем рисунке показан **адаптер класса**, поскольку он одновременно реализует интерфейс и наследует от адаптируемого класса Adaptee. Альтернативой наследованию от Adaptee может стать агрегация объекта типа Adaptee. В этом случае получится **адаптер объекта**. Разница в этих двух способах в том, что при наследовании легче **переопределить** поведение Adaptee, а при агрегации – **добавить** к Adaptee поведение.



Использование интерфейса ITarget позволит адаптируемыми объектами быть заменяемыми с другими объектами, реализующими тот же интерфейс. В то же время, адаптируемые объекты могут не удовлетворять сигнатурам методов из ITarget, поэтому сам по себе интерфейс не может решить задачу адаптации. Поэтому и понадобился паттерн Адаптер. Adaptee имеет функциональность, соответствующую методу Request, но возможно она находится в методе с другим названием или другими параметрами. Adaptee является полностью независимым от других классов и от соглашений, принятых в системе, поэтому в нем ничего изменить не получится.

Рассмотрим роли классов из шаблона.

**ITarget**

Интерфейс, который требуется Client

**Adaptee**

Реализация, которая требует адаптации

**Adapter**

Класс, который реализует ITarget в соответствии с Adaptee

**Request**

Метод, который нужен Client

**SpecificRequest**

Реализация метода Request в Adaptee

Роли	
Client	Любое приложение из Mac OS X
ITarget	Спецификации инструкций из библиотеки AltiVec
Request	Вызов инструкции из AltiVec
Adapter	Фреймворк Accelerate
Adaptee	Процессор Intel с набором инструкций SSE
DifferentRequest	Вызов инструкции SSE

## Пример

Допустим данные (числа) хранятся в некотором компоненте с высокой точностью, а клиент умеет оперировать ими только как с целыми числами. Поэтому методы интерфейса, которым пользуется клиент, содержат параметры целого типа, а методы компонента - параметры вещественного типа. В интерфейсе `ITarget` содержится метод, интересующий клиента.

```
1 using System;
2
3
4
5
6 // Existing way requests are implemented
7 class Adaptee {
8     // Provide full precision
9     public double SpecificRequest(double a, double b) {
10         return a / b;
11     }
12 }
13
14 // Required standard for requests
15 interface ITarget {
16     // Rough estimate required
17     string Request(int i);
18 }
19
20 // Implementing the required standard via Adaptee
21 class Adapter : Adaptee, ITarget {
22     public string Request(int i) {
23         return "Rough estimate is " + (int)Math.Round(SpecificRequest(i, 3));
24     }
25 }
26
27 class Client {
28
29     static void Main() {
30         // Showing the Adaptee in standalone mode
31         Adaptee first = new Adaptee();
32         Console.Write("Before the new standard\nPrecise reading: ");
33         Console.WriteLine(first.SpecificRequest(5, 3));
34
35         // What the client really wants
36         ITarget second = new Adapter();
37         Console.WriteLine("\nMoving to the new standard");
38         Console.WriteLine(second.Request(5));
39     }
40 }
41 /* Output
42 Before the new standard
43 Precise reading: 1.666666666666667
44
45 Moving to the new standard
46 Rough estimate is 2
47 */
```

Программа содержит два сценария. В первом (строка 33) метод `Adaptee` вызывается напрямую (вывод на строке 43). Однако клиенты могут иметь другой интерфейс (строки 17 и 38). `Adapter` реализует `ITarget` и наследует `Adaptee` (строка 21). В результате, он

может принимать вызов метода Request с параметрами целого типа и перенаправлять их Adaptee, вызывая метод с параметрами вещественного типа (строка 23). Вывод – на строке 46.

Особенность адаптеров в том, что они могут добавлять дополнительное поведение к тому поведению, что специфицируется в ITarget и в Adaptee. Другими словами, адаптеры могут быть прозрачными для клиента и непрозрачными. Выше показан пример последнего случая, где Adapter добавляет "Rough estimate is". Эта добавка показывает, что вызов Request был адаптирован (изменен) перед тем, как был вызван метод SpecificRequest (строка 23).

Адаптеры могут прикладывать разные усилия для того, чтобы адаптировать Adaptee к целевому интерфейсу Target. Простейшей адаптацией является перенаправление вызова методу с другим именем. Однако клиенту может потребоваться от адаптера поддержка дополнительного набора методов. Например, фреймворк Accelerate проделывает довольно много дополнительной работы, чтобы преобразовать инструкции из Altivec в аналогичные интеловские инструкции.

Поэтому можно сказать, что существуют следующие комбинации адаптируемого объекта и адаптера.

- Интерфейсы Adapter и Adaptee имеют одинаковые сигнатуры.
  - Это тривиальный случай. В сети много примеров именно этого типа адаптера, но это не самый его полезный вариант.
- Методы интерфейса Adapter имеют немного больше параметров, чем методы интерфейса Adaptee.
  - Адаптер вызывает метод Adaptee с заглушками вместо параметров.
- Методы интерфейса Adapter имеют намного больше параметров, чем методы интерфейса Adaptee.
  - Адаптер добавляет недостающую функциональность, становясь наполовину адаптером, наполовину компонентом.
- Методы интерфейса адаптера имеют другие типы параметров, чем методы интерфейса Adaptee.
  - Адаптер выполняет преобразование типов.

Возможны также комбинации этих вариантов.

## ***Двусторонние адаптеры***

Адаптеры обеспечивают доступ к некоторым методам Adaptee (требуемые методы перечислены в интерфейсе ITarget), но объекты адаптера не взаимозаменяемы с объектами Adaptee. Они не могут использоваться там, где могут использоваться объекты Adaptee, т.к. они работают с реализацией Adaptee, а не с их интерфейсом. Иногда нам нужны объекты, которые могли трактоваться и как объекты типа ITarget и как объекты типа Adaptee. Этого можно легко добиться, если при создании Adapter будет использовано множественное наследование, однако в C# его нет, и придется искать другой способ.

Необходимость в двухсторонних адаптерах возникает в случаях, когда есть две системы, причем свойства одной из них должны быть использованы в другой и наоборот. Класс

адаптера проектируется так, чтобы он содержал все важные методы обеих систем и обеспечивал их одновременное использование. Результирующий адаптер можно использовать с двух точек зрения или с двух сторон. В теории можно создать и многосторонние адаптеры, но все упрется в невозможность множественного наследования: нужно будет вводить дополнительный интерфейс между исходным классом и адаптером.

Приведем пример двустороннего адаптера. На компьютере Mac с процессором Intel можно запускать Windows в виртуальной машине. Windows, конечно, рассчитана на работу с Intel и использует инструкции SSE. В этом случае Windows и Mac OS X можно рассматривать как клиентов, желающих получить доступ к Adaptee, т.е. процессору Intel. Адаптер должен уметь обрабатывать инструкции обоих типов и преобразовывать их при необходимости. С точки зрения клиента (т.е. ОС) ситуацию можно описать следующим псевдокодом:

```
В Mac OS X имеется вызов метода  
ExecuteAltiVec(instruction);
```

```
В Windows имеется вызов метода  
ExecuteSEE(instruction);
```

Adapter должен уметь обработать вызовы методов и из Windows, и из Mac OS X:

```
void ExecuteAltiVec(instruction) {  
    ExecuteSSE(ConvertToSSE(instruction));  
}  
void ExecuteSSE(instruction) {  
    Intel.ExecuteSSE(instruction);  
}
```

В итоге процессор может представляться и процессором типа PowerPC (в этом случае вызываются команды AltiVec) и процессором типа Intel. Для этого процессор «адаптируется» двусторонним способом.

Рассмотрим другой пример.

Допустим, разрабатывается система управления для летательного аппарата, способного садиться на воду (амфибия=корабль+самолет). Пусть он называется Seabird. Аппарат имеет свойства и самолета и корабля: например, корпус самолета, а двигатель корабля. Причем соединены эти части так, что управлять аппаратом можно через органы управления от обеих частей. В терминах подхода Seabird – это двусторонний адаптер классов Aircraft и Seacraft. Во время экспериментов с Seabird нужно иметь доступ к методам и данным из обоих классов. Другими словами, Seabird ведет себя одновременно и как Seacraft, и как Aircraft. Мы можем создать обычный адаптер, чтобы реализовать поведение Aircraft и воспользоваться свойствами Seacraft (наследуя интерфейс IAircraft и класс Seacraft), но воспользоваться уже реализованным поведением Seacraft применительно к Seabird уже не получится. Т.е. если нам нужно, чтобы при увеличении скорости (есть метод IncreaseRevs в Seacraft), аппарат автоматически взлетал, то придется переопределять метод IncreaseRevs, иначе скорость увеличиваться будет, а взлетать аппарат не будет.

Поэтому нам нужен двусторонний адаптер. Интерфейс ITarget, т.е. IAircraft, имеет два свойства: Airborne (признак взлета), Height и метод TakeOff. Класс Aircraft реализует этот интерфейс, т.е. делает аппарат летательным. Интерфейс IAdaptee, т.е. ISeacraft, имеет два метода Speed и IncreaseRevs, которые реализуются классом Seacraft. В этом классе

аппарат получает поведение корабля. Адаптер наследует от Adaptee (Seacraft) и реализует ITarget (IAircraft). При этом адаптеру придется кое-что сделать дополнительно, чтобы клиент, управляя аппаратом как кораблем, мог использовать привычные для корабля (Seacraft) методы, а управляя, как летательным аппаратом – методы Aircraft. Покажем это соответствие в таблице:

Aircraft (Target)	Seacraft (Adaptee)	Seabird (Adapter)	Experiment (Client)
		Наследует Seacraft, реализует Aircraft	создает экземпляр seabird
<b>Методы</b>			
TakeOff — устанавливает Airborne в true и Height равным 200		TakeOff — задействует Airborne и IncreaseRevs	seabird.TakeOff — перенаправляет запрос Seabird
	IncreaseRevs— меняет скорость на 10	IncreaseRevs— вызывает IncreaseRevs, Airborne из Seacraft и устанавливает высоту	(seabird as ISeacraft) IncreaseRevs— перенаправляет запрос Seabird
<b>Свойства</b>			
Airborne=true после взлета		Airborne=true, если Height > 50	seabird.Airborne— перенаправляет запрос Seabird
	Speed—возвращает значение скорости		(seabird as Seacraft) Speed — перенаправляет запрос Seacraft
Height— возвращает значение высоты		Height—возвращает значение высоты	seabird.Height— перенаправляет запрос Seabird

Таким образом, у клиента есть готовый корабль, а он хочет сделать из него амфибию (адаптер), обладающую поведением самолета. Требования к самолету собраны в интерфейсе IAircraft. И реализованы они будут уже в амфибии, в то время как корабль был готов до этого, что доказывает наследование от Seacraft. И в дополнение ко всему клиент хочет пользоваться амфибией так, как он привык пользоваться кораблем.

Классы, представляющие различные части аппарата, имеют различные методы: TakeOff - для самолета и IncreaseRevs - для корабля. В случае использования обычного адаптера работал бы только метод TakeOff. Метод IncreaseRevs был бы доступен, но напрямую его вызывать не имело бы смысла. В случае двустороннего адаптера метод IncreaseRevs из адаптируемого объекта переопределяется таким образом, чтобы при изменении скорости выше некоторого порога аппарат начинал взлетать. Т.о. этот метод адаптируемого объекта приобретает новый смысл при вызове из адаптера.

Двусторонний адаптер содержит свойства - Airborne, Speed и Height. Они получены из Aircraft и определены в адаптере.

В итоге клиент может использовать Seabird следующим образом:

```
1 Console.WriteLine("\nExperiment 3: Increase the speed of the Seabird:");
2 (seabird as ISeacraft).IncreaseRevs( );
3 (seabird as ISeacraft).IncreaseRevs( );
4 if (seabird.Airborne)
```

```

5 Console.WriteLine("Seabird flying at height "
6 + seabird.Height +
7 " meters and speed "+(seabird as ISeacraft).Speed + " knots");
8 Console.WriteLine("Experiments successful; the Seabird flies!");

```

Возможность обращения к `seabird.Airborne` и `seabird.Height` – это результат, который можно было получить и с помощью обычного адаптера. А вот возможность трактовать `Seabird` как `Seacraft` – это особенность двустороннего адаптера.

В итоге можно сказать, что обычный адаптер занимается перенаправлением вызовов к адаптируемому объекту, возможно меняя их сигнатуру. Протокол вызовов к самому адаптеру задается в интерфейсе `ITarget`.

Двусторонний адаптер не просто реализует интерфейс `ITarget`, но и переопределяет методы адаптируемого класса, возможно меняя их смысл. При этом клиент может пользоваться адаптером через те же методы, которые были в адаптируемом объекте. Также при таком подходе клиент всегда может привести тип адаптера к типу адаптируемого объекта (см. `(seabird as ISeacraft).IncreaseRevs()`).

Полный текст программы:

```

using System;
// Two-Way Adapter Pattern
// Embedded system for a Seabird flying plane
// ITarget interface
public interface IAircraft
{
    bool Airborne { get; } // в полете или нет
    void TakeOff();
    int Height { get; }
}

// Target
public sealed class Aircraft : IAircraft
{
    int height;
    bool airborne;
    public Aircraft()
    {
        height = 0;
        airborne = false;
    }
    public void TakeOff()
    {
        Console.WriteLine("Aircraft engine takeoff");
        airborne = true;
        height = 200; // Meters
    }
    public bool Airborne
    {
        get { return airborne; }
    }
    public int Height
    {
        get { return height; }
    }
}

// Adaptee interface
public interface ISeacraft
{
    int Speed { get; }
    void IncreaseRevs();
}

// Adaptee implementation
public class Seacraft : ISeacraft
{
    int speed = 0;
    public virtual void IncreaseRevs()
    {
        speed += 10;
    }
}

```

```

        Console.WriteLine("Seacraft engine increases revs to " + speed + " knots");
    }
    public int Speed
    {
        get { return speed; }
    }
}
// Adapter
public class Seabird : Seacraft, IAircraft
{
    int height = 0;
    // Двусторонний адаптер переопределяет этот метод и перенаправляет запрос в Seacraft
    public void TakeOff()
    {
        while (!Airborne)
            IncreaseRevs();
    }
    // Перенаправление прямо в Aircraft
    public int Height
    {
        get { return height; }
    }

    // Общий метод для Target и Adaptee
    public override void IncreaseRevs()
    {
        base.IncreaseRevs();
        if (Speed > 40)
            height += 100;
    }
    public bool Airborne
    {
        get { return height > 50; }
    }
}
class Experiment_MakeSeaBirdFly
{
    static void Main()
    {
        // Без адаптера
        Console.WriteLine("Experiment 1: test the aircraft engine");
        IAircraft aircraft = new Aircraft();
        aircraft.TakeOff();
        if (aircraft.Airborne) Console.WriteLine(
            "The aircraft engine is fine, flying at "
            + aircraft.Height + "meters");
        // Обычное использование адаптера
        Console.WriteLine("\nExperiment 2: Use the engine in the Seabird");
        IAircraft seabird = new Seabird();
        seabird.TakeOff(); // скорость автоматически увеличивается
        Console.WriteLine("The Seabird took off");
        // Двусторонний адаптер: используются команды от seacraft, примененные к объекту

        // IAircraft object
        // (которых нет в интерфейсе IAircraft)
        Console.WriteLine("\nExperiment 3: Increase the speed of the Seabird:");
        (seabird as ISeacraft).IncreaseRevs();
        (seabird as ISeacraft).IncreaseRevs();
        if (seabird.Airborne)
            Console.WriteLine("Seabird flying at height " + seabird.Height +
                " meters and speed " + (seabird as ISeacraft).Speed + " knots");
        Console.WriteLine("Experiments successful; the Seabird flies!");
    }
}
/* Output
Experiment 1: test the aircraft engine
Aircraft engine takeoff
The aircraft engine is fine, flying at 200 meters
Experiment 2: Use the engine in the Seabird
Seacraft engine increases revs to 10 knots
Seacraft engine increases revs to 20 knots
Seacraft engine increases revs to 30 knots
Seacraft engine increases revs to 40 knots
Seacraft engine increases revs to 50 knots
The Seabird took off
Experiment 3: Increase the speed of the Seabird:
Seacraft engine increases revs to 60 knots
Seacraft engine increases revs to 70 knots
Seabird flying at height 300 meters and speed 70 knots
Experiments successful; the Seabird flies!
*/

```

\*/

## Встраиваемые адаптеры

При разработке зачастую необходимо использование сторонних компонентов. Компоненты могут не быть определены заранее, поэтому важно предусмотреть те места в системе, которые могут потребовать таких изменений и оснастить их соответствующими адаптерами. Чем меньше интерфейс таких адаптеров, тем вероятнее, что к ним можно будет легко подключиться. С технической точки зрения эти адаптеры не отличаются от обычных, их отличие в размере интерфейса. Адаптеры с небольшим количеством методов в интерфейсе называются встраиваемыми. Отличительной чертой такого адаптера также является то, что название метода, вызываемого клиентом, может не совпадать с названием метода в интерфейсе ITarget. Адаптер должен уметь обрабатывать изменение названий методов. Для рассмотренных выше вариантов адаптеров это требование касалось методов адаптируемого объекта, а клиент был обязан использовать методы из ITarget. Предположим, что клиент хочет использовать свои названия методов или существуют несколько клиентов, у каждого из которых своя терминология. Чтобы добиться возможности динамического изменения методов мы будем использовать делегаты. Рассмотрим пример, в котором показано, как можно создать встраиваемый адаптер с делегатами.

```
1 using System;
2
3 // Adapter Pattern
4 // Адаптер может принимать любое число встраиваемых адаптируемых объектов
5 // и перенаправлять запросы через делегаты, в некоторых случаях
6 // используются анонимные делегаты
7
8 // Старая форма запроса (адаптируемый объект)
9 class Adaptee {
10     public double Precise(double a, double b) {
11         return a / b;
12     }
13 }
14
15 // Новая форма запроса
16 class Target {
17     public string Estimate(int i) {
18         return "Estimate is " + (int)Math.Round(i / 3.0);
19     }
20 }
21
22 // Адаптация запросов в старой форме к новой
23 class Adapter : Adaptee {
24     public Func<int, string> Request;
25
26     // Различные конструкторы для ожидаемых адаптируемых объектов
27
28     // Adapter-Adaptee
29     public Adapter(Adaptee adaptee) {
30         // Инициализация делегата старым способом запроса
31         Request = delegate(int i) {
32             return "Estimate based on precision is " +
33                 (int)Math.Round(adaptee.Precise(i, 3));
34         };
35     }
36
37     // Adapter-Target
```

```

38     public Adapter(Target target) {
39         // Инициализация делегата новым способом запроса
40         Request = target.Estimate;
41     }
42 }
43
44 class Client {
45
46     static void Main() {
47
48         Adapter adapter1 = new Adapter(new Adaptee());
49         Console.WriteLine(adapter1.Request(5));
50
51         Adapter adapter2 = new Adapter(new Target());
52         Console.WriteLine(adapter2.Request(5));
53
54     }
55 }
56 /* Output
57 Estimate based on precision is 2
58 Estimate is 2
59 */

```

Делегат содержится в адаптере. Его экземпляр создается на строке 24 и имеет форму одного из стандартных делегатов. На строках 33 и 40, делегат получает значение в виде методов Precise и Estimate. Precise является методом адаптируемым к виду метода Estimate. В строках 31-34 показано использование анонимной функции для дополнения строки результата, полученного из Adaptee. Обратите внимание, что клиент использует метод с названием, которое ему хочется использовать.

Ассоциация между названием типа объекта, чей метод был присвоен делегату, и методом теряется в момент присвоения делегату метода, т.е. в конструкторе адаптера.

Встраиваемый адаптер имеет конструктор для каждого типа объектов, который он собирает адаптировать. В каждом конструкторе происходит инициализация делегата одним или несколькими нужными методами.

В C# 3.0 введены так называемые стандартные типы делегатов:

```

delegate R Func<R>( );
delegate R Func<A1, R>(A1 a1);
delegate R Func<A1, A2, R>(A1 a1, A2 a2);

```

есть и с большим количеством аргументов.

Здесь R – тип возвращаемого значения, Ai – типы аргументов.

Поэтому можно использовать эти объявления для своих делегатов, также их ожидают некоторые методы классов .NET Framework 3.0.

Для объявления экземпляра делегата нужно:

объявить делегат с именем Request, который принимает целый параметр и возвращает строку:

```
public Func <int,string> Request;
```

и присвоить делегату требуемый метод, например:

```
Request = Target.Estimate;
```

Делегат может быть вызван командой:

```
string s = Request(5);
```

Этот вызов приведет к выполнению метода Estimate класса Target.

## Порождающие паттерны. Абстрактная фабрика (Abstract Factory). Строитель (Builder)

Среди порождающих шаблонов особенно популярны фабрики. Фабрики создают семейства взаимосвязанных объектов.

### **Абстрактная фабрика**

#### **Назначение**

Предназначен для создания семейств объектов, которые должны существовать только совместно. Абстрактные фабрики воплощаются в конкретные фабрики, которые и создают объекты различных типов и в различных комбинациях. Подход изолирует определения объектов и имена их классов от клиента, чтобы он мог создать их только с помощью фабрики. Поэтому семейства объектов могут быть заменены и клиент об этом не должен узнать.

#### **Пример**

Сейчас в мире полно подделок различных товаров. Продукция известных фирм копируется и выдается за оригинал. Иногда такие товары называют репликами и пользователь сознательно покупает копии. Однако в большинстве случаев пользователь оказывается в неведении, какая фабрика на самом деле сделала товар, который он покупает.

Шаблон Абстрактная фабрика создает похожий непрозрачный слой, через который клиент не видит, какая фабрика производит объекты. Другими словами, появляется возможность заменить фабрику на другую, производящую похожие объекты и при этом сохраняются требования контракта (интерфейса), который она реализует.

Другой пример

#### **Дизайн**

Клиент пользуется конкретной фабрикой, реализующей интерфейс `AbstractFactory`. Через нее он запрашивает объекты типов `A` и `B`. Однако, первоначально клиент не знает, какие конкретно объекты ему предоставит фабрика, поэтому у него они имеют абстрактные типы `IProductA`, `IProductB`. Это позволяет сделать систему независимой от того, как именно реализованы конкретные классы объектов. Клиент даже не знает названия конкретных классов, объекты которых делает для него фабрика.

`IFactory` – интерфейс с операцией `Create` для каждого абстрактного объекта (абстрактная фабрика).

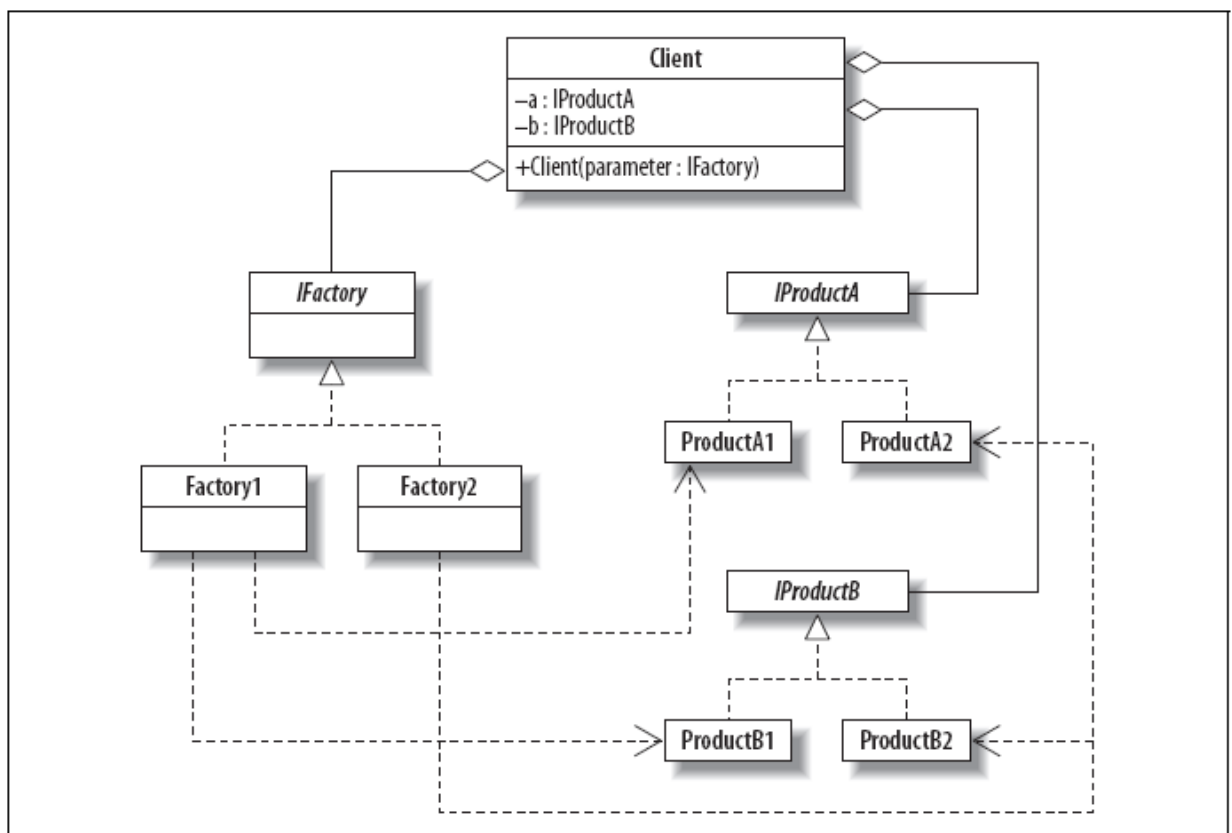
Factory1, Factory2 – реализации абстрактной фабрики, конкретные фабрики с реализацией метода Create.

IProductA, IProductB – интерфейсы, которым удовлетворяют объекты из создаваемых групп.

ProductA1, ProductA2, ProductB1, ProductB2 – объекты, создаваемые соответствующими фабриками.

Client – клиент, который знает только о IFactory и IProductA, IProductB.

Интересной особенностью шаблона является то, что семейства целиком могут быть заменены во время выполнения программы. Это возможно из-за того, что объекты одного семейства реализуют соответствующие интерфейсы.



Client	Человек, который покупает, например, сумку и обувь
IFactory	Интерфейс фабрик, в котором есть методы, возвращающие сумку или обувь
Factory1	Gucci
Factory2	Poochy
IProductA	Интерфейс сумок
IProductB	Интерфейс обуви
ProductA1	Сумка от Gucci
ProductA2	Сумка от Poochy
ProductB1	Обувь от Gucci
ProductB2	Обувь от Poochy

Воспользуемся параметризованными интерфейсами для того, чтобы описать в одном классе все разновидности фабрик, а не использовать наследование.

```
interface IFactory<Brand> where Brand : IBrand
{
    IBag CreateBag();
    IShoes CreateShoes();
}
// Фабрики
class Factory<Brand> : IFactory<Brand>
where Brand : IBrand, new()
{
    public IBag CreateBag()
    {
        return new Bag<Brand>();
    }
    public IShoes CreateShoes()
    {
        return new Shoes<Brand>();
    }
}
```

Здесь мы применили возможность C# задавать ограничения на тип-параметр в виде: `class SortedList<T> where T: IComparable<T> {...}`. Т.е. указывается, что тип T должен реализовывать интерфейс IComparable. `new()` означает, что тип T должен иметь конструктор.

В нашем случае, конкретные объекты – это обувь и сумки. Классы этих объектов также можно задать обобщенным образом.

```
// Товар 1
interface IBags
{
    string Material { get; }
}
// Конкретный Товар 1
class Bag<Brand> : IBag
where Brand : IBrand, new()
{
    private Brand myBrand;
    public Bag()
    {
        myBrand = new Brand();
    }
    public string Material { get { return myBrand.Material; } }
}
```

Фабрика создается так:

```
IFactory<Brand> factory = new Factory<Brand>( );
```

Полный текст примера:

```
using System;

namespace AbstractFactoryPattern
{
    interface IFactory<Brand>
    where Brand : IBrand
    {
        IBag CreateBag();
    }
}
```

```

        IShoes CreateShoes();
    }

    // Конкретные Фабрики
    class Factory<Brand> : IFactory<Brand>
        where Brand : IBrand, new()
    {
        public IBag CreateBag()
        {
            return new Bag<Brand>();
        }

        public IShoes CreateShoes()
        {
            return new Shoes<Brand>();
        }
    }

    // Product 1
    interface IBag
    {
        string Material { get; }
    }

    // Product 2
    interface IShoes
    {
        int Price { get; }
    }

    // Конкретный Product 1
    class Bag<Brand> : IBag
        where Brand : IBrand, new()
    {
        private Brand myBrand;
        public Bag()
        {
            myBrand = new Brand();
        }

        public string Material { get { return myBrand.Material; } }
    }

    // Конкретный Product 2
    class Shoes<Brand> : IShoes
        where Brand : IBrand, new()
    {
        private Brand myBrand;

        public Shoes()
        {
            myBrand = new Brand();
        }

        public int Price { get { return myBrand.Price; } }
    }

    interface IBrand
    {
        int Price { get; }
        string Material { get; }
    }

```

```

class Gucci : IBrand
{
    public int Price { get { return 1000; } }
    public string Material { get { return "Crocodile skin"; } }
}

class Poochy : IBrand
{
    public int Price { get { return new Gucci().Price / 3; } }
    public string Material { get { return "Plastic"; } }
}

class Groundcover : IBrand
{
    public int Price { get { return 2000; } }
    public string Material { get { return "South african leather"; } }
}

class Client<Brand>
    where Brand : IBrand, new()
{
    public void ClientMain()
    {
        IFactory<Brand> factory = new Factory<Brand>();

        IBag bag = factory.CreateBag();
        IShoes shoes = factory.CreateShoes();

        Console.WriteLine("I bought a Bag which is made from " +
bag.Material);
        Console.WriteLine("I bought some shoes which cost " +
shoes.Price);
    }
}

static class Program
{
    static void Main()
    {
        // Call Client twice
        new Client<Poochy>().ClientMain();
        new Client<Gucci>().ClientMain();
        new Client<Groundcover>().ClientMain();
    }
}

/* Output
I bought a Bag which is made from Plastic
I bought some shoes which cost 333
I bought a Bag which is made from Crocodile skin
I bought some shoes which cost 1000
I bought a Bag which is made from South african leather
I bought some shoes which cost 2000
*/

```

Ограничение этого шаблона заключается в том, что не так просто добавить новые группы объектов-продуктов. Название и количество интерфейсов объектов-продуктов жестко заданы в классе абстрактной фабрики. Следовательно, при добавлении нового класса продукта и интерфейс абстрактной фабрики и реализации в конкретных фабриках должны быть изменены. И даже клиент должен быть изменен.

## Задачи.

1. Добавить к товарам, которые производят фабрики, джинсы.
2. Создать фабрики, производящие автомобильные колеса разного диаметра (например, 16 и 20 дюймов) и соответствующие диски к ним.
3. В примере заменить параметризацию созданием подклассов.

## Шаблон Строитель

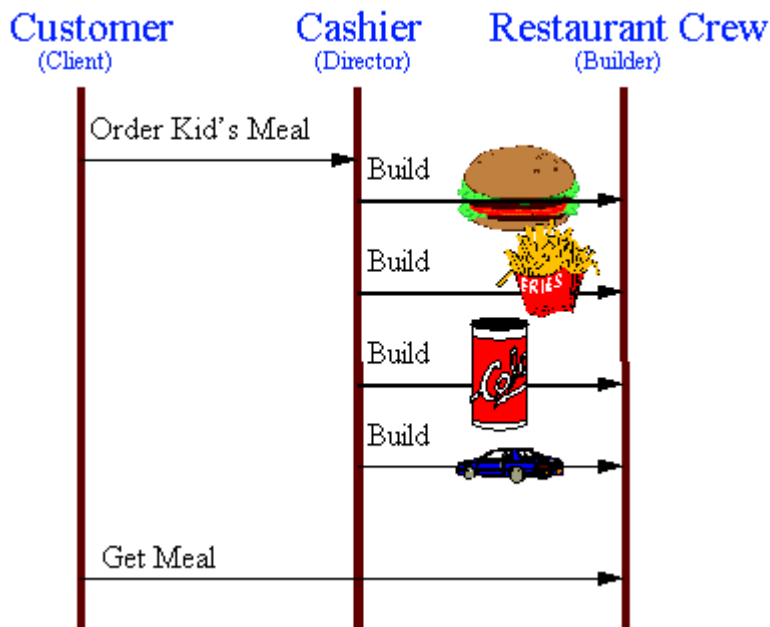
### Назначение

Строитель отделяет спецификацию сложного объекта от его создания. Один и тот же технологический процесс может давать разные по свойствам объекты.

Продолжим пример с подделками. Пусть есть онлайн-магазин, который принимает заказы и на настоящие товары и на имитации. Естественно, что времени на изготовление настоящего товара потребуется больше. Естественно, между товарами будет и различие. Каждая сумка состоит из нескольких частей. Причем настоящая сумка наверняка будет иметь больше частей, чем подделка.

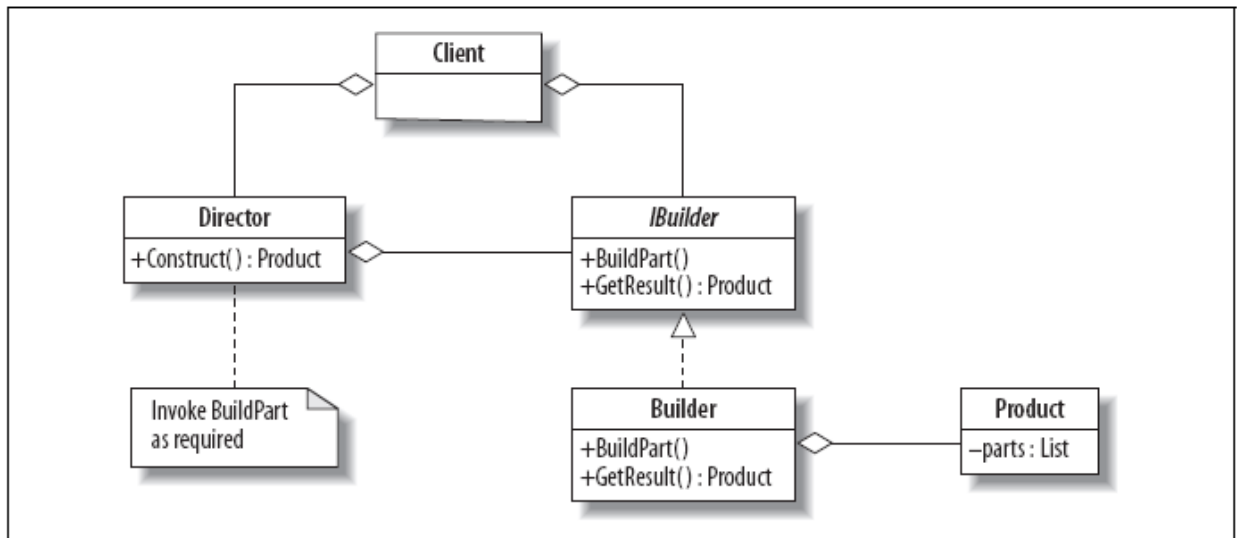
Строитель содержит все подробности того, как будет сделан товар и некоторую информацию об этом процессе передает пользователю. Например, требуемое время на изготовление товара.

### Другой пример



### Дизайн

В шаблоне Строитель используются Директоры и Строители. В схеме может быть любое число Строителей, удовлетворяющих интерфейсу `IBuilder`. Каждый из них может быть вызван Директором, чтобы в итоге создать продукт согласно спецификации. Строители создают части продуктов, из которых Директор собирает целый Продукт. Разные Строители могут создавать разные части продуктов.



В данной схеме могут быть несколько Директоров и несколько Строителей. Клиент вызывает метод `Construct` у определенного Директора, у которого есть определенный Строитель. Директор знает, как из частей продукта, которые создает Строитель, собрать продукт целиком. В объекте `Product` накапливаются создаваемые части продукта и этот набор является результатом общего процесса конструирования.

<code>IBuilder</code>	Интерфейс, описывающий действия с тем, что должно быть построено
<code>Director</code>	Содержит последовательность технологических операций, которая должна быть выполнена, чтобы создать <code>Product</code>
<code>Builder</code>	Вызывается Директором, чтобы создать части Продукта
<code>Product</code>	Конструируемый по частям объект

Преимущество введения и директора и строителя в том, что продукты не обязательно должны получаться одинаковыми. Строитель может изменить свой способ создания продукта и даже если директор останется тем же, продукт получится другим. Итак, при использовании шаблона Строитель продукт конструируется по шагам под управлением директора.

Для нашего примера клиент должен создать Директора:

```

class Director
{
    // Build a Product from several parts
    public void Construct(IBuilder builder)
    {
        builder.BuildPartA();
        builder.BuildPartB();
        builder.BuildPartB();
    }
}

```

Здесь видно, что сначала создается часть А, а потом создаются две части В. Однако, результат будет разным в зависимости от того, какой именно Строитель будет передан в этот метод.

Таким образом, полный текст программы:

```
using System;
using System.Collections.Generic;

class Director
{
    // Builder uses a complex series of steps
    public void Construct(IBuilder builder)
    {
        builder.BuildPartA();
        builder.BuildPartB();
        builder.BuildPartB();
    }
}

interface IBuilder
{
    void BuildPartA();
    void BuildPartB();
    Product GetResult();
}

class Builder1 : IBuilder
{
    private Product product = new Product();
    public void BuildPartA()
    {
        product.Add("PartA ");
    }

    public void BuildPartB()
    {
        product.Add("PartB ");
    }

    public Product GetResult()
    {
        return product;
    }
}

class Builder2 : IBuilder
{
    private Product product = new Product();
    public void BuildPartA()
    {
        product.Add("PartX ");
    }

    public void BuildPartB()
    {
        product.Add("PartY ");
    }

    public Product GetResult()
    {
        return product;
    }
}
```

```

}

class Product
{
    List<string> parts = new List<string>();
    public void Add(string part)
    {
        parts.Add(part);
    }

    public void Display()
    {
        Console.WriteLine("\nProduct Parts -----");
        foreach (string part in parts)
            Console.Write(part);
        Console.WriteLine();
    }
}

public class Client
{
    public static void Main()
    {
        // Создаем одного директора и двух строителей
        Director director = new Director();

        IBuilder b1 = new Builder1();
        IBuilder b2 = new Builder2();

        // Создаем два продукта
        director.Construct(b1);
        Product p1 = b1.GetResult();
        p1.Display();

        director.Construct(b2);
        Product p2 = b2.GetResult();
        p2.Display();
    }
}
/* Output
Product Parts -----
PartA PartB PartB

Product Parts -----
PartX PartY PartY
*/

```

Переделаем пример, приведенный в разделе Абстрактная Фабрика так, чтобы использовались строители. В целом, реализация похожа, но отличие в том, что Директор содержит больше функциональности, чем требуется Фабрикам. Рассмотрим одного из двух Директоров:

```

class Gucci : IBrand
{
    public IBag CreateBag()
    {
        Bag b = new Bag();
        Program.DoWork("Cut Leather", 250);
        Program.DoWork("Sew leather", 1000);
        b.Properties += "Leather";
    }
}

```

```

        Program.DoWork("Create Lining", 500);
        Program.DoWork("Attach Lining", 1000);
        b.Properties += " lined";
        Program.DoWork("Add Label", 250);
        b.Properties += " with label";
        return b;
    }
}

```

Метод DoWork моделирует задержку при производстве частей товара.

Полный текст примера:

```

using System;
using System.Diagnostics;
using System.IO;
using System.Threading;

namespace BuilderPattern
{
    interface IBuilder<Brand>
        where Brand : IBrand
    {
        IBag CreateBag();
    }

    // Abstract Factory теперь реализуется с помощью Builder
    class Builder<Brand> : IBuilder<Brand>
        where Brand : IBrand, new()
    {
        Brand myBrand;
        public Builder()
        {
            myBrand = new Brand();
        }

        public IBag CreateBag()
        {
            return myBrand.CreateBag();
        }
    }

    // Продукт 1
    interface IBag
    {
        string Properties { get; set; }
    }

    // Конкретный Продукт 1
    class Bag : IBag
    {
        public string Properties { get; set; }
    }

    // Directors
    interface IBrand
    {
        IBag CreateBag();
    }

    class Gucci : IBrand
    {
        public IBag CreateBag()
    }
}

```

```

    {
        Bag b = new Bag();
        Program.DoWork("Cut Leather", 250);
        Program.DoWork("Sew leather", 1000);
        b.Properties += "Leather";
        Program.DoWork("Create Lining", 500);
        Program.DoWork("Attach Lining", 1000);
        b.Properties += " lined";
        Program.DoWork("Add Label", 250);
        b.Properties += " with label";
        return b;
    }
}

class Poochy : IBrand
{
    public IBag CreateBag()
    {
        Bag b = new Bag();
        Program.DoWork("Hire cheap labour", 200);
        Program.DoWork("Cut Plastic", 125);
        Program.DoWork("Sew Plastic", 500);
        b.Properties += "Plastic";
        Program.DoWork("Add Label", 100);
        b.Properties += " with label";
        return b;
    }
}

class Client<Brand>
    where Brand : IBrand, new()
{
    public void ClientMain()
    {
        IBuilder<Brand> factory = new Builder<Brand>();

        DateTime date = DateTime.Now;
        Console.WriteLine("I want to buy a bag!");
        IBag bag = factory.CreateBag();

        Console.WriteLine("I got my Bag which took " +
            DateTime.Now.Subtract(date).TotalSeconds * 5 + " days");
        Console.WriteLine("  with the following properties " +
            bag.Properties + "\n");
    }
}

static class Program
{
    static void Main()
    {
        // Call Client twice
        new Client<Poochy>().ClientMain();
        new Client<Gucci>().ClientMain();
    }

    public static void DoWork(string workitem, int time)
    {
        Console.Write("'" + workitem + " : 0%");
        Thread.Sleep(time);
        Console.Write("....25%");
        Thread.Sleep(time);
    }
}

```



При этом клиент уже не содержит поля связанные с типом продуктов, как в Абстрактной фабрике (ProductA и ProductB), потому что Product содержит лишь список составляющих его частей, каждая часть может иметь разное наполнение. Это зависит от Директора.

<b>Действие</b>	<b>Абстрактная Фабрика</b>	<b>Строитель</b>
Клиент содержит	Фабрику и Продукты	Директора, Строителя и Продукт
Продукт создается через	Фабрику	Директора
Создание Продукта начинается с вызова	CreateProductA	Construct(builder)
Фабрика/Строитель возвращает	Определенный продукт	Часть продукта
Продукт содержит	Определенные свойства	Список составных частей

# Структурные паттерны. Паттерн Фасад (Façade)

## Назначение

Назначение паттерна Фасад - обеспечить высокоуровневое представление подсистем, детали которых скрыты от пользователей. Операции, необходимые пользователям фасада, могут быть вызваны из разных подсистем.

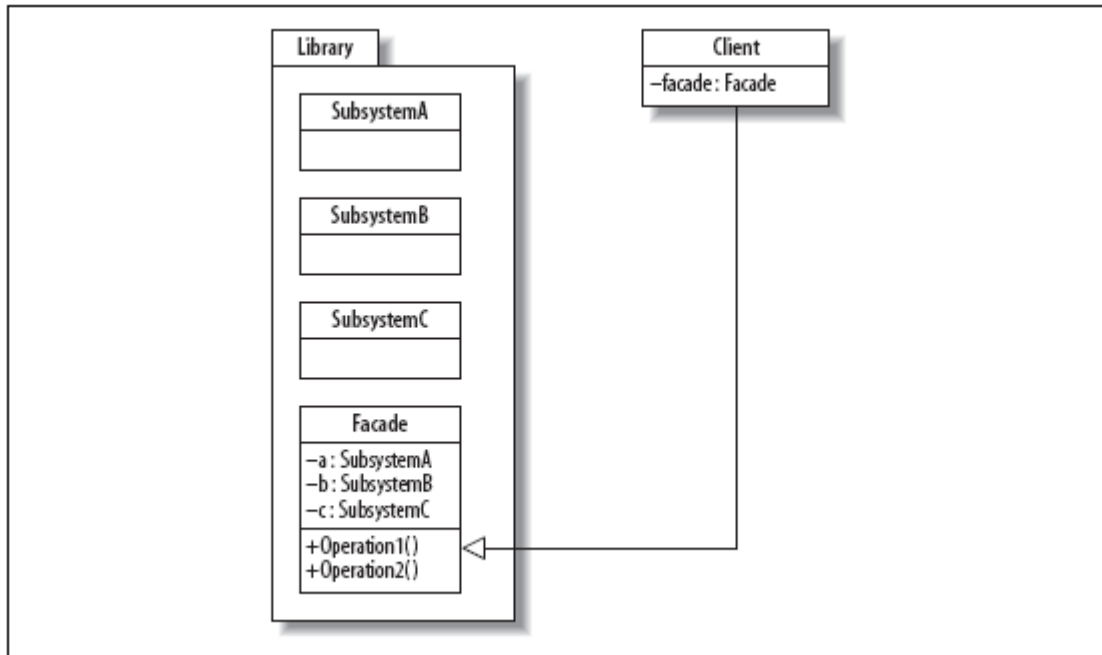
## Исходная ситуация, в которой может возникнуть необходимость в шаблоне

Фасад может быть полезен в разных ситуациях. Существует множество примеров, когда компьютерная система построена на наборе независимых подсистем. Один из всем известных примеров - компилятор: он состоит из четко определяемых подсистем, которые называются: лексический анализатор, семантический анализатор, синтаксический анализатор, генератор кода и некоторые оптимизаторы. В современных компиляторах каждая подсистема решает несколько подзадач. Разные задачи и подзадачи вызываются в одной последовательности, но не все из них обязательны. Например, если возникает ошибка во время работы одного из анализаторов, то компилятор не должен переходить к следующим этапам работы. Скрытие этой особенности за Фасадом позволяет программе вызывать задачи в подсистемах в логическом порядке, передавая необходимые структуры данных между ними.

## Сравнение с другими известными структурными шаблонами

Что отличает Фасад от, скажем, Декоратора или Адаптера, это то, что интерфейс, который он строит, может быть совершенно новым. Он может не отвечать существующим требованиям и не реализовывать существующие интерфейсы. Также может быть создано несколько Фасадов вокруг одного существующего набора подсистем. Термин "подсистема" здесь используется в более широком смысле; мы говорим о более высоком уровне, нежели классы.

## Диаграмма



### Роли в диаграмме

#### Namespace 1

Библиотека подсистем - исходный элемент, существовавший до применения шаблона

#### Subsystem

Класс, предоставляющий конкретные операции

#### Facade

Класс, предоставляющий несколько высокоуровневых операций, таких как выбор конкретных подсистем

#### Namespace 2

Место нахождения Client

#### Client

Вызывает высокоуровневые операции в Facade в Namespace 1

## Порождающие паттерны. Шаблон Одиночка (Singleton)

### Назначение

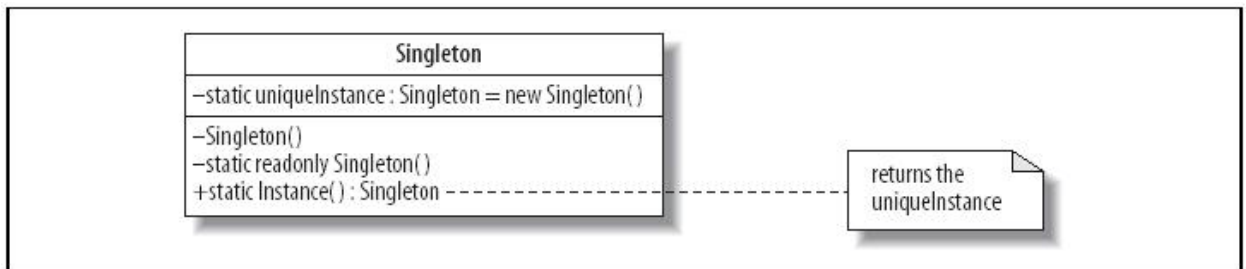
Шаблон гарантирует, что создается лишь один экземпляр некоторого класса, и предоставляет к нему глобальную точку доступа. Сам класс контролирует то, что у него есть только один экземпляр и может запретить создание дополнительных экземпляров, перехватывая запросы на создание новых объектов. Он же предоставляет доступ к своему экземпляру. Используется, если в системе должен быть ровно один экземпляр класса, легко доступный всем остальным участникам.

Многие другие шаблоны - такие как Строитель, Прототип и Абстрактная Фабрика - могут использовать шаблон Одиночка, чтобы гарантировать создание только одного экземпляра класса.

## Сравнение с другими известными структурными шаблонами

Все три шаблона - Прототип, Фабричный метод и Одиночка - используются для генерации экземпляров классов. Этот шаблон отличается тем, что гарантирует создание только одного экземпляра класса.

## Диаграмма



## Роли в диаграмме

### Singleton

Класс, содержащий механизм генерации уникального экземпляра. Исходным элементом является класс, которому необходимо обеспечить единственность экземпляра.

### Instance

Свойство, используемое для создания и доступа к экземпляру класса Singleton.

## Порождающие паттерны. Шаблон Фабричный метод (Factory Method)

### Назначение

Шаблон дает возможность выбрать тип создаваемого объекта во время выполнения программы. Это позволяет манипулировать абстрактными объектами. Используется, когда классу заранее не известно, объекты каких типов ему надо создать.

### Исходная ситуация, в которой может возникнуть необходимость в шаблоне

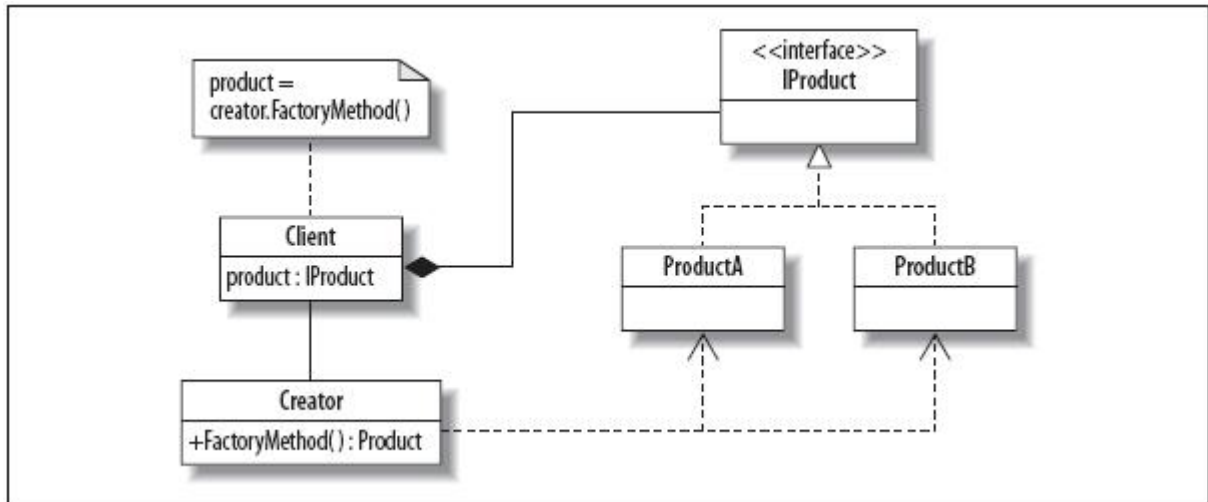
Фабричный метод - это шаблон, который позволяет достичь независимости от классов, специфичных для данного приложения. Клиент программирует, основываясь на интерфейсе (в данном случае IProduct) и оставляет шаблон разбираться с созданием экземпляров. Конкретное преимущество Фабричного метода в том, что он позволяет соединять параллельные иерархии классов. Например, пользователю нужно получить авокадо. Существует два канала их поставки и способы, которыми эти каналы получают

авокадо пользователя не интересуют. Если каждый канал имеет своего Creator, который содержит FactoryMethod, то каждый из этих FactoryMethod может быть использован пользователем для получения авокадо.

## Сравнение с другими известными структурными шаблонами

Все три шаблона - Прототип, Фабричный метод и Одиночка - используются для генерации экземпляров классов. Этот шаблон отличается тем, что позволяет выбрать экземпляр какого подкласса создавать.

## Диаграмма



## Роли в диаграмме

IProduct  
Интерфейс продуктов.

Product A и Product B  
Классы, реализующие IProduct. Исходные элементы.

Creator  
Реализует FactoryMethod.

FactoryMethod  
Решает, экземпляр какого класса создавать.

## Порождающие паттерны. Шаблон Прототип (Prototype)

### Назначение

Шаблон Прототип предоставляет возможность создания новых объектов путем копирования некоторого существующего прототипа. Шаблон используется, когда система не должна зависеть от того, как в ней создаются, компонуются и предоставляются объекты или классы.

## Исходная ситуация, в которой может возникнуть необходимость в шаблоне

Шаблон Прототип поглощает один экземпляр объекта и использует его как образец для копирования для всех будущих экземпляров. Проекты, которые активно используют Декоратор и Композит, также могут получить преимущества от использования прототипа. Прототипы полезны, когда инициализация объекта ресурсоемка и ожидаются несколько вариантов инициализирующих параметров. Возможны несколько вариантов использования шаблона:

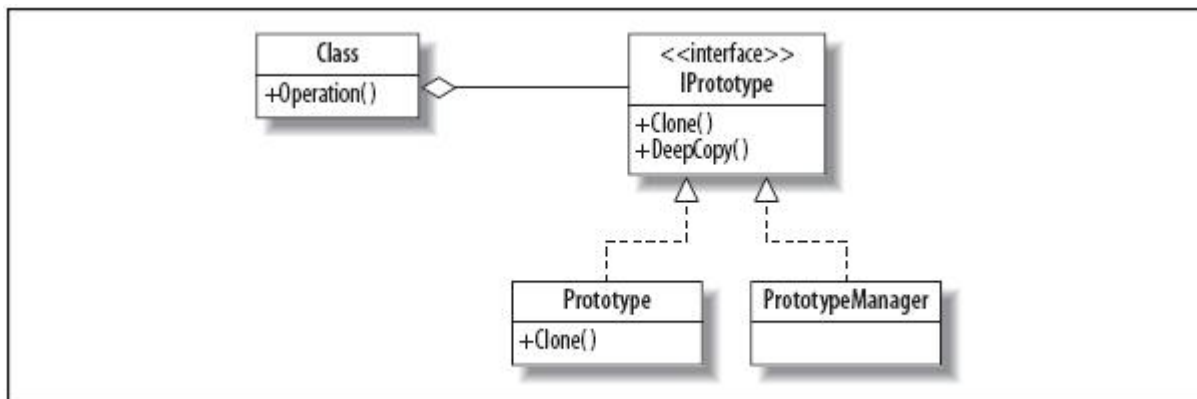
1) скажем, вы загружаете набор объектов в память, и пользователь выбирает копируемые объекты. При реализации выбора пользователя, обычно существуют только несколько вариантов объектов. Они описаны как прототипы и определяются в программе путем сравнения с допустимыми вводимыми значениями (такими как строки, числа или символы).

2) в программе существуют составные структуры, и некоторые их части должны быть скопированы, допустим, для архивации. Какую часть копировать обычно определяет ввод пользователя. Копия станет прототипом, который не должен быть изменен.

## Сравнение с другими известными структурными шаблонами

Все три шаблона - Прототип, Фабричный метод и Одиночка - используются для генерации экземпляров классов. Этот шаблон отличается тем, что является генератором копий единожды заданного объекта.

## Диаграмма



## Роли в диаграмме

IPrototype

Интерфейс, который определяет, что прототипы должны быть копируемы.

Prototype

Класс с возможностями копирования, исходный элемент.

PrototypeManager

Хранит индексированный список копируемых структур данных.

Client

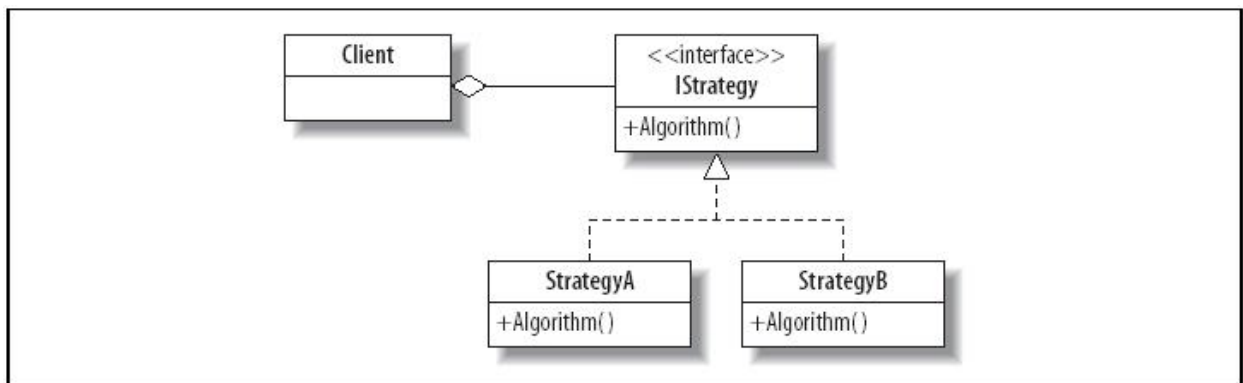
Добавляет прототипы в словарь и запрашивает клоны.

## Поведенческие паттерны. Паттерн Стратегия (Strategy)

### Назначение шаблона

Шаблон Стратегия дает возможность сделать выбор алгоритма из группы на основании контекста. Эта возможность осуществляется с помощью отделения процедуры выбора алгоритма от его реализации. Шаблон используется в ситуации, когда выбор алгоритма зависит от текущего состояния системы или обрабатываемых данных.

### Основные участники:



Client - класс, содержащий информацию, которая требуется для выбора алгоритма

IStrategy - интерфейс для всех алгоритмов, обладающий абстрактным методом

Algorithm(), который определяет способ вызова алгоритма

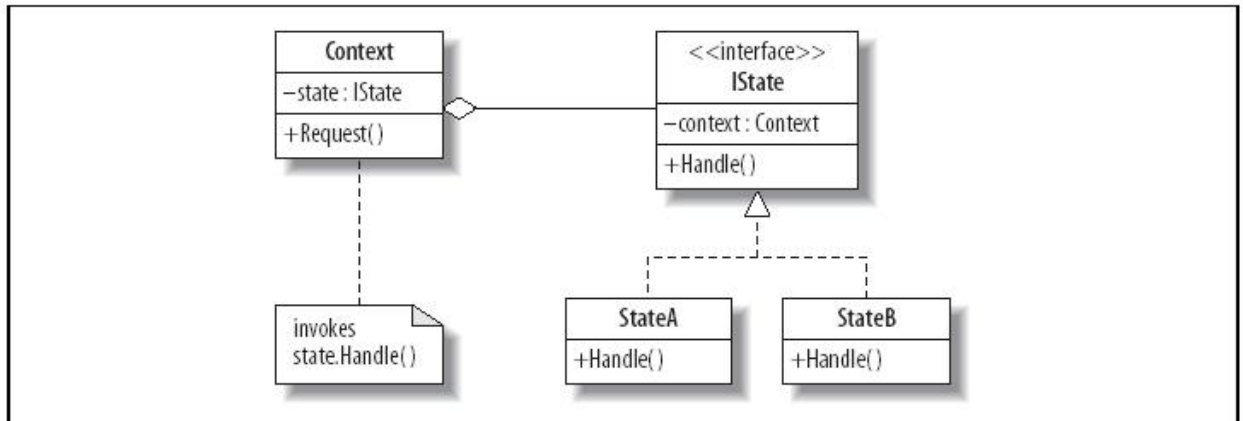
StrategyA и StrategyB - классы, реализующие интерфейс IStrategy

## Поведенческие паттерны. Паттерн Состояние (State)

### Назначение шаблона

Шаблон Состояние позволяет менять поведение объекта в зависимости от его состояния во время выполнения программы. Используется, когда есть объекты, которым понадобится менять свое поведение во время выполнения, исходя из некоторых условий. Шаблон можно рассматривать как динамичную версию шаблона Стратегия.

## Основные участники:

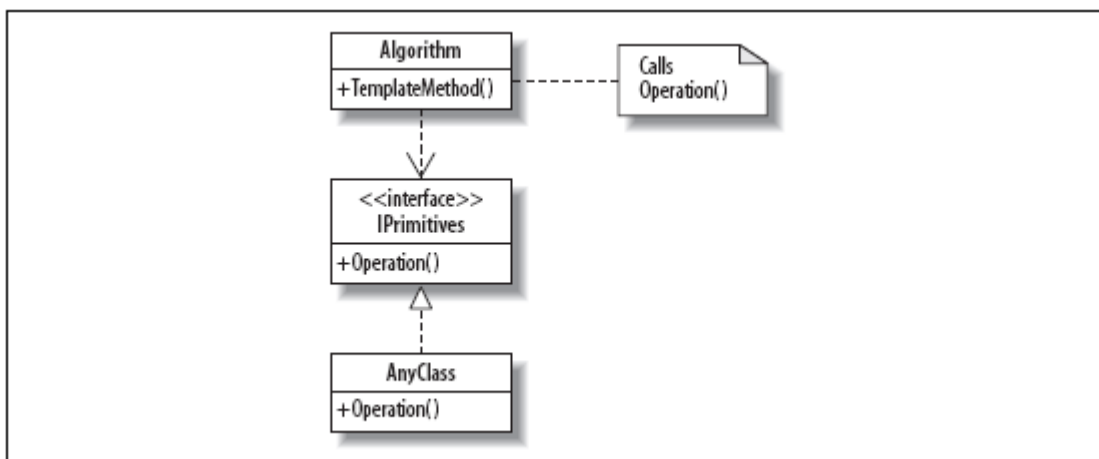


Context - класс, объекты которого должны менять свое поведение в зависимости от состояния, имеет переменную типа IState, которая сначала имеет конкретное состояние. Все запросы передаются с помощью вызова метода Handle()

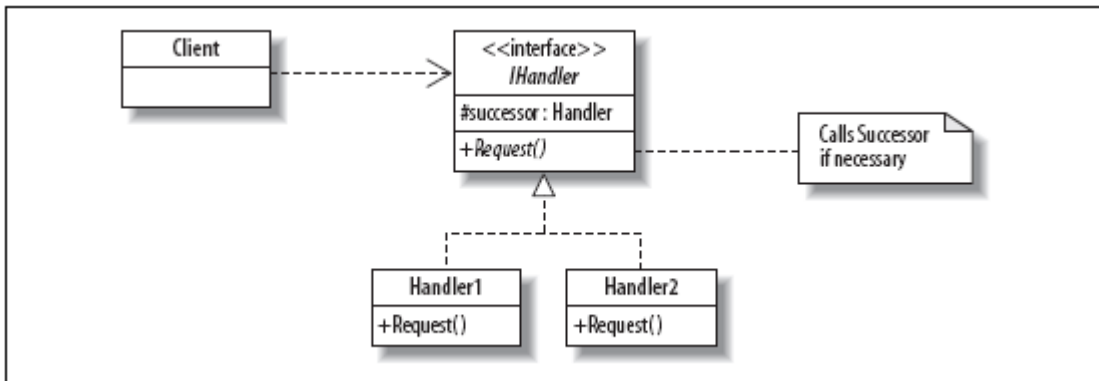
IState - интерфейс, который реализуют все состояния, содержит объект класса Context, поведение которого нужно будет изменить. Это необходимо для того, чтобы в процессе выполнения программы заменять объект состояния при появлении запроса.

StateA и StateB - классы конкретных состояний, содержат информацию о том, при каких условиях и в какие состояния может переходить объект из текущего состояния.

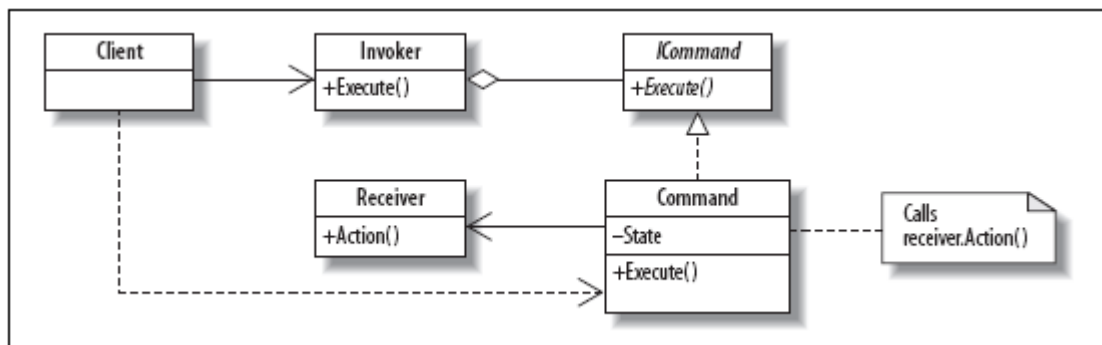
## Поведенческие паттерны. Паттерн Шаблонный метод (Template Method)



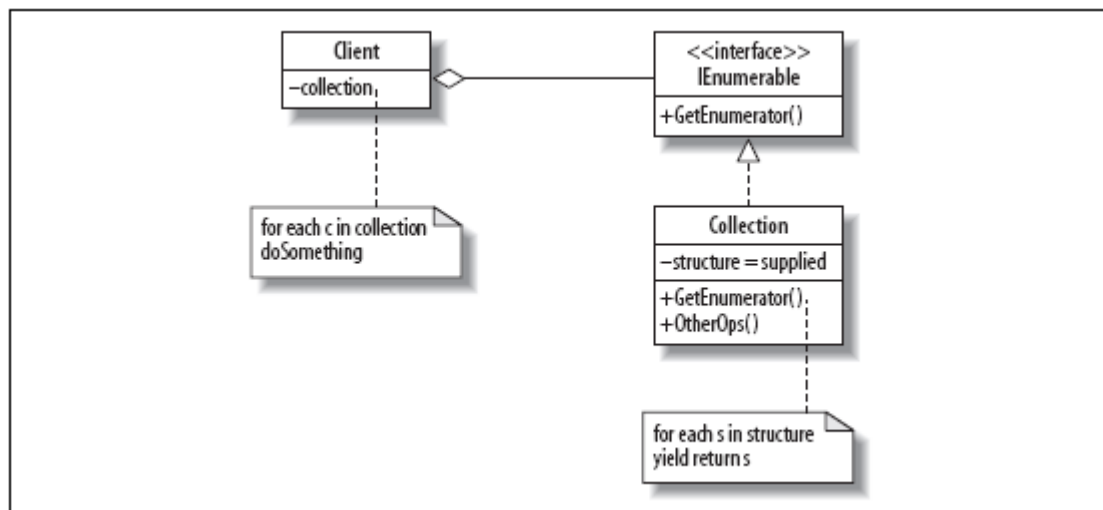
## Поведенческие паттерны. Паттерн Цепочка обязанностей (Chain of Responsibility)



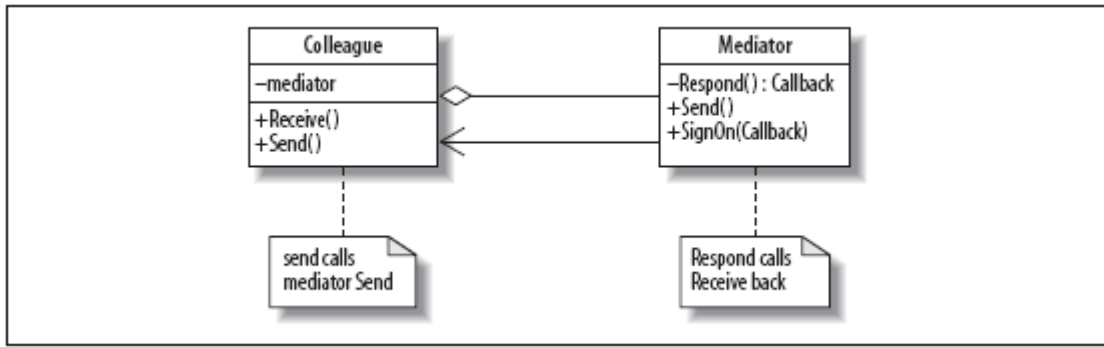
## Поведенческие паттерны. Паттерн Команда (Command)



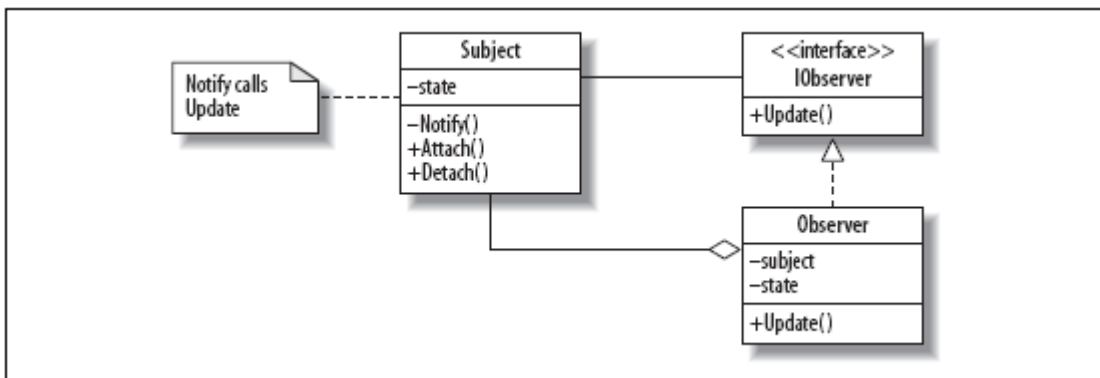
## Поведенческие паттерны. Паттерн Итератор (Iterator)



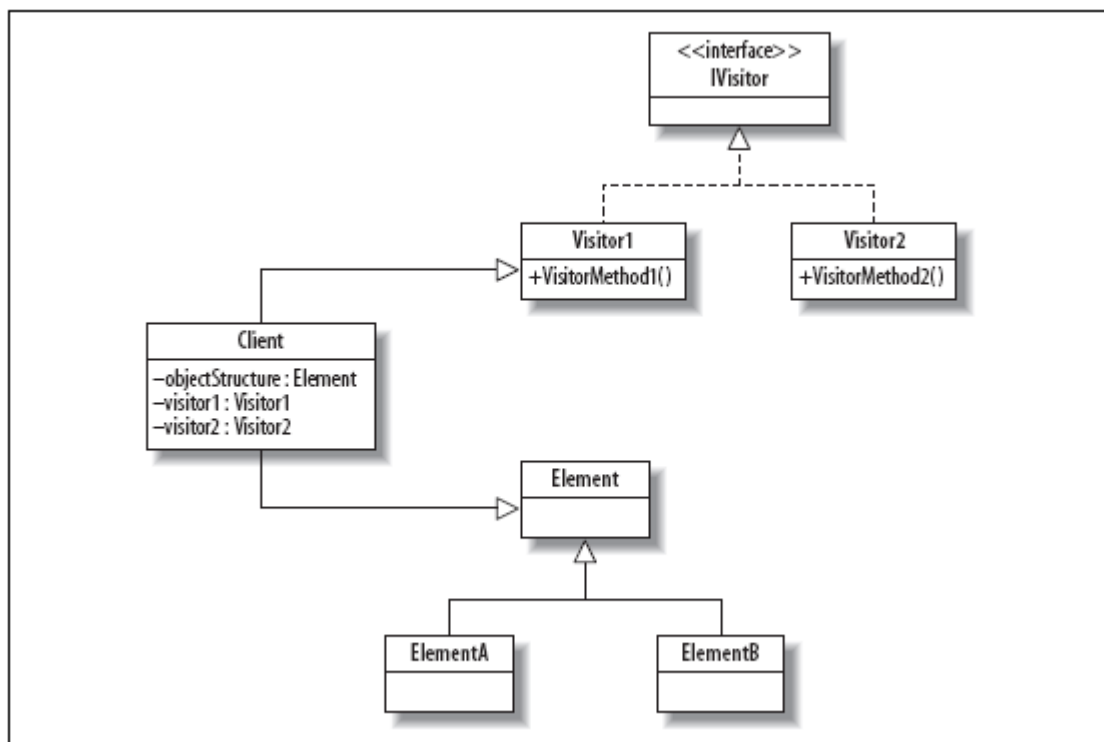
## Поведенческие паттерны. Паттерн Медиатор (Mediator)



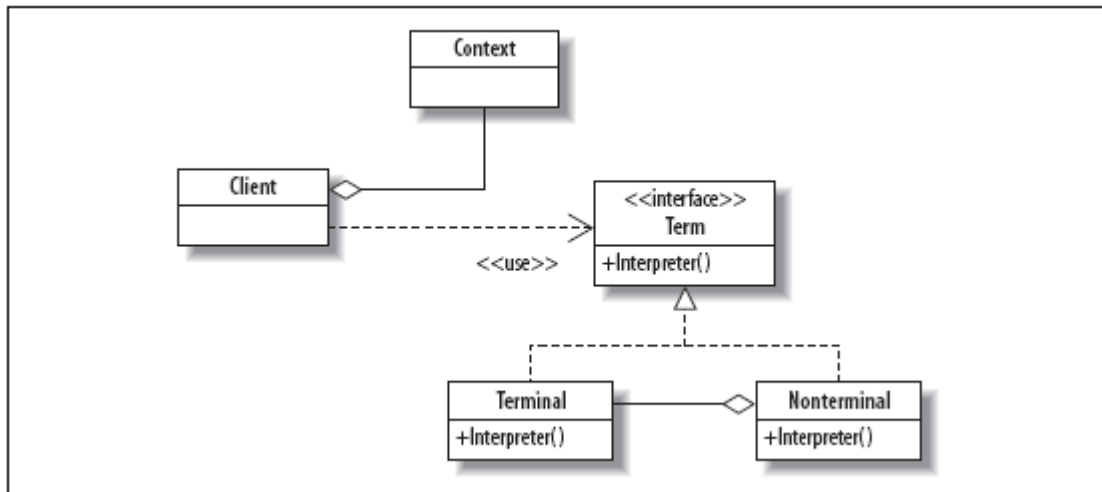
## Поведенческие паттерны. Паттерн Наблюдатель (Observer)



## Поведенческие паттерны. Паттерн Посетитель (Visitor)



## Поведенческие паттерны. Паттерн Интерпретатор (Interpreter)



## Поведенческие паттерны. Паттерн Хранитель (Memento)

